ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

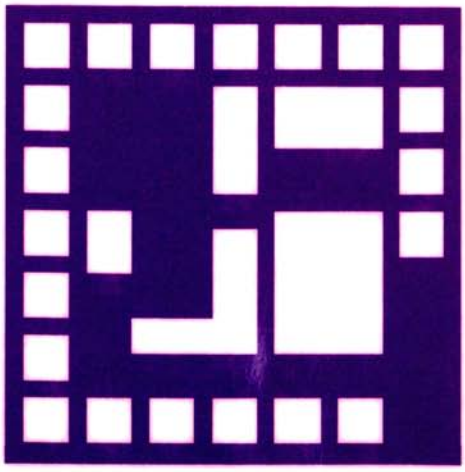
Traité d'Électricité

PUBLIÉ SOUS LA DIRECTION DE JACQUES NEIRYNCK

VOLUME XIV

CALCULATRICES

Jean-Daniel Nicoud



PRESSES POLYTECHNIQUES ROMANDES

XIV
CALCULATRICES

TRAITÉ D'ÉLECTRICITÉ

DE L'ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
PUBLIÉ SOUS LA DIRECTION DE JACQUES NEIRYNCK

VOLUME XIV

CALCULATRICES

par Jean-Daniel Nicoud

PRESSES POLYTECHNIQUES ROMANDES

Cet ouvrage fait partie d'une série de vingt-deux volumes
dont les titres sont les suivants :

I INTRODUCTION À L'ÉLECTROTECHNIQUE
II MATÉRIAUX DE L'ÉLECTROTECHNIQUE
III ÉLECTROMAGNÉTISME
IV THÉORIE DES RÉSEAUX DE KIRCHHOFF
V ANALYSE ET SYNTHÈSE DES SYSTÈMES LOGIQUES
VI THÉORIE ET TRAITEMENT DES SIGNAUX
VII DISPOSITIFS À SEMICONDUCTEUR
VIII ÉLECTRONIQUE
IX TRANSDUCTEURS ÉLECTROMÉCANIQUES
X MACHINES ÉLECTRIQUES
XI MACHINES SÉQUENTIELLES
XII ÉNERGIE ÉLECTRIQUE
XIII HYPERFRÉQUENCES
XIV CALCULATRICES
XV ÉLECTRONIQUE DE PUISSANCE
XVI ÉLECTRONIQUE DE RÉGLAGE ET DE COMMANDE
XVII SYSTÈMES DE MESURE
XVIII SYSTÈMES DE TÉLÉCOMMUNICATIONS
XIX FILTRES ÉLECTRIQUES
XX TRAITEMENT NUMÉRIQUE DES SIGNAUX
XXI ÉLECTROACOUSTIQUE
XXII HAUTE TENSION

Le Traité d'Electricité est une publication des
Presses polytechniques romandes, fondation scientifique
dont le but est principalement la diffusion des travaux de
l'Ecole polytechnique fédérale de Lausanne.
Le catalogue de ces publications peut être obtenu aux
Presses polytechniques romandes, CH-1015 Lausanne.

Deuxième édition
ISBN (série) : 2-604-00002-4
ISBN (ce volume) : 2-88074-054-1

CH-1015 Lausanne
Imprimé en Suisse

INTRODUCTION

Place du volume XIV dans le Traité d'Electricité

Dans la hiérarchie des modèles utilisés par l'ingénieur, le modèle logique traité en détail dans le volume V est suivi par le modèle numérique, dans lequel les éléments logiques sont associés pour agir sur des mots binaires représentant des nombres ou toute information codable sous forme de nombres. Le modèle numérique ne fait pas l'objet d'un volume particulier du Traité, car il ne prend de valeur et d'intérêt que lorsque les modules logiques complexes sont associés pour former un système programmable, dont le comportement ne dépend pas du câblage, mais de l'état d'une mémoire.

Appelons modèle de von Neumann ou modèle des calculatrices ce modèle qui guide le développement de l'informatique depuis 30 ans, et après lequel se développe un modèle, encore mal dégagé, qui couvre les systèmes informatiques et les réseaux d'ordinateurs. Les problèmes de l'intelligence artificielle dominent le tout; le modèle associatif se rapproche de celui du cerveau humain, mais la technologie est encore loin de permettre sa mise en valeur.

Le modèle de von Neumann permet de définir l'architecture d'une machine et de la programmer. Les contraintes technologiques sont omniprésentes dans cette définition, et l'évolution très rapide de la technologie entraîne l'évolution de tout le domaine informatique à un rythme qui ne se rencontre dans aucune autre science.

Ce livre tient compte de la technologie des circuits intégrés et de son évolution. Il est fortement imprégné des possibilités actuelles et des perspectives de développement des microprocesseurs. Toutefois les notions relatives aux microprocesseurs et aux gros ordinateurs y sont confondues, car le modèle est identique, et chaque nouvelle génération de microprocesseurs incorpore les caractéristiques précédemment réservées aux ordinateurs. L'optique choisie est toutefois résolument celle du concepteur et de l'ingénieur amené à inclure dans ses équipements des systèmes programmables plus ou moins spécialisés. L'aspect programmation a été développé dans le sens des applications en temps réel exigeant une efficacité maximale des programmes.

Objectifs

Ce volume n'est pas un livre d'introduction à l'informatique au sens traditionnel du terme. Il est destiné à des étudiants ayant eu une introduction sur les systèmes logiques et les microprocesseurs, et à des praticiens soucieux de mieux comprendre leur domaine de travail et désireux d'évoluer au même rythme que la technologie. Ce livre peut aussi aider l'informaticien traditionnel à «redescendre» au niveau du matériel et de ses contraintes afin de pouvoir configurer un système adapté à une application particulière, et l'animer par des programmes écrits dans les langages appropriés.

Les exemples sont généralement traités dans leur aspect fonctionnel, indépendant d'une technologie ou d'une architecture particulière. L'originalité, et nous espérons la valeur, de ce livre réside dans son niveau de généralité relativement invariant, mais proche des multiples réalisations utilisant des microprocesseurs et ordinateurs individuels, qui visent un champ d'application en continuelle expansion.

Ce volume n'est pas unique sur ce sujet. Il cherche à être complémentaire à la fois aux quelques bons livres orientés du côté des microprocesseurs et à l'abondante littérature sur l'informatique traditionnelle; de plus il cherche à établir le lien entre ces deux approches.

Le titre de ce livre surprendra plus d'un lecteur. Le terme «calculatrice» a un peu passé de mode, mais correspond à la généralité recherchée dans cet ouvrage, et convient mieux que les termes de calculettes, microprocesseurs ou ordinateurs qui recouvrent respectivement des calculatrices n'effectuant que des calculs arithmétiques simples, des unités de traitement d'information réalisées sous forme de circuits intégrés ou des équipements informatiques complexes. L'orientation générale est toutefois celle des microsystèmes, c'est-à-dire des systèmes informatiques basés sur un microprocesseur, et dont la complexité n'atteint pas celle des ordinateurs utilisés dans les centres de calcul, bien que la fonctionnalité soit équivalente.

Organisation générale du Volume XIV

Un premier chapitre introduit le sujet, justifie la structure générale du volume, et rappelle quelques notions importantes liées aux modules de base utilisés lors de la définition d'une architecture de calculatrice.

Le second chapitre est très complet et présente tous les problèmes liés à la représentation des nombres et aux opérations arithmétiques. Ce chapitre insiste sur les opérations arithmétiques, qui sont implémentées par les machines décrites au chapitre 3 et qui se retrouvent au chapitre 4 sous forme d'instructions en langage d'assemblage.

Une grande partie du chapitre 2 peut être sautée en première lecture; les problèmes de représentation, détection de dépassement de capacité, de comparaison de nombres, etc., sont très importants mais sont aussi plus faciles à assimiler lorsqu'ils apparaissent dans une application précise.

Le chapitre 3 introduit tous les concepts importants de l'architecture des ordinateurs en partant des petites calculettes bien connues de tous, en tous cas en ce qui concerne leur comportement extérieur. Il détaille les principes d'adressage et d'exécution des instructions dans les machines de von Neumann.

Le chapitre 4 développe surtout la programmation en langage d'assemblage, très importante pour comprendre la philosophie des systèmes programmables. Il insiste sur les concepts de base et les modes de représentation qui permettent à l'utilisateur de dominer son problème et de le résoudre efficacement.

Le chapitre 5 entre dans le détail de la structure et de la programmation des interfaces simples, en insistant sur les notions d'interruption, d'interfaces programmables et de communication série.

Le chapitre 6 aborde les caractéristiques d'un système et les problèmes de mise au point des programmes. Une étude superficielle des petits systèmes d'exploitation a pour but de familiariser le lecteur avec les outils qu'il peut être amené à utiliser tous les jours, et lui faciliter l'accès à une documentation plus spécialisée.

Conventions

Le traité d'Electricité est composé de volumes (vol.) repérés par des chiffres romains (vol. VI). Chaque volume est partagé en chapitres (chap.) repérés par des nombres arabes (chap. 2.). Chaque chapitre est divisé en sections (sect.) repérées par deux nombres arabes séparés par un point (sect. 2.3). Chaque section est divisée en paragraphes (§) repérés par trois nombres arabes séparés par deux points (§ 2.3.11). Les références internes stipulent le volume, le chapitre, la section ou le paragraphe du Traité auxquels on renvoie. Dans le cas de la référence à une partie du même volume, on omet le numéro de celui-ci. Les références bibliographiques sont numérotées continûment, et repérées par un seul nombre arabe entre crochets.

Un terme apparait en *italique maigre* la première fois qu'il est défini dans le texte. Etant donné l'importance de l'anglais dans ce domaine, l'équivalent d'un terme important est donné en italique entre parenthèses. Autant que possible, une terminologie correspondant aux usages reconnus ou les plus fréquents a été utilisée [3, 4].

Un paragraphe délicat ou compliqué est marqué par le signe ■ précédant son repère numérique; dans les exercices ce même signe peut également annoncer des calculs longs et fastidieux. Un paragraphe qui n'est pas indispensable à la compréhension de ce qui suit est précédé par le signe □ .

Les équations et éléments de programme hors texte sont numérotés continûment par chapitre et repérés par deux nombres arabes placés entre parenthèses et séparés par un point (3.14). Les figures, photographies, tableaux et programmes sont numérotés continûment par chapitre et repérés par deux nombres arabes précédés de Fig. (Fig. 4.12).

TABLE DES MATIÈRES

CHAPITRE 1

PRÉLIMINAIRES

1.1 MODÈLE GÉNÉRAL ET EXEMPLES

1.1.1 Modèle en trois couches

Une *calculatrice* (*computer*) ou système microinformatique, au sens le plus large, est un système complexe réalisé avec des composants électroniques évolués; une partie importante de ces composants, les mémoires, contiennent des *programmes* permettant d'obtenir des fonctionnements très variés pour un même système.

Vue de l'utilisateur non informaticien, une calculatrice résout à un instant donné une application donnée, la communication s'effectuant par le moyen de dispositifs de lecture, d'affichage et de stockage d'information. La calculatrice est une boîte noire qui, après quelques manipulations très simples (mise sous-tension) ou très complexes (transferts successifs de toute une hiérarchie de programmes) est prête à travailler pour son application.

Le but de ce livre est d'aider à comprendre le fonctionnement interne de cette boîte noire, dans laquelle on peut distinguer trois couches principales (fig. 1.1).

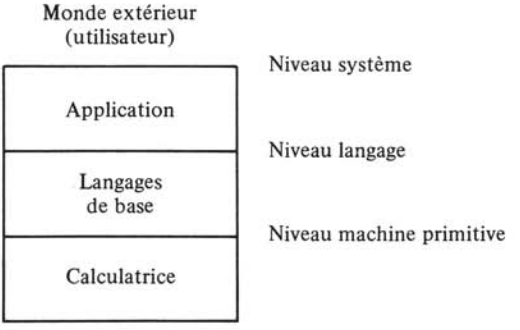


Fig. 1.1

La couche d'application, en contact avec l'utilisateur, consiste usuellement dans des programmes écrits par des spécialistes ayant des connaissances en informatique. L'analyse de l'application et les problèmes d'interfaces homme-machine y jouent un rôle prédominant. La couche intermédiaire consiste dans un ou plusieurs langages développés par les informaticiens pour faciliter la programmation des applications. La couche inférieure concerne essentiellement l'infrastructure concrète et matérielle de la machine primitive qui traite l'information selon les instructions du langage utilisé, après traduction dans le langage propre de la machine.

Ce volume recouvre essentiellement cette troisième couche, dont le développement est pleinement du ressort des ingénieurs électroniciens mettant en œuvre des petits systèmes.

1.1.2 Logiciel et matériel

L'ensemble des programmes que l'on peut charger dans les mémoires de la machine forment le *logiciel* (*software*). La machine avec sa circuiterie et ses périphériques électro-mécaniques forme le *matériel* (*hardware*).

La distinction entre matériel et logiciel devient difficile à faire lorsque les fonctions matérielles sont réalisées avec des programmes stockés dans des mémoires permanentes. Ces programmes, souvent appelés microprogrammes, sont définis par le fabricant, comme pour le reste du matériel. Le terme anglais correspondant (*firmware*) a un équivalent français, le *mollaciel*, encore mal accepté.

Réciproquement, des fonctions traditionnellement logicielles peuvent se figer sous forme de microprogrammes fixes, et être considérées comme du matériel.

Les trois couches de la figure 1.1 n'évoquent donc pas de façon précise une hiérarchie matériel-logiciel. L'évolution de la technologie et les besoins des applications conduisent à un développement considérable de la microprogrammation, indiscernable des fonctions câblées, avec un déplacement général de la frontière logiciel matériel vers le haut, mais avec simultanément une extension considérable de la complexité et de la richesse de chaque couche.

1.1.3 Exemples de calculatrices

Avec l'évolution de la technique et des besoins, les calculatrices ont pris des formes variées et multiples, et leur prix recouvre une gamme s'étendant sur 7 ordres de grandeur (10 francs à 100 millions de francs).

Dans toute cette gamme, les principes de base sont les mêmes, avec évidemment des perfectionnements significatifs et des concepts supplémentaires dans les modèles plus coûteux.

1.1.4 Automate programmable

Un *automate programmable* (*programmable controller*) simule et remplace un automatisme à relais tel qu'il en existe depuis de nombreuses années. Le "programme" de ces automatismes à relais se construisait en câblant les liaisons entre les bobines et les contacts de travail et repos des relais.

Dans un automate programmable moderne (fig. 1.2), le schéma des liaisons entre contacts s'établit au moyen d'un clavier et d'un affichage spécial, donc avec un langage extrêmement proche de l'utilisateur. Ce langage est écrit dans le langage de la machine ou dans un langage plus évolué, et la machine de base est un microprocesseur standard ou spécialisé, mais cette architecture interne logicielle et matérielle est cachée à l'utilisateur, qui ne voit en gros du système que des symboles représentant des contacts de relais ou des équations logiques. Du point de vue physique, il voit des bornes sur lesquelles il peut visser les fils de ses capteurs et moteurs, en respectant naturellement certaines règles [5].

Fig. 1.2 Automate programmable (photo Gould-Ghielmetti).

1.1.5 Calculette

Une *calculette* (*pocket calculator*) permet d'effectuer les opérations arithmétiques de base. Anciennement, une circuiterie spéciale s'occupait complètement de l'utilisateur, avec une couche langage et programme d'application inexistante. Les calculettes programmables (fig. 1.3) disposent d'un langage, permettant d'effectuer des calculs très complexes, et évoluent vers des ordinateurs individuels (§ 1.3.3) dont les entrées/sorties sont miniaturisées et parfois simplifiées, bien que des concepts évolués de réseaux apparaissent. Certains modèles sont spécialisés pour certains types de calcul ou pour des jeux avec dans ce cas un affichage et un clavier spécial [6,7].

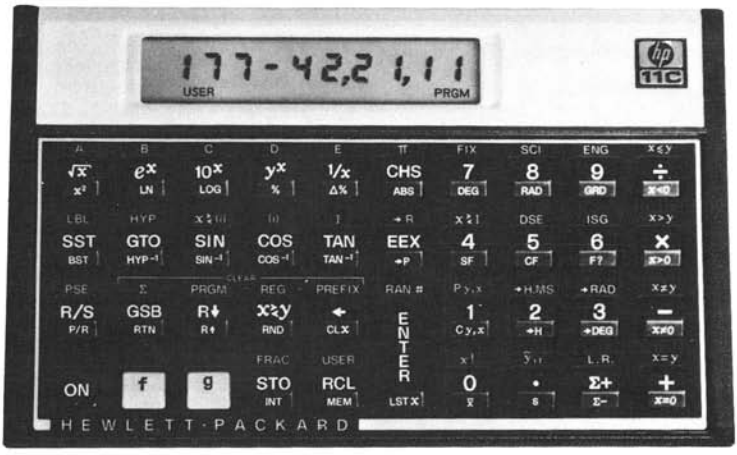


Fig. 1.3 Calculette (photo Hewlett-Packard).

1.1.6 Ordinateur individuel

Un *ordinateur individuel* (*personal computer*) (fig. 1.4) est une calculatrice prévue pour des applications simples le plus fréquemment programmées dans un langage facile à apprendre, le Basic (§ 4.1.3).

En référence à la figure 1.1, la couche inférieure est formée d'un microprocesseur, d'une certaine quantité de mémoire et des entrées-sorties. La couche intermédiaire, qui permet d'interpréter le langage Basic, est un programme dans des mémoires mortes (§ 1.2.7) ou sur disquette (sect. 5.3). La couche d'application est par exemple un programme de comptabilité écrit en Basic.

Cette structure est très universelle, et permet d'utiliser différents langages et résoudre des applications variées [8, 9]. Les miniordinateurs et gros ordinateurs sont similaires, avec un niveau de performance plus élevé, et une possibilité d'exécution quasi simultanée de plusieurs applications.

1.1.7 Station de travail

Une *station de travail* (*workstation*) (fig. 1.5) individuelle est un ordinateur personnel professionnel, dont le niveau de performance se compare à un puissant miniordinateur de quelques années plus ancien, avec en plus une qualité d'interaction

Fig. 1.4 Ordinateur individuel (photo Tandy Miniper).

inégalée. Un écran graphique, sur lequel des textes et dessins joliment présentés peuvent aussi apparaître, caractérise ce type de stations, qui sont supportées par un logiciel performant et prennent toute leur valeur lorsque plusieurs stations sont reliées entre elles et à des serveurs (disques, imprimantes) grâce à un réseau local, permettant la communication entre utilisateurs, le partage des ressources coûteuses et l'utilisation intensive des répertoires graphiques [10, 11].

1.1.8 Ordinateur centralisé

Un *ordinateur centralisé* (*mainframe computer*) (fig. 1.6) consiste en plusieurs armoires réalisant les fonctions de calcul et stockage d'information pour un nombre élevé d'utilisateurs. Ceux-ci accèdent à la machine au moyen d'un *terminal* consistant dans un clavier et un écran simple transmettant son information à faible vitesse (30 à 200 caractères par seconde) au travers de lignes téléphoniques. Le prix des ordinateurs centralisés varie beaucoup entre les miniordinateurs et les ordinateurs ultrapuissants; leur architecture est influencée par leur domaine d'application, commercial, scientifique ou technique (commande de processus) [12, 13].

1.1.9 Miniordinateur

Le *miniordinateur* (fig. 1.7) est un ordinateur centralisé de taille réduite, n'occupant qu'une ou deux armoires. Il est le plus souvent incorporé dans des systèmes industriels de commande de processus, où il tend à être remplacé par des systèmes basés sur un ou plusieurs microprocesseurs interconnectés en réseau [14, 15].

1.1.10 Microprocesseur

Le *microprocesseur* (fig. 1.8) est au sens strict un circuit intégré unique qui interprète et exécute les instructions d'un programme stocké en mémoire (chap. 3). Ce terme

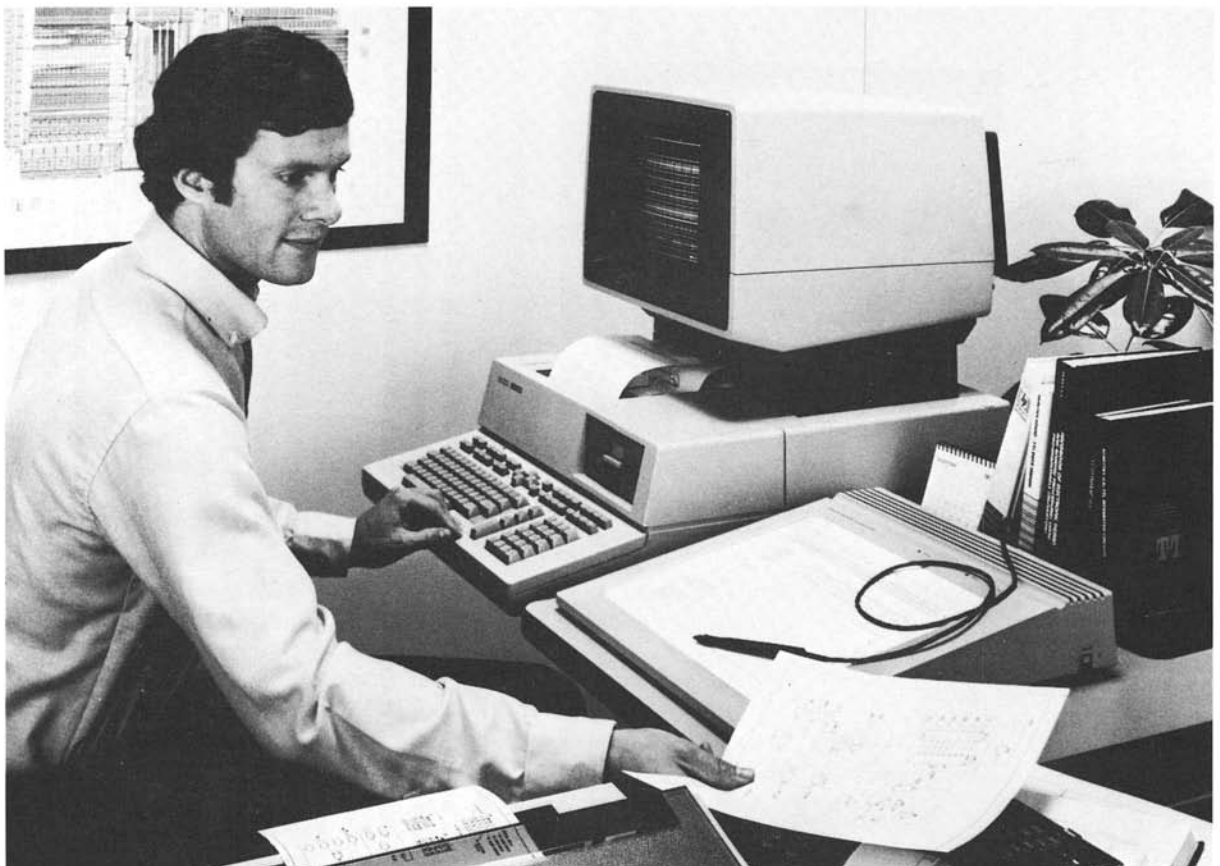


Fig. 1.5 Station de travail (photo Hewlett-Packard).

Fig. 1.6 Petit ordinateur centralisé (photo Digital Equipment).

recouvre en fait tous les circuits miniaturisés qui traitent de l'information, et qui sont utilisés soit seuls dans des applications de commande de processus simples (machines à laver), soit associés avec d'autres composants complexes (mémoire, processeurs esclaves) pour des applications variées (machine à écrire, robot industriel) [15, 21, 74].

Les microprocesseurs, comme les gros ordinateurs, existent par familles comportant un nouveau membre tous les 2 à 4 ans, chaque membre ayant une durée de vie active de 4 à 6 ans. On se référera plusieurs fois dans ce livre aux familles 8080 (Intel), 6800 et 68000 (Motorola), 6500 (MOS-Technology/Rockwell), VAX (Digital Equipment) comme si ces familles étaient bien connues du lecteur. Si ce n'est pas le cas, ces exemples peuvent être sautés en première lecture, et une abondante littérature distribuée par les fabricants et résumée dans de nombreux livres [50, 51, 53, 54, 61, 62, 74, 78] permettra au nouveau venu dans ce domaine de compléter progressivement ses connaissances.

1.1.11 Modèle détaillé

La couche de base des calculatrices (fig. 1.1) peut se détailler comme montré dans la figure 1.9.

Les bases mathématiques résident dans la théorie des nombres [16], l'algèbre de Boole [17, 18], les systèmes logiques combinatoires et séquentiels (vol. V et XI) et la technologie électronique (vol. VII et VIII).

Les éléments matériels de base, appelés systèmes digitaux ou numériques car ils agissent sur des mots binaires, sont rappelés à la fin de ce premier chapitre. La représentation des nombres et les règles concernant les opérations font l'objet du chapitre 2, et éclairent l'architecture des calculettes et ordinateurs, traitée dans le chapitre 3.

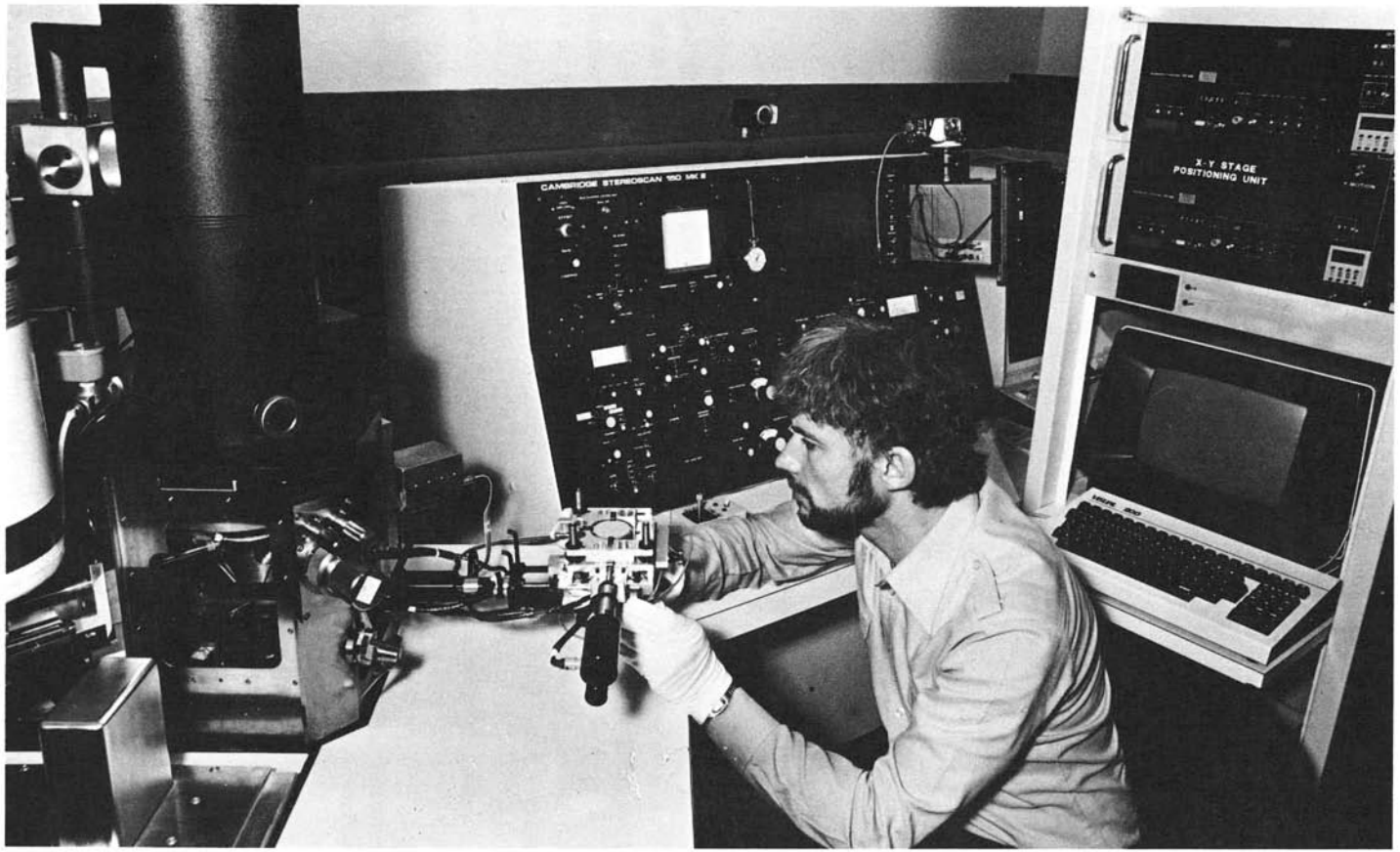


Fig. 1.7 Miniordinateur de commande de processus et appareillage associé (photo Landis et Gyr, Zoug).

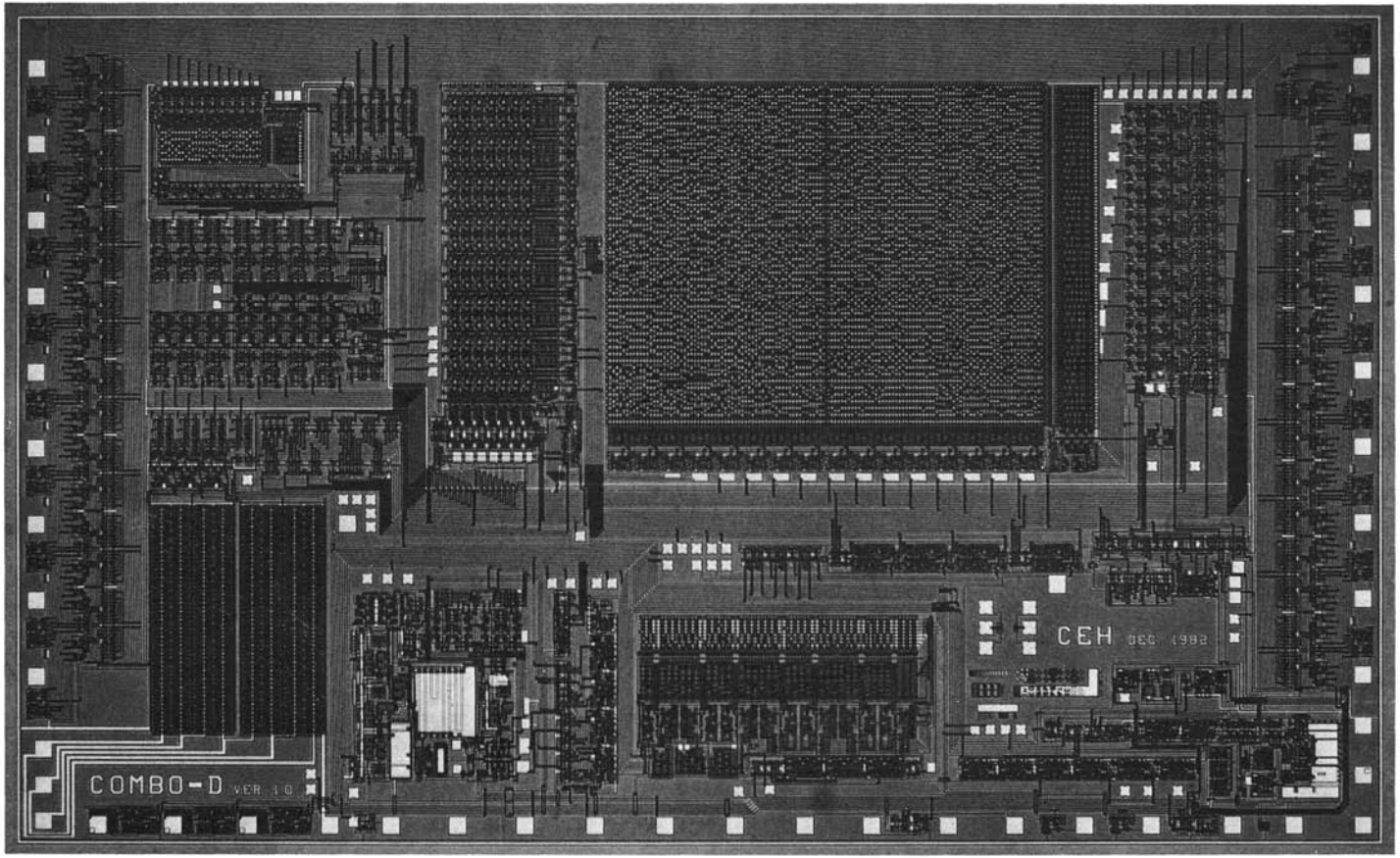


Fig. 1.8 Microprocesseur CMOS, développé par le Centre Electronique Horloger pour ETA S.A., Granges (photo CEH).

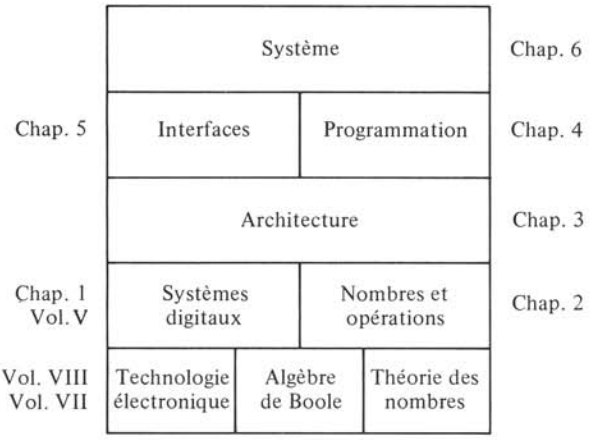


Fig. 1.9

La programmation en assembleur est fortement liée aux principes architecturaux et est détaillée dans le chapitre 4. Le chapitre 5 poursuit les deux chapitres précédents en montrant l'architecture détaillée des interfaces, et leur gestion avec les instructions prévues dans le processeur.

La gestion correcte de l'ensemble des ressources de la calculatrice est confiée à un programme système dont le noyau tend à faire partie du matériel, et dont l'exposé dans le chapitre 6 n'est qu'amorcé.

L'étude des systèmes d'exploitation, des systèmes temps réel et des systèmes multiprocesseurs et en réseaux forme un vaste domaine sortant du cadre de ce livre.

1.2 ÉLÉMENTS DES CALCULATRICES

1.2.1 Modules fonctionnels

Une calculatrice est formée avec des modules plus ou moins facilement discernables, et chaque module réalise une fonction de traitement, de stockage ou de transfert d'information. Un *module fonctionnel* peut être réalisé en logiciel et avoir une nature immatérielle, ou être réalisé avec du matériel facile à identifier. Il est important de bien distinguer la fonction, et ses implémentations sous différentes formes.

Les modules qui nous intéressent ici traitent de l'*information* ou des *données* (*data*), codées sous forme de caractères qui sont eux-mêmes codés dans la machine sous forme *numérale* (*digital*).

Par exemple, un module d'information (on dit aussi bloc ou paquet d'information) tel que par exemple un nombre ou un texte, peut être stocké en mémoire et être facilement modifié ou perdu, ou être câblé de façon permanente comme l'identificateur d'appel dans une station télex.

1.2.2 Communication entre modules

Chaque module manipule ou transforme de l'information, et communique par un *canal* ou *voie* (*channel*).

Le canal assure le transfert de l'information selon une succession d'opérations élémentaires de demande, transfert et vérification appelée *protocole*.

1.2.3 Transfert unidirectionnel

Physiquement, la partie d'un canal permettant le transfert de l'information sur une certaine distance est un *faisceau de lignes* ou *câbles*. Un câble *unidirectionnel* transfère l'information dans un seul sens (fig. 1.10). Des *circuits transmetteurs* (*line-driver*) et *récepteurs* (*line receiver*) sont utilisés pour respecter les caractéristiques électriques ou physiques du câble. Des éléments aiguillages peuvent être utilisés pour prélever l'information dans l'une de plusieurs sources [19].

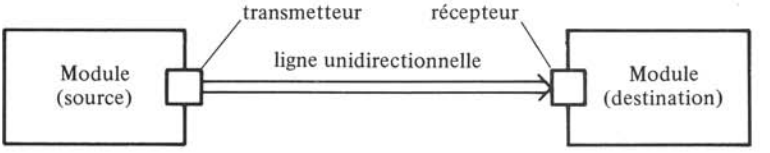


Fig. 1.10

1.2.4 Transferts bidirectionnels

Un *câble bidirectionnel* permet de transférer l'information dans les deux sens (fig. 1.11). A l'entrée de chaque module, un circuit *passeur* (*bidirectional driver*) joue alternativement le rôle de transmetteur ou récepteur, selon le sens du transfert et en fonction du protocole. Ces passeurs sont dits *à trois états* (*three-state*) car ils ont un état de sortie actif (état logique 0 ou 1) et un état d'entrée à haute impédance.

L'utilisation de deux câbles unidirectionnels permet également des transferts dans les deux sens, avec possibilité de simultanéité.

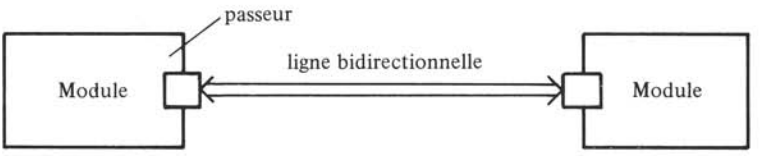


Fig. 1.11

1.2.5 Transfert par bus

Plus de deux modules peuvent être physiquement liés sur un *chemin de données* souvent appelé *bus* (fig. 1.12). Les transferts se font deux à deux après sélection des deux partenaires au moyen de lignes du bus sur l'initiative de l'un des deux partenaires ou d'un troisième module assurant la supervision du bus (chap. 5).

Lors d'un transfert d'information du module C vers le module B par exemple, le passeur C est actif en mode transmetteur, le passeur C est en mode récepteur et le passeur A est inactif.

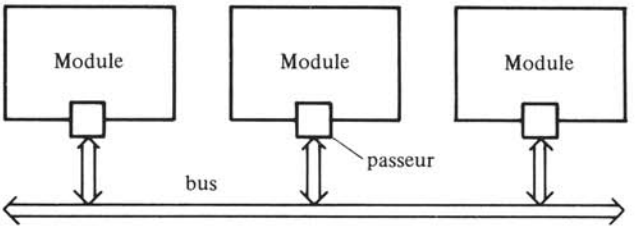


Fig. 1.12

Le bus est physiquement réalisé sous forme de connecteurs parallèles dont les broches correspondantes sont reliées par un *circuit arrière* (*backplane*) (fig. 1.13).

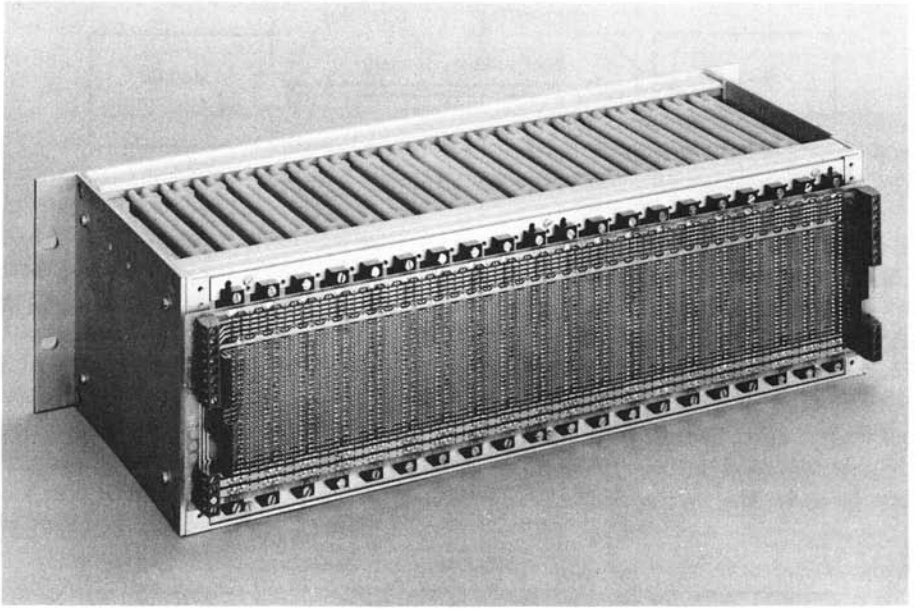


Fig. 1.13 Bus arrière dans un système microprocesseur (photo Panel-Gardy, Préverenges).

1.2.6 Module mémoire

Un module *mémoire* (*memory*) permet de stocker de l'information et de la retrouver [20]. La clé d'accès peut être de quatre types différents.

- Accès aléatoire. A chaque information se trouve associée une adresse. La connaissance de l'adresse permet de trouver l'information. Physiquement, l'adresse est transmise par un câble, et deux lignes de commande au moins définissent l'instant de lecture et d'écriture de l'information;
- accès en silo. La mémoire joue le rôle de stockage temporaire, la première information écrite est la première à être relue;
- accès en pile. La mémoire joue aussi le rôle de stockage temporaire, la dernière information écrite est la première à être relue;

- accès associatif. L'information est écrite avec l'un des mécanismes précédents, mais est relue en donnant un élément de l'information cherchée. Une comparaison interne avec tous les mots mémoire permet de retrouver le ou les mots ayant l'élément d'information cherché et l'information associée.

1.2.7 Mémoire à accès aléatoire

Une *mémoire aléatoire* avec lecture et écriture ou *mémoire vive* est traditionnellement abrégée *RAM* (*Random Access Memory*). Elle comporte un nombre de cellules qui est généralement une puissance de 2, adressée par un mot binaire (sect. VIII.7.10). Des noms symboliques sont souvent associés aux adresses et aux contenus de certaines positions mémoire.

Le transfert d'information avec les autres éléments du système (processeur, périphérique, etc.) se fait au moyen d'un bus. Il en est de même pour la sélection de la position mémoire grâce à l'ensemble des lignes d'adresse (fig. 1.14).

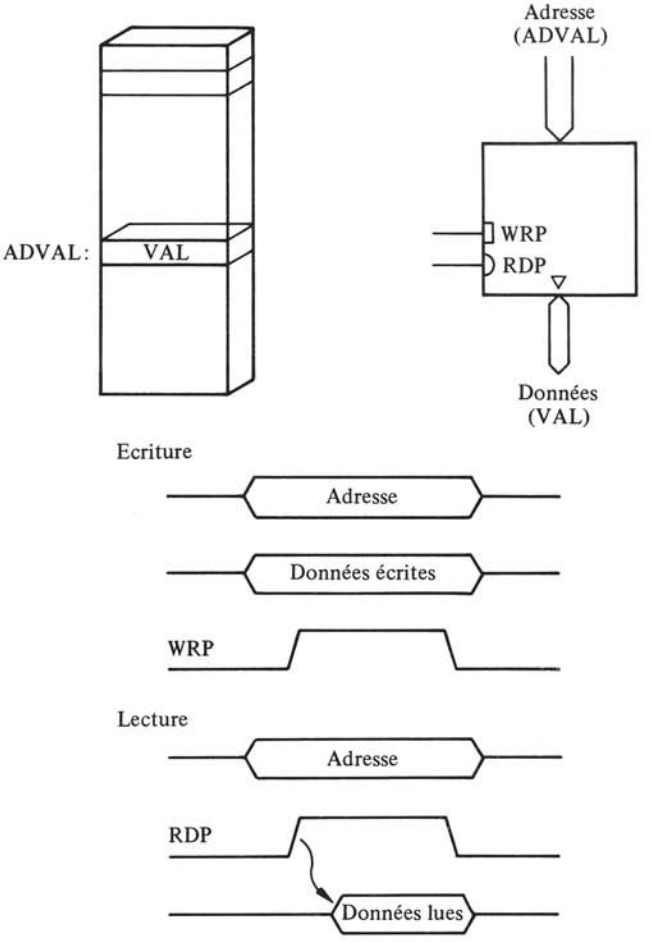


Fig. 1.14

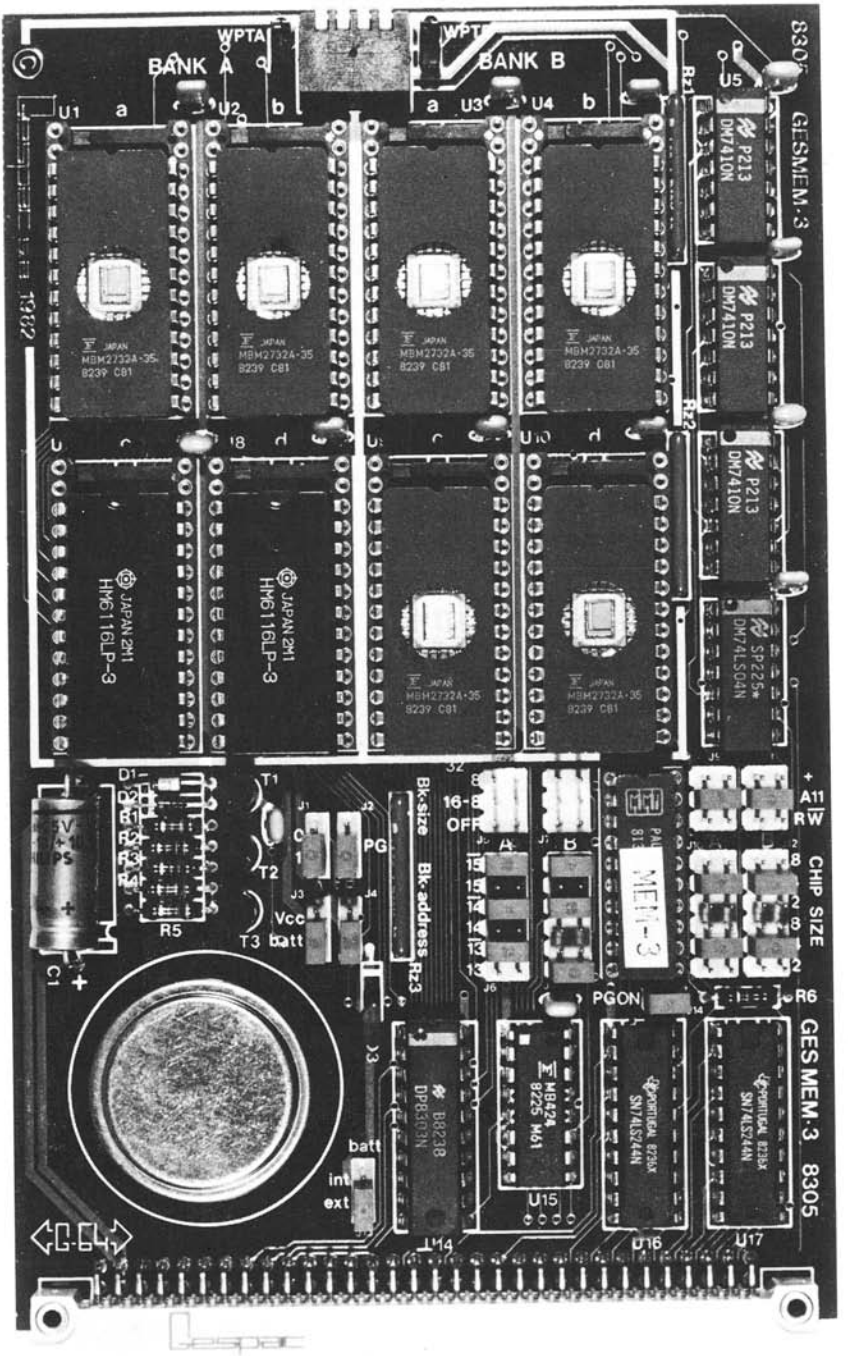


Fig. 1.15 Carte mémoire ROM effaçable par ultraviolets et reprogrammable (photo GESPAC, Genève).

Dans un cycle d'écriture, une impulsion d'écriture effectue le transfert d'information à un instant où l'information et les adresses sont stables. Dans un cycle de lecture, l'adresse doit être stable pendant qu'une impulsion de lecture demande l'information et la maintient sur le chemin de données. Cette impulsion de lecture active les passeurs de sortie à trois états.

Des modules mémoire de toute taille (fig. 1.15) sont construits à partir de circuits mémoire intégrés (vol. VIII) et de circuits auxiliaires [20, 21].

Une mémoire aléatoire est un élément universel que l'on peut utiliser pour simuler les autres types de mémoires. Une mémoire aléatoire que l'on ne peut écrire est appelée *mémoire morte* ou *ROM* (*Read Only Memory*).

Les *mémoires silo* sont souvent appelées *FIFO* (*First In First Out*), et les *mémoires piles* sont appelées *LIFO* (*Last in First Out*). Les mémoires associatives sont coûteuses à réaliser et ont encore un domaine d'application très restreint (§ 3.8.8).

1.2.8 Registre

Une mémoire ne peut contenir qu'un seul mot. On parle alors de *verrou* (*latch*) et de *registre* (fig. 1.16). Le verrou a un comportement identique à une mémoire, et suppose que l'information reste présente pendant toute la durée de l'impulsion d'écriture, appelée horloge. Un registre est formé de bascules (sect. V.5.3) et mémorise l'information au front descendant (ou montant) de l'horloge. Un verrou ou un registre contient souvent des éléments transmetteurs ou passeurs (dits à trois états), pour faciliter la communication avec d'autres modules [22, 23].

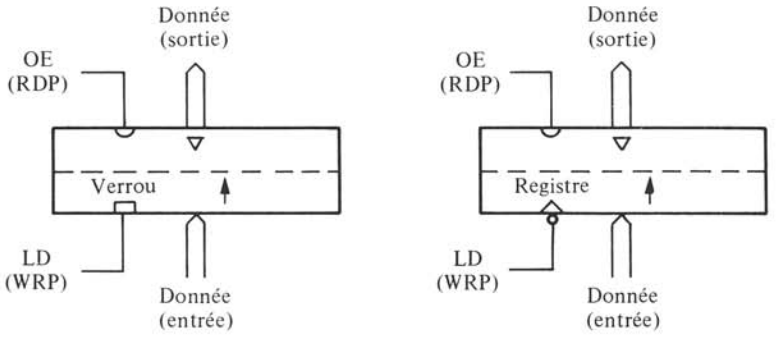


Fig. 1.16

1.2.9 Compteurs

Un *compteur* est formé de bascules bistables (chap. V.4) et génère des mots binaires successifs, correspondant par exemple à la suite des adresses d'une mémoire. La fonction verrou ou registre est en général simultanément présente dans un registre, afin de permettre l'initialisation à une valeur quelconque (fig. 1.17).

Un compteur peut avoir un mode décompteur. L'association d'un compteur et d'une mémoire permet de parcourir facilement tous les mots de cette mémoire si les sorties du compteur commandent les lignes d'adresse de la mémoire. L'exécution d'un programme (§ 3.2.3) est basée sur cette association.

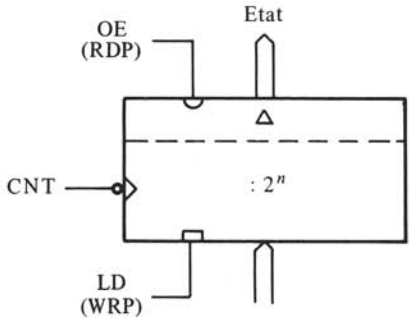


Fig. 1.17

1.2.10 Module de traitement

Un *module de traitement* (*processing module*) effectue une opération ou un groupe d'opérations sur les informations qui lui sont présentées. Les opérations concernent des nombres, des chaînes de caractères, des tableaux, etc. et se décomposent en opérations élémentaires réalisées par des opérateurs simples. L'addition binaire a été étudiée au paragraphe V.2.7.8 et est reprise au chapitre 2 de ce volume, avec les autres opérations de base pour traiter des nombres.

Comme pour les fonctions logiques (sect. V.1.4), un petit nombre d'opérateurs permet d'obtenir par combinaison toutes les opérations souhaitables. Ces opérateurs seront étudiés de façon fonctionnelle aux chapitres 2 et 3, et on se référera à la section VIII.7.1 pour l'étude des circuits électroniques intégrés qui réalisent certains de ces opérateurs et sont utilisés dans la réalisation des calculatrices électroniques.

1.2.11 Séquenceur

Un type de module important dans les calculatrices, mais souvent implicite et non détaillé, est le *séquenceur*, qui garantit la succession correcte des événements et réalise les protocoles de transfert d'information entre les modules [23]. Le module de traitement et le séquenceur général des opérations forment le *processeur*, cœur de tout système informatique.

Dans toute calculatrice, le séquencement se produit à deux niveaux. Au niveau de la machine, des opérations élémentaires du point de vue de l'utilisateur (une multiplication par exemple), sont converties dans des séquences d'opérations plus élémentaires (chap. XI.2). Au niveau des langages et de l'application, le programmeur définit le séquencement de la tâche à effectuer selon l'ordre d'exécution des instructions du programme (chap. 4). Le parallélisme dans l'exécution des opérations à chacun des deux niveaux est possible si l'on dispose de plusieurs unités pouvant effectuer des opérations élémentaires simultanément, et les protocoles de synchronisation nécessaires [24].

1.2.12 Processeur et processus

Une opération complexe, comme par exemple la fabrication d'une automobile, se décompose toujours en sous-opérations, appelées *tâches* ou *processus*. Un processus détaille la séquence des opérations élémentaires à effectuer pour atteindre le résultat.

Un processus a besoin d'un processeur pour exécuter les opérations élémentaires. Plusieurs processeurs peuvent se partager le travail et effectuer simultanément les différentes parties d'un processus. On parle alors de *système multiprocesseur*.

Un seul processeur peut aussi se consacrer à plusieurs tâches quasisimultanément, en donnant un peu de son temps à tour de rôle. On parle alors de *temps partagé* (*time sharing*) ou *multitâche* (*multitasking*).

Enfin, plusieurs processeurs peuvent coopérer pour travailler à plusieurs tâches, on parle alors de *multitraitement* (*multiprocessing*).

Par exemple dans la fabrication d'une automobile, les processeurs sont des hommes et des machines, les processus s'appellent visser, souder, peindre, et l'on imagine facilement les différentes organisations de travail avec un ou plusieurs processeurs.

Le terme de processeur n'évoque pas seulement une machine concrète qui effectue un travail donné, mais aussi un programme qui permet à une machine universelle de réaliser ce travail.

1.3 ÉVOLUTION

1.3.1 Historique

Si l'on exclut les bouliers, la première calculatrice a été construite en 1623 et réinventée par Pascal en 1642, qui la fit largement connaître [12, 25]. Perfectionnées peu après par Leibnitz, qui leur ajouta les quatre opérations, ces machines restèrent curiosités de savant jusqu'au début de notre siècle.

En 1834, après avoir abandonné la construction d'une première machine déjà remarquable, l'anglais Charles Babbage conçut son "Analytical Engine" formée d'une mémoire de 1000 nombres de 50 chiffres (1 roue par chiffre) et d'une unité de calcul [12]. Des cartes perforées, selon le principe des métiers à tisser Jacquard, définissaient les instructions et les variables en mémoire. Des sauts dans le programme, et même des sauts conditionnels en fonction du signe du résultat précédent, étaient prévus. La vitesse de calcul devait atteindre une addition par seconde et une multiplication par minute.

Cette machine si extraordinaire et si impossible pour l'époque ne fut que très partiellement construite, et Ada, Comtesse de Lovelace et collaboratrice de Babbage, n'eut que peu de programmes à écrire.

L'idée de la carte perforée par contre fit son chemin grâce à Hollerith et sa société, qui prit le nom d'IBM en 1924.

Le premier calculateur électromécanique à relais, le Z1 fut construit en 1938 par Zuse. Le Z4 construit de 42 à 46 fut racheté par l'Ecole Polytechnique de Zurich et remis en service en 1950; il a fonctionné jusqu'en 1959 [26].

Indépendamment, Aiken construisait à Harvard une machine à relais similaire terminée en 1944, avec une mémoire de 72 nombres décimaux de 23 chiffres.

1.3.2 Calculatrices électroniques

La première calculatrice à tubes radio, l'ENIAC, fut terminée en 1946 et amena un gain en vitesse de 1000. Programmée par des câbles mobiles, elle travaillait en décimal avec dix bascules par chiffre. Avec ses 30 tonnes et 18 000 tubes radio, elle pouvait multiplier deux nombres de 10 chiffres en 3 millisecondes.

John von Neumann, dans un rapport célèbre publié en 1945 [27], proposa de mémoriser le programme de la même façon que les données, d'utiliser le système binaire et une architecture avec un seul registre accumulateur avec un répertoire d'instruction qui a été repris par les premiers miniordinateurs comme le PDP-8 [28].

Il est intéressant de noter que von Neumann limitait son ambition à un superordinateur travaillant sur 4000 mots de 40 bits, avec un temps d'accès de 100 microsecondes.

La première génération de calculatrices de 1945 à 1955 se caractérisa par l'amélioration technologique des circuits électroniques, et surtout des mémoires, d'abord à écran cathodique, puis à tambour et enfin à tores de ferrite.

La seconde génération, de 1955 à 1965 environ fut influencée par UNIVAC, IBM et l'Université de Manchester, et dégagea les concepts plus avancés liés à la gestion de la mémoire et des entrées-sorties. Grâce au FORTRAN (1957) et au COBOL (1959), la programmation se libéra des contraintes très dures du langage machine.

La troisième génération de grosses machines, dès 1965, a essentiellement vu un progrès technologique marqué et une baisse de prix à performance constante, dus au remplacement des transistors par des circuits intégrés. On peut éventuellement voir une quatrième génération d'ordinateurs dans les machines vectorielles et multiprocesseurs en service dès 1975 (Illiac IV, Cray I, CDC Star) et dont la puissance est encore insuffisante pour résoudre les problèmes de simulation aérodynamiques et météorologiques pour lesquelles elles sont partiellement destinées [24].

1.3.3 Miniordinateurs et microprocesseurs

Dès 1965, les transistors ont permis de réaliser des calculatrices de format réduit et d'un coût abordable. Ces ordinateurs de taille réduite et appelés miniordinateurs (§ 1.1.9) se sont développés rapidement au début des années 70, grâce aux circuits intégrés. Ils ont permis le développement des applications de commande de processus en temps réel.

Appelés parfois *midiordinateurs* dans leur configuration la plus complète servant plusieurs utilisateurs, ils équivalent actuellement facilement à la puissance des ordinateurs de la 3e génération. Quant aux miniordinateurs simples, ils deviennent difficilement distingables des *microordinateurs*, dont la caractéristique est d'avoir un processeur contenu dans un seul circuit intégré, appelé microprocesseur (§ 1.1.10).

L'évolution des microprocesseurs a été extrêmement rapide depuis 1971, année de lancement de l'Intel 4004. Plusieurs générations peuvent être distinguées depuis cette époque; elles correspondent à un raccourci de l'évolution des ordinateurs et miniordinateurs, le point de convergence du développement de ces trois familles étant prochainement atteint.

1.3.4 Evolution future

Les éléments de base des calculatrices sont actuellement des circuits intégrés complexes contenant plusieurs centaines de milliers de transistors avec une très grande régularité de structure [29] (fig. 1.18). Ces circuits sont les mêmes pour les calculettes, ordinateurs individuels, stations de travail et ordinateurs centralisés. Seule la rapidité et la quantité de ces circuits varient, de façon à s'adapter aux contraintes de coût ou de performance.

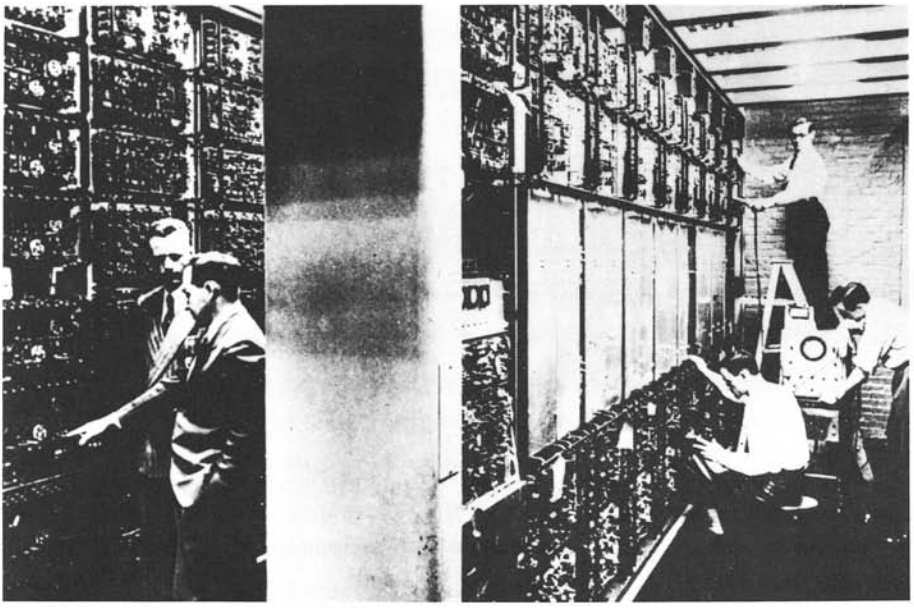


Fig. 1.18 Ordinateur Whirlwind, 1950 (photo R. Weiss).

Une génération entièrement nouvelle de machines, atteignant des rapidités extrêmes grâce à l'effet Josephson, apparaîtra probablement avant la fin du siècle, mais la principale caractéristique des années 80 est la prolifération de stations de performance moyenne basées sur des microprocesseurs 8 et 16 bits (chap. 3), reliées en réseau et offrant de nombreux services informatiques pour un coût réduit [30, 31].

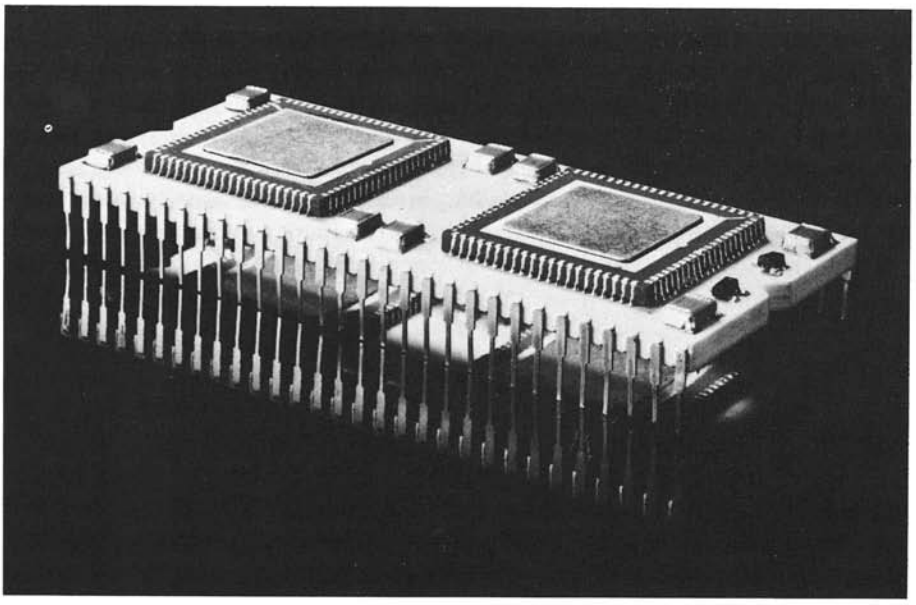


Fig. 1.19 Circuit intégré complexe (photo Digital Equipment).

CHAPITRE 2

NOMBRES ET OPÉRATIONS

2.1 NUMÉRATION DE POSITION

2.1.1 Introduction

Pendant de nombreux siècles, l'humanité a cherché le meilleur moyen de représenter les nombres et les grandeurs [32, 33]. Les peuples primitifs, peu préoccupés de caractériser les grands nombres, se sont contentés d'aligner des bâtonnets ou des encoches.

L'idée de représenter des groupements de 2, 5, 10, 20 ou 60 objets par de nouveaux symboles, ou par les mêmes symboles placés à des emplacements déterminés, apparut dans chaque grande civilisation. Mais il fallut attendre l'invention du zéro aux Indes, et la propagation de cette idée par les Arabes, pour que l'humanité dispose d'un système de numération souple et efficace. Les Mayas ont également introduit un zéro dans leur numération en base 20.

La notation actuelle pour les nombres fractionnaires date de la fin du XVIe siècle seulement.

La base de ce système de numération est dix, car les doigts sont un auxiliaire commode pour le calcul, mais de nombreux autres systèmes ont subsisté pour la mesure du temps et des longueurs.

Le plus simple de tous les systèmes de numération est le système binaire, utilisé en Chine il y a plus de 5000 ans et préconisé à nouveau par Leibnitz. Si notre système décimal autorise la conception de machines à calculer mécaniques, l'existence de nombreux systèmes physiques ayant deux états d'équilibre bien distincts, notamment des circuits électroniques très rapides, a imposé l'emploi du système binaire et des systèmes qui en sont dérivés dans tous les ensembles de calcul actuels.

2.1.2 Nombres entiers positifs et systèmes de numération

Un *nombre entier positif* ou *cardinal* est une entité abstraite associée à une collection d'objets indépendants. L'absence d'objet est représentée par le nombre zéro (symbole 0), un objet unique par le nombre 1, etc. L'ensemble des nombres entiers positifs est un ensemble ordonné qui s'écrit traditionnellement

$$\mathbb{N} = \{0, 1, ..., N, N+1, ...\} \tag{2.1}$$

où N représente un nombre entier positif quelconque.

Les notions d'ensemble, d'opération et de comparaison sont supposées connues.

Un *système de numération* fait correspondre à un nombre positif N un certain symbolisme écrit et oral. Dans un système de *base* p, p étant un nombre entier positif supérieur à 1, les nombres entiers 0, 1, ..., $p-1$ sont appelés *chiffres* (*digits*).

2.1.3 Assertion

Tout nombre entier peut être représenté de façon unique par une expression de la forme

$$N = a_n \cdot p^n + a_{n-1} \cdot p^{n-1} + \ldots + a_1 \cdot p + a_0 = \sum_{i=0}^{n} a_i \cdot p^i \qquad (2.2)$$

avec $a_i \epsilon \{0, 1, \ldots, p-1\}$ et $a_n \neq 0$

La notation condensée

$$N = a_n\, a_{n-1} \ldots a_1\, a_0 \qquad (2.3)$$

est équivalente.

Il résulte de la définition des chiffres que

$$a_i \cdot p^i < p \cdot p^i < p^{i+1}, \quad \text{et} \quad N < p^{n+1} \qquad (2.4)$$

2.1.4 Définitions

L'indice courant i est appelé le *rang* du chiffre a_i et le facteur multiplicatif p^i est appelé le *poids* du chiffre a_i. Le poids de a_0 est 1; ce chiffre est dit chiffre des *unités*, ou chiffre de *poids le plus faible* (*least significant digit*, *LSD*). Le poids de a_n est p^n; c'est le chiffre de *poids le plus fort* (*most significant digit*, *MSD*).

2.1.5 Systèmes usuels

Le *système décimal* dont la base est dix, nous est familier. Les chiffres décimaux 0, 1, ..., 8, 9 sont parfois appelés *digits*. Le *système binaire*, de base 2, n'utilise que les chiffres 0 et 1, appelés *bits*.

La longueur des nombres écrits en binaire et les risques de confusion qui en résultent ont généralisé l'emploi du *système octal*, de base 8, et du *système sexadécimal* ou *hexadécimal* [1] de base 16., dans tout ce qui concerne le niveau le plus bas de la description de la structure et de la programmation des ordinateurs. Dans ces deux cas, la base est une puissance de 2, et la conversion directe et inverse avec le système binaire est immédiate (§ 2.9.12). Pour préciser la base dans laquelle un nombre est écrit, la valeur de la base en indice peut suivre ce nombre: $AF3B_H$. Une lettre en préfixe est préférable: un *B* pour le binaire, un *O* pour l'octal, un *H* pour le sexadécimal et un *D* pour le décimal. Par exemple: H'AF3B. La convention générale dans plusieurs chapitres de ce livre sera de considérer le système sexadécimal comme le système naturel et de représenter les nombres sexadécimaux sans préfixes. En d'autres termes, le sexadécimal sera le système de numération par défaut (sous-entendu par défaut d'indication supplémentaire). Pour faciliter la distinction, les nombres décimaux seront suivis d'un point. Pour les chiffres de 0 à 9, il n'y a pas de confusion possible, et le point sera omis, pour retrouver ultérieurement une signification différente (§ 2.8.1).

2.1.6 Remarques

Dans un système de base p, la base s'écrit toujours 10, car $p = 1 \cdot p^1 + 0 \cdot p^0$. Le terme "dizaine" est toutefois réservé à la base 10 qui nous est familière. Le système décimal sera toujours utilisé comme système de référence pour les bases, indices et exposants.

Dans un système de numération de base $p < 36.$, les chiffres successifs sont pris dans la liste 0, 1, 2, ..., 8, 9, A, B, ..., Z. L'exercice 2.2.6 montre un exemple pratique d'utilisation de la base 40.

2.1.7 Exemple

Dans le système décimal (base 10.), le numéro atomique de l'or est $79. = 7 \cdot 10. + 9.$

En base 16. (hexadécimal), les chiffres sont 0, 1, ..., E, F et le numéro atomique de l'or est $H'4F = 4 \cdot H'10 + H'F = 4 \cdot 16. + 15. = 79.$

2.1.8 Exercice

Ecrire en hexadécimal les 8 nombres qui suivent chaque fois les nombres: H'57, H'98, H'B9, H'9F9.

Ecrire en octal les 4 nombres qui suivent chaque fois les nombres: O'57, O'175, O'657, O'7774.

2.1.9 Codage des chiffres

Souvent les chiffres d'une base p doivent être notés en utilisant des nombres d'une autre base. On appelle *code* la table de correspondance entre les deux notations.

Un code est dit *naturel* lorsque l'ordre des chiffres dans les 2 bases est conservé, et que seuls les chiffres sont codés dans l'autre base. Par exemple, les chiffres hexadécimaux 0, 1, ..., 9, A, ..., F peuvent être codés par les nombres octaux 00, 01, ..., 11, 12, ..., 17.

Les chiffres décimaux 0, 1, ..., 9 peuvent être codés en binaire par les nombres 0000, 0001, ..., 1001 (code naturel). Par contre, le codage par les nombres 0000000001, 0000000010, ..., 1000000000 (code 1 parmi 10) n'est pas naturel.

Un code est dit *pondéré* lorsqu'une expression du type $a_i = \Sigma_{j=0}^{f} q_j \cdot c_{ij}$ lie les chiffres a_i du nombre exprimé dans la première base aux chiffres c_{ij} du nombre exprimé dans la 2ème base. Les coefficients $q_0, q_1, \ldots q_f$ sont les poids du code. Il est évident que les codes naturels sont pondérés, les poids $q_0, q_1, \ldots q_f$ étant égaux à $1, q, \ldots, q^f$ si la 2ème base vaut q. Les deux codes binaires de l'alinéa précédent sont pondérés.

Dans le premier cas, $q_3 = 8$ $q_2 = 4$ $q_1 = 2$ $q_0 = 1$ et par exemple $9 = 1 \cdot q_3 + 0 \cdot q_2 + 0 \cdot q_1 + 1 \cdot q_0$.

Dans le second cas, $q_9 = 9 \ldots q_0 = 0$ et $9 = 1 \cdot q_9$.

D'autes exemples de codes décimaux naturels, pondérés ou non, seront donnés au paragraphe 2.2.4.

2.1.10 Représentation en champ fixe

Dans un système de base p, un nombre entier positif peut être précédé d'un nombre quelconque de *zéros non significatifs*: $000\, a_n \ldots a_1 a_0 = a_n \ldots a_1 a_0$. Si le nombre de chiffres utilisés pour représenter un nombre entier positif est fixe et égal à k, y compris

les éventuels zéros non significatifs, on parle de *représentation en champ fixe*. Seuls les nombres entiers positifs de 0 à p^{k-1} peuvent être représentés dans un champ de longueur k. Un rectangle entourant le nombre, avec parfois des cloisons entre chaque paire de chiffres, sera dessiné pour caractériser les nombres en champ fixe (fig. 2.1).

Fig. 2.1

Le rectangle dans lequel un nombre en champ fixe peut être écrit est appelé *mot* de *k positions* ou *registre* de *k cellules*.

Chaque cellule est caractérisée par le rang ou le poids du chiffre qu'elle contient (fig. 2.2).

Fig. 2.2

La *capacité d'un registre* est égale au plus grand nombre entier positif qu'il peut contenir, soit p^{k-1}. La *longueur d'un registre* est égale au nombre de ses cellules. Dans la figure 2.2 cette longueur est égale à k.

En toute rigueur, la notion de nombre est abstraite et celle de registre est concrète. Physiquement, un registre est un ensemble de roues codeuses ou de bascules électriques dont l'état représente un nombre.

2.1.11 Exemple

Dans un registre de 4 cellules en base 8, le nombre atomique de l'or se note comme dans la figure 2.3 :

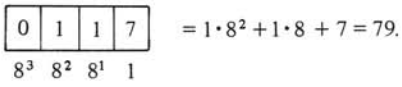


Fig. 2.3

En base 2, ce même nombre ne peut pas s'écrire dans un registre de 4 cellules; le nombre dépasse la capacité du registre, qui est au maximum $2^4 - 1 = 15$..

2.1.12 Relation

La longueur k d'un registre en base p est liée à sa capacité C par la relation

$$C = p^k - 1 \cong p^k \quad \text{pour } k \text{ grand} \tag{2.5}$$

d'où l'on tire

$$k \cong \log_p C \tag{2.6}$$

2.1.13 Exercice

Quelle doit être la longueur d'un registre permettant de représenter en binaire le nombre d'Avogadro ($6 \cdot 10^{23}$)?

2.1.14 Exercice

Supposons que le prix d'un registre est proportionnel à sa longueur et au nombre de chiffres différents que chaque cellule peut contenir (c'est-à-dire à la base du système de numération utilisé). Déterminer dans ce cas la valeur de la base conduisant à un coût minimal et calculer les rapports des prix des registres en base 2, 3, 4, 10, pour une capacité donnée.

2.1.15 Remarque

Si les chiffres d'un nombre en champ fixe sont codés dans une autre base, chaque chiffre correspond à un sous-champ ou à un groupe de cellules dans la nouvelle base. Par exemple, la figure 2.4 montre la représentation du nombre décimal exprimant le numéro atomique de l'or lorsque les chiffres décimaux sont codés en octal et en binaire (code naturel).

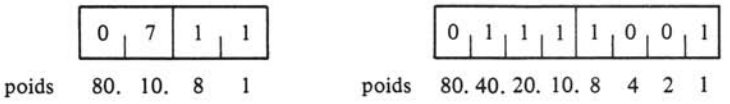


Fig. 2.4

2.1.16 Bases mixtes

A chaque position des chiffres d'un nombre peut correspondre un poids qui n'est pas une puissance de la même base. Le système utilisé pour représenter le temps (heures-minutes-secondes) est un exemple d'un tel système. La représentation des heures en base 24 n'utilise toutefois pas les chiffres 0 à N en s'inspirant de ce qui a été fait pour l'hexadécimal, mais les nombres décimaux de 0 à 23. (code naturel) ou d'autres notations incohérentes telles que 5 PM (post meridiem).

2.1.17 Nombres entiers relatifs

L'ensemble des *nombres entiers relatifs* est un ensemble ordonné comportant des nombres positifs et négatifs. Les *nombres négatifs* sont caractérisés par leur signe et leur valeur absolue.

$$\mathbb{Z} = \{\ldots -N, -N+1, \ldots -1, 0, 1, \ldots, N, N+1, \ldots\} \qquad (2.7)$$

L'utilisation du signe + est optionnelle devant les nombres positifs. Dans une représentation en champ fixe, un champ supplémentaire doit être ajouté pour le signe (fig. 2.5).

Fig. 2.5

Le champ du signe est souvent placé devant, mais cette représentation est purement conventionnelle et il faut remarquer que le signe et les chiffres sont des objets de nature fondamentalement différente, devant se traiter différemment. Ceci nous amènera à considérer ultérieurement d'autres représentations pour les nombres négatifs (§ 2.4.10).

2.1.18 Exercice

Peut-on définir un système de numération dont la base est négative, les chiffres étant toujours des entiers positifs inférieurs en valeur absolue à la base? Si oui, écrire en base -5 les entiers positifs de 0 à 20. et les entiers négatifs de -1 à -8 . Quel peut être l'intérêt d'un tel système?

2.1.19 Exercice

Peut-on définir un système de numération dont la base p est positive, mais dont les p chiffres sont certains positifs, d'autres négatifs? Définir un système de base 3 avec pour chiffres 0 (noté o), $+1$ (noté +) et -1 (noté $-$). Compter dans cette base de -10. à $+10$. Quels sont les équivalents décimaux de $+o-o$ et de $-++$? Quel peut être l'intérêt d'un tel système?

2.1.20 Nombres purement fractionnaires

Un *nombre purement fractionnaire,* c'est-à-dire compris entre zéro et l'unité, peut toujours s'exprimer dans un système de numération de base p sous la forme:

$$N = 0, a_{-1}\, a_{-2}\, a_{-3} \ldots = \sum_{i=1}^{\infty} a_{-i} \cdot p^{-i} \tag{2.8}$$

La représentation d'un nombre purement fractionnaire dans un champ fixe de l positions (registre de l cellules) ne permet pas de représenter les chiffres de poids inférieur à p^{-l} (fig. 2.6).

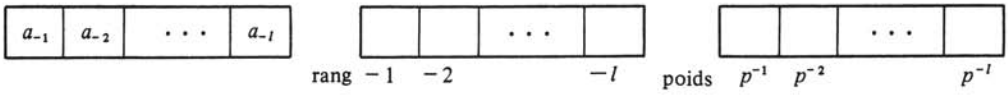


Fig. 2.6

La suppression des chiffres $a_{-l-1}, a_{-l-2}, \ldots$ crée une erreur absolue positive appelée *erreur d'arrondi*, au plus égale à p^{-l}. En effet:

$$N = \sum_{i=1}^{l} a_{-i} \cdot p^{-i} + \sum_{j=l+1}^{\infty} a_{-j} \cdot p^{-j} < \sum_{i=1}^{l} a_{-i} \cdot p^{-i} + p^{-l} \tag{2.9}$$

Si le terme $\Sigma_{j=l+1}^{\infty} a_{-j} \cdot p^{-j}$ est supérieur à $\frac{1}{2} p \cdot p^{-l-1} = \frac{1}{2} p^{-l}$, le nombre $N^+ = \Sigma_{i=1}^{l} a_{-i}\, p^{-i} + p^{-l}$ est plus proche de N que le nombre $N^- = \Sigma_{i=1}^{l} a_{-i} \cdot p^{-i}$. Le choix de la meilleure des deux valeurs N^+, N^- s'appelle *arrondi*. Ce choix permet de diminuer l'erreur d'arrondi à $\frac{1}{2} p^{-l}$ et de faire tendre la moyenne des erreurs sur un grand ensemble de nombres vers zéro (*erreur centrée*). Dans les calculs itératifs, les considérations sur les erreurs d'arrondi sont très importantes [35, 48].

On remarque dans la figure 2.6 que dans la représentation d'un champ fixe, le zéro et la virgule que l'on a l'habitude d'écrire en Europe devant la partie fractionnaire ne sont pas conservés.

Comme pour les entiers positifs, le signe d'un nombre fractionnaire négatif peut se représenter dans un champ supplémentaire.

Remarquons encore qu'un registre de k cellules peut contenir en base p indifféremment un entier positif inférieur à p^k ou un nombre fractionnaire positif avec une erreur inférieure à $\frac{1}{2} p^{-k}$. Le contexte, ou un champ supplémentaire, doit préciser la nature du nombre (fig. 2.7).

La façon dont les différents champs sont répartis est appelée *format.*

Fig. 2.7

2.1.21 Nombres réels

Tout *nombre réel positif* est la somme d'un nombre entier positif et d'un nombre purement fractionnaire positif.

$$N = a_n \ldots a_1 a_0, a_{-1} a_{-2} \ldots = \sum_{i=n}^{-\infty} a_i \cdot p^i \qquad (2.10)$$

Dans une représentation en champ fixe d'un nombre réel, un certain nombre de cellules sont réservées pour la partie entière et pour la partie fractionnaire; la grandeur maximum des nombres est limitée, ainsi que leur précision.

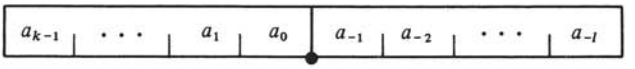


Fig. 2.8

C'est ainsi que le nombre de la figure 2.8 ne peut représenter que les nombres réels compris entre 0 et p^k, avec une *erreur absolue* inférieure à p^{-l} et une *erreur relative* inférieure à p^{-k-l}.

Le rang de la séparation entre la partie entière et la partie fractionnaire peut être explicité dans un champ supplémentaire.

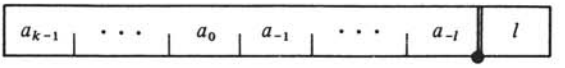


Fig. 2.9

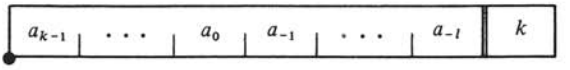


Fig. 2.10

Avec la représentation de la figure 2.9, la virgule a été déplacée vers la droite. Le nombre est représenté sous forme d'un entier de $l+k$ chiffres et d'un facteur correctif égal à p^{-l}.

La virgule a par contre été déplacée à gauche dans le cas de la figure 2.10. Le nombre est devenu un nombre fractionnaire pur avec un facteur correctif égal à p^k.

La justification algébrique de ces deux cas de figure est évidente:

$$\sum_{i=k-1}^{-l} a_i \cdot p^i = \left(\sum_{i=k-1}^{-l} a_i \cdot p^{i+l} \right) \cdot p^{-l} = \left(\sum_{i=k-1}^{-l} a_i \cdot p^{i-k} \right) \cdot p^k \qquad (2.11)$$

2.1.22 Nombres flottants

Tout nombre peut être représenté sous forme d'un nombre réel et d'un facteur multiplicatif qui est une puissance de la base. Cette représentation est dite *flottante*. La *partie réelle* est appelée *mantisse* et la puissance de la base est l'*exposant* ou la *caractéristique*. Cette représentation n'est pas unique et pour diminuer au maximum les erreurs d'arrondi dans une représentation en champ fixe, on choisit en général l'exposant de façon telle qu'il n'y ait pas de zéro non significatif au début de la mantisse. Dans ce cas, la représentation est dite *normalisée*. L'erreur relative, dans le cas d'un champ normalisé de k cellules est égale à p^{-k+1}.

Le plus fréquemment, la mantisse est considérée comme un nombre purement fractionnaire plutôt que comme un nombre entier. Sa valeur est par conséquent comprise entre p^{-1} (inclus) et 1 (exclu).

2.1.23 Exemple

Avec un champ de 5 chiffres pour la mantisse, le nombre π admet les formats ou représentations flottants suivants (fig. 2.11).

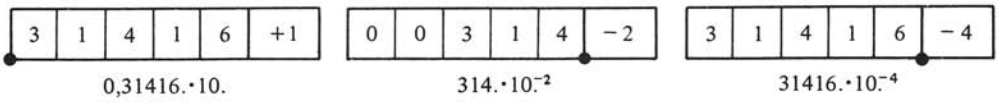


Fig. 2.11

Le premier et le dernier exemple sont normalisés, avec une précision relative du résultat meilleure que 10^{-4}. Le problème du signe de la mantisse et de l'exposant sera revu au paragraphe 2.7.3.

2.1.24 Exercice

Dans un registre, un champ de 6 chiffres décimaux a été réservé pour la mantisse (positive) et un champ de deux chiffres pour l'exposant (positif). Exprimer en décimal le plus grand et le plus petit nombre que l'on peut représenter, dans le cas d'une mantisse entière non normalisée, entière normalisée et fractionnaire normalisée.

2.2 SYSTÈMES USUELS

2.2.1 Système binaire

La base du système binaire est 2 et les chiffres, appelés bits, sont 0 et 1.

Un mot de 8 bits est appelé *octet* (*byte*). Un mot de 4 bits est appelé *quartet* (*nibble*). Un octet peut contenir deux quartets, et deux octets forment un mot de

16 bits pour lequel il n'y a pas de terme bien défini. Proposons *bisoctet*. En anglais, on trouve doublet *word* (Digital Equipment) et *half-word* (IBM). Les mots de 32 bits sont appelés *quadlets* (terme non normalisé). A chaque bit d'un mot binaire correspond un rang et un poids qui peuvent s'exprimer en décimal (fig. 2.12) ou en sexadécimal (fig. 2.13).

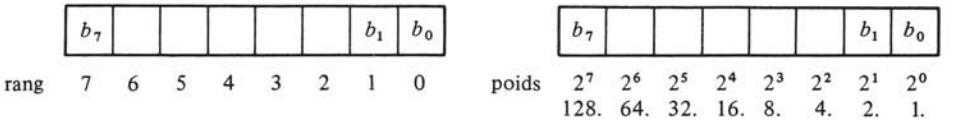


Fig. 2.12

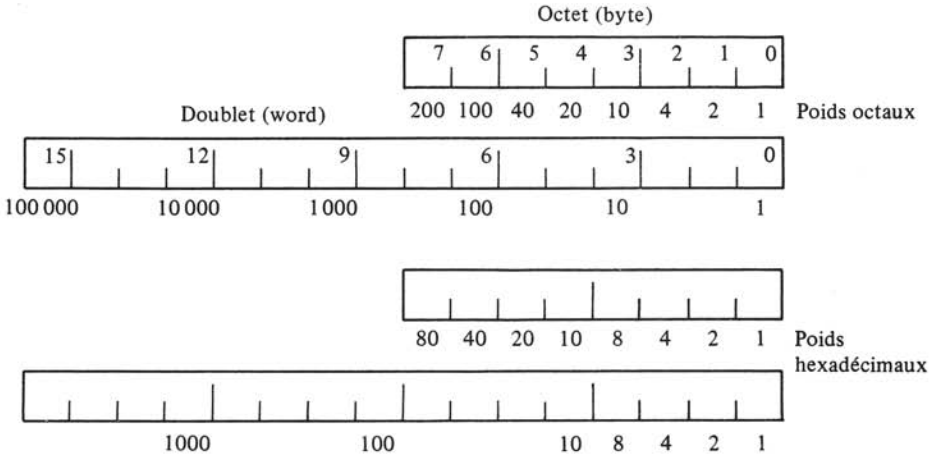


Fig. 2.13

L'octet est le groupement de bits le plus fréquemment rencontré; 256. ordres peuvent être codés dans un octet. L'alphabet, les chiffres et les signes spéciaux se codent facilement dans des mots de 8 bits (§ 7.1.3). Deux quartets représentant chacun un chiffre décimal codé en BCD, se placent dans un octet; tous les nombres entiers positifs de 0 à 99. peuvent se représenter de cette manière.

La valeur des puissances de 2 est donnée en annexe (§ 7.1.1) avec les valeurs octales et sexadécimales de quelques constantes (§ 7.1.2). Il faut se souvenir que $2^6 = 64.$, $2^8 = 256.$, et surtout $2^{10} = 1024. = 1\,K$. Cette dernière quantité est souvent utilisée pour les tailles mémoire et est appelée *kilo*.

Avec seulement deux chiffres différents, la table d'addition et la table de multiplication binaire sont particulièrement simples (fig. 2.14).

+	0	1
0	0	1
1	1	10

·	0	1
0	0	0
1	0	1

Fig. 2.14

Par contre, les nombres binaires sont longs et peu pratiques pour un être humain. Le rapport entre la longueur d'un nombre N exprimé en binaire et la longueur du même nombre exprimé en décimal selon la formule 2.6 tend vers

$$\frac{\log_2 N}{\log_{10} N} = \frac{1}{\log_{10} 2} \cong 3{,}32 \qquad (2.12)$$

2.2.2 Système octal

Le système octal utilise la base 8 et les chiffres de 0 à 7. Il est employé comme sténographe du binaire, le binaire-codé-octal étant identique à la conversion en octal (§ 2.9.12).

Le système octal est le plus proche du système décimal et donne une "sensation" analogue. Les tables d'addition et de multiplication sont les suivantes (fig. 2.15).

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

·	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

Fig. 2.15

Pour effectuer une opération simple en octal, on peut soit utiliser la table d'addition ou de multiplication ci-dessus, soit effectuer les calculs partiels en décimal et convertir chaque résultat partiel en octal.

Par exemple, pour effectuer $0'54 + 0'47$, la lecture de la table d'addition donne :

- pour les unités : $4 + 7 = 3$ avec un report valant 1
- pour les "huitaines" : $1 + 5 + 4 = 2$ avec un report valant 1

le résultat est donc $0'123$.

En passant par le système décimal pour les calculs intermédiaires, cette opération peut être pensée :

- $4 + 7 = 11. = 8 + 3 = 3$ avec un report valant 1
- $5 + 4 + 1 = 10. = 8 + 2 = 2$ avec un report valant 1

d'où le résultat $0'123$.

Ces deux méthodes peuvent s'appliquer, bien qu'avec moins de facilité, aux autres opérations arithmétiques, et dans n'importe quelle base.

2.2.3 Système hexadécimal

Le principal inconvénient du système sexadécimal est la présence de 6 symboles supplémentaires (lettres A à F) pour représenter les chiffres. Le nombre et les symboles alphanumériques ne sont plus facilement distingables et les opérations simples ne peuvent pas s'effectuer mentalement. Ce système est néanmoins très utilisé de par la compacité de la représentation des nombres binaires en sexadécimal, ainsi que la commodité de séparation des mots de 16 bits en mots de 8 et 4 bits.

Dans la manipulation des nombres hexadécimaux, il faut au moins mémoriser les valeurs de A, C et F (fig. 2.16).

	0	0
	1	1
	2	2
	⋮	⋮
	9	9
→	A	10.
	B	11.
→	C	12.
	D	13.
	E	14.
→	F	15.

Fig. 2.16

Quelques opérations simples peuvent alors être entreprises de tête, par exemple C + 4 = 10 (12. + 4 = 16.), 7 + 7 = E (= 14. = 15. − 1, donc lettre avant F).

La table complète d'addition et de multiplication hexadécimale est donnée dans la figure 2.17. Il est souvent nécessaire de calculer en sexadécimal ou en octal pour vérifier les opérations binaires effectuées par la machine. Des calculatrices spécialisées ou programmables peuvent aider dans ce but.

2.2.4 Système décimal

Dans une calculatrice, les nombres décimaux sont généralement codés en binaire. Les codes les plus connus sont donnés dans la figure 2.18.

Le code naturel est appelé *décimal codé binaire* ou *BCD* (de l'anglais *Binary Coded Decimal*). C'est le code utilisé dans la majorité des applications, avec le *code ASCII* (American Standard Code for Information Interchange) (§ 7.1.3) qui a 4 bits compatibles avec le code BCD.

Les codes *XS3* et *Aiken* amenaient quelques petites simplifications dans la circuiterie des calculatrices et ont eu leur heure de gloire lorsque les tubes radio et les transistors étaient coûteux. Le code *Gray*, dont plusieurs variantes existent, est encore utilisé dans les capteurs de position car un seul bit change d'un mot au suivant. Les autres codes de la figure 2.18 apparaissent à certains niveaux des périphériques clavier et affichage.

2.2.5 Remarque

Le codage d'un nombre décimal en binaire (BCD) conserve la structure décimale et est très différent du nombre binaire équivalent. Par exemple, pour reprendre le cas

TABLE D'ADDITION HEXADÉCIMALE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

TABLE DE MULTIPLICATION HEXADÉCIMALE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	0	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	0	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	0	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	0	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	0	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	0	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	0	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	0	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	0	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	0	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	0	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	0	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	0	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	0	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

Fig. 2.17

DIGIT	BCD	ASCII 7 bits	XS3	AIKEN	GRAY	1 PARMI 10	7-SEGMENTS gfedcba	
0	0000	0110000	0011	0000	0000	0000000001	0111111	0
1	0001	0110001	0100	0001	0001	0000000010	0000110	1
2	0010	0110010	0101	0010	0011	0000000100	1011011	2
3	0011	0110011	0110	0011	0010	0000001000	1001111	3
4	0100	0110100	0111	0100	0110	0000010000	1100110	4
5	0101	0110101	1000	1011	1110	0000100000	1101101	5
6	0110	0110110	1001	1100	1010	0001000000	1111101	6
7	0111	0110111	1010	1101	1011	0010000000	0000111	7
8	1000	0111000	1011	1110	1001	0100000000	1111111	8
9	1001	0111001	1100	1111	1000	1000000000	1101111	9
Poids	8421	0008421	----	2421	----	9876543210	-------	
	Naturel pondéré	Pondéré		Pondéré		Pondéré		

Fig. 2.18

du numéro atomique de l'or, c'est un nombre qui s'écrit:

- en décimal 79.;
- en BCD (0 1 1 1) (1 0 0 1)
  80. 40. 20. 10. 8 4 2 1
- en décimal codé ASCII 7 bits (0110111) (0111001);
- en XS3 (1010) (1100);
- en binaire 1 0 0 1 1 1 1
  64. 32. 16. 8 4 2 1
- en binaire code ASCII 7 bits (0110001) (0110000)...

Le codage binaire d'un nombre décimal conduit à des mots plus longs que la conversion de ce nombre en binaire, mais ces nombres n'utilisent que 2 symboles. On a vu (formule 2.12 du § 2.2.1) que les nombres binaires sont asymptotiquement 3.32 fois plus longs que les nombres décimaux correspondants. Les nombres BCD sont eux 4 fois plus longs. Le rapport des longueurs est 0,83, c'est-à-dire qu'un nombre BCD est asymptotiquement 20% plus long que sont équivalent binaire.

2.2.6 Exercice

La base 40. est utilisée dans certains ordinateurs pour coder des programmes et des textes de façon compacte [14]. L'alphabet, les chiffres et quatre signes spéciaux sont mis en correspondance avec les chiffres de 0 à 39. en base 40. (codés en octal de 0 à 0'47, base 0'50).

Caractère	Equivalent ASCII octal	Code base O'50
Espace	0'40	0
A-Z	0'101 – 0'132	1 – 0'32
$	0'44	0'33
.	0'56	0'34
libre		0'35
0-9	0'60 – 0'71	0'36 – 0'47

Trois caractères successifs d'un texte sont considérés, après conversion, comme les trois chiffres d'un nombre en base 0'50. La valeur maximale de ce nombre est $40^3 - 1 = 64000. - 1 < 65536. = 2^{16}$, ce qui permet donc de mémoriser trois caractères dans un mot de 16 bits.

Calculer en octal la représentation en "code ASCII"; la chaîne est complétée par des espaces à la fin pour atteindre une longueur multiple de 3 avant conversion.

2.3 ADDITION DE NOMBRES ENTIERS

2.3.1 Opérandes et opérateurs

Un *opérateur* fait correspondre à un ou plusieurs mots ou nombres appelés *opérandes* un mot ou nombre appelé *résultat*. Le *schéma fonctionnel* de cette définition est donné dans la figure 2.19.

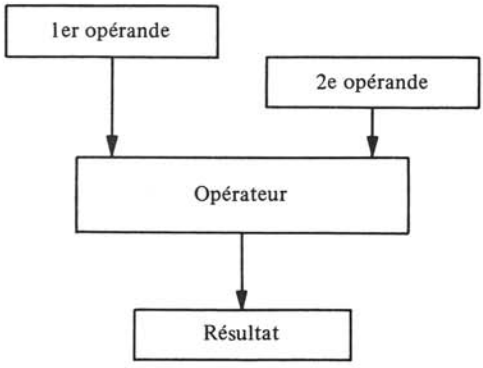


Fig. 2.19

Ce chapitre s'occupe plus particulièrement des opérateurs arithmétiques, effectuant sur des nombres les opérations arithmétiques usuelles.

Un opérateur peut n'avoir qu'un seul opérande (§ 2.4.8). Les opérations à plus de deux opérandes sont en général décomposées en suites d'opérations à deux opérandes au plus.

Les opérandes seront toujours représentés en champ fixe, avec en général pour chaque opérande et pour le résultat la même longueur de champ et le même type de représentation. Le cas des nombres fractionnaires ou flottants sera étudié dans la section 2.7.

La *longueur d'un opérateur* est égale à la longueur de l'opérande principal sur lequel il agit.

Un opérateur est souvent appelé *dyadique* s'il a deux opérandes et *monadique* s'il a un seul opérande.

Une opération dyadique peut être notée de façon abrégée:

OPÉRATEUR RÉSULTAT, 1er OPÉRANDE, 2e OPÉRANDE (2.13)

Une opération monadique est notée:

OPÉRATEUR RÉSULTAT, OPÉRANDE (2.14)

2.3.2 Addition de 2 nombres entiers positifs

Soient A et B deux nombres entiers positifs de k chiffres au plus représentés en base p dans des champs de k positions (registres de k cellules).

$$A = a_{k-1} \ldots a_1 a_0 = \sum_{i=0}^{k-1} a_i \cdot p^i$$
$$B = b_{k-1} \ldots b_1 b_0 = \sum_{i=0}^{k-1} b_i \cdot p^i \qquad (2.15)$$

La somme de ces deux nombres est égale à:

$$R = A + B = \sum_{i=0}^{k-1} (a_i + b_i) \cdot p^i \qquad (2.16)$$

Cette formule n'est pas directement utile, car $a_i + b_i$ n'est généralement pas un chiffre en base p. Il faut donc la transformer.

Remarquons que la somme R est un nombre entier positif de $k+1$ positions au plus, car à cause de la formule (2.4) du paragraphe 2.1.3:

$$A \leqslant p^k - 1, \quad B \leqslant p^k - 1$$
$$R = A + B \leqslant 2p^k - 2 < 2p^k - 1 = p^k + (p^k - 1) \qquad (2.17)$$

R est donc inférieur à la somme de p^k et d'un nombre de k chiffres. On peut donc écrire:

$$R = c_k \cdot p^k + \sum_{i=0}^{k-1} r_i \cdot p^i = c_k \cdot p^k + R',$$
$$\text{avec } c_k \in \{0, 1\} \quad \text{et} \quad r_i \in \{0, \ldots p^{-1}\} \qquad (2.18)$$

$R' = \Sigma_{i=0}^{R-1} r_i \cdot p^i$ est un nombre exprimable dans un champ de k positions.

Si le résultat, comme c'est usuellement le cas, doit occuper comme A et B un champ de k positions, ce n'est possible que si $c_k = 0$. Si $c_k = 1$, il y a *dépassement de capacité* (*overflow*) et le résultat R' de k positions correspond en fait au résultat de l'addition de A et B modulo p^k.

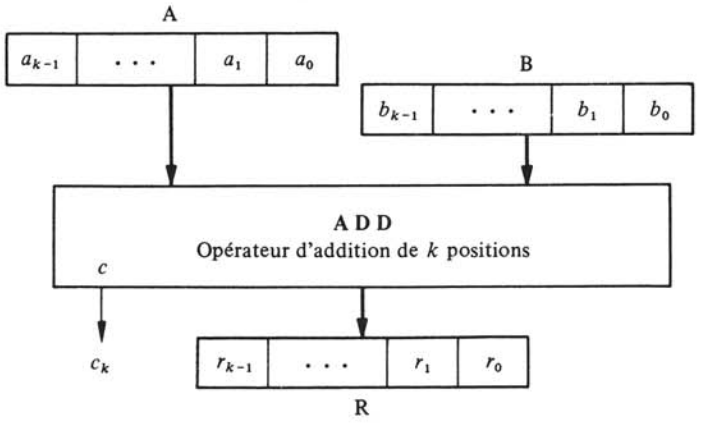


Fig. 2.20

La valeur binaire c_k est appelée *report* ou *retenue* et est souvent désignée par la lettre C et le terme anglais *Carry*.

La figure 2.20 illustre cette opération d'addition en champ fixe. L'opérateur d'addition est abrégé par les 3 lettres *ADD* et l'on peut décrire l'opération par l'expression ADD. k R, A, B qui met en évidence les opérandes sans mentionner explicitement l'effet sur le report, mais précise la longueur k des opérandes. R, A et B désignent des cases (registres, positions mémoire) qui contiennent des nombres, et pas les nombres eux-mêmes.

Dans le cas de l'addition, les deux opérandes s'appellent *cumulande* (*augend*) et *cumulateur* (*addend*) et le résultat *somme* (*sum*).

2.3.3 Exemple

Le mécanisme de l'addition décimale est bien connu de chacun. La figure 2.21 montre une addition type.

1	1	1			←	reports
3	4	7	9	2	←	A
3	7	2	5	4	←	B
7	2	0	4	6	←	R

Fig. 2.21

Le problème du dépassement de capacité ne se rencontre en général pas sur une feuille de papier, mais surgit avec n'importe quel boulier ou calculatrice en virgule fixe.

L'addition binaire a un mécanisme identique. La figure 2.22 donne 2 exemples en champ fixe avec des opérandes similaires.

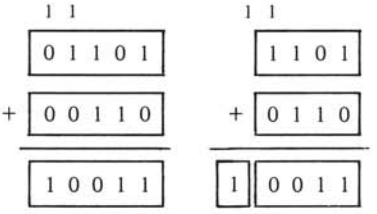


Fig. 2.22

Ces 2 opérations ne sont en fait pas identiques. La première est une addition de deux nombres de 5 bits conduisant à un résultat 5 bits correct. La seconde est une addition de deux nombres de 4 bits fournissant un résultat 4 bits incorrect (ou mieux incomplet) et un report indiquant un dépassement de capacité. Une bonne compréhension des mécanismes d'opérations en champ fixe est fondamentale dans l'étude des processeurs et de leur programmation.

2.3.4 Représentation graphique de l'addition

Sur l'axe des entiers positifs, tout entier en format fixe peut se représenter par un vecteur de longueur inférieure à p^{k-1} (fig. 2.23). La somme de 2 vecteurs illustre la

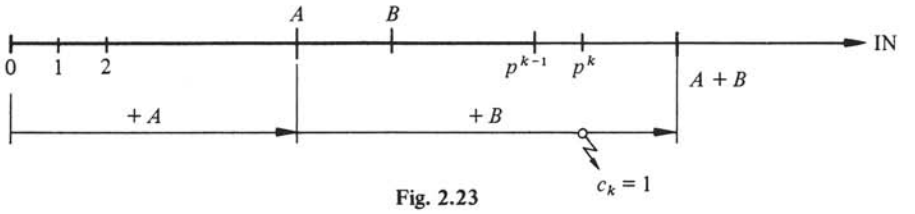


Fig. 2.23

somme de 2 nombres, et il y a dépassement de capacité (report $c_k = 1$) toutes les fois que le vecteur somme a une longueur supérieure ou égale à p^k.

Dans une représentation circulaire sous forme d'angles orientés ou segments circulaires, il y a dépassement toutes les fois que la somme est supérieure ou égale à un tour complet (fig. 2.24).

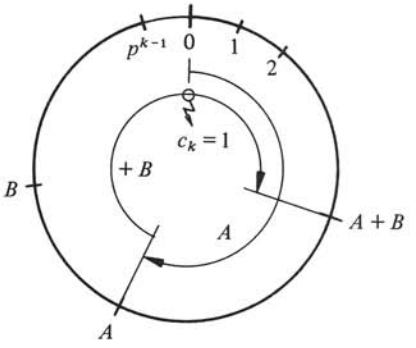


Fig. 2.24

2.3.5 Décomposition élémentaire de l'addition

La formule (2.16) vue au paragraphe 2.3.2 s'écrit développée :

$$R = (a_{k-1} + b_{k-1}) \cdot p^{k-1} + \ldots + (a_1 + b_1) \cdot p + (a_0 + b_0) \tag{2.19}$$

Considérons cette expression depuis la fin, c'est-à-dire à partir des poids faibles.

Etant donné que l'on a :

$$a_0 + b_0 \leqslant 2(p-1) = p + (p-2) \tag{2.20}$$

on peut écrire :

$a_0 + b_0 = c_1 \cdot p + r_0$, avec la définition suivante pour c_1 et r_0 :

$$\text{Si} \quad a_0 + b_0 < p, c_1 = 0 \quad \text{et} \quad r_0 = a_0 + b_0 < p \tag{2.21}$$

$$\text{Si} \quad a_0 + b_0 \geqslant p, c_1 = 1 \quad \text{et} \quad r_0 = a_0 + b_0 - p \geqslant 0,$$

avec de plus à cause de l'inégalité (2.20) $r_0 \leqslant p-2$.

On en déduit donc que dans tous les cas r_0 est bien un chiffre en base p. En termes mathématiques, on dit que r_0 est le résultat de l'addition de a_0 et b_0 *modulo p*.

La valeur binaire c_1, de poids p, doit être ajoutée aux termes de poids p de l'addition.

On démontre de la même manière que :

$$a_1 + b_1 + c_1 \leqslant p + (p-1) \tag{2.22}$$

d'où

$$a_1 + b_1 + c_1 = c_2 \cdot p + r_1 \quad \text{avec} \quad c_2 \in \{0,1\} \quad \text{et} \quad r_1 \in \{0, ..., p-1\} \tag{2.23}$$

En résumé on a la formule:

$$\begin{aligned} a_0 + b_0 &= c_1 \cdot p + r_0 \\ a_1 + b_1 + c_1 &= c_2 \cdot p + r_1 \end{aligned} \tag{2.24}$$

On peut continuer par récurrence; le terme c_k est identique au terme de la formule (2.18).

La première ligne fait appel à un opérateur élémentaire d'addition appelé *demi-additionneur*: les opérandes sont des chiffres a_0 et b_0, le résultat est un chiffre r_0 et une valeur binaire c_1 appelée report. Les deux chiffres et le résultat ont un poids $p^0 = 1$ mais le report c_1 a le poids $p^1 = p$. Cet opérateur est désigné par les lettres ADD.

La deuxième ligne et les suivantes font appel à un opérateur plus complet appelé *additionneur*. Les opérandes sont deux chiffres a_i et b_i, plus une variable binaire c_1 de même poids appelée C_{in} (Carry in). Le résultat est un chiffre de même poids et un report c_{i+1} de poids immédiatement supérieur appelé C_{out} (Carry out). Cet opérateur est désigné par les lettres *ADDC* (*add with carry*). Graphiquement la formule (2.24) peut se représenter comme dans la figure 2.25.

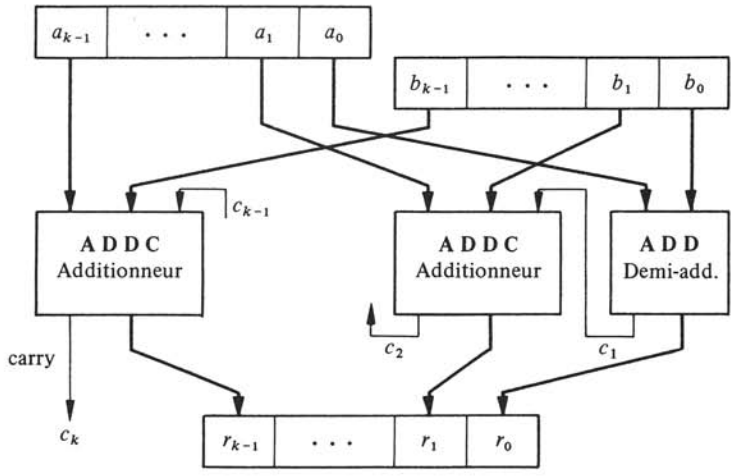


Fig. 2.25

2.3.6 Décomposition spatiale

Supposons que pour opérer sur des nombres de 12. chiffres, seuls des registres et opérateurs d'addition de 4 chiffres soient disponibles. L'addition peut dans ce cas se faire sur les 4 chiffres de poids faible, avec un report ne signifiant pas un dépassement de capacité, mais une correction d'une unité à apporter à l'addition des 4 chiffres de poids supérieur. Seul le dernier report indique un dépassement de capacité (fig. 2.26).

2.3.7 Exercice

Démontrer les propriétés énoncées dans le paragraphe 2.3.6.

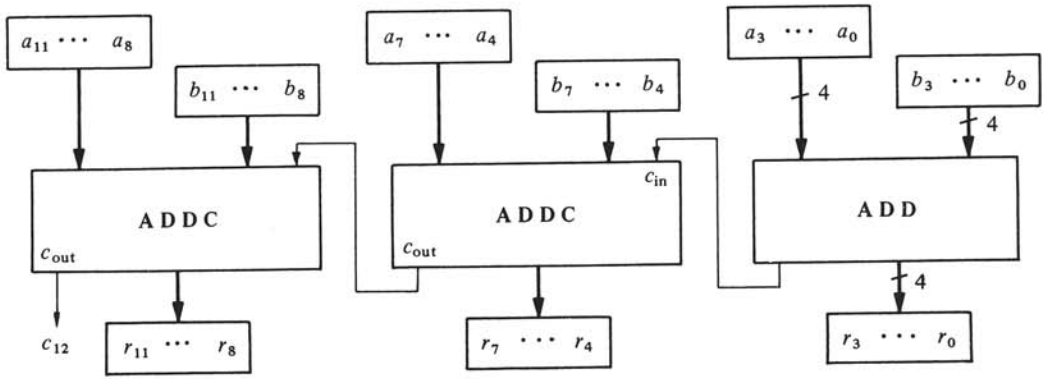


Fig. 2.26

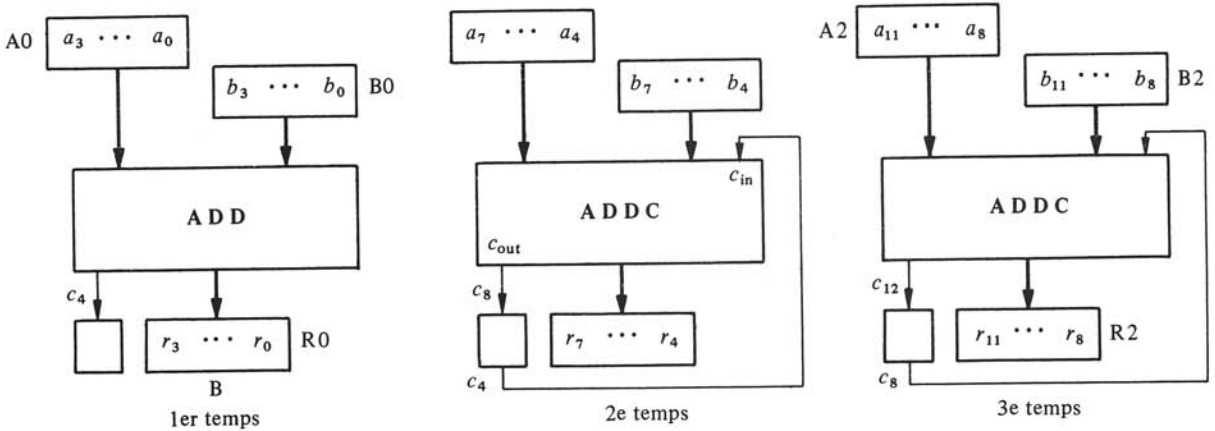


Fig. 2.27

2.3.8 Décomposition temporelle

L'opération donnée en exemple dans le paragraphe précédent peut être effectuée par un opérateur d'addition unique agissant successivement sur les 4 chiffres de poids faible, sur la partie médiane, et sur les poids forts des nombres donnés. Le report éventuel créé lors de chaque opération partielle doit être mémorisé temporairement pour agir comme correction sur les poids faibles lors de l'addition suivante (fig. 2.27).

Cette séquence d'opérations peut se noter:

```
ADD. 4   R0, A0, B0   (1er temps)
ADDC. 4  R1, A1, B1   (2ème temps)          (2.25)
ADDC. 4  R2, A2, B2   (3ème temps)
```

Ceci est un "programme" de trois instructions.

2.3.9 Exemple

La figure 2.28 donne un exemple d'addition binaire sur des nombres de 12 bits avec un additionneur 4 bits.

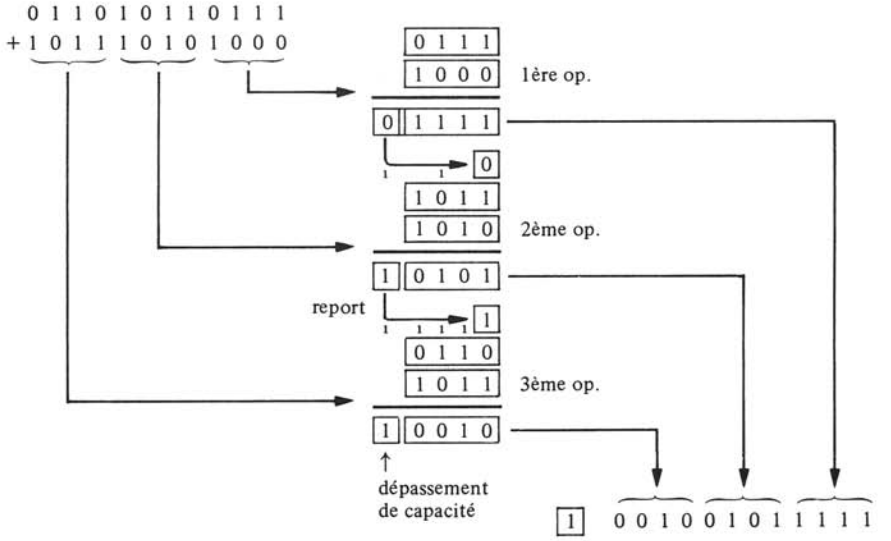


Fig. 2.28

2.3.10 Exemple

L'addition de deux nombres de 24 bits en mémoire de 8 bits, dont les poids faibles sont respectivement aux adresses NB1 et NB2, avec copie du résultat dans RES (fig. 2.29) se note par l'instruction:

```
ADD.24   RES, NB1, NB2                      (2.26)
```

Avec un microprocesseur 8 bits, cette opération doit être décomposée temporellement en n'utilisant que des opérations d'addition 8 bits pour lesquelles l'opérande

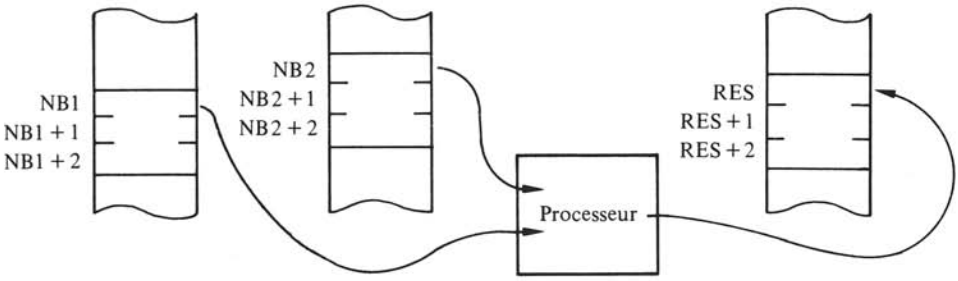


Fig. 2.29

résultat est identique au 1er opérande source et n'est pas répété. Dans le cas d'un microprocesseur de type 6800, 6802, 6809, 6502, etc. [53] et avec les notations dont le détail sera donné au chapitre 4, on écrira le programme suivant, dans lequel le mnémonique *LOAD* signifie qu'il y a transfert du 2e opérande dans le registre du 1er opérande

```
LOAD    A, NB1
ADD     A, NB2      ADD.8   RES, NB1, NB2
LOAD    RES, A

LOAD    A, NB1
ADDC    A, NB2 + 1  ADDC.8  RES + 1, NB1 + 1, NB2 + 1      (2.27)
LOAD    RES + 1, A

LOAD    A, NB1 + 2
ADDC    A, NB2 + 2  ADDC.8  RES + 2, NB1 + 2, NB2 + 2
LOAD    RES + 2, A
```

2.3.11 Multiprécision

La décomposition temporelle (§ 2.3.8) est une technique abondamment utilisée pour effectuer avec un opérateur "court", et donc bon marché, des opérations sur des nombres "longs". On parle d'opération en *multiprécision*, car la longueur des nombres est un multiple de la longueur de l'opérateur.

La séquence temporelle à exécuter pour effectuer une opération décomposée dans le temps est décrite le plus souvent par un *organigramme*, dont l'étude générale se fera au chapitre 4. Un premier exemple d'organigramme, correspondant à la décomposition temporelle de la figure 2.27, est donné dans la figure 2.30. Chaque rectangle d'un organigramme caractérise une opération s'effectuant à un instant donné. Le libellé caractérisant l'opération est souvent peu précis, car le but de l'organigramme est de mette en évidence l'aspect temporel, et l'aspect fonctionnel.

2.3.12 Propagation du report

La décomposition spatiale de l'addition met en évidence une propagation en cascade du carry qui augmente le temps de calcul. Une décomposition différente de

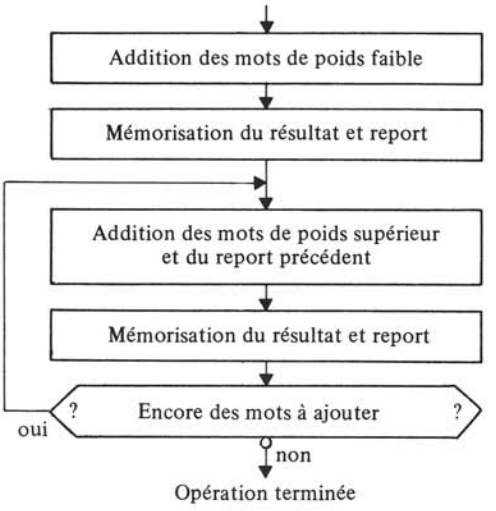


Fig. 2.30

l'opération d'addition [37-39] conduit dans le cas du binaire à la génération de deux signaux P et G dont la combinaison dans un circuit spécial de *report anticipé* (*carry-look ahead*) accélère l'opération (fig. 2.31).

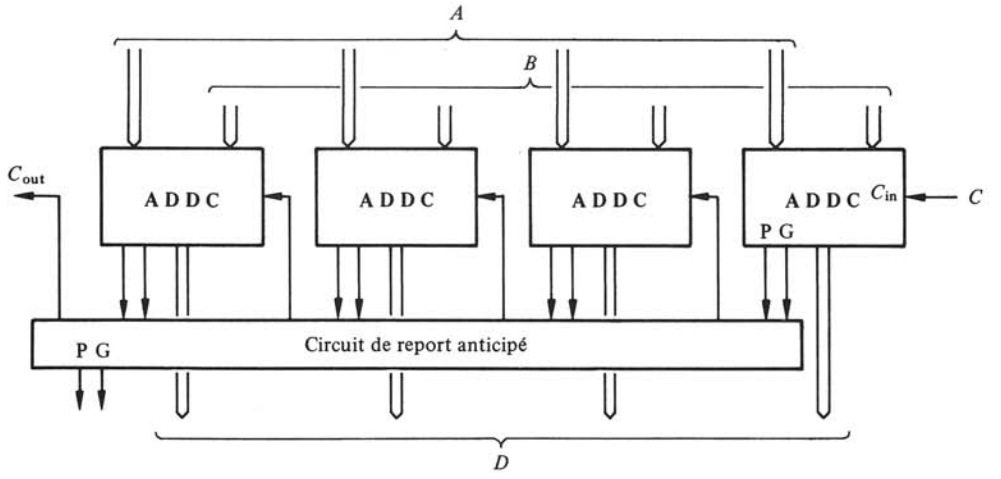


Fig. 2.31

Des circuits intégrés additionneurs binaires 4 bits et report anticipé existent et peuvent être cascadés pour calculer avec 10 circuits une addition 32 bits en moins de 20 nanosecondes.

2.3.13 Addition de un

Une opération fréquente est l'addition d'une unité à un nombre. Cette opération est appelée *incrémentation*; l'opérateur correspondant est désigné par les lettres *INC*

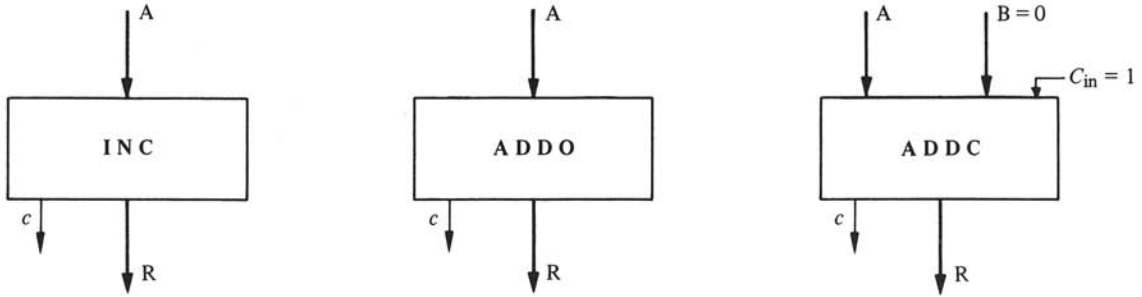


Fig. 2.32

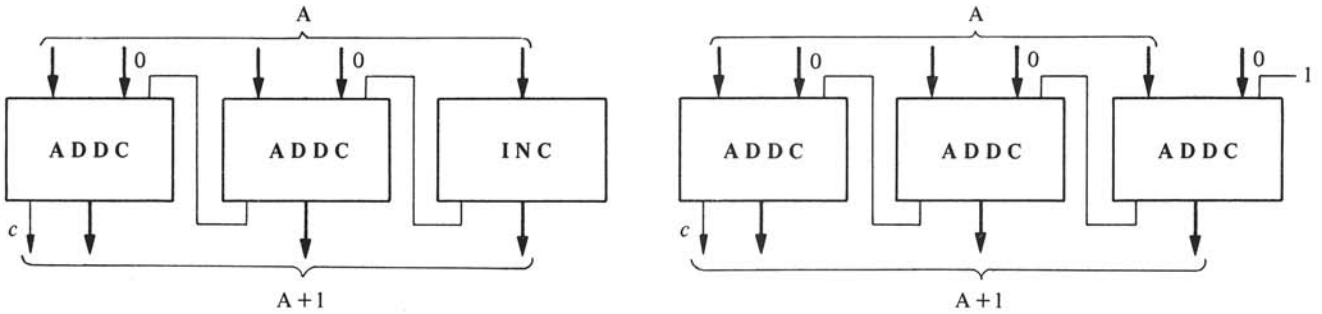


Fig. 2.33

ou plus rarement par les lettres ADDO (Add one). L'opérateur ADDC peut également être utilisé si le 2ème opérande est nul et si le report d'entrée C_{in} est égal à 1. (fig. 2.32).

Le schéma fonctionnel de la décomposition spatiale de l'incrémentation est donné à la figure 2.33. La démonstration est laissée comme exercice.

Dans le cas d'une décomposition temporelle avec des opérandes identiques à ceux du paragraphe 2.3.8, on peut noter que

$$\begin{array}{lll} & & \text{INC.4} \quad \text{R0, A0} \\ \text{INC.12} \quad \text{R, A} & \text{est identique à} & \text{ADDC.4} \quad \text{R1, A1} \\ & & \text{ADDC.4} \quad \text{R2, A2} \end{array} \qquad (2.28)$$

2.3.14 Exercice

Démontrer algébriquement le schéma fonctionnel de la figure 2.33 et établir que si le report créé par l'un des opérateurs vaut 1, tous les chiffres de poids plus faibles que ce report sont nuls.

2.4 SOUSTRACTION ET COMPLÉMENT

2.4.1 Soustraction de 2 nombres entiers positifs

Soustraire deux nombres A et B, c'est trouver un nombre R qui, ajouté au 2ème (B), donne le 1er (A). Contrairement à l'addition, l'opération de soustraction n'est pas commutative et le résultat n'est positif que si le 1er nombre (A) est supérieur au second (B). Si le résultat est négatif, la pratique courante veut que l'on soustraie au 2ème nombre le 1er, et que l'on donne au résultat le signe moins. La figure 2.34 donne le schéma fonctionnel de cette opération, et met en évidence un résultat hybride constitué d'un nombre positif et d'un signe.

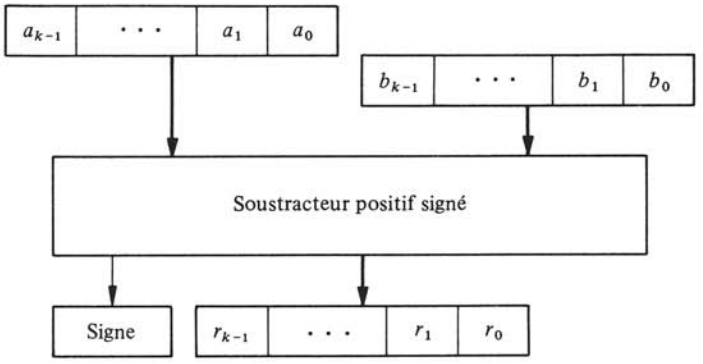


Fig. 2.34

Le signe est une variable dite *binaire* ou *booléenne*, codable par les bits 0 et 1. La convention la plus fréquente fait correspondre au + le bit 0 et au − le bit 1.

Dans une soustraction, le 1er opérande est appelé *diminuande* (*minuend*), le second (que l'on soustrait) *diminuteur* (*subtrahend*) et le résultat *différence* (*difference*).

2.4.2 Décomposition de la soustraction

Considérons la soustraction de 2 nombres en champ fixe. On peut écrire:

$$R = A - B = (a_{k-1} - b_{k-1}) \cdot p^{k-1} + \ldots + (a_1 - b_1) \cdot p + (a_0 - b_0) \qquad (2.29)$$

Analysons cette expression à partir de la fin. On a $|a_0 - b_0| < p$ car $0 \leqslant a_0 < p$ et $0 \leqslant b_0 < p$. On peut écrire:

$$R = (a_{k-1} - b_{k-1}) \cdot p^{k-1} + \ldots + (a_1 - b_1 - c_1) \cdot p + (c_1 \cdot p + a_0 - b_0)$$

avec $c_1 = 0$ si $a_0 - b_0 \geqslant 0$ (2.30)

et $c_1 = 1$ si $a_0 - b_0 < 0$

Si l'on pose $r_0 = c_1 \cdot p + a_0 - b_0$, il est facile de vérifier que $0 \leqslant r_0 \leqslant p$. Considérons maintenant l'avant-dernier terme $a_1 - b_1 - c_1$. Ce terme satisfait à l'inégalité:

$$|a_1 - b_1 - c_1| \leqslant p$$

car (2.31)

$$a_1 \leqslant p - 1, \quad b_1 \leqslant p - 1, \quad c_1 \leqslant 1.$$

On peut donc écrire:

$$R = (a_{k-1} - b_{k-1}) \cdot p^{k-1} + \ldots + (\ldots - c_2) \cdot p^2 + (c_2 \cdot p + a_1 - b_1 - c_1)\, p + r_0$$

avec $c_2 = 0$ si $a_1 - b_1 - c_1 \geqslant 0$ (2.32)

et $c_2 = 1$ si $a_1 - b_1 - c_1 < 0$

Posons $r_1 = c_2 \cdot p + a_1 - b_1 - c_1$; on peut montrer comme avant que r_1 est un chiffre en base p. Par récurrence, on arrive à l'expression:

$$\begin{aligned} R &= -c_k \cdot p^k + (c_k \cdot p + a_{k-1} - b_{k-1} - c_{k-1}) \cdot p^{k-1} + \ldots + \\ &\quad + (c_2 \cdot p + a_1 - b_1 - c_1) \cdot p + (c_1 \cdot p + a_0 - b_0) = \\ &= -c_k p^k + \sum_{j=0}^{k=1} [c_{j+1} \cdot p + a_j - b_j - c_j] \cdot p^j \qquad \text{avec } c_0 = 0 \end{aligned} \qquad (2.33)$$

$$R = -c_k \cdot p^k + r_{k-1} \cdot p^{k-1} + \ldots + r_1 \cdot p + r_0 = -c_k \cdot p^k + R' \qquad (2.34)$$

avec $c_k \in \{0,1\}$ $\quad r_i \in \{0, \ldots, p-1\}$

La formule (2.34) peut se représenter par la figure 2.35. Cette décomposition conduit à la définition d'opérateurs élémentaires appelés *demi-soustracteurs* et *soustracteurs*. Les symboles *SUB* (*subtract*) et *SUBC* (*subtract and subtract carry*) caractérisent les opérateurs.

2.4.3 Représentation naturelle des résultats négatifs

L'hypothèse $A \geqslant B$ n'a jamais été utilisée dans les développements du paragraphe précédent. Si $A \geqslant B$, on a nécessairement $c_k = 0$ et $R = R'$. Si $A < B$, on a $c_k = 1$ et

$$R = -p^k + R' \quad \text{ou} \quad R' = p^k + R \qquad (2.35)$$

Le résultat R' est un nombre positif (alors que R est négatif); c'est la *représentation naturelle d'un résultat négatif* en champ fixe, dite aussi représentation sous forme de *complément vrai* ou *complément à* p^k.

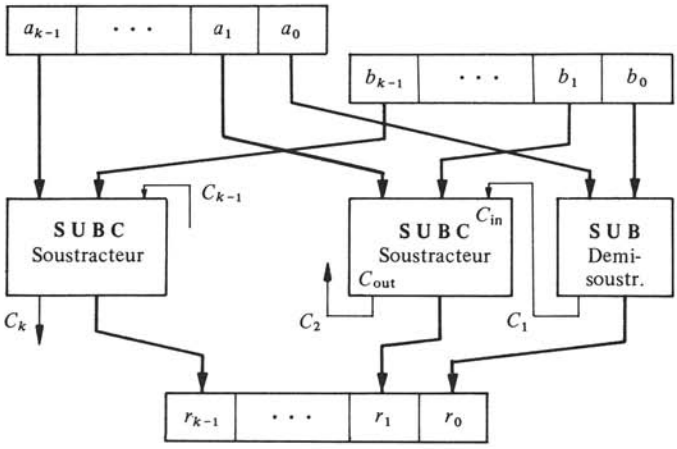


Fig. 2.35

La figure 2.36 concrétise la formule (2.34) et les considérations qui précèdent. Le résultat de la soustraction est un nombre positif, égal au résultat cherché si $c_k = 0$, et égal au complément vrai du résultat si $c_k = 1$. La variable binaire c_k représente le signe du résultat. Si le résultat attendu est positif, c_k signifie un *soupassement* (*underflow*) ou dépassement de capacité par valeurs inférieures.

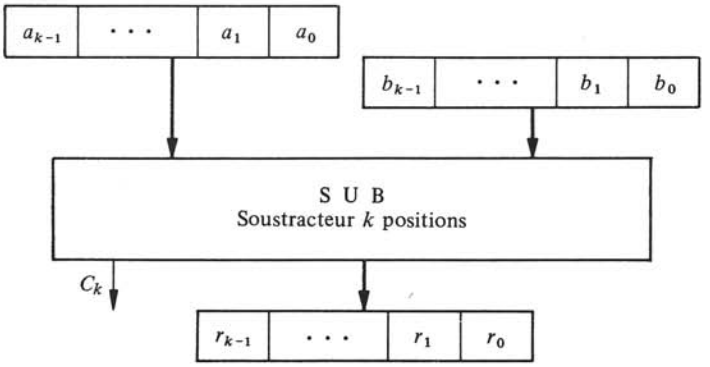


Fig. 2.36

2.4.4 Exemple

Effectuons dans le système décimal une soustraction en champ fixe de 4 positions dont le résultat est négatif (fig. 2.37). Si les opérandes ne sont pas permutés, le résultat apparaît sous forme complémentaire. La différence à 10 000 donne la valeur absolue du résultat.

Cette technique du complément décimal est bien connue de par son application aux logarithmes. La plupart des anciennes machines à calculer mécaniques affichaient les résultats négatifs sous forme complémentaire. Les calculatrices modernes obtiennent également un résultat sous forme complémentaire, mais ce résultat est immédiatement retransformé par une opération supplémentaire.

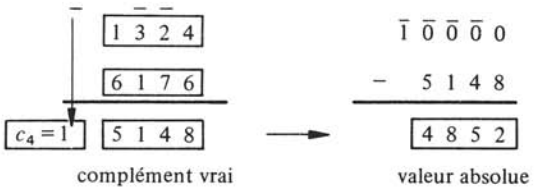


Fig. 2.37

2.4.5 Exercice

Effectuer en octal dans un champ de 5 chiffres et en appliquant strictement l'algorithme de soustraction, les opérations suivantes: 324 − 17, 32547 − 63217.

Effectuer en sexadécimal (champ de 2 chiffres) les opérations suivantes: 3F − 17, 3 − 10.

Effectuer en binaire (champ de 8 bits) les opérations suivantes: 1101 − 110110, 01101101 − 01011011, 11 − 111.

2.4.6 Décomposition spatiale de la soustraction

Dans l'expression (2.33), des groupements peuvent être effectués pour mettre en évidence l'effet de la soustraction sur des nombres décomposés en sous-champs (fig. 2.38).

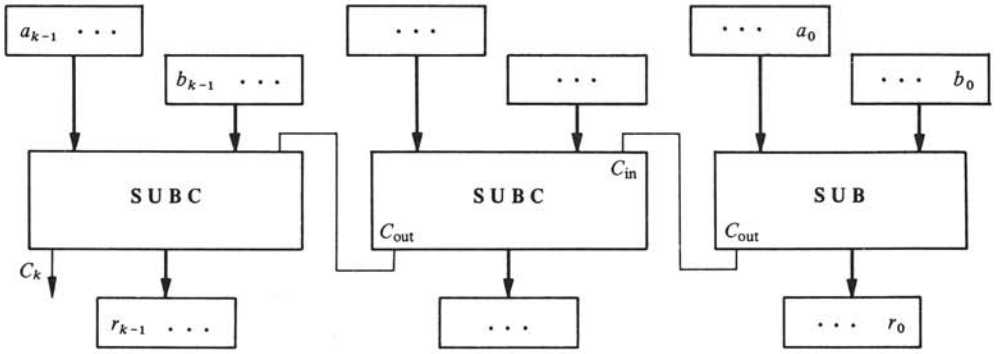


Fig. 2.38

Deux types d'opérateurs apparaissent dans la figure 2.38. L'opérateur SUB calcule $A - B$ et donne un résultat R et un emprunt éventuel C_{out}. L'opérateur SUBC tient compte de l'emprunt de l'étage précédent. Il est évident que si cet emprunt est nul ($C_{in} = 0$), l'opérateur SUBC peut être utilisé à la place de l'opérateur SUB.

De façon très similaire à ce qui a été vu au paragraphe 2.3.6, la décomposition spatiale de la figure 2.38 conduit à une décomposition temporelle analogue à celle de la figure 2.27 (§ 2.3.8). On peut noter que

		SUB.4 R0, A0, B0	
SUB.12 R, A, B	est identique à	SUBC.4 R1, A1, B1	(2.36)
		SUBC.4 R2, A2, B2	

2.4.7 Représentation graphique de la soustraction

Dans une représentation linéaire des entiers positifs et négatifs (fig. 2.39), l'opération de soustraction correspond à une succession de vecteurs correctement orientés.

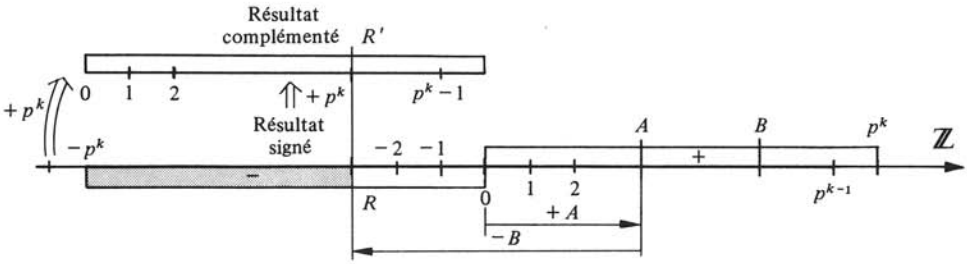


Fig. 2.39

Si le résultat est négatif, la valeur complémentée s'obtient en ajoutant p^k (formule (2.35)).

Un résultat négatif peut être noté sous forme signée ou sous forme complémentaire.

Etant donné que les nombres sont limités à la valeur p^k et que les opérations s'effectuent modulo p^k, une représentation des nombres sur un cercle de périmètre p^k (fig. 2.40) correspond exactement aux opérations effectuées et explicite le caractère naturel de la définition du complément.

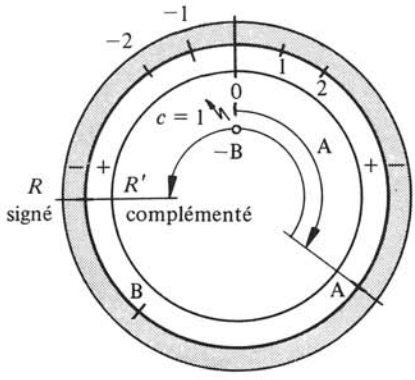


Fig. 2.40

2.4.8 Représentation des nombres négatifs

Tout nombre négatif est l'opposé de sa valeur absolue. A cause de (2.35), on peut écrire :

$$-B = 0 - B = -p^k + B' \qquad \text{(modulo } p^k\text{)} \tag{2.37}$$

d'où l'on tire :

$$B' = p^k - B \tag{2.38}$$

B' est le *complément vrai* de B; à chaque nombre négatif $(-B)$ correspond un complé-

ment vrai B' unique. A chaque nombre positif B correspond un opposé $-B$ et un complément vrai B'. Ces deux opérations sont involutives : le complément du complément est le nombre lui-même. Pour distinguer un nombre positif et son complément, qui est également positif, on associe au complément une variable binaire appelée signe.

L'opérateur qui fait correspondre à un nombre positif son complément vrai est désigné par les lettres *NEG* (*negate*). La figure 2.41 illustre l'effet de cet opérateur et de l'opérateur "opposé", qui n'agit que sur le signe par l'opération *CHS* (*change sign*).

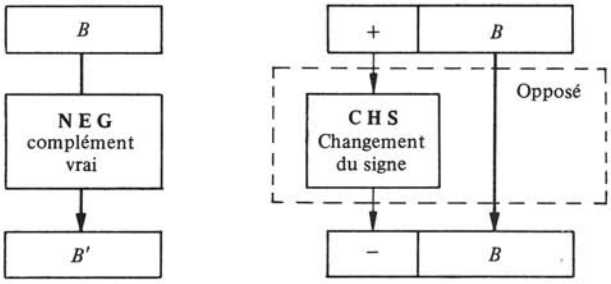


Fig. 2.41

2.4.9 Soustraction par addition du complément

Toute soustraction de deux nombres représentés dans un champ fixe de k positions se ramène à l'addition du complément du 2ème nombre. En effet :

$$A - B = A + (p^k - B) - p^k = A + B' - p^k \qquad (2.39)$$

Le terme p^k à soustraire prouve que l'opération $A + B'$ produit un dépassement de capacité, de poids p^k, lorsque le résultat est positif. Au contraire, si le résultat de $A - B$ est négatif, il n'y a pas de dépassement de capacité.

En champ fixe, la différence entre soustraction et addition du complément vrai est donc relativement importante. La figure 2.42 montre cette différence; le nom ACOO pour l'opérateur qui ajoute le complément vrai, sera justifié au paragraphe 2.4.26.

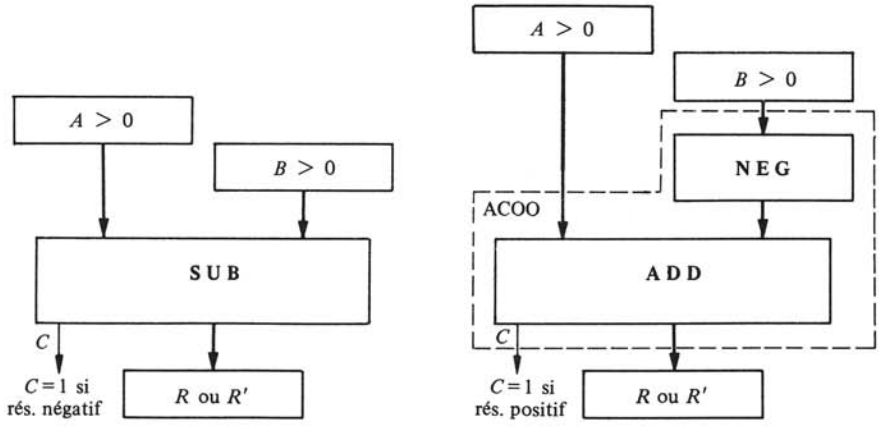


Fig. 2.42

2.4.10 Chiffre de signe

La représentation du signe d'un nombre négatif représenté sous forme de complément (fig. 2.41) n'est pas cohérente. Transformons la formule (2.39) en considérant un chiffre supplémentaire dans le champ des nombres et écrivons:

$$\begin{aligned} A-B=R &= -c_k \cdot p^k + R' = -p^{k+1} + (p-1)\cdot p^k + R' \quad \text{si} \quad c_k = 1 \\ &= 0 + 0 \cdot p^k + R' \qquad\qquad\qquad\quad \text{si} \quad c_k = 0 \end{aligned} \tag{2.40}$$

On voit que le chiffre de poids fort du résultat vaut 0 dans tous les cas où le résultat est positif, et $(p-1)$ lorsque le résultat est négatif (fig. 2.43). Le chiffre de poids fort contient l'information concernant le signe du résultat; ce chiffre est alors appelé *chiffre de signe*.

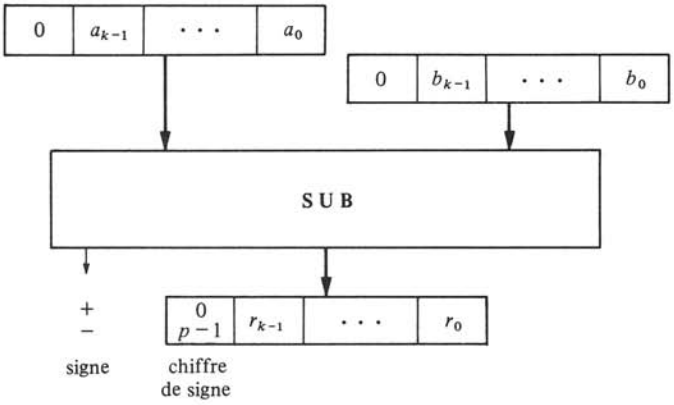


Fig. 2.43

La convention du chiffre de signe se généralise à la représentation des nombres négatifs sous forme de complément vrai. A chaque nombre positif ou négatif, correspond un nombre positif unique compris entre 0 et p^{k-1} pour les nombres positifs, et $(p-1)\cdot p^k$ et $p^{k+1}-1$ pour les nombres négatifs. La figure 2.44 représente cette nouvelle convention et doit être comparée avec la figure 2.39.

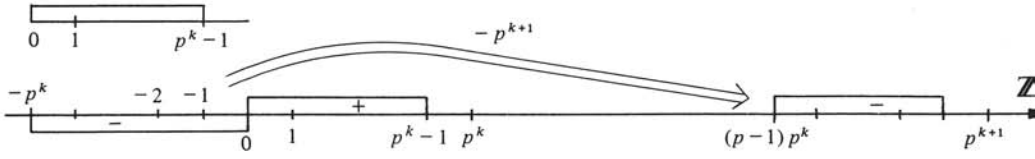


Fig. 2.44

2.4.11 Nombres logiques et arithmétiques

On appelle *nombre logique* un nombre entier dont le premier chiffre n'est pas un chiffre de signe. Les nombres logiques sont toujours positifs, même lorsqu'ils représentent le complément vrai d'un nombre négatif. Un *nombre arithmétique* est un nombre entier représenté sous forme de complément vrai s'il est négatif, et dont le premier chiffre est un chiffre de signe. On peut prendre le complément vrai aussi bien d'un nombre logique, que d'un nombre arithmétique. Dans le cas d'un nombre logique, rien

dans le nombre lui-même ne permet de reconnaître la forme directe et la forme complémentaire. Le premier chiffre d'un nombre arithmétique permet par contre de reconnaître si ce nombre est positif ou négatif; s'il est négatif, il est représenté sous forme du complément vrai de sa valeur absolue.

Par exemple, si les nombres sont décimaux avec un nombre total de 6 chiffres ($k = 5$) 034721 est soit un nombre logique, soit un nombre arithmétique positif. Le nombre 999321 est soit un nombre logique, soit un nombre arithmétique négatif, égal à -679. Le nombre 543210 ne peut être qu'un nombre logique.

Un compteur kilométrique de voiture affiche des nombres logiques. Les anciennes machines à calculer mécaniques affichaient des nombres arithmétiques (résultats négatifs commençant par un 9).

Remarquons que, dans ces définitions, les termes de logique et arithmétique ne sont pas utilisés dans un sens usuel. L'habitude semble toutefois être bien établie dans la documentation des fabricants.

2.4.12 Cercles logiques et arithmétiques

On appelle *cercle logique* de longueur k le cercle sur lequel on peut représenter les nombres entiers modulo p^k, comme expliqué au paragraphe 2.4.7. Les compléments des nombres peuvent se représenter sur le même cercle et le cercle logique doit être considéré comme deux cercles superposés, l'un représentant les nombres positifs, l'autre les nombres négatifs en complément vrai (fig. 2.45).

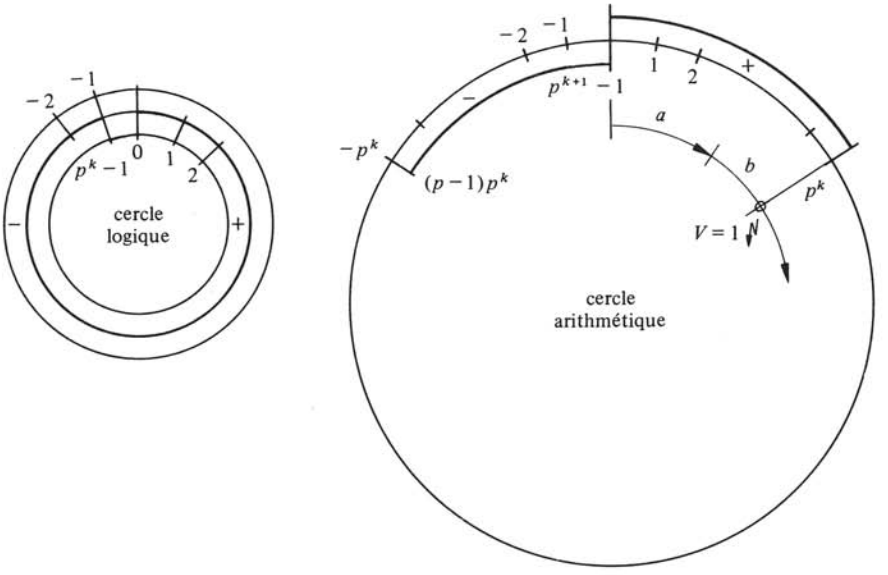


Fig. 2.45

On appelle *cercle arithmétique* de longueur $k+1$ le cercle sur lequel on peut représenter les nombres entiers arithmétiques modulo p^{k-1} dont le premier chiffre est le chiffre de signe. Sur ce cercle, les zones occupées par les nombres positifs et les nombres négatifs (sous forme de complément) sont disjointes (fig. 2.45) sauf si $p = 2$, ce qui est un avantage du binaire.

Une opération d'addition de deux nombres arithmétiques peut avoir un dépassement de capacité qu'il faut bien distinguer du dépassement de capacité sur les nombres logiques, tel que nous l'avons mis en évidence par la figure 2.24 du paragraphe 2.3.4.

La figure 2.45 donne un exemple d'opération tel que la somme de 2 nombres positifs dépasse la capacité, en ce sens que le résultat n'a plus un chiffre de signe correct. Ce type de dépassement de capacité est caractérisé par la lettre V (oVerflow) (§ 2.4.20).

2.4.13 Complément à 2

En binaire, le complément vrai ou complément à 2^k est appelé plus simplement *complément à* 2 et le chiffre de signe est appelé *bit de signe.*

Soit un champ de $k+1$ positions, appelé aussi champ de $k+1$ bits ou registre de $k+1$ bits. Ce champ peut représenter tous les nombres entiers binaires arithmétiques inférieurs en valeur absolue à 2^k (fig. 2.46).

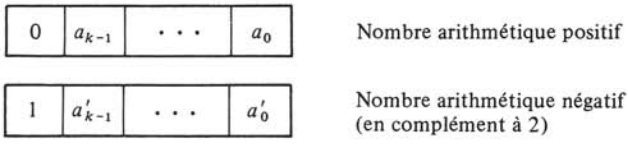


Fig. 2.46

La relation liant un nombre arithmétique à son complément (§ 2.4.8) devient en binaire :

$$B' = 2^{k+1} - B \tag{2.41}$$

Cette opération est particulièrement simple en binaire et conduit à une règle pratique de calcul du complément s'énonçant comme suit.

Pour obtenir le complément à 2 d'un nombre binaire, on examine les bits à partir de la droite (poids faible). Tous les bits rencontrés égaux à 0 et le premier bit égal à 1 sont conservés, tous les suivants sont inversés (fig. 2.47).

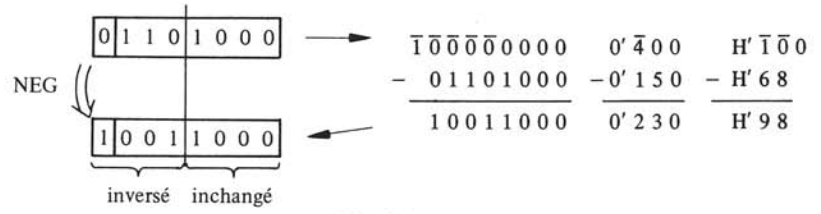


Fig. 2.47

Cette règle de calcul du complément est facile pour l'être humain, mais elle ne conduit pas, en général, à des implémentations technologiques avantageuses. La solution du paragraphe 2.4.26 est généralement préférée.

On remarque dans la figure 2.47 que dans la représentation octale et sexadécimale, le bit de signe est noyé dans le chiffre de poids fort. Il est toutefois immédiat que les nombres arithmétiques binaires 8 bits négatifs sont supérieurs ou égaux à 200 en octal (1er chiffre égal à 2 ou 3) et à 80 en sexadécimal (1er chiffre égal à 8, 9, A, B, C, D, E ou F).

2.4.14 Cercle des nombres arithmétiques binaires

La représentation des nombres arithmétiques sur un cercle (§ 2.4.12) conduit dans le cas du binaire à une situation particulièrement favorable. A chaque point du cercle correspond un nombre positif ou négatif; le nombre positif le plus grand est à côté du nombre négatif le plus grand en valeur absolue (fig. 2.48). Ceci facilite la plupart des opérations, mais nécessite une plus grande prudence dans l'utilisation de certains résultats.

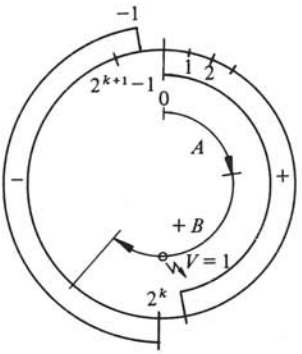


Fig. 2.48

La figure 2.48 montre par exemple que l'addition de 2 nombres arithmétiques positifs peut donner un résultat négatif. L'indicateur de dépassement de capacité V signale ce dépassement.

La figure 2.49 donne dans le cas de mots de 8 bits et de 7 bits, quelques correspondances entre nombres signés et arithmétiques. Les valeurs décimales sont données comme point de référence.

Mots de 8 bits				Mots de 7 bits			
Décimal	Octal	Binaire	Hexa	Décimal	Octal	Binaire	Hexa
− 1	377	11111111	FF	− 1	177	1111111	7F
− 2	376	11111110	FE	− 2	176	1111110	7E
− 8	370	11111000	F8	− 8	170	1111000	78
− 9	367	11110111	F7	− 9	167	1110111	77
− 10.	366	11110110	F6	− 10.	166	1110110	76
− 16.	360	11110000	F0	− 16.	160	1110000	70
− 17.	357	11101111	EF	− 17.	157	1101111	6E
− 127.	201	10000001	81	− 63.	101	1000001	41
− 128.	200	10000000	80	− 64.	100	1000000	40
+ 127.	177	01111111	7F	+ 63.	77	0111111	3F
+ 126.	176	01111110	7E	+ 62.	76	0111110	3E
⋮	⋮		0	⋮	⋮		⋮
+ 0	0	00000000	0	+ 0	0	0000000	0

Fig. 2.49

2.4.15 Addition et soustraction de nombres négatifs

La représentation des nombres négatifs sous forme complémentaire simplifie considérablement les opérations d'addition et de soustraction. Les opérations signées

impliquent des opérations différentes selon le signe et une détermination délicate du signe du résultat. Les opérations sur les nombres complémentaires sont incomparablement plus simples, mais le problème des dépassements de capacité n'est pas trivial.

Si nous supposons que A et B sont des nombres positifs, trois cas seulement sont à étudier :

- $A + B = A - (-B)$ Somme de deux nombres positifs ou différence entre un nombre positif et un nombre négatif;
- $A + (-B) = A - B$ Somme d'un nombre positif et d'un nombre négatif, ou différence entre deux nombres positifs;
- $(-A) + (-B) = (-A) - B = -(A + B)$ Somme de deux nombres négatifs ou différence entre un nombre négatif et un nombre positif, ou encore opposé de la somme de deux nombres positifs.

Etudions ces cas dans des paragraphes différents, en considérant simultanément les nombres logiques et arithmétiques.

□ 2.4.16 Somme de deux nombres positifs

La somme de deux nombres positifs a été étudiée au paragraphe 2.3.2 dans le cas des nombres logiques. Le dépassement de capacité est dans ce cas caractérisé par le report C de l'opérateur d'addition (fig. 2.50).

Si les nombres sont arithmétiques, le dépassement de capacité est caractérisé par le fait que le chiffre de signe du résultat ne correspond pas au chiffre 0 caractérisant les nombres positifs (fig. 2.52). Les exemples numériques en décimal illustrent les opérations.

La même opération peut être effectuée par un soustracteur opérant sur le complément de B.

En effet :

$$A - (-B) = A - (p^k - B) + p^k = A - B' + p^k \qquad (2.42)$$

Si le résultat est correct, c'est-à-dire s'il est inférieur à p^k, l'opération $A - B'$ crée un emprunt égal à p^k. Ceci est vrai à la fois pour les nombres logiques et pour les nombres arithmétiques (fig. 2.51 et 2.53).

Il ne revient donc pas au même, du point de vue du report C généré par l'opérateur, d'additionner un nombre positif, ou de soustraire son complément vrai.

□ 2.4.17 Somme d'un nombre positif et d'un nombre négatif

Le résultat de la somme d'un nombre positif et d'un nombre négatif peut être soit positif, soit négatif. Les deux cas doivent s'étudier séparément.

- Si le résultat est positif, on peut écrire :

  $$A + (-B) = A + (p^k - B) - p^k = A + B' - p^k \qquad (2.43)$$

  La somme $A + B'$ crée un report de poids p^k, qui ne doit pas être considéré comme un dépassement de capacité.

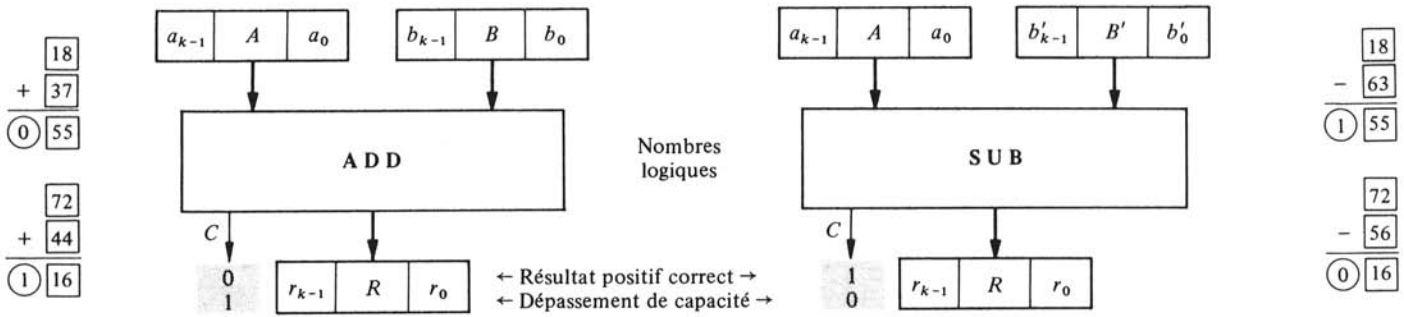


Fig. 2.50

Fig. 2.51

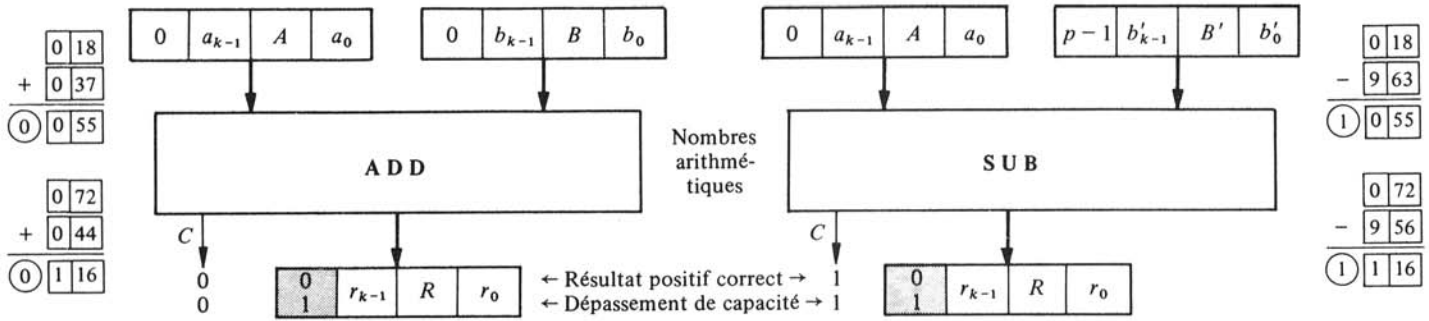


Fig. 2.52

Fig. 2.53

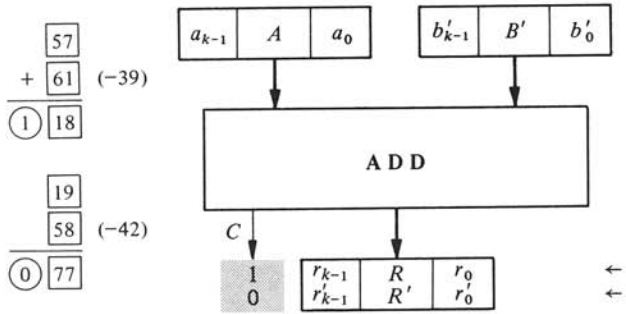


Fig. 2.54

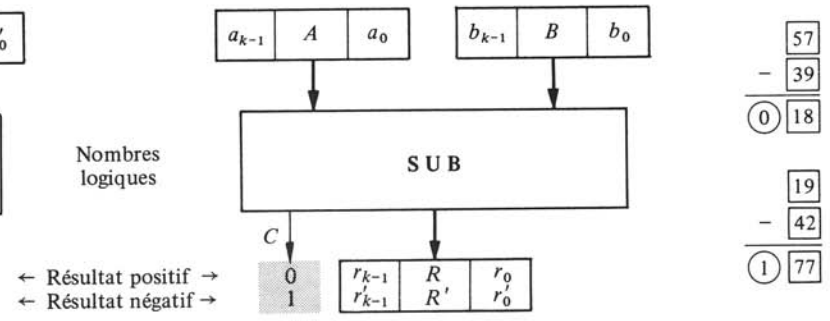


Fig. 2.55

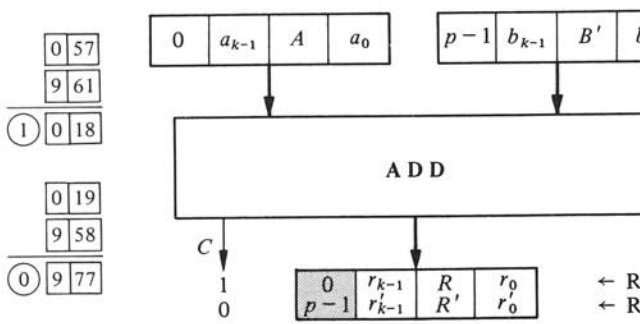


Fig. 2.56

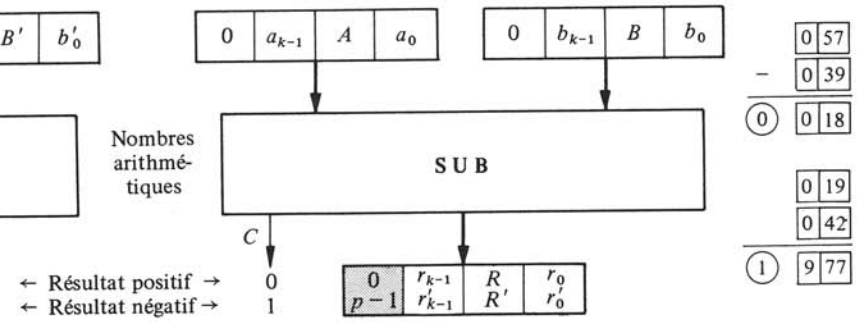


Fig. 2.57

- Si le résultat est négatif, il apparaît naturellement sous forme complémentaire, et il faut écrire :

$$A + (-B) = -(B-A) = p^k - (B-A) = A + (p^k - B) = A + B' \quad (2.44)$$

Dans ce cas, il n'y a pas de report.

La valeur du report C est la même dans le cas des nombres logiques et des nombres arithmétiques. Evidemment, dans le cas des nombres arithmétiques, le chiffre de signe indique le signe du résultat (fig. 2.54 et 2.56).

La même opération, effectuée par un soustracteur, a été étudiée au paragraphe 2.4.2 et conduit au même résultat, sauf en ce qui concerne la valeur finale du report C (fig. 2.55 et 2.57).

□ 2.4.18 Somme de deux nombres négatifs

Pour varier, utilisons la représentation circulaire des nombres logiques et arithmétiques pour justifier les résultats de ce paragraphe, très proches de ceux du paragraphe 2.4.17. Avec des nombres logiques, l'opération $(-A) + (-B)$, effectuée par addition des compléments vrais produit un report si le résultat est correct, mais n'en produit pas s'il y a dépassement de capacité (fig. 2.58 et 2.60).

La même opération avec des nombres arithmétiques produit dans tous les cas un report; le dépassement de capacité se reconnaît à la valeur du chiffre de signe (fig. 2.62 et 2.64).

La même opération peut être effectuée avec un soustracteur en écrivant $-(A) - B$. Les figures 2.59, 2.61, 2.63 et 2.65 montrent l'effet de cette opération sur le report et sur le chiffre de signe.

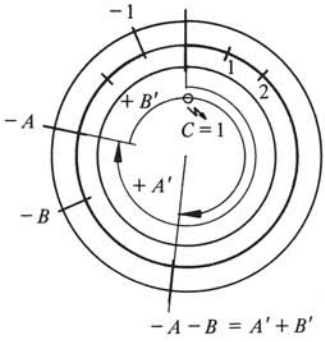


Fig. 2.58

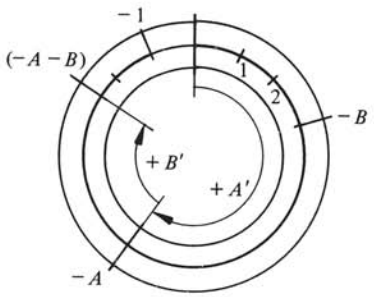


Fig. 2.59

Une dernière façon de procéder revient à écrire $-(A+B)$, c'est-à-dire à calculer la somme et en prendre le complément vrai. Les figures 2.66, 2.67, 2.68 et 2.69 illustrent cette façon de faire pour des nombres logiques et arithmétiques.

2.4.19 Exercice

Effectuer à l'aide des opérateurs NEG et ADD les opérations suivantes sur les mots de 12 bits: $0'1017 - 0'3106$; $-H'37 - H'712$; $H'37F - H'1F7$.

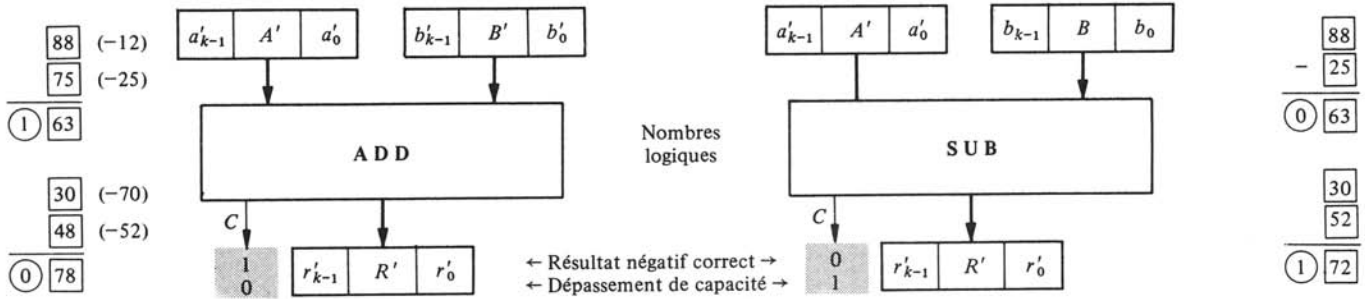


Fig. 2.60

Fig. 2.61

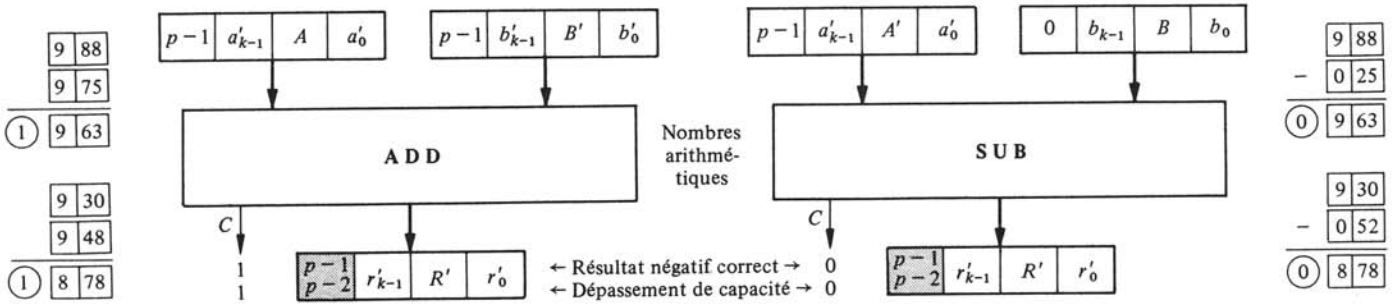


Fig. 2.62

Fig. 2.63

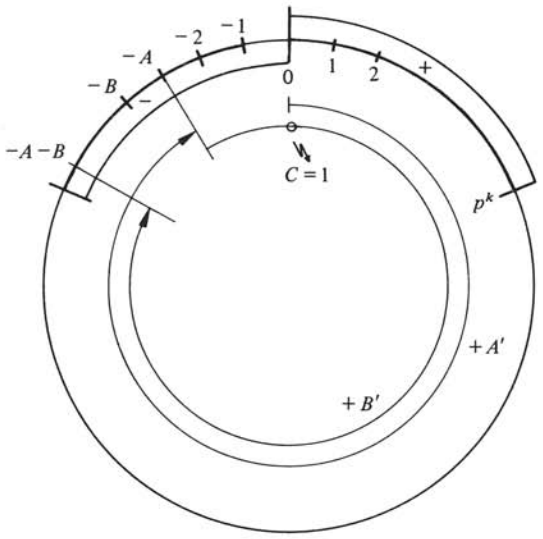


Fig. 2.64

2.4.20 Tests de dépassement de capacité

Les considérations des paragraphes précédents mettent en évidence les problèmes de tests de dépassement de capacité et la différence de traitement des nombres logiques et arithmétiques. Le report C (Carry) détecte tous les cas de dépassement de capacité des opérations sur des nombres logiques. Le dépassement de capacité des opérations sur des nombres arithmétiques peut être caractérisé par un indicateur dit de *dépassement V* (*oVerflow*), qui dépend du chiffre de signe des opérandes et du résultat. Deux informations supplémentaires sont souvent utiles après une addition ou soustraction : le *signe S* du résultat et la *nullité Z* (zéro) du résultat.

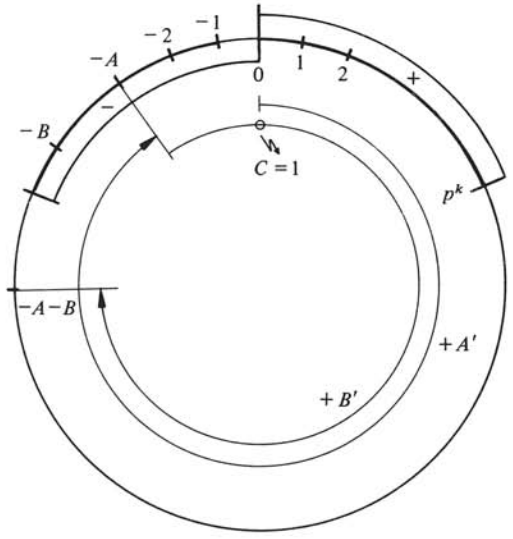


Fig. 2.65

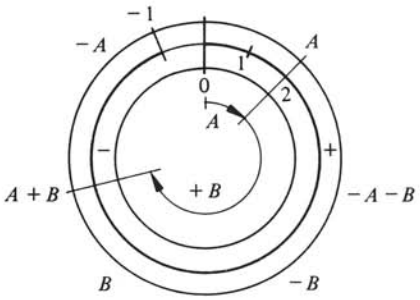


Fig. 2.66

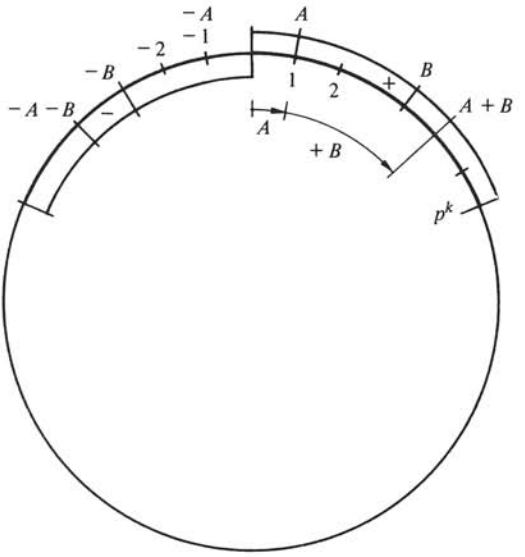


Fig. 2.67

Les valeurs booléennes C, S, Z, V sont appelées *indicateurs ou fanions* (*flag*).

On a $S = 0$ si le résultat est positif (chiffre de signe = 0) et $S = 1$ si le résultat est négatif (chiffre de signe = $p - 1$). La nullité Z est égale à 1 si le résultat est égal à zéro (*EQual*), et on a $Z = 0$ si le résultat est différent de zéro (*NE Non Equal*).

Le symbole de l'opérateur complet d'addition et de soustraction est représenté dans la figure 2.70. Un tel opérateur est utilisable pour des nombres logiques, mais dans ce cas les indicateurs V (overflow) et S (signe) n'ont pas de signification.

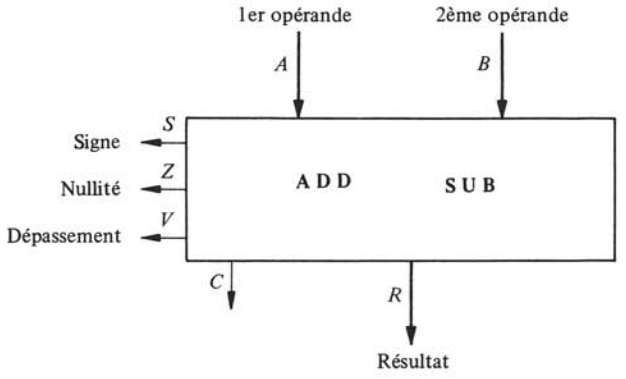


Fig. 2.70

2.4.21 Addition et soustraction binaires

L'addition et la soustraction binaires, décomposées ou non, s'effectuent de la même façon sur des nombres logiques ou arithmétiques. L'indicateur de dépassement est toutefois différent, comme cela a été montré au paragraphe 2.4.20. Le report C (carry) indique un dépassement en cas d'opération sur des nombres logiques. L'indicateur de dépassement

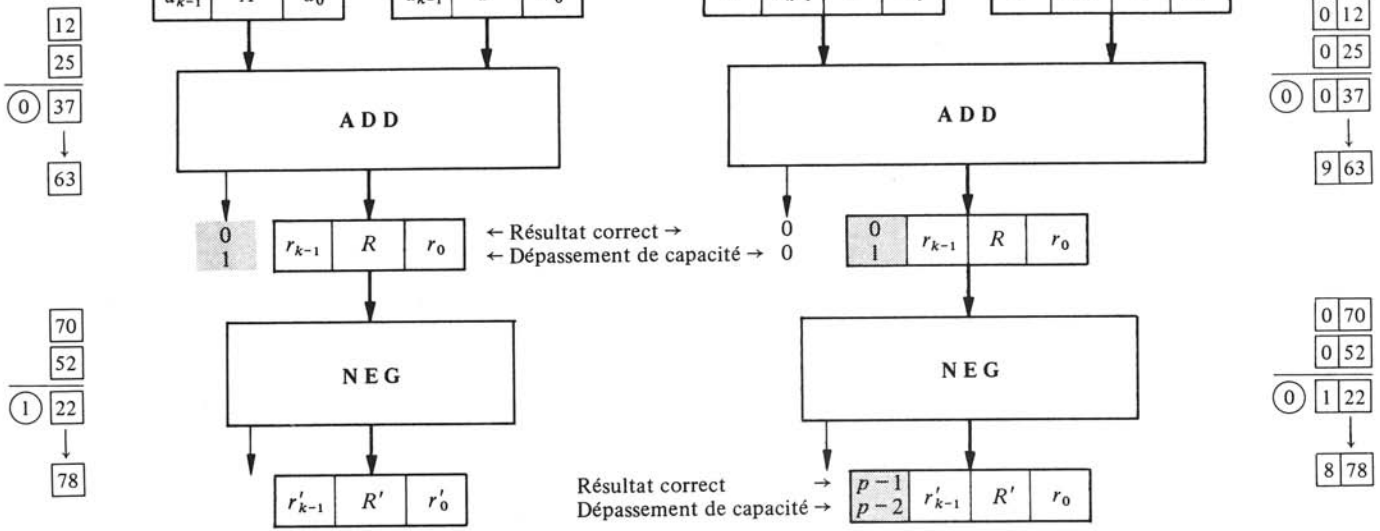


Fig. 2.68

Fig. 2.69

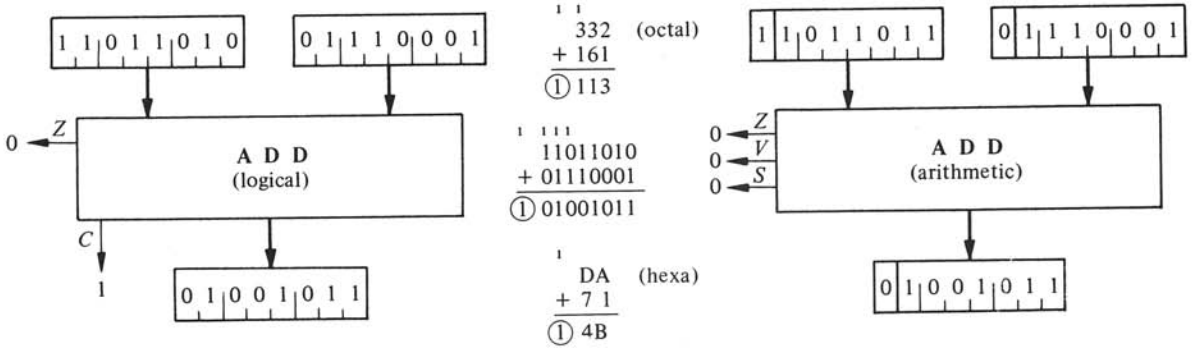


Fig. 2.71

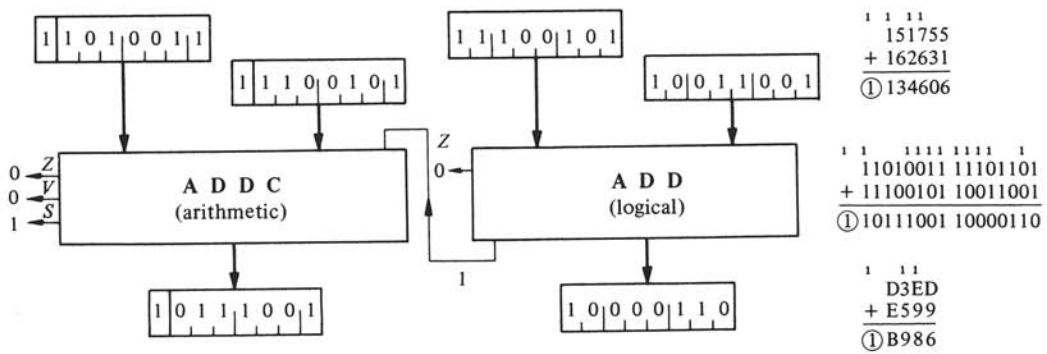


Fig. 2.72

V (overflow) doit par contre être utilisé pour les nombres arithmétiques. Une caractéristique du système binaire est que l'indicateur de signe S est identique au bit de poids fort (bit de signe).

Les figures 2.71 et 2.72 donnent quelques exemples types d'opérations et mettent en évidence l'application des considérations des paragraphes 2.4.15 à 2.4.20 au binaire.

La figure 2.71 montre l'addition de 2 nombres de 8 bits considérés comme nombres logiques, puis comme nombres arithmétiques. Dans le premier cas, il y a dépassement de capacité car $C = 1$. Dans le second, le résultat est correct ($V = 0$) car la somme d'un nombre positif et d'un nombre négatif ne crée jamais de dépassement de capacité. On peut remarquer dans cet exemple, que si l'opération est effectuée en octal, elle doit se calculer modulo 400. Un report est créé parce que la somme des chiffres de poids fort vaut $1 + 3 + 1 = 5 = 4 + 1$.

2.4.22 Extension du signe

Il arrive souvent qu'un nombre en format fixe doive changer de format. Pour un nombre entier positif, il suffit d'ajouter ou d'enlever des zéros non significatifs; toute suppression d'un chiffre non nul entraîne naturellement une erreur qui doit être signalée.

Pour étudier le cas des nombres négatifs, partons de la relation (2.37) liant un nombre négatif et son complément. Si le champ doit être étendu de x positions, écrivons:

$$-A = -p^k + A' = -p^{k+x} + (p-1)\cdot p^{k+x-1} + \ldots + (p-1)\cdot p^k + A'$$

$$= -p^{k+x} + \sum_{j=1}^{x} [(p-1)\, p^{k+j-1}] + A' \qquad (2.45)$$

Pour qu'il y ait équivalence entre le nombre et le nombre étendu, il faut donc que les chiffres ajoutés, de poids p^k, ... p^{k+x-1}, soient égaux à $p-1$, c'est-à-dire au chiffre de signe.

Cette démonstration est vraie pour les nombres logiques et arithmétiques. Pour les nombres arithmétiques, la règle d'extension n'a pas besoin de distinguer les nombres positifs et négatifs: il suffit de recopier le chiffre de signe, et l'opérateur correspondant (fig. 2.73) se note *SEX* (*sign extension*).

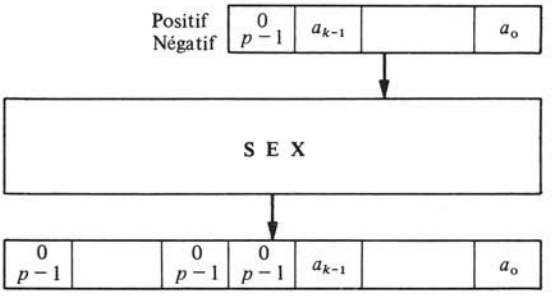


Fig. 2.73

2.4.23 Opérations sur les nombres de longueurs différentes

Lorsque deux nombres arithmétiques occupant des champs de longueurs différentes doivent être additionnés ou soustraits, une extension du signe du nombre occupant le

champ le plus court est nécessaire. La figure 2.74 montre une opération type fréquente avec les microprocesseurs: l'addition à une adresse 16 bits d'un déplacement arithmétique de 8 bits. L'adresse est un nombre logique et le déplacement donné en exemple (372) est égal à −6. L'additionneur crée un report $C = 1$, sans signification. Cet exemple montre une façon de travailler avec des nombres à la fois logiques et arithmétiques, compatibles avec le contexte de l'application.

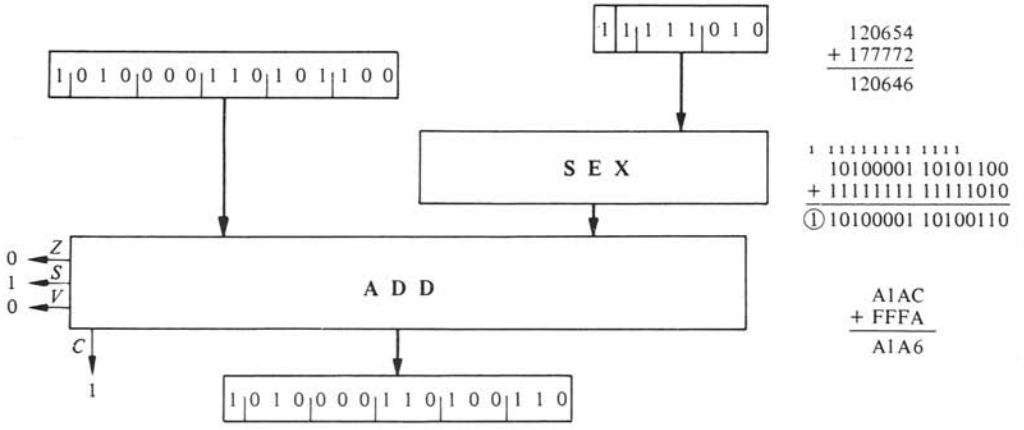


Fig. 2.74

2.4.24 Complément restreint

Le complément vrai d'un nombre est relativement difficile à calculer et différentes techniques permettent de simplifier l'opérateur *NEG* défini au paragraphe 2.4.8.

Partons à nouveau de l'expression (2.38) vue au paragraphe 2.4.8, dans laquelle B est négatif:

$$B' = p^k - B = (p^k - 1) - B + 1$$

En décomposant $p^k - 1$, on peut écrire:

$$\begin{aligned} B' &= ((p-1)\cdot p^{k-1} + \ldots + (p-1)) - (l_{k-1}\cdot p^{k-1} + \ldots + l_1\cdot p + l_0) + 1 \\ &= (p-1-l_{k-1})\cdot p^{k-1} + \ldots + (p-1-l_1)\cdot p + (p-1-l_0) + 1 = B'' + 1 \end{aligned} \tag{2.46}$$

B'' est un nombre de longueur k, car $0 \leqslant p-1-l_i < p$.

$B'' = (p^k - 1) - B$ est appelé *complément restreint* de B. C'est une expression facile à calculer car il suffit de faire la différence à $p-1$ de chaque chiffre. Cette différence est un chiffre et ne crée jamais d'emprunt.

L'opérateur complément restreint est abrégé par les lettres *NOT*, parfois par les lettres CPL. En cas de décomposition spatiale ou temporelle, il n'y a pas de report (fig. 2.75).

Le complément restreint, de par les relations très simples qui le lient au complément vrai, est souvent utilisé comme étape intermédiaire pour calculer ce complément vrai.

Les relations $B' = B'' + 1$, $B' = (B-1)''$ (cf. exercice 2.4.25) se laissent représenter par les schémas fonctionnels de la figure 2.76.

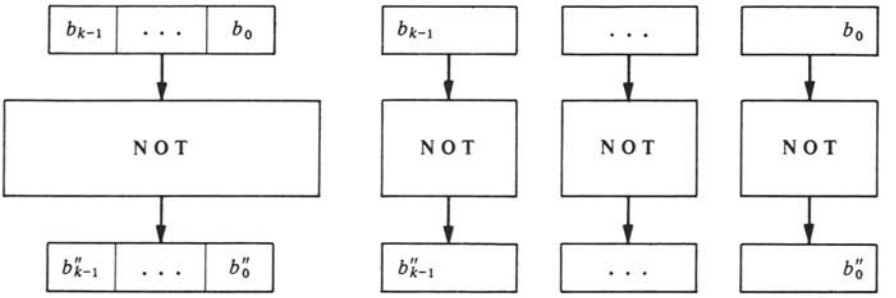


Fig. 2.75

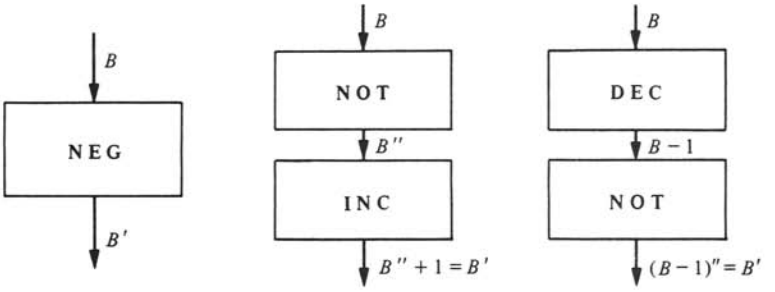


Fig. 2.76

L'opérateur de *décrémentation* (soustraction de un) noté DEC, utilisé dans la figure 2.76 se définit de façon analogue à l'incrémentation (§ 2.3.13).

2.4.25 Exercice

Justifier algébriquement que $B' = (B - 1)''$.

2.4.26 Complément à 1

Le complément restreint en binaire s'appelle *complément à 1* et est particulièrement simple à calculer. Il faut calculer la différence de chaque bit à $2 - 1 = 1$ ce qui revient simplement à changer les 1 en 0 et les 0 en 1. Cette opération est une simple *inversion* des bits facile à effectuer avec des portes NON (§ V 1.2.2). Le complément à 2 est égal au complément à 1 augmenté de 1. Cette propriété est généralement utilisée pour calculer le complément à 2, la correction de 1 étant apportée au niveau de l'additionneur.

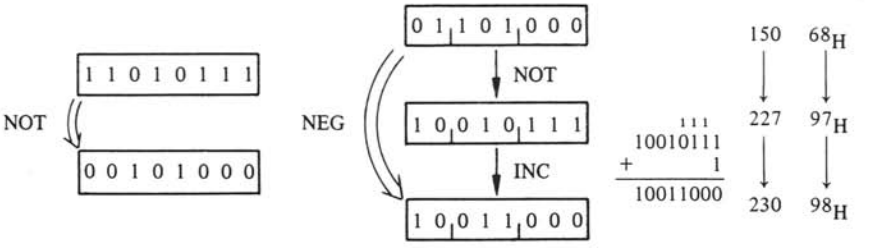


Fig. 2.77

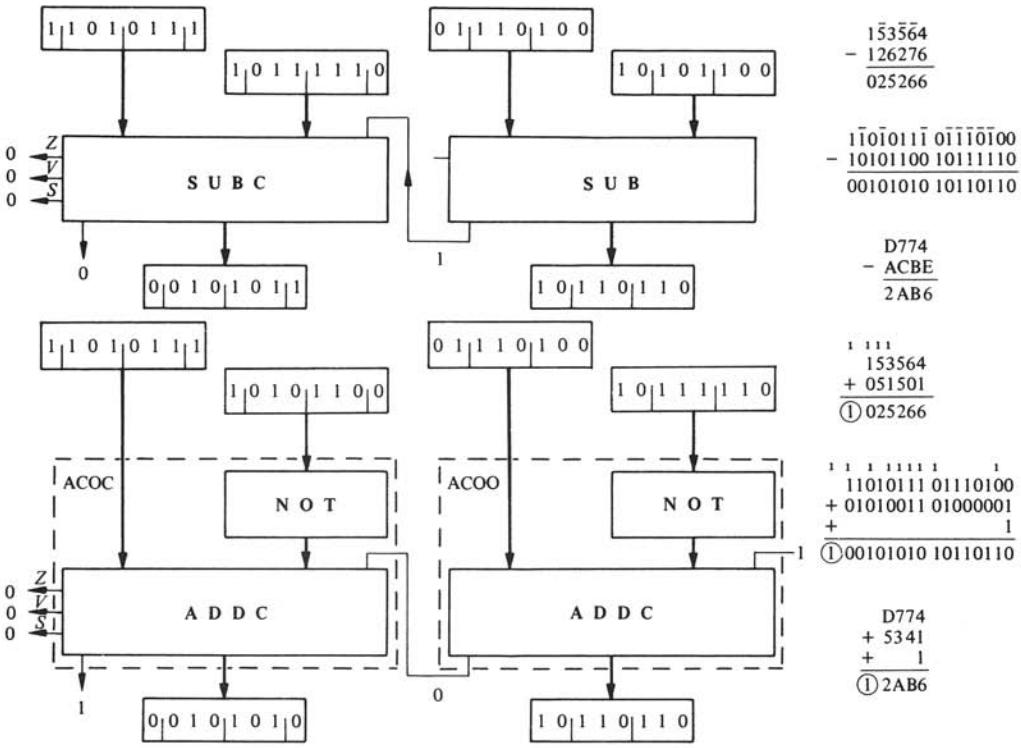


Fig. 2.78

La figure 2.77 et la figure 2.78 rappellent ces propriétés et montrent en particulier que l'addition du complément génère un report inverse de celui produit par la soustraction.

L'opération effectuée dans la figure 2.78 est soit une soustraction de deux nombres logiques de 16 bits, donnant un résultat 16 bits correct, soit une soustraction de deux nombres arithmétiques négatifs (15 bits plus signe), dont le résultat est positif.

La soustraction en complément vrai est toutefois souvent implémentée avec des opérateurs en complément restreint, lorsque la simplicité de ceux-ci est plus grande.

Ces opérateurs sont abrégés *ACO* (*add complement*) (fig. 2.79), *ACOC* (*add complement plus carry*) (fig. 2.80) et *ACOO* (*add complement plus one*). L'opérateur ACOO a déjà été rencontré au paragraphe 2.4.9, où la différence entre ACOO et SUB a été mise en évidence (fig. 2.81).

2.4.27 Exercice

Définir les règles d'addition et de soustraction en complément à un, c'est-à-dire avec les nombres négatifs représentés sous forme de complément à 1. Etablir en particulier que l'opérateur d'addition en complément à 1 est un additionneur dont l'entrée de report c_0 est liée à la sortie du report c_k.

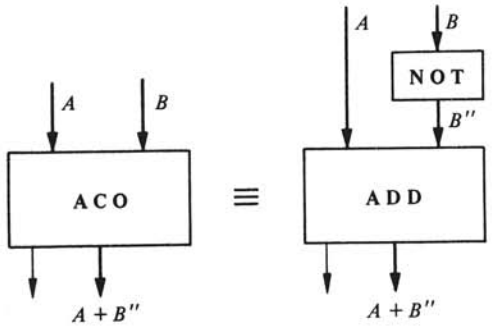


Fig. 2.79

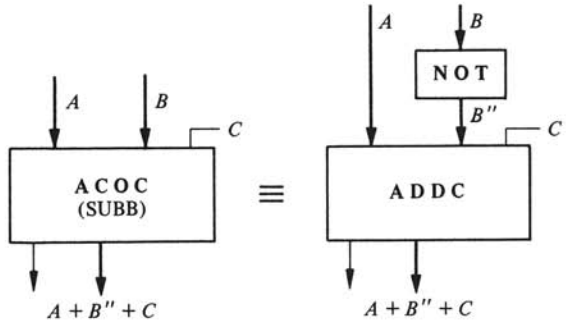


Fig. 2.80

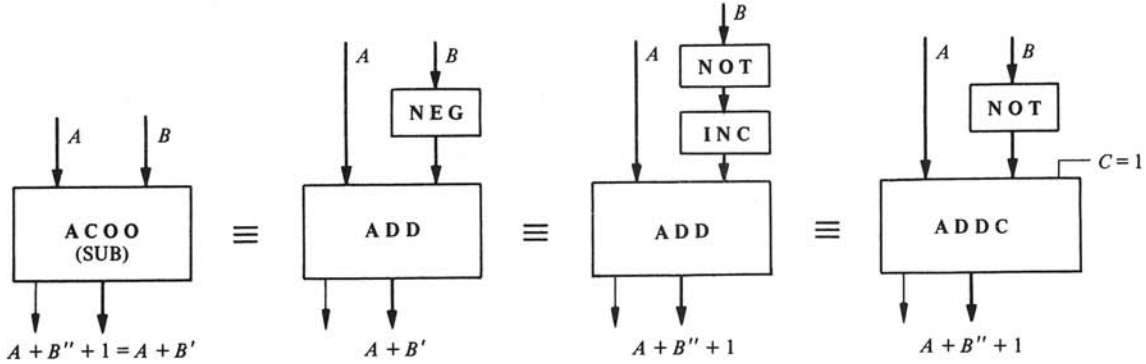


Fig. 2.81

2.4.28 Opérations en complément restreint

Le complément restreint peut être utilisé pour la représentation des nombres négatifs, à la place du complément vrai. Cette représentation n'étant pas naturelle, des corrections d'une unité doivent parfois être effectuées sur les résultats. Une étude détaillée sort du cadre de cet ouvrage.

Remarquons toutefois que plusieurs ordinateurs anciens (par exemple le CDC 6600) utilisent une représentation des nombres négatifs sous forme de complément à 1, étant donné quelques légers avantages en complexité et vitesse au niveau de l'unité arithmétique.

Vu de l'utilisateur, l'emploi du complément à 1 se remarque par la double représentation du zéro, qui peut être codé avec tous les bits à zéro (zéro positif) ou en complément avec tous les bits à 1 (zéro négatif). Cette double représentation ne gêne pas pour les opérations internes, mais est mise en évidence lors de l'impression des résultats.

2.5 COMPARAISONS ET DÉCALAGES

■ 2.5.1 Opérateur de comparaison

La comparaison de 2 nombres permet de savoir s'ils sont égaux, si le premier est inférieur à l'autre, ou si le premier est supérieur à l'autre.

Ces propriétés sont des variables binaires vraies ou fausses et l'opérateur de comparaison *COMP*, représenté dans la figure 2.82, rappelle les signes utilisés lorsque ces trois propriétés, plus trois autres propriétés similaires bien connues, sont vraies.

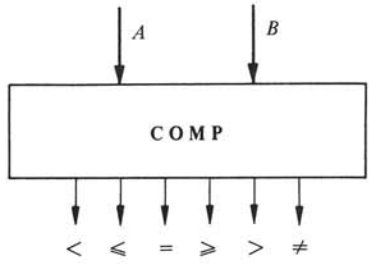


Fig. 2.82

La représentation des nombres en format fixe, et l'utilisation habituelle d'un soustracteur pour effectuer les comparaisons, obligent à étudier séparément la comparaison des nombres logiques et arithmétiques. Dans le cas des nombres entiers positifs, deux algorithmes différents sont utilisés pour comparer deux nombres en format fixe.

Ecrivons les expressions générales des 2 nombres sous la forme :

$$\begin{aligned} A &= a_{k-1} \cdot p^{k-1} + A^* \qquad A^* < p^{k-1} \\ B &= b_{k-1} \cdot p^{k-1} + B^* \qquad B^* < p^{k-1} \end{aligned} \tag{2.47}$$

Si $a_{k-1} > b_{k-1}$, alors $A > B$.

En effet, $a_{k-1} > b_{k-1}$ entraîne $a_{k-1} - b_{k-1} \geqslant 1$ car les chiffres a_{k-1} et b_{k-1} sont des nombres entiers. On a donc, étant donné que $|A^* - B^*| < p^{k-1}$.

$$A - B = (a_{k-1} - b_{k-1})\, p^{k-1} + (A^* - B^*) \geqslant p^{k-1} + (A^* - B^*) > 0 \tag{2.48}$$

De même, si $a_{k-1} < b_{k-1}$, $A < B$.

Si $a_{k-1} = b_{k-1}$, on ne peut par contre pas déduire que $A = B$. Dans ce cas, $A - B = A^* - B^*$ et le raisonnement précédent se poursuit par récurrence sur A^* et B^*.

Cet algorithme de comparaison par les poids forts est bien connu; rappelons qu'il n'est valable que pour des nombres positifs ou logiques.

Pour des nombres arithmétiques, le chiffre de signe du résultat de la soustraction fournit avec l'indicateur de nullité les informations de comparaison nécessaires:

Si $A - B$ est positif, alors $A \geqslant B$

Si $A - B$ est négatif alors $A < B$

Les problèmes de dépassement de capacité en champ fixe compliquent cette règle (§ 2.4.16).

2.5.2 Comparaison des nombres positifs

Considérons le cas de 2 nombres positifs ou logiques. La figure 2.83 rappelle la structure de ces nombres et définit les lettres usuelles caractérisant les 6 types de comparaisons concernant des nombres logiques.

• *LO*	(Lower)	$A < B$ (strictement inférieur)	Nombres logiques
• *LS*	(Lower or same)	$A \leqslant B$ (inférieur ou égal)	
• EQ	(Equal)	$A = B$ (égal)	
• *HS*	(Higher or same)	$A \geqslant B$ (supérieur ou égal)	
• *HI*	(Higher)	$A > B$ (strictement supérieur)	
• NE	(Not equal)	$A \neq B$ (différent)	

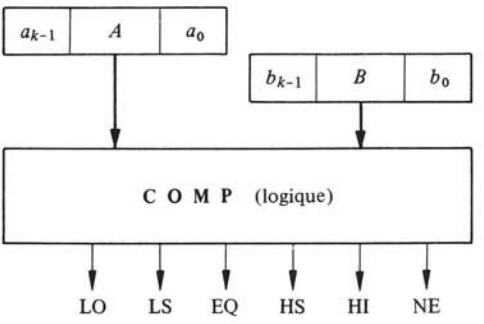


Fig. 2.83

Si un soustracteur est utilisé pour effectuer la comparaison, les valeurs du report C et de l'indicateur de nullité du résultat Z déterminent entièrement les différents cas de comparaison (fig. 2.84). Par exemple, si $C = 1$, il y a emprunt dans l'opération de soustraction $A - B$, donc $A < B$ (§ 2.4.17). Si $C = 0$, on a $A \geqslant B$. La condition $A > B$ est vraie lorsque la condition $A \leqslant B$ est fausse, c'est-à-dire lorsque $C = 1$ ou $Z = 1$. Ceci est noté $C + Z = 1$, selon les conventions utilisées dans le Traité d'Electricité (§ V. 1.3.4). Le contexte permettra en général d'éviter la confusion entre le + de l'addition et le + du ou logique. Un *v* prononcé vel est utilisé à la place du + dans de nombreux ouvrages pour éviter cette confusion.

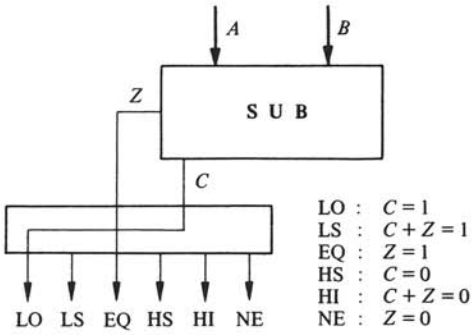


Fig. 2.84

2.5.3 Comparaison des nombres arithmétiques

Le schéma fonctionnel de la comparaison de deux nombres arithmétiques est donné dans la figure 2.85. On retrouve les mêmes types de comparaisons, mais les inégalités sont désignées par des lettres différentes, pour rappeler la nature différente des nombres comparés.

• *LT*	(Lower than)	$A < B$	Nombres arithmétiques
• *LE*	(Lower or Equal)	$A \leqslant B$	
• EQ	(Equal)	$A = B$	
• *GE*	(Greater or equal)	$A \geqslant B$	
• *GT*	(Greater than)	$A > B$	
• NE	(Not equal)	$A \neq B$	

La distinction entre comparaison de nombres logiques et arithmétiques est importante, car souvent les deux types de nombres sont mélangés dans des opérations, et le même soustracteur est utilisé dans chaque cas.

Dans le cas des nombres arithmétiques, le report C n'a pas d'influence pour la comparaison. Il suffit de considérer les figures 2.53 et 2.65 pour s'en convaincre : la valeur de C ne change pas selon que le résultat est positif ou négatif. Par contre, les indicateurs de dépassement V (overflow) et de signe S interviennent de façon combinée.

Démontrons la relation qui caractérise l'inégalité stricte par épuisement des cas (fig. 2.86).

Quatre cas sont à considérer, comme dans l'étude de la soustraction (§ 2.4.16 à 2.4.18).

Dans tous les cas où $A \geqslant B$, le ou exclusif $S \oplus V$ de S et V est égal à 0. Si $A < B$, $S \oplus V = 1$. Toutefois, ceci n'est valable qu'avec une convention d'alternance des signes sur le cercle des nombres arithmétiques. En base 3, cette convention ne peut pas être respectée.

La figure 2.85 résume la valeur des différentes conditions d'inégalité, en fonction de Z, S et V.

2.5.4 Décomposition spatiale de la comparaison

Si deux nombres doivent être fractionnés en plusieurs morceaux pour pouvoir être comparés, des regroupements dans l'expression générale de ces nombres (2.2) permettent

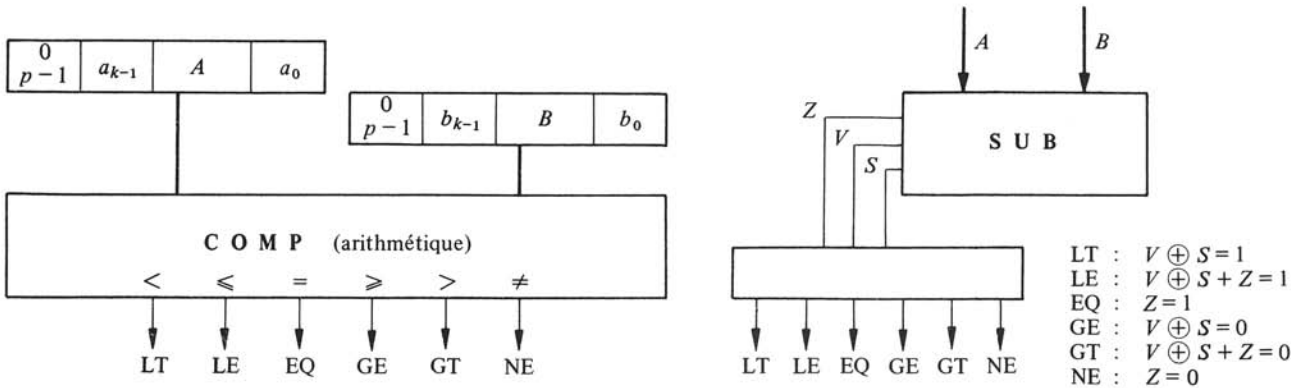

0
p − 1
a_{k-1}
A
a_0
0
p − 1
b_{k-1}
B
b_0
COMP (arithmétique)
< ⩽ = ⩾ > ≠
LT LE EQ GE GT NE
A
B
Z
V
S
SUB
LT LE EQ GE GT NE
LT : V ⊕ S = 1
LE : V ⊕ S + Z = 1
EQ : Z = 1
GE : V ⊕ S = 0
GT : V ⊕ S + Z = 0
NE : Z = 0

Fig. 2.85

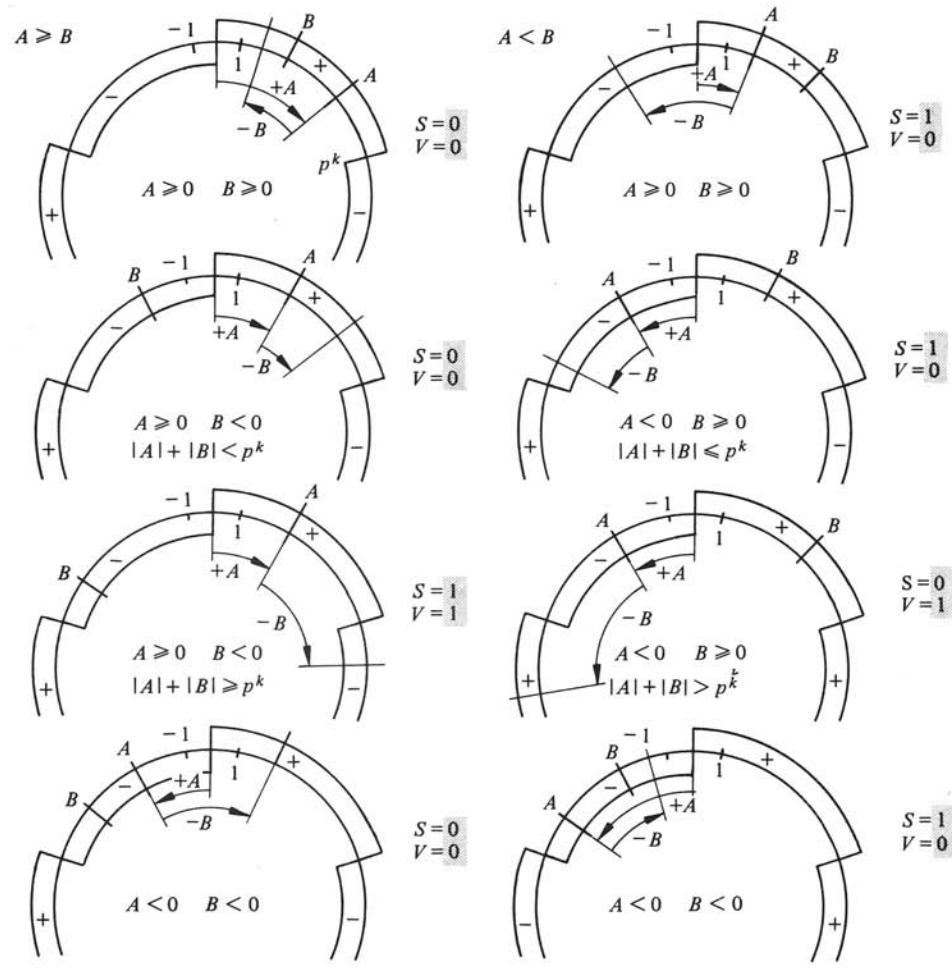


Fig. 2.86

d'écrire :

$$A = A_x \cdot p^x + A_y \cdot p^y + A_z$$
$$B = B_x \cdot p^x + B_y \cdot p^y + B_z \qquad (2.49)$$

Une démonstration analogue à celle du paragraphe 1.5.1 montre que si $A_x < B_x$, alors $A < B$. Si $A_x = B_x$, alors $A < B$ si $A_y < B_y$. La comparaison de A_z et B_z n'est nécessaire que si $A_x = B_x$ et $A_y = B_y$. L'ensemble de ces conditions logiques peut s'exprimer par un logigramme (§ V.1.3.3), représenté dans le schéma fonctionnel de la figure 2.87.

La même opération, utilisant des soustracteurs, conduit à la figure 2.88. L'opérateur de poids fort fournit toute l'information en cas d'inégalité stricte. La détection d'égalité nécessite la connaissance des indicateurs de nullité de chacun des opérateurs de soustraction participant à l'opération.

La comparaison des nombres signés impose l'emploi de soustracteurs. bien que l'on puisse également traiter séparément le chiffre de signe (exercice 2.5.6). La figure 2.89 précise ce cas, très semblable au cas précédent.

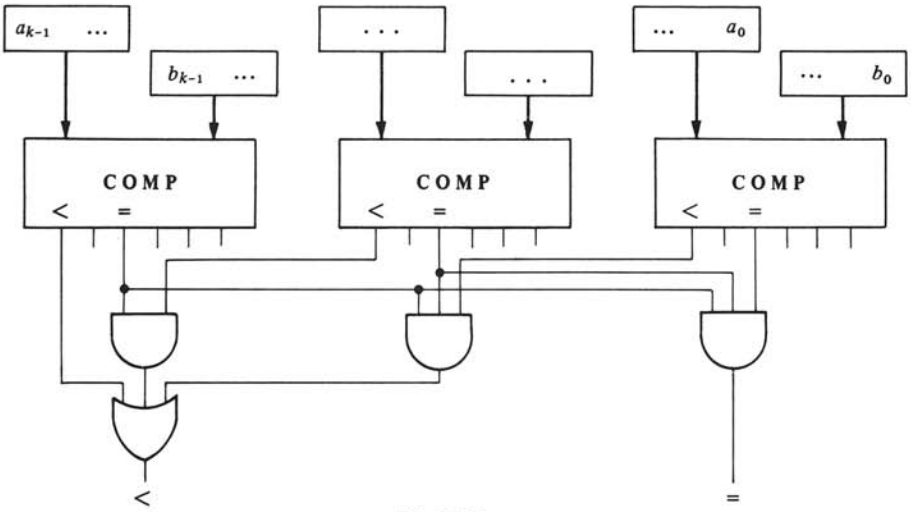


Fig. 2.87

2.5.5 Exemple pratique

Pour le lecteur familier avec les processeurs de la famille M6800 [53], la comparaison de deux nombres logiques de 24 bits en mémoire (fig. 2.90) peut s'écrire en transposant la figure 2.87 en comparant d'abord les poids forts.

```
LOAD        A, NB1 + 2
COMP        A, NB2 + 2
JUMP, LO    PLUSPETIT
JUMP, NE    PLUSGRAND
LOAD        A, NB1 + 1
COMP        A, NB2 + 1
JUMP, LO    PLUSPETIT
JUMP, NE    PLUSGRAND
LOAD        A, NB1
COMP        A, NB2
JUMP, LO    PLUSPETIT
JUMP, NE    PLUSGRAND
JUMP, EQ    EGAL
```

Il est plus simple de partir de la figure 2.88 en traitant d'abord les poids faibles.

```
LOAD        A, NB1
SUB         A, NB2
LOAD        A, NB1 + 1
SUBC        A, NB2 + 1
LOAD        A, NB1 + 2
SUBC        A, NB2 + 2
JUMP, LO    PLUPETIT
JUMP, NE    PLUSGRAND
JUMP, EQ    EGAL
```

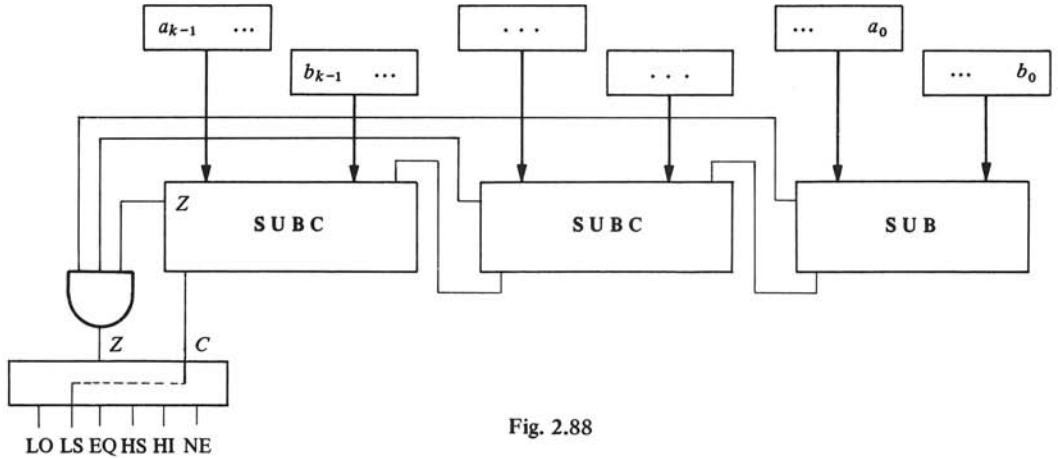


Fig. 2.88

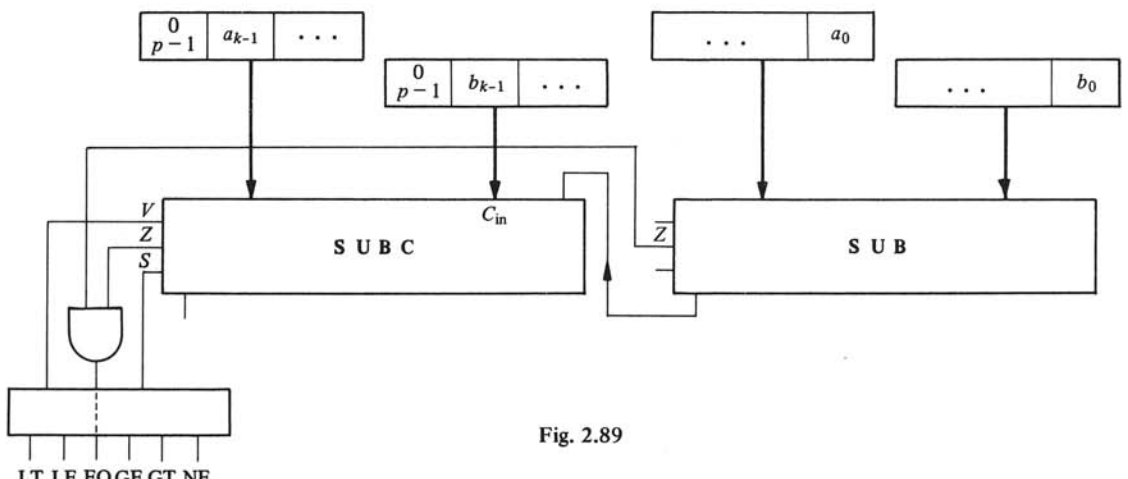


Fig. 2.89

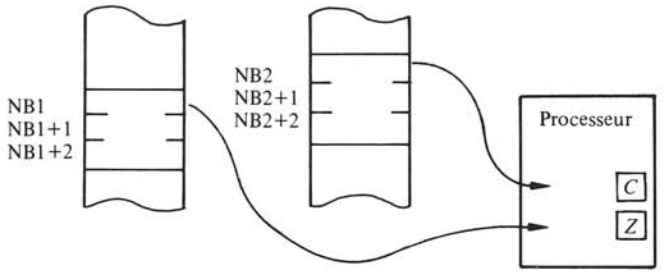


Fig. 2.90

2.5.6 Exercice

Etudier un algorithme de comparaison de 2 nombres arithmétiques dans lequel on compare tout d'abord les 2 chiffres de signe, puis, si c'est nécessaire, les 2 nombres logiques qui suivent.

2.5.7 Décalage à gauche

La multiplication d'un nombre entier positif par la base p revient à *décaler* (*shift*) ce nombre d'une position vers la gauche. Si:

$$A = a_{k-1} \cdot p^{k-1} + \ldots + a_1 \cdot p + a_0 = a_{k-1} \ldots a_1 a_0 \qquad (2.50)$$

alors

$$A \cdot p = a_{k-1} \cdot p^k + \ldots + a_1 \cdot p^2 + a_0 p = a_{k-1} a_{k-2} \ldots a_0 0 \qquad (2.51)$$

En champ fixe, le décalage entraîne la perte du chiffre de poids fort, sans conséquence si ce chiffre est un zéro non significatif. L'opérateur de décalage à gauche, appelé *SL* (*shift left*) est représenté dans la figure 2.91. Si le nombre décalé est un nombre arithmétique, le résultat n'est correct que si l'information de signe subsiste après décalage. Une démonstration inverse de celle qui a été faite au paragraphe 2.4.20 montre que si a_{k-1} n'est pas égal au chiffre de signe, le résultat est incorrect et on parle à nouveau de dépassement de capacité. L'opérateur SL (fig. 2.92) peut, comme l'additionneur, générer une variable V (overflow), ainsi que S (signe du résultat) et Z (résultat égal à zéro).

En toute rigueur, on pourrait distinguer l'opérateur *SL* de l'opérateur *ASL* (*arithmetic shift left*), mais l'opération est la même, à part la génération de V.

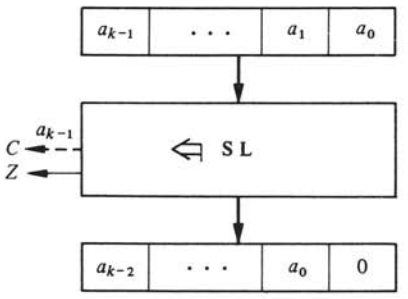


Fig. 2.91

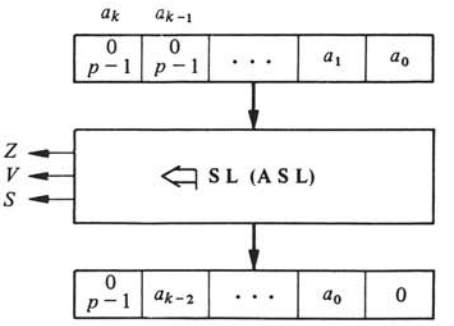


Fig. 2.92

2.5.8 Décalage à droite

La division d'un nombre entier par la base p décale ce nombre à droite. L'opérateur n'est toutefois pas le même pour les nombres logiques (positifs) et pour les nombres arithmétiques.

Pour les nombres logiques, un zéro non significatif doit remplacer le chiffre de poids fort qui s'est décalé. L'opérateur *SR* (*shift right*) effectue cette opération (fig. 2.93).

Pour les nombres arithmétiques, le chiffre de signe doit être recopié en première position. La démonstration de ce fait, similaire à celle du paragraphe 2.4.22, est laissée en exercice. L'opérateur est différent du précédent et s'appelle *ASR* (*arithmétic shift right*). L'opération ne peut pas créer de dépassement; l'indicateur V n'est jamais vrai, mais il peut être utile de connaître S et Z (fig. 2.94).

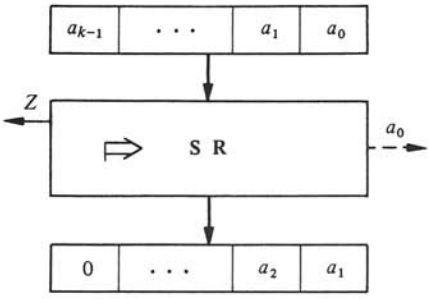


Fig. 2.93

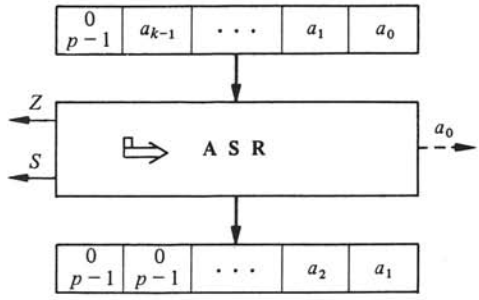


Fig. 2.94

2.5.9 Décomposition des opérations de décalage

L'opérateur de décalage à gauche SL introduit un zéro à la place du chiffre de poids faible. Dans le cas général, pour permettre la multiprécision, un chiffre quelconque doit pouvoir être transféré dans cette position, et le chiffre de poids fort a_{k-1} doit pouvoir être transmis à un autre opérateur. L'opérateur correspondant s'appelle *SLL* (*shift left link*) (fig. 2.95) et permet une décomposition spatiale de l'opérateur SL (fig. 2.96).

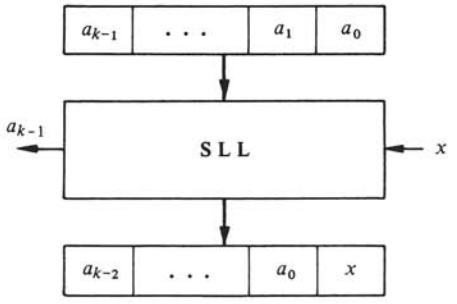


Fig. 2.95

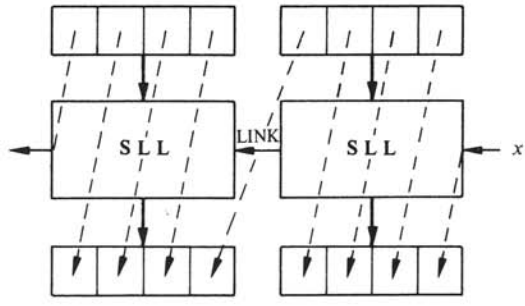


Fig. 2.96

On peut définir de même un opérateur de décalage à droite *SRL*. La figure 2.97 montre comment cet opérateur s'utilise pour étendre la précision de l'opérateur ASR.

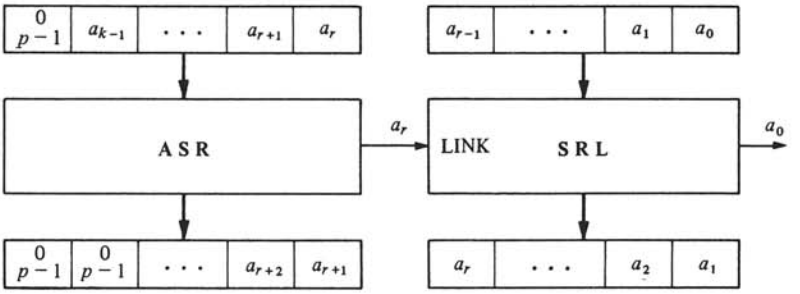


Fig. 2.97

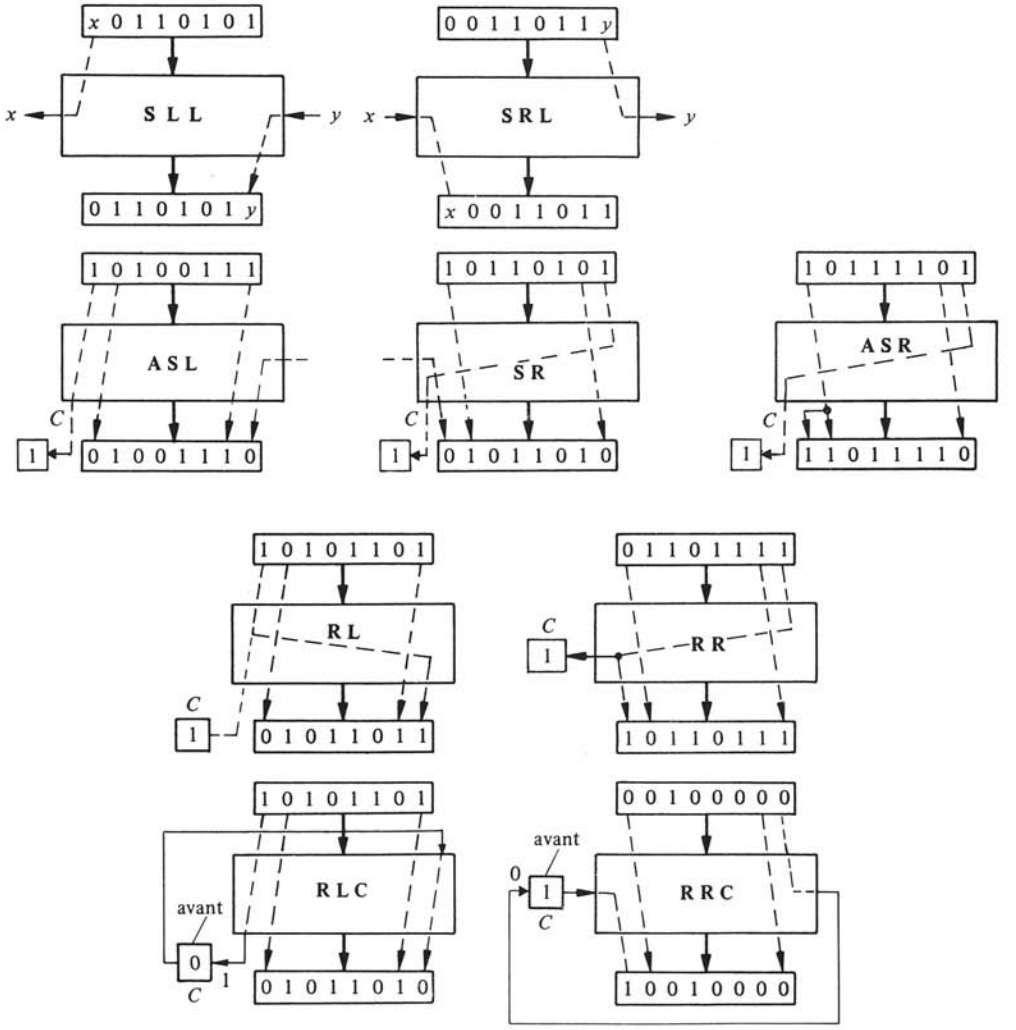


Fig. 2.98

2.5.10 Décalage en binaire

Dans le cas du binaire, le *lien L* (*link*) nécessaire au transfert des chiffres d'extrémité d'un opérateur à l'autre est un simple bit. La plupart des processeurs identifient ce bit de lien *L* avec le report *C* (carry). Les opérateurs les plus courants sont représentés dans la figure 2.98 qui tient de définition pour les opérateurs ASL, SR, ASR, *RL* (*Rotate Left*), *RR* (*Rotate Right*), RLC (Rotate Left through Carry), RRC (Rotate Right through Carry).

Les opérateurs SLL, SRL sont rarement utilisés dans les mini et microprocesseurs. Les opérations qu'ils effectuent (fig. 2.99), dans le cas usuel de la décomposition tempo-

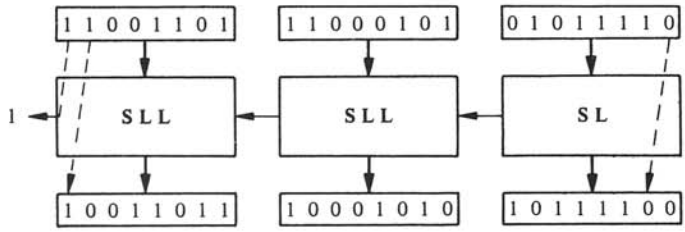


Fig. 2.99

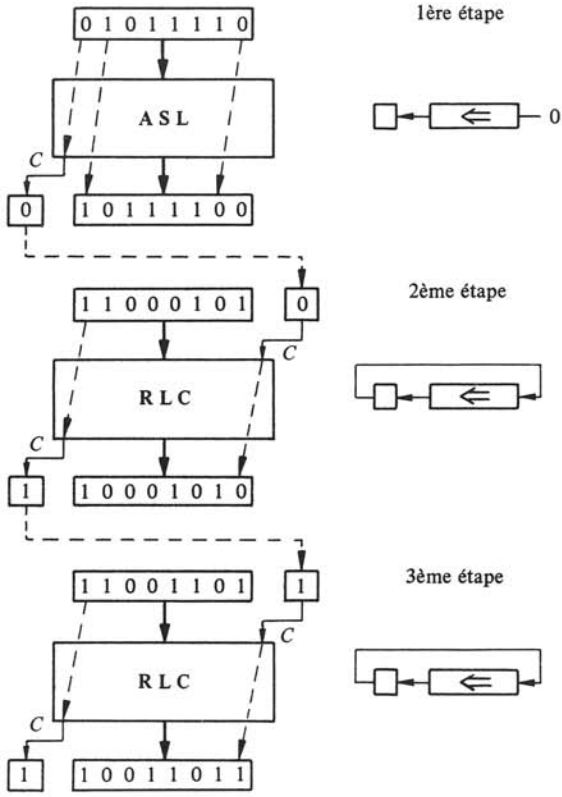


Fig. 2.100

relle, sont effectuées par les opérateurs RLC et RRC, la cellule C mémorisant temporairement les bits transférés (fig. 2.100).

A noter qu'avec le processeur M68000 [54], le concept d'un bit de lien séparé du report a été retenu, mais le bit de lien s'appelle C et le report X. Les opérations d'addition et soustraction modifient C et X. Les opérations de décalage et les opérations de transfert ne modifient que C.

2.5.11 Exemple d'application

Le processeur 8085 n'a que 4 instructions de décalage:

```
RL      A
RLC     A
RR      A
RRC     A
```

Pour exécuter l'instruction SR, il faut utiliser RRC en ayant préalablement mis le Carry à zéro avec une instruction à disposition.

On écrira par exemple à la place de SR A

```
ADD     A,≠0  ; ajoute la valeur 0 et force Carry = 0, donc ne modifie pas A.
RRC     A
```

L'instruction ASR A peut se réaliser de façon assez simple et astucieuse par les ins-

```
RL      A     ; copie le bit de poids fort dans C.
RRC     A
RRC     A
```

2.5.12 Opérateur de sélection

L'*opérateur de sélection* ou *multiplexeur* (*multiplexer*) permet de choisir l'un des 2 nombres logiques ou arithmétiques selon la valeur d'une variable booléenne (fig. 2.101). Cet opérateur est désigné par les lettres *CASE*. La variable de contrôle est implicite ou explicite. Le plus souvent il s'agit d'un indicateur de comparaison ou de dépassement de capacité.

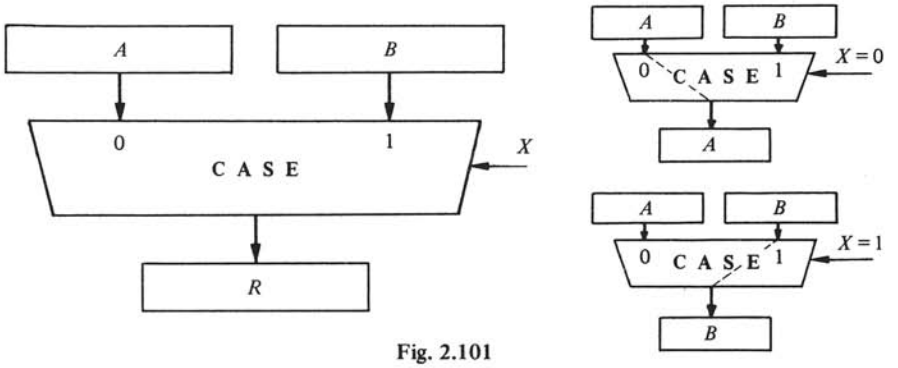


Fig. 2.101

La notion d'opérateur de sélection se généralise facilement à n entrées, avec une variable de sélection X prenant les valeurs 0,1, ..., $n-1$.

2.5.13 Opérateur de mise à zéro

Une initialisation ou une condition d'erreur peut imposer qu'un nombre ou résultat soit nul. L'opérateur *CLR* (*clear*) effectue cette mise à zéro (fig. 2.102), et transforme un nombre logique ou arithmétique en un nombre nul.

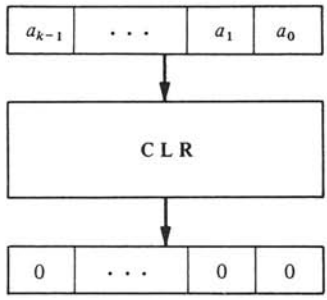


Fig. 2.102

2.5.14 Opérateur de transfert

Un simple transfert entre registres est une opération fréquente, imposée par les limitations de place en mémoire et dans les registres des processeurs. L'opérateur s'appelle soit *MOVE*, soit *LOAD* selon l'optique adoptée (fig. 2.103). Move implique que la source est d'abord mentionnée. LOAD mentionne par contre d'abord la destination.

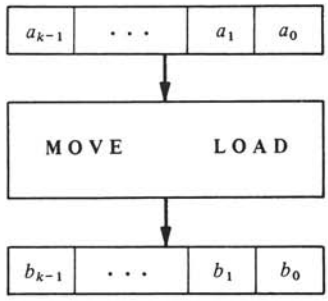


Fig. 2.103

Les deux optiques "LOAD" et "MOVE" ont des répercussions sur l'ordre des opérandes pour toutes les opérations arithmétiques. Cet ouvrage utilise principalement la notation "LOAD", c'est-à-dire mentionne la destination d'abord, ce qui est plus naturel pour la soustraction, la comparaison et les décalages. Les exemples avec le M68000 (chap. 4) utiliseront l'optique "MOVE", qui tend à se généraliser pour les microprocesseurs 16 bits.

2.5.15 Opérations logiques

Les variables booléennes étant des variables binaires, il peut être intéressant d'effectuer des fonctions logiques entre mots binaires. Les opérations les plus intéressantes sont le *ET* logique (*AND*), le *OU* logique (*OR*) et le *OU exclusif* (*XOR*). Ces opérations effectuent l'opération logique correspondante sur les bits de même rang. La figure 2.104 illustre l'effet de ces opérateurs sur un exemple.

Les opérateurs logiques sont souvent utilisés pour modifier de façon sélective les bits d'un mot. Un second mot appelé *masque* est utilisé comme référence et permet de mettre à zéro, mettre à un, ou inverser certains bits ou groupes de bits (fig. 2.104).

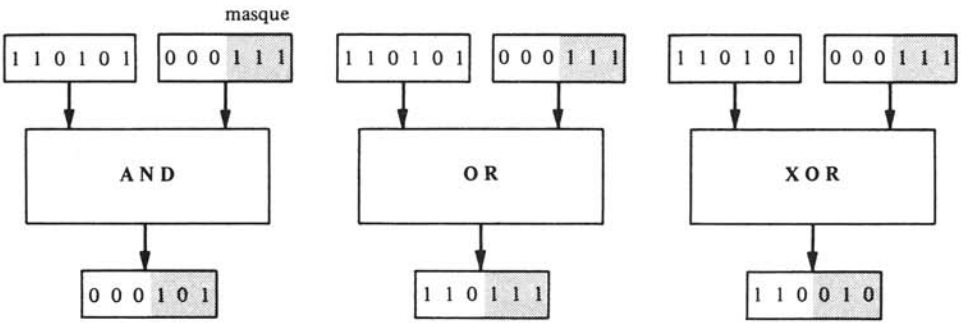


Fig. 2.104

2.5.16 Exercice

L'opérateur OR force à 1 tous les bits du résultat qui correspondent à des 1 dans le masque. A cause de cette propriété, certains fabricants appellent *BIS* (*bit set*) cette fonction et définissent une fonction *BIC* (*bit clear*) qui force à 0 tous les bits du résultat correspondant à des 1 dans le masque. Montrer comment la fonction BIC peut être réalisée en utilisant les opérateurs vus précédemment.

2.5.17 Exercice

Montrer comment l'opérateur OR peut se réaliser à partir des opérateurs AND et NOT.

2.5.18 Exercice

Montrer comment l'opérateur XOR peut être remplacé par une combinaison d'opérateurs AND, OR et NOT.

2.6 MULTIPLICATION ET DIVISION ENTIÈRE

2.6.1 Multiplication d'entiers positifs

L'opérateur de multiplication de deux nombres entiers de longueur k fournit un résultat de longueur $2k$ (fig. 2.105).

Les opérandes sont traditionnellement appelés *multiplicande* et *multiplicateur* ou *facteurs*; le résultat s'appelle *produit*. L'opérateur est caractérisé par les lettres *MUL*. On écrit :

$$\text{MUL}.k \qquad P, A, B \tag{2.52}$$

ou

$$\text{MUL}. \qquad P.\,2k, A.\,k, B.\,k \tag{2.53}$$

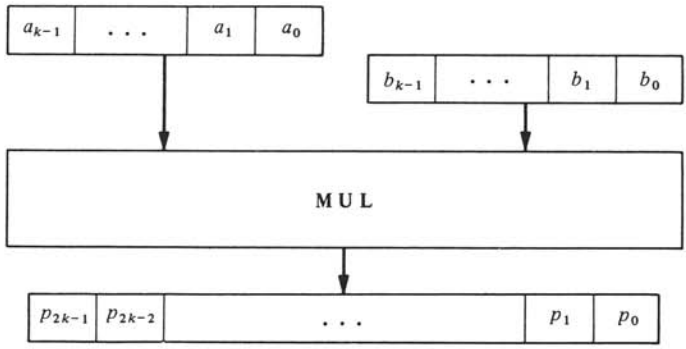


Fig. 2.105

L'opération de multiplication, de par sa complexité, est toujours décomposée en opérations plus simples. Cette décomposition conduit à différents algorithmes et à une multitude de variantes selon la base. Bornons-nous dans ce paragraphe à mettre en évidence un certain nombre de principes.

Si A et B sont les deux opérandes, on déduit de (2.2) que

$$A \cdot B = A \cdot (b_{k-1} \cdot p^{k-1} + \ldots + b_0) = \tag{2.54}$$

$$= (\ldots (((0 + A \cdot b_0) + A \cdot p \cdot b_1) + (Ap) p \cdot b_2) + \ldots + (Ap^{k-2}) \cdot p \cdot b_{k-1}) \tag{2.55}$$

Cette expression fait intervenir trois types d'opérateurs

- *ADD*: les groupements mis en évidence par les parenthèses montrent qu'une succession d'additions comportant chacune deux termes conduit au résultat;
- *SL*: le multiplicande A est multiplié par p, puis encore par p, etc. Chacune de ces opérations correspond à un décalage à gauche. Il pourrait revenir au même de décaler le résultat de l'opération;
- *MULP*: la multiplication de A par b_i définit un nouvel opérateur, dit de *multiplication partielle*.

Le schéma fonctionnel de cette décomposition est donné dans la figure 2.106. La longueur de chaque opérateur est égale à $2k$, mais seule une partie (tramée sur la figure) est active, c'est-à-dire modifie les chiffres correspondants.

2.6.2 Exercice

Effectuer suivant la méthode définie à la figure 2.106 la multiplication décimale 113 x 216.

2.6.3 Décomposition temporelle de la multiplication

Pour éviter le nombre important d'opérateurs de la figure 2.106, on peut calculer le résultat par étapes successives, en faisant intervenir l'un après l'autre les chiffres du multiplicateur.

Les registres opérandes et produit sont modifiés à chaque cycle (défini par la transition d'un signal de synchronisation), et un registre auxiliaire X mémorise le résultat de la multiplication partielle. Le registre qui génère le produit est initialisé à zéro et un comp-

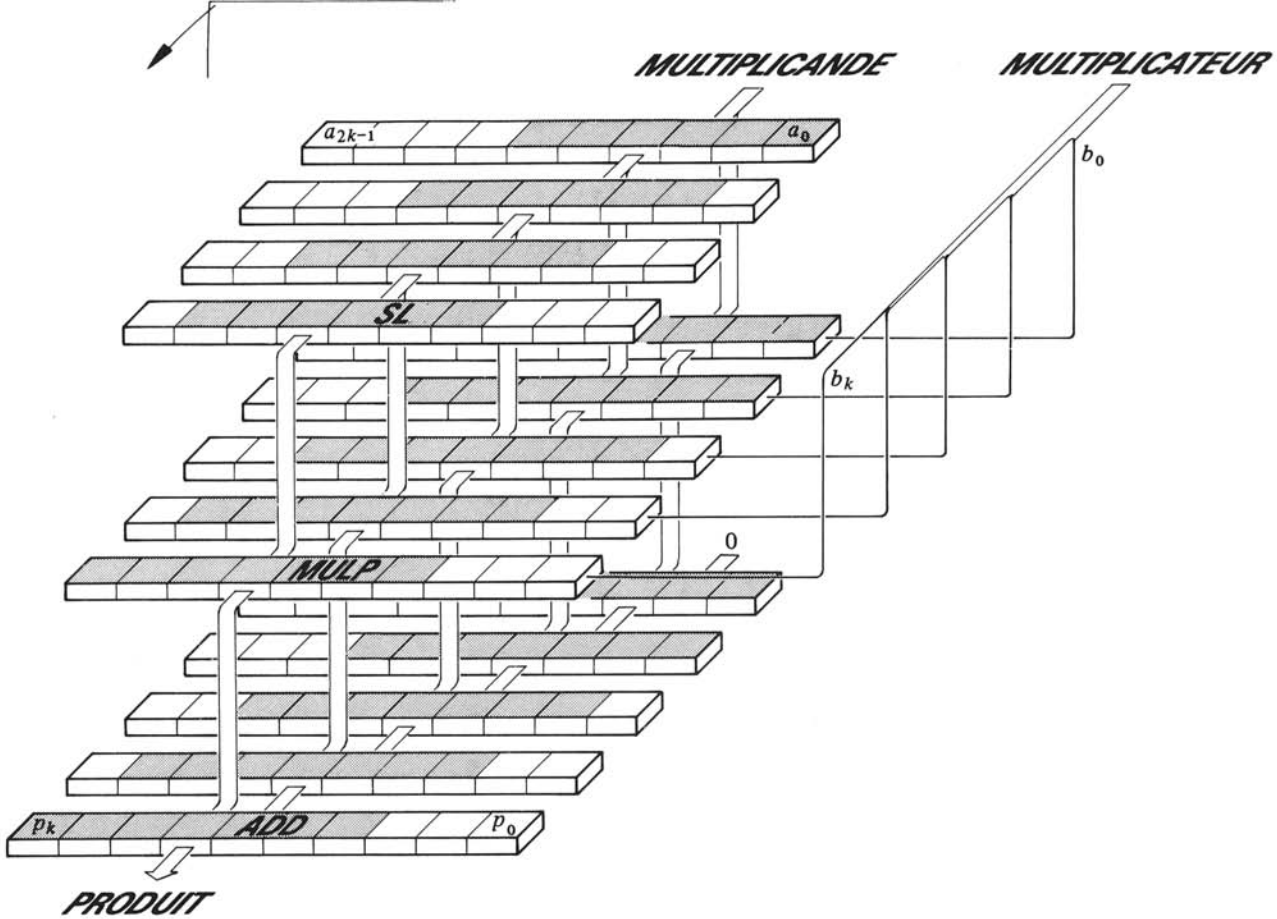

MULTIPLICANDE
MULTIPLICATEUR
a_{2k-1}
a_0
b_0
SL
b_k
0
MULP
p_k
ADD
p_0
PRODUIT

Fig. 2.106

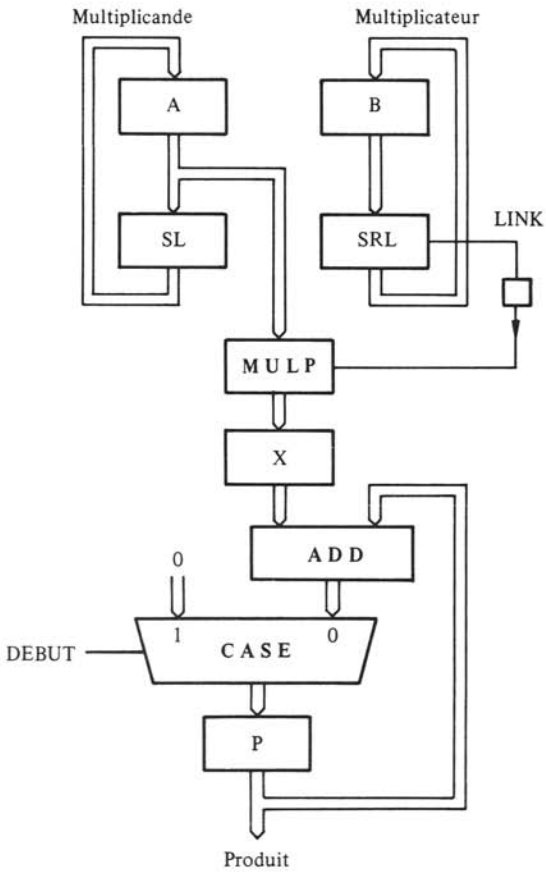


Fig. 2.107

teur de cycle, non représenté sur la figure 2.107, détermine la fin de l'opération. Le système évolue vers le résultat et les registres A, B et P sont modifiés tous simultanément ou selon un ordre compatible avec l'opération.

Le schéma-bloc de la multiplication a un équivalent sous forme d'un programme:

```
CLR          P
répéter k fois
   SRL       B, B
   MULP      X, A                    (2.56)
   ADD       P, A, X
   ASL       A, A
fin répéter
```

2.6.4 Décomposition de la multiplication partielle

L'opérateur de multiplication MULP est plus simple que l'opérateur de multiplication. Il mutiplie un nombre de longueur k par un chiffre, et fournit un résultat de longueur $k + 1$ (fig. 2.108).

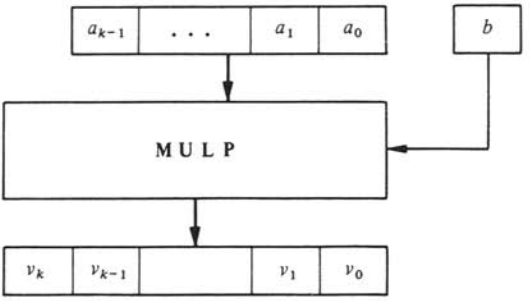


Fig. 2.108

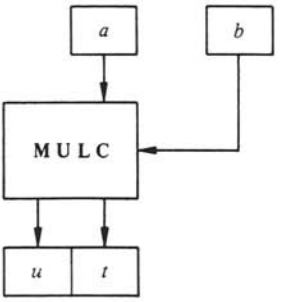


Fig. 2.109

Développons l'expression qui caractérise cet opérateur

$$A \cdot b = a_{k-1} \cdot b \cdot p^{k-1} + \ldots + a_1 \cdot b \cdot p + a_0 \cdot b \qquad (2.57)$$

On voit que l'opération $a_i \cdot b$, qui multiplie deux chiffres et fournit un résultat inférieur à p^2 (donc un nombre de longueur 2) est une opération de base à laquelle nous pouvons associer l'opérateur *MULC* (*multiplication de chiffres*) (fig. 2.109). Des opérateurs MULC et un additionneur permettent donc d'implémenter l'opérateur MULP (fig. 2.110).

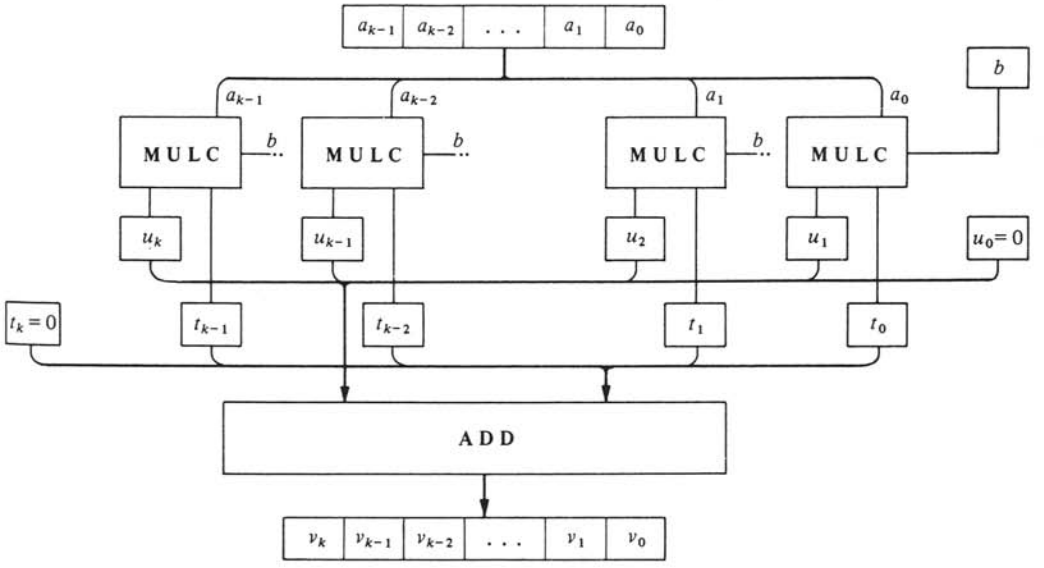


Fig. 2.110

A son tour, l'opérateur MULC peut être décomposé en opérations plus élémentaires d'addition, de décalage, de sélection, de comptage, etc. Cette décomposition dépend fortement de la base et sera étudiée plus loin pour le binaire (§ 2.6.8) et le décimal codé en binaire (§ 2.8.12).

2.6.5 Remarque

Le développement de l'expression (2.55) conduit à la relation

$$A \cdot B = p^{k-1} \cdot [(...(A \cdot b_{k-1} + (A \cdot p^{-1})\, b_{k-2}) + ...) + (A \cdot p^{-k+1}) \cdot b_0] \quad (2.58)$$

Cette relation montre qu'un réseau très similaire à celui de la figure 2.106 peut être défini pour la multiplication. Ce réseau est presque identique à celui dessiné dans la figure 2.106 avec des opérateurs *SR* de décalage à droite et une action du chiffre de poids fort sur le premier opérateur de multiplication partielle. Cette décomposition correspond à l'algorithme de multiplication "par les poids forts", plus rarement enseigné à l'école que l'algorithme "par les poids faibles".

2.6.6 Exercice

Effectuer suivant la méthode du paragraphe 2.6.5 la multiplication 113 x 216.

□ 2.6.7 Multiplication de nombres arithmétiques

L'algorithme et l'opérateur de multiplication vus précédemment permettent aussi de multiplier des nombres arithmétiques. En effet si les nombres A et B sont signés de longueur $k+1$ (a_k et b_k sont les chiffres de signe)

$$A' = p^{k+1} - A,\ B' = p^{k+1} - B$$

$$A \cdot B' = A \cdot p^{k+1} - AB = (A-1) \cdot p^{k+1} - AB + p^{k+1} = (A-1) \cdot p^{k+1} + (AB)'$$

donc

$$AB' \equiv (AB)' \text{ (modulo } p^{k+1}) \quad (2.59)$$

$$A' \cdot B' = p^{2k+2} - (A+B) \cdot p^{k+1} + AB$$

donc

$$A'B' \equiv AB \text{ modulo } p^{k+1} \quad (2.60)$$

Le même algorithme de multiplication peut donc être appliqué pour des nombres arithmétiques et des nombres logiques, mais dans le cas des nombres arithmétiques, les dépassements de capacité en cours d'opération ne sont pas significatifs.

Dans la plupart des applications, la multiplication de nombres arithmétiques est toutefois effectuée en se ramenant à des nombres signés (valeur absolue et signe), en déterminant le signe du résultat d'après la règle des signes bien connue, puis en effectuant le produit des valeurs absolues.

□ 2.6.8 Multiplication binaire

En binaire, l'opérateur MULP vu au paragraphe 2.6.4 est particulièrement simple, étant donné la table de multiplication binaire donnée par la figure 2.14 (§ 2.2.1). Deux cas seulement sont à envisager: la multiplication par 1, qui est l'identité, et la multiplication par 0, qui donne un résultat nul. L'opérande AND (§ 2.5.15) joue ce rôle, chaque bit de A étant combiné avec b_i.

L'opérateur MULP et l'additionneur qui le suit sont souvent associés pour former un additionneur conditionnel dont les schémas fonctionnels équivalents sont donnés dans la figure 2.111.

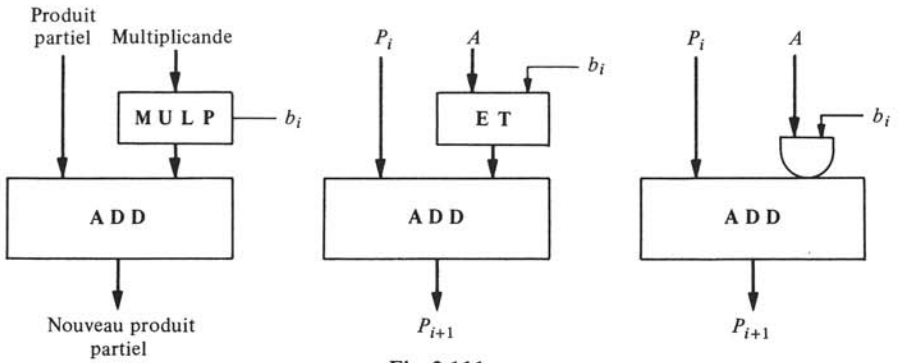


Fig. 2.111

Un additionneur et un multiplexeur permettent également la réalisation d'un additionneur conditionnel (fig. 2.112).

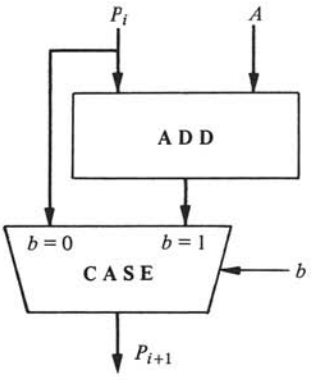


Fig. 2.112

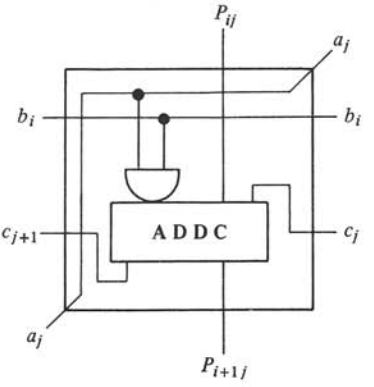


Fig. 2.113

L'additionneur conditionnel de longueur k peut être décomposé en *cellules de multiplication* opérant chacune sur un seul bit du multiplicande. La figure 2.113 donne la structure d'une telle cellule, tenant compte du report qui doit se propager d'une cellule à l'autre.

La juxtaposition de cellules dans le plan permet de réaliser l'opérateur de multiplication complet. Le décalage physique des cellules à gauche réalise l'opérateur *SL* et la moitié des cellules peuvent être remplacées par des liaisons directes. La figure 2.114 donne la représentation de ce réseau, appelé *réseau itératif* à cause de la répétition d'un même motif.

Un réseau très similaire, avec décalage des cellules vers la droite, correspond à l'algorithme de multiplication par les poids forts (§ 2.6.5). La structure des cellules est identique.

De nombreuses autres variantes de multiplication sont possibles, destinées à minimiser les temps de propagation ou simplifier la structure des cellules [38, 39].

□ 2.6.9 Algorithme rapide de multiplication binaire

Différentes techniques permettent d'accélérer l'opération de multiplication en tenant compte de groupements particuliers de bits dans le multiplicande ou dans le multi-

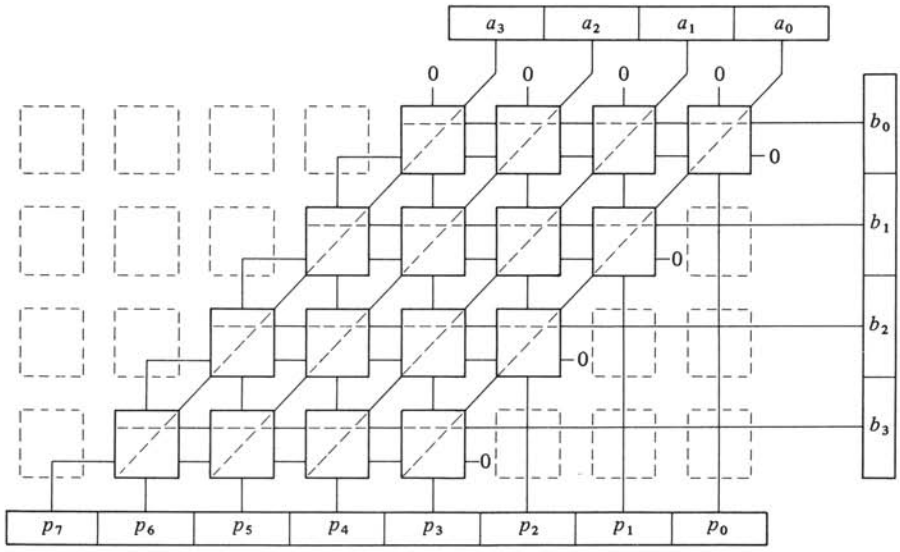


Fig. 2.114

plicateur. En particulier, il est évident que si l'un des opérandes est nul, le résultat est également nul.

Une suite d'additions du même terme décalé peut être remplacée par une soustraction et une addition. Par exemple, si le multiplicande A doit être multiplié par 11 111, on peut écrire :

$$A \cdot 11111 = A \cdot 100000 - A \tag{2.61}$$

ce qui correspond à la figure 2.115.

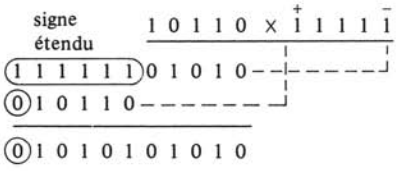


Fig. 2.115

La figure 2.116 donne un autre exemple d'opération plus complexe.

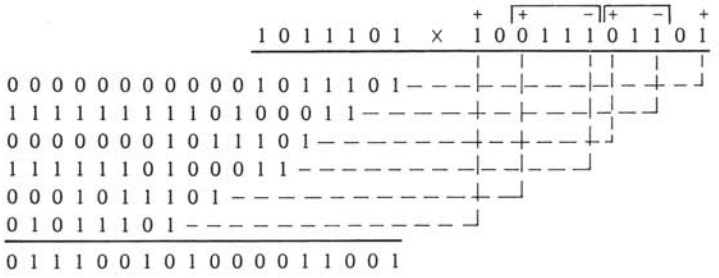


Fig. 2.116

Pour chaque bit du multiplicateur examiné, il y a trois comportements possibles selon la valeur du bit immédiatement à gauche et l'opération effectuée précédemment:

- passer au bit suivant;
- additionner le multiplicande;
- additionner le complément à 2 du multiplicande.

Chaque fois, un décalage est effectué pour aligner le résultat partiel.

Remarquons qu'une variante un peu plus rapide conduit à examiner également les chaînes de "1" avec des "0" isolés. La présence d'un zéro isolé conduit à soustraire le terme correspondant. Dans ce cas, l'exemple ci-dessus aurait donné pour les tests faits sur le multiplicateur:

$$\begin{array}{c} + \quad \overset{\downarrow}{+} \qquad \overset{\downarrow}{-} \qquad \overset{\downarrow}{-} \quad - \\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 1 \end{array} \tag{2.62}$$

On peut remarquer en comparant avec la figure 2.116 qu'une opération de moins est nécessaire.

Ces algorithmes sont appelés algorithmes de Booth [36-38]. Ils ne sont plus rapides que dans la mesure où le test des bits du multiplicateur et la décision d'addition, de soustraction ou de non opération est plus rapide que l'addition conditionnelle.

Etant donné la rapidité des ordinateurs actuels dans les opérations d'addition et de décalage, ces algorithmes ont beaucoup perdu de leur intérêt. Par contre la décomposition spatiale et le recours à des additionneurs dits sans report organisés en réseaux de Wallace permet d'obtenir des unités de multiplication très rapides [38, 40].

□ 2.6.10 Division entière en base p

Diviser A (*dividende*) par B (*diviseur*), c'est trouver un *quotient* Q et un *reste* $R < B$ tels que $A = B \cdot Q + R$. L'opérateur est représenté dans la figure 2.117.

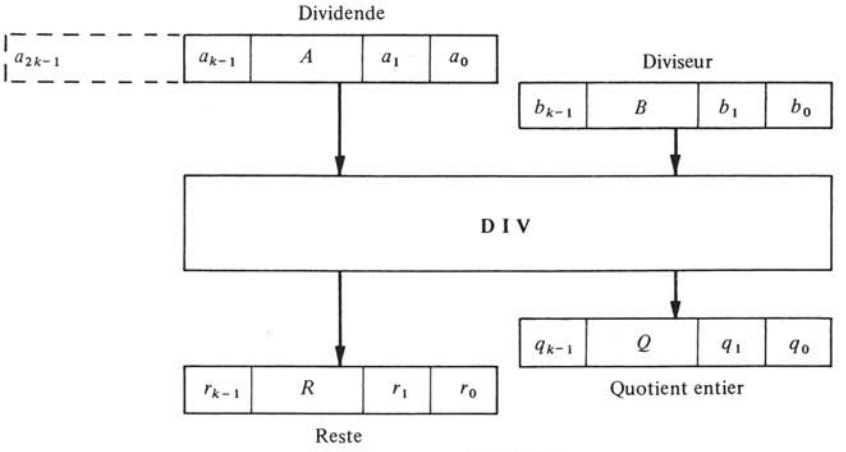


Fig. 2.117

L'opérateur peut se noter $DIV.k\,R, Q, A, B$.

Posons:

$$Q = q_s \cdot p^s + \ldots + q_1 \cdot p + q_0 \tag{2.63}$$

Si A et B sont de longueurs k, Q est de longueur $\leqslant k$, car $Q \leqslant A < p^k$. Ecrivons en partant de la définition

$$A = B \cdot q_s \cdot p^s + B \cdot q_{s-1} \cdot p^{s-1} + \ldots B \cdot q_0 + R \tag{2.64}$$

$$A - B \cdot q_s \cdot p^s = B \cdot (q_{s-1} \cdot p^{s-1} + \ldots + q_0) + R \tag{2.65}$$

$$A - B \cdot q_s \cdot p^s \leqslant B \cdot (p^s - 1) + R < B(p^s - 1) + B = B \cdot p^s \tag{2.66}$$

Cette relation définit q_s comme étant un chiffre tel que

$$A - B \cdot q_s \cdot p^s < B \cdot p^s \tag{2.67}$$

Pratiquement, un alignement de A et B détermine s et la valeur q_s est déterminée par essais successifs. Pour éviter l'alignement initial, un alignement correspondant au cas pire, c'est-à-dire à $s = k - 1$ peut être choisi au départ.

Définissons un opérateur de division partielle *DIVP* calculant le plus grand produit $B \cdot q_s$ et soustrayant ce nombre. Un soustracteur soustrait ce terme et des opérateurs de décalage tiennent compte des termes p^s, p^{s-1}, ... Un réseau très similaire à celui de la figure 2.106 résulte de cette décomposition.

L'opération de division partielle est délicate et conduit à de nombreuses variantes, dont certaines seront étudiées dans les paragraphes 2.6.11 et 2.8.14 dans le cas du système binaire et du système décimal.

La division des nombres arithmétiques, c'est-à-dire sous forme complémentaire n'est pas possible en général. Il faut toujours passer par une représentation signée et opérer sur les valeurs absolues.

□ 2.6.11 Division binaire entière

Dans la division binaire de A par B, les bits du quotient valent 0 et 1. L'opérateur de division partielle et le soustracteur associé *DIVP* est donc simplement une unité qui,

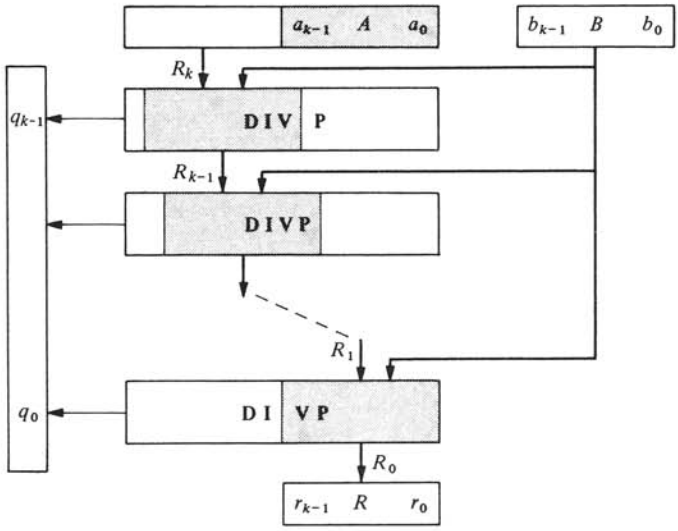


Fig. 2.118

après un décalage convenable soustrait B si cela est possible. Une représentation simplifiée de la division est donnée dans la figure 2.118 dans laquelle les opérateurs de décalage sont implicites. On remarque dans cette figure que si A et B sont de longueur k, les opérateurs doivent être de longueur $2^k - 1$. Seule une partie de ces opérateurs est utilisée.

L'opérateur de division partielle peut être réalisé comme dans la figure 2.119. Le multiplexeur est commandé par le report C du soustracteur.

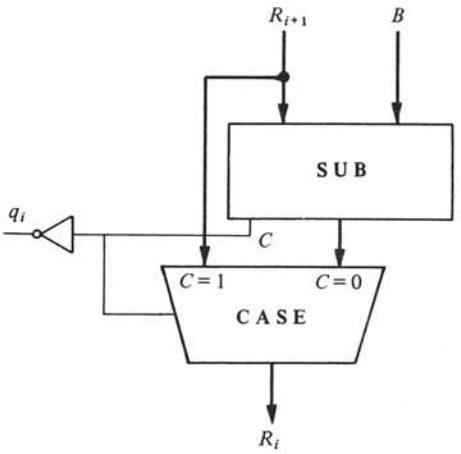


Fig. 2.119

Si $C = 1$, il n'est pas possible d'extraire le diviseur du résultat partiel; le bit du quotient vaut 0 et le multiplexeur prend le résultat précédent non modifié pour le décaler.

Si $C = 0$, le résultat de la soustraction est positif ou nul et doit être transmis pour l'étape suivante; le bit du quotient vaut 1.

Cette méthode est appelée méthode par *comparaison*, car elle admet une variante dans laquelle un comparateur commande le multiplexeur qui décide si la soustraction doit s'effectuer ou non.

□ 2.6.12 Méthode avec et sans rétablissement

Un algorithme de division fréquemment utilisé consiste à effectuer la soustraction et, en cas de résultat négatif, rétablir le résultat correct par addition. La figure 2.120 illustre cette méthode par un exemple numérique et par le schéma fonctionnel des opérateurs nécessaires au calcul d'un bit du quotient. Cette méthode est dite *division avec rétablissement* du résultat.

Remarquons que lorsqu'il y a addition de B pour correction, puis décalage de B et soustraction de B décalé, cette opération est équivalente à l'addition de B décalé. En effet, le décalage à droite de B est équivalent à une division par 2, et l'opération citée s'écrit $X + B - B/2 = X + B/2$.

Un algorithme souvent plus simple, dit *division sans rétablissement*, se déduit de cette règle. Il est représenté dans la figure 2.121 et présente l'inconvénient de fournir un reste sous forme complémentaire lorsque le bit de poids faible du quotient est nul. Une addition sans décalage permet de rétablir le résultat correct mais cette opération rompt la régularité de l'algorithme.

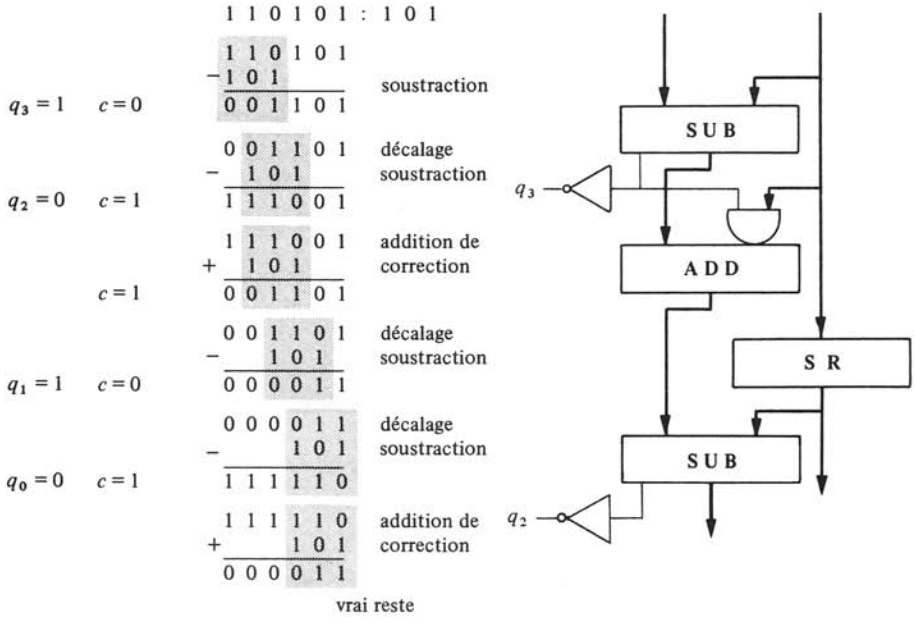


Fig. 2.120

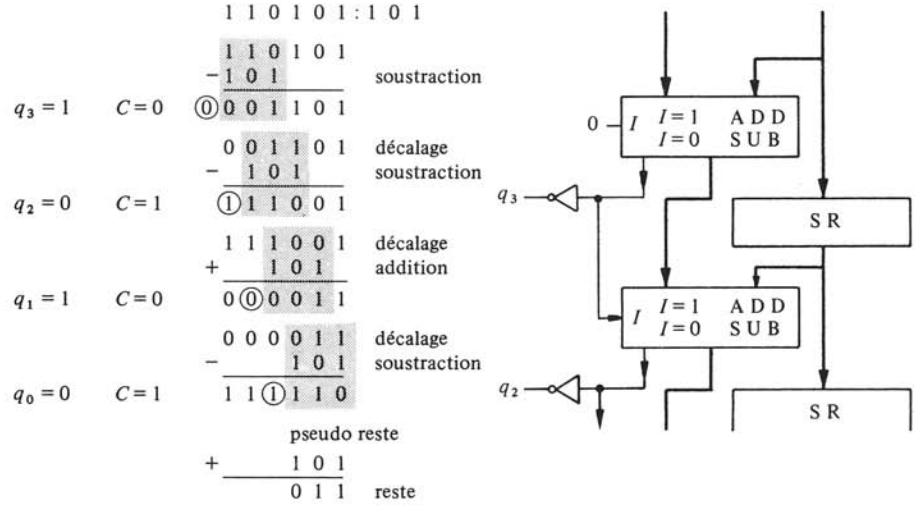


Fig. 2.121

□ 2.6.13 Autres méthodes de division

La division de deux nombres apparaissant fréquemment dans les programmes d'ordinateur et nécessitant un temps de calcul appréciable, de nombreuses méthodes particulières ont été proposées. Elles impliquent généralement une complexité extrême de l'organe de calcul [37, 38].

2.6.14 Exercice

Effectuer les divisions binaires suivantes par la méthode sans rétablissement: 11010110 : 0110; 11011111 : 100011.

2.7 OPÉRATIONS SUR LES NOMBRES RÉELS

2.7.1 Addition et soustraction sur des nombres purement fractionnaires

Les nombres purement fractionnaires (§ 2.1.20) se traitent de façon très similaire aux nombres entiers. Les développements faits dans la section 2.3 se transposent sans peine et conduisent aux résultats suivants:

- le complément vrai A' d'un nombre logique purement fractionnaire A est défini par la relation $A = +1 - A'$, ou $A' = 1 - A$ (§ 2.4.3 et 2.4.8);
- il est naturel de représenter les nombres négatifs purement fractionnaires par le complément vrai de leur valeur absolue (§ 2.4.3);
- soustraire un nombre fractionnaire ou ajouter son complément vrai revient au même, sauf en ce qui concerne la valeur du report c_0 généré par l'opérateur de soustraction ou d'addition (§ 2.4.15);
- le chiffre de signe est un chiffre de poids 1 valant 0 pour les nombres positifs et $p - 1$ pour les nombres négatifs représentés en complément vrai (§ 2.4.10). La figure 2.122 représente un nombre logique et un nombre arithmétique purement fractionnaires. Le gros point dans le champ du nombre arithmétique est un repère visuel sans signification réelle;

Logique Arithmétique

Fig. 2.122

- le complément restreint d'un nombre fractionnaire diffère du complément vrai par une unité de plus faible poids (§ 2.4.24). Il peut être envisageable d'identifier les deux types de compléments, si l'erreur systématique ainsi créée est négligeable;
- la comparaison et le décalage des nombres purement fractionnaires logiques ou arithmétiques suivent les mêmes règles et utilisent les mêmes opérateurs que pour les nombres entiers (sect. 2.5).

□ 2.7.2 Opérations en virgule fixe

Les nombres en virgule fixe sont utilisés lorsque l'application conduit naturellement à ce type de nombre, avec un format fixe. Comme exemple, citons les applications commerciales avec leurs francs et centimes.

L'addition et la soustraction en virgule fixe sont identiques à l'addition et à la soustraction de nombres entiers. La multiplication doit être suivie d'une correction de la position de la virgule; la division doit être précédée de cette correction.

Les règles associées aux nombres en virgule fixe sont suffisamment familières pour ne pas les développer ici. Leur application dans les calculatrices est par ailleurs relativement restreinte.

□ 2.7.3 Nombre arithmétique purement fractionnaire normalisé

La définition d'une mantisse normalisée (§ 2.1.22) a été donnée pour les nombres positifs. Pour les nombres arithmétiques, le chiffre de signe joue un rôle particulier.

Si le nombre est positif, le chiffre de signe vaut zéro, et le chiffre suivant est différent de zéro si le nombre est normalisé.

Si le nombre est négatif, le chiffre de signe vaut $p-1$, et le chiffre suivant est différent de $p-1$ si le nombre est normalisé. Ceci revient à dire qu'un nombre négatif est normalisé si son complément est normalisé. Faisons la démonstration pour un nombre arithmétique purement fractionnaire normalisé négatif.

Soit

$$A = (p-1) + a_{-1}\cdot p^{-1} + a_{-2}\cdot p^{-2} + \ldots = (p-1) + a_{-1}\cdot p^{-1} + x \qquad (2.68)$$

avec

$$a_{-1} \leqslant p-2 \text{ (normalisé négatif) et } x = a_{-2}\cdot p^{-2} + \ldots < p^{-1} \qquad (2.69)$$

Le complément vrai s'écrit :

$$A' = p - A = p - (p-1) - a_{-1}\cdot p^{-1} - x = 1 - a_{-1}\cdot p^{-1} - x$$

On déduit en tenant compte de (2.69) que :

$$A' \geqslant 1 - (p-2)\cdot p^{-1} - x = 2\cdot p^{-1} - x > p^{-1} \qquad (2.70)$$

Comme par ailleurs il est facile de montrer que $A' < 1$, on en déduit que A est normalisé.

□ 2.7.4 Addition de nombres flottants

Soient A et B deux nombres flottants, que l'on peut écrire :

$$A = a\cdot p^{\alpha} \qquad \text{et} \qquad B = b\cdot p^{\beta} \qquad (2.71)$$

Les mantisses a et b sont des nombres arithmétiques, de même que les exposants α et β.

Exprimons la somme de A et B :

$$A + B = a\cdot p^{\alpha} + b\cdot p^{\beta} = a\cdot p^{\alpha} + (b\cdot p^{\beta-\alpha})\cdot p^{\alpha} = (a + b\cdot p^{\beta-\alpha})\cdot p^{\alpha} \qquad (2.72)$$

La mantisse $b\cdot p^{\beta-\alpha}$ s'obtient à partir de la mantisse b par décalage.

Si $\beta > \alpha$, un décalage de $\beta - \alpha$ positions à gauche multiple par $p^{\beta-\alpha}$. Les conditions de dépassement de capacité vues au paragraphe 2.5.7 font que l'opération ne peut pas nécessairement être effectuée dans le champ initialement prévu.

Si $\beta < \alpha$, un décalage à droite de $\alpha - \beta$ positions multiplie par $p^{\alpha-\beta}$. Ce décalage doit conserver le signe de b, c'est-à-dire que l'opérateur ASR (§ 2.5.8) doit être utilisé.

Si $\beta = \alpha$, aucun décalage ne doit être effectué.

La même opération peut être effectuée en écrivant :

$$A + B = a\cdot p^{\alpha} + b\cdot p^{\beta} = (a\cdot p^{\alpha-\beta})\,p^{\beta} + b\cdot p^{\beta} = (a\cdot p^{\alpha-\beta} + b)\cdot p^{\beta} \qquad (2.73)$$

Dans une représentation en champ fixe, l'erreur d'arrondi qui résulte de ces deux opérations peut être très différente. Considérons en effet dans le système décimal les nombres $A = 0{,}0440.\cdot 10^{+2}$ et $B = 0{,}1234.\cdot 10^{-1}$.

$$A + B = 0{,}0440.\cdot 10^{2} + 0{,}1234.\cdot 10^{-3}\cdot 10^{2} = (0{,}0440. + 0{,}0001234.)\cdot 10^{2}$$
$$= 0{,}0441234.\cdot 10^{+2} \qquad (2.74)$$

$$A + B = 0{,}0440.\cdot 10^{3}\cdot 10^{-1} + 0{,}1234.\cdot 10^{-1} = 44.\cdot 10^{-1} + 0{,}1234.\cdot 10^{-1}$$
$$= 44.1234.\cdot 10^{-1} \qquad (2.75)$$

Si nous supposons maintenant que les mantisses sont des nombres purement fractionnaires de 4 chiffres, le premier résultat est $0{,}0441.\cdot 10^{2}$ et le second $0{,}1234.\cdot 10^{-1}$. Dans ce second cas, le décalage du 1er nombre à gauche avant addition a signalé un dépassement enlevant toute signification au résultat.

Si les mantisses sont normalisées, un décalage à gauche crée nécessairement un dépassement de capacité. Il est nécessaire de décaler à droite avant l'addition le nombre dont l'exposant est le plus faible (en valeur arithmétique).

□ 2.7.5 Remarque

L'addition de deux nombres flottants normalisés peut créer un dépassement, c'est-à-dire un nombre non normalisé. Le résultat est généralement récupérable par décalage et reconstitution du chiffre de signe (normalisation), ainsi que le montrent les exemples suivants en base 10 (fig. 2.123).

	$0{,}976.\cdot 10^{2}$	$0{,}321.\cdot 10^{-3}$		$9{,}143.\cdot 10^{6}$	$(-0{,}857.\cdot 10^{6})$
	$+\ 0{,}321.\cdot 10^{2}$	$+\ 9{,}676.\cdot 10^{-3}$	$(-0{,}324.\cdot 10^{-3})$	$+\ 9{,}312.\cdot 10^{6}$	$(-0{,}688.\cdot 10^{6})$
somme	$1{,}297.\cdot 10^{2}$	$9{,}997.\cdot 10^{-3}$	$(-0{,}003.\cdot 10^{-3})$	$(1)8{,}455.\cdot 10^{6}$	$(-1{,}545.\cdot 10^{6})$
normalisation	$0{,}129.\cdot 10^{3}$	$9{,}700.\cdot 10^{-5}$	$(-0{,}300.\cdot 10^{-5})$	$9{,}845.\cdot 10^{7}$	$(-0{,}154.\cdot 10^{7})$
					complément

Fig. 2.123

La normalisation n'est pas possible si l'exposant ne peut pas être augmenté ou diminué sans créer un dépassement de capacité de l'exposant, donc un résultat complètement erroné. Les termes de dépassement (overflow) et soupassement (underflow) caractérisent les deux débordements possibles de l'exposant.

□ 2.7.6 Soustraction et comparaison de nombres flottants

La soustraction de nombres fractionnaires ou flottants n'est pas différente de l'addition, spécialement si les nombres négatifs sont représentés sous forme de complément vrai. Toutes les considérations des précédents paragraphes sont donc valables pour la soustraction.

La comparaison de deux nombres flottants produit les indicateurs LT, LE, EQ, GE, GT et NE comme pour les nombres arithmétiques (§ 2.5.3). Ces indicateurs peuvent être déterminés par le résultat de la soustraction des 2 nombres à comparer. Une comparaison des exposants et mantisses peut conduire plus rapidement à ce résultat.

□ 2.7.7 Représentation du zéro

En virgule flottante, le zéro peut être à priori représenté par une mantisse égale à zéro et un exposant quelconque. Le zéro n'est pas normalisable et il apparaît un problème dans l'addition mis en évidence par l'exemple décimal suivant, dans lequel l'exposant du zéro est supérieur à l'exposant du nombre à additionner:

$$0\cdot 10^{2} + 0{,}1234.\cdot 10^{-1} = 0\cdot 10^{2} + 0{,}0001234.\cdot 10^{2} = 0{,}0001234.\cdot 10^{2} \qquad (2.76)$$

En format fixe de 4 chiffres pour la mantisse, le résultat est $0{,}0001 \cdot 10^2$. L'application de la règle du paragraphe 2.7.4 a conduit à une erreur d'arrondi non justifiée.

Dans ces conditions, il faut soit compliquer la règle d'addition, soit convenir que le zéro est toujours représenté en virgule flottante avec l'exposant minimum. Le décalage s'effectue alors sur la mantisse nulle, sans arrondi sur l'autre nombre. De plus, l'opération $0 + 0$ n'est pas précédée d'un test ou de décalages inutiles.

Ainsi en base p, si un champ de r chiffres est réservé pour l'exposant, plus un $r + 1^e$ chiffre de signe, l'expression du zéro est donnée par la formule (2.77) et la figure 2.124.

$$0 = 0 \cdot p^{-p^r} \tag{2.77}$$

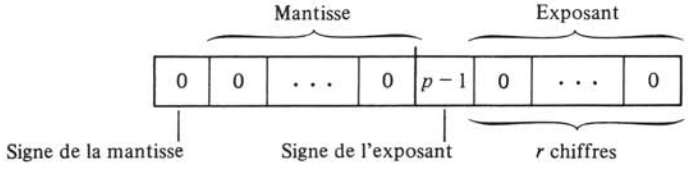


Fig. 2.124

□ 2.7.8 Représentation des exposants positifs et négatifs

La représentation du nombre flottant zéro par un mot différent de zéro (fig. 2.124) n'est pas pratique et conduit à un troisième type de représentation des nombres négatifs, généralement utilisée pour les exposants des nombres flottants.

Soit α un exposant dont la valeur absolue a r chiffres :

$$-p^r \leqslant \alpha < p^r \tag{2.78}$$

Posons

$$\alpha^+ = \alpha + p^r \tag{2.79}$$

alors :

$$0 \leqslant \alpha^+ < 2 \cdot p^r \tag{2.80}$$

La valeur α^+ est dite *valeur en excédent* p^r de α ou *valeur biaisée*; l'*excédent* ou *biais*

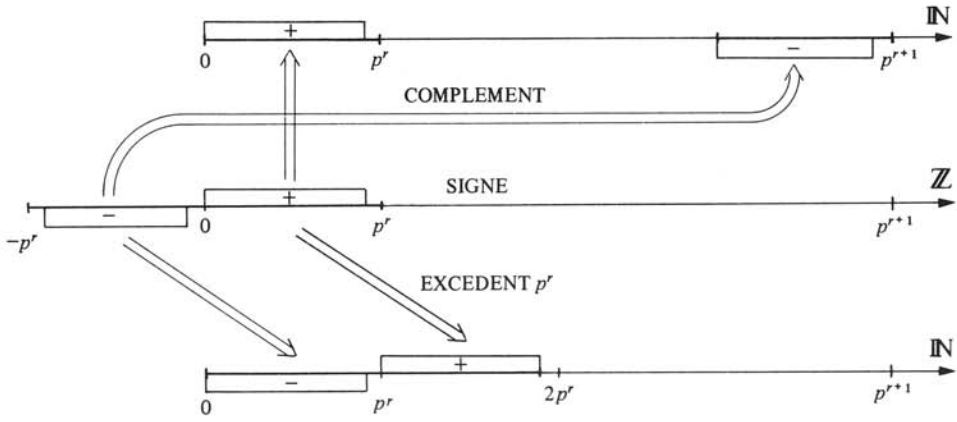


Fig. 2.125

(*bias*), vaut p^r. Tous les nombres α^+ sont positifs et la figure 2.125 montre les différences et relations entre les trois représentations possibles des nombres entiers positifs et négatifs.

Avec cette représentation, l'exposant le plus petit vaut 0 et la représentation du zéro est $0 \cdot p^0$. Les règles d'addition de nombres flottants ne changent pas, car la différence entre les exposants, déterminante pour le décalage des mantisses, a la même valeur dans les deux cas.

□ 2.7.9 Multiplication de nombres purement fractionnaires

L'opération de multiplication des nombres purement fractionnaires est très similaire à la multiplication de nombres entiers. Il est toutefois plus avantageux d'effectuer le produit en commençant par les chiffres de poids fort du multiplicateur, comme expliqué au paragraphe 2.6.5 et dessiné dans la figure 2.126.

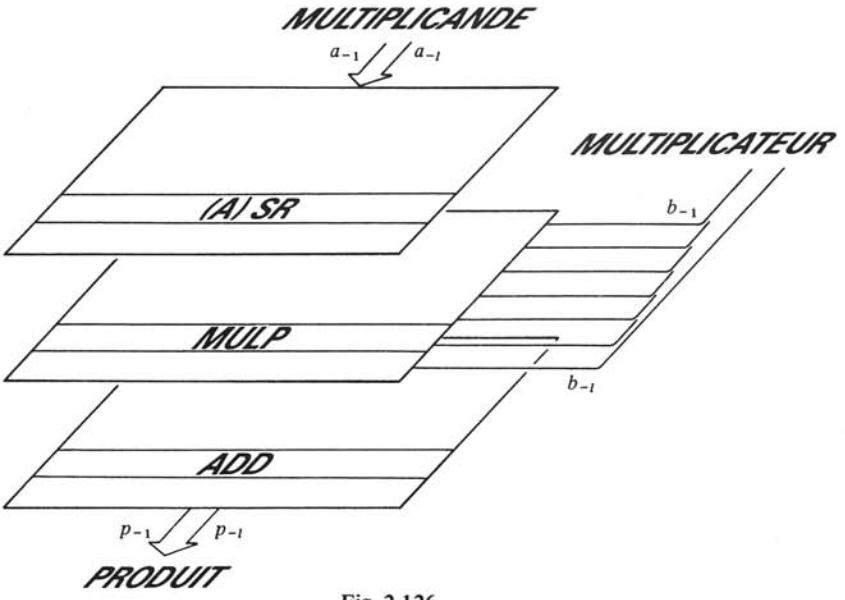


Fig. 2.126

De plus, la multiplication de 2 nombres purement fractionnaires de longueur l peut se faire avec des opérateurs de longueur l, et non pas de longueur double. L'arrondi en dessous de chaque résultat partiel fausse les derniers chiffres du résultat, mais l'erreur peut être minimisée par des techniques adéquates.

Le produit de nombres normalisés n'est pas normalisé et le même opérateur peut multiplier des nombres arithmétiques normalisés, à condition que le signe soit reconstitué correctement (§ 2.6.7 et 2.7.5).

□ 2.7.10 Division de nombres purement fractionnaires

La division de nombres purement fractionnaires pose, comme dans le cas de la division de nombres entiers, un difficile problème d'alignement initial. Le résultat n'est purement fractionnaire que si le diviseur est supérieur au dividende. L'opérateur de division

de nombres entiers positifs peut alors être utilisé tel quel pour la division de nombres fractionnaires positifs.

Si le diviseur est inférieur au dividende, un décalage à droite du diviseur ou un décalage à gauche du dividende permet de retrouver la condition précédente. Chaque décalage multiplie le quotient par un facteur égal à la base et les erreurs d'arrondi peuvent être minimisées par un choix adéquat des opérandes décalés.

2.7.11 Multiplication et division de nombres flottants

Soient $A = a \cdot p^{\alpha}$ et $B = b \cdot p^{\beta}$. On a :

$$A \cdot B = a \cdot b \cdot p^{\alpha+\beta} \tag{2.81}$$

et

$$\frac{A}{B} = \frac{a}{b} \cdot p^{\alpha-\beta} \tag{2.82}$$

La multiplication et la division de nombres flottants se ramènent d'une part à la multiplication et à la division de leur mantisse, qui est en général un nombre fractionnaire normalisé (§ 2.7.3), et d'autre part, à l'addition ou à la soustraction de leurs exposants.

Seule cette opération nécessite un commentaire particulier, étant donné la représentation biaisée utilisée pour les exposants.

Si α et β sont représentés par leur valeur en excédent p^r :

$$\alpha^+ = \alpha + p^r \qquad \text{et} \qquad \beta^+ = \beta + p^r \tag{2.83}$$

alors :

$$\alpha^+ + \beta^+ = (\alpha + \beta + p^r) + p^r = (\alpha + \beta)^+ + p^r \tag{2.84}$$

$$\alpha^+ - \beta^+ = (\alpha - \beta + p^r) - p^r = (\alpha - \beta)^+ - p^r \tag{2.85}$$

Une correction négative ou positive égale à la valeur de l'excédent est nécessaire dans chacun des cas.

2.7.12 Arrondi

L'utilisation d'une mantisse normalisée permet de conserver la précision la plus grande pour les nombres. L'erreur sur une mantisse fractionnaire de l bits est de 2^{-l} (§ 2.1.21) dans le cas d'un *arrondi par troncature* ou $\frac{1}{2} \cdot 2^{-l} = 2^{-l-1}$ dans le cas de l'*arrondi au plus près*.

Les calculs se font le plus souvent avec des formats *étendus* comportant des chiffres de poids faibles supplémentaires, pour éviter que les opérations de soustraction de nombres proches créent des erreurs de normalisation trop importantes. Les arrondis qui sont effectués en fin d'opération doivent avoir une distribution statistique symétrique, pour éviter de biaiser les résultats dans un sens qui est toujours le même, comme par exemple l'arrondi par troncature. Un arrondi au plus près avec un biais faible (arrondi vers les valeurs paires) peut être défini par exemple comme suit [1].

On ajoute 1 au chiffre de poids le plus faible de la partie conservée (en effectuant les reports éventuels qui en résultent), si l'une des conditions suivantes est remplie :

- le chiffre supprimé de plus fort poids est supérieur à la moitié de la base de numérotation du rang de ce chiffre ;

- le chiffre supprimé de plus fort poids est égal à la moitié de la base de numération du rang de ce chiffre et un ou plusieurs autres des chiffres supprimés sont différents de zéro;
- le chiffre supprimé de plus fort poids est égal à la moitié de la base de numération du rang de ce chiffre, tous les autres chiffres supprimés sont nuls et le chiffre conservé de plus faible poids est impair.

Par exemple, les numéraux 12,6375 et 15,0625, arrondis au plus près à trois décimales deviennent respectivement: 12,638 et 15,062.

□ 2.7.13 Nombres binaires flottants

La mantisse et l'exposant d'un nombre flottant sont positifs ou négatifs. Ils peuvent donc être représentés sous forme signée, arithmétique ou biaisée (§ 2.7.8). Le choix est fait en fonction de la complexité des opérations résultantes. La mantisse est presque toujours représentée sous forme signée, à cause des opérations de normalisation (§ 2.7.5), de multiplication et surtout de division (§ 2.7.11). L'exposant est généralement sous forme excédentaire. Indépendamment de ce choix de la représentation des nombres, la longueur des champs et leur position relative entraîne une grande variété de formats flottants. La longueur de la mantisse fixe la précision relative des nombres et des résultats des opérations; le plus souvent, la mantisse est un nombre fractionnaire normalisé. La longueur de l'exposant fixe la grandeur en taille des nombres représentables. Le choix de ces deux longueurs dépend d'une part de l'application, et d'autre part de la longueur des opérateurs utilisés pour effectuer les opérations. Par exemple, si l'opérateur a 8 bits, un format de 9 bits est peu efficace à cause des opérations supplémentaires de transfert et de masquage nécessaires sur le bit additionnel.

Dans les gros ordinateurs, la mantisse est souvent un nombre entier. La conversion de nombre entier en nombre réel (flottant) en est facilitée. Le CDC 7600 par exemple a une mantisse entière en complément à 1 de 49 bits et un exposant de 11 bits en excédent 1024. Des circuits spéciaux sont réalisés pour accélérer la normalisation des nombres et les différentes opérations [38].

□ 2.7.14 Exemple

Les miniordinateurs de la famille PDP11 utilisent un format flottant de 32 bits, extensible à 64 bits, représenté dans la figure 2.127.

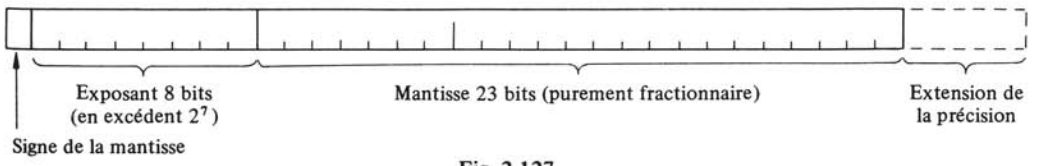


Fig. 2.127

La mantisse est un nombre signé purement fractionnaire normalisé de 23 bits. Le signe est séparé de la mantisse, car dans les opérations signées le signe est traité de façon totalement indépendante de la valeur absolue du nombre. La présence du signe en 1ère position simplifie les tests, et la présence de la mantisse comme 3e champ permet l'augmentation de la précision par simple allongement de ce champ.

L'exposant est un nombre entier 8 bits représentant en excédent 2^7 les nombres de $-128.$ à $+127$. Le plus petit nombre en valeur absolue que l'on peut représenter dans ce format flottant est donc

$$N_{min} = 0{,}1 \cdot 2^{-128.} \simeq 1{,}47 \cdot 10^{-39} \tag{2.86}$$

le plus grand est

$$N_{max} = 0{,}11111...1 \cdot 2^{127.} \simeq 1{,}7. \cdot 10^{38.} \tag{2.87}$$

la précision de la mantisse est de $2^{-23.} \simeq 10^{-7}$

2.7.15 Exercice

Calculer le plus grand nombre, le plus petit nombre et la précision dans le cas du format précédent étendu à 64 bits.

□ 2.7.16 Exemple

Le format précédent convient mal si les opérateurs ont une longueur de 8 bits, comme c'est le cas avec la plupart des microprocesseurs. Le format de la figure 2.128 sacrifie un bit sur l'exposant, limitant à environ $10^{\pm 19.}$ la valeur des nombres représentés.

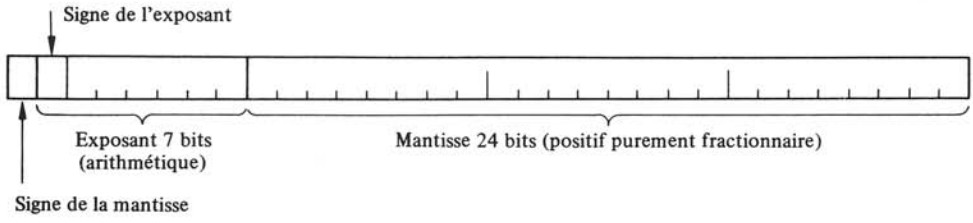


Fig. 2.128

Ce format a été utilisé dans un circuit interface calculateur (AMD 9511 et Intel 8231) prévu pour être associé à un microprosseur 8 bits. On remarque l'utilisation peu usuelle d'une représentation en complément à 2 des exposants négatifs et la limitation très gênante dans la grandeur maximum des nombres ($2^{64} \cong 10^{19}$).

□ 2.7.17 Exemple

La limitation du format précédent à des exposants équivalents à 10^{19} peut être gênante dans plusieurs applications physiques. Pour étendre cette valeur, on peut utiliser une base supérieure.

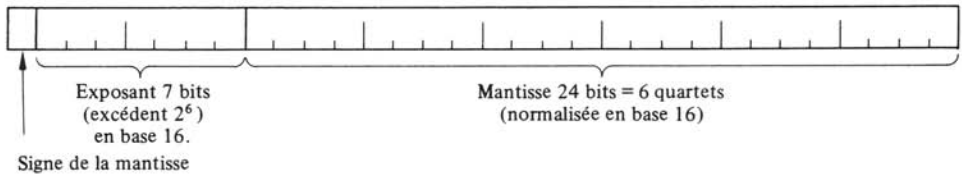


Fig. 2.129

Le format de la figure 2.129 utilise une base 16. et un codage binaire naturel. Il se trouve dans les ordinateurs IBM 360/370 et DG NOVA/ECLIPSE. La mantisse est normalisée si son premier chiffre est différent de zéro. Il en résulte une perte de précision maximale de 3 bits par rapport au binaire, due à la normalisation qui nécessite 4 décalages.

La figure 2.130 montre la représentation d'un nombre égal à 4 et les résultats successifs d'une suite de divisions par 2. La normalisation n'intervient qu'après 4 divisions et entraîne une perte de précision plus grande que dans le cas précédent. Par contre, le plus grand et le plus petit nombre représentables valent respectivement:

$$N_{max} = 7{,}2.\cdot 10^{75.} \qquad N_{min} = 4{,}3.\cdot 10^{-78.} \tag{2.88}$$

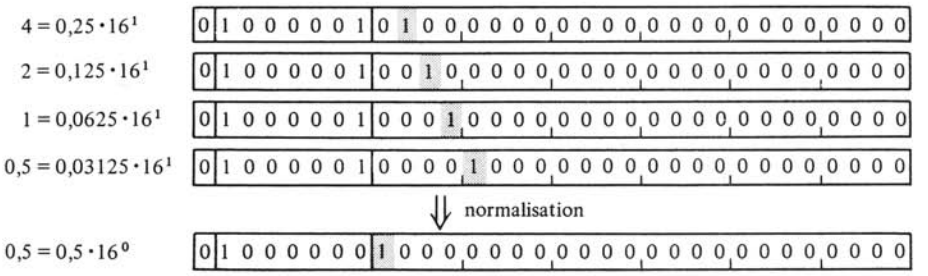


Fig. 2.130

2.7.18 Exemple : format IEEE 754

Un format flottant proposé pour standardisation par l'IEEE (Institute of Electrical and Electronic Engineers) et la CEI (Commission Electrotechnique Internationale) prévoit plusieurs variantes [42, 43]. Il formalise la représentation des nombres particuliers ($\pm\infty$), et permet la représentation d'objets qui ne sont pas des nombres, appelés *non-nombres* ou *NaN* (*not a number*).

Les NaN peuvent par exemple permettre de définir un pointeur ou une chaîne de caractères au milieu d'un groupe de nombres flottants.

La mantisse est fractionnaire normalisée et un nombre N s'écrit:

$$N = (-1)^{s}\cdot 2^{a}\cdot(\alpha) \tag{2.89}$$

Le format dit en simple précision code l'exposant sous forme biaisée en excédent à $2^7 - 1 = 127.$, dans un champ de 8 bits contenant un exposant codé noté e. On a donc $e = a + 127$.

Seule la partie purement fractionnaire f de la mantisse est codée. La partie entière, valant toujours 1 puisque les nombres sont normalisés, est implicite, et non représentée. On peut donc écrire

$$\alpha = 1,f \tag{2.90}$$

Le format résultant est donné dans la figure 2.131 et permet de représenter des nombres selon l'étendue suivante:

$$\text{plus grand: } 2^{254.-127}\cdot 2 = 2^{128.} = 10^{38} \tag{2.91}$$

$$\text{plus petit (en valeur absolue): } 2^{0-127}\cdot 1 = 2^{-127} = 10^{-38} \tag{2.92}$$

Si $e = 0$ et $f = 0$, on a la représentation du zéro, qui peut être signé.

Si $e = 255.$ et $f = 0$ on a une représentation conventionnelle de l'infini avec son signe.

Si $e = 255.$ et $f \neq 0$ il s'agit d'un non-nombre NaN dont la signification dépend de l'utilisateur.

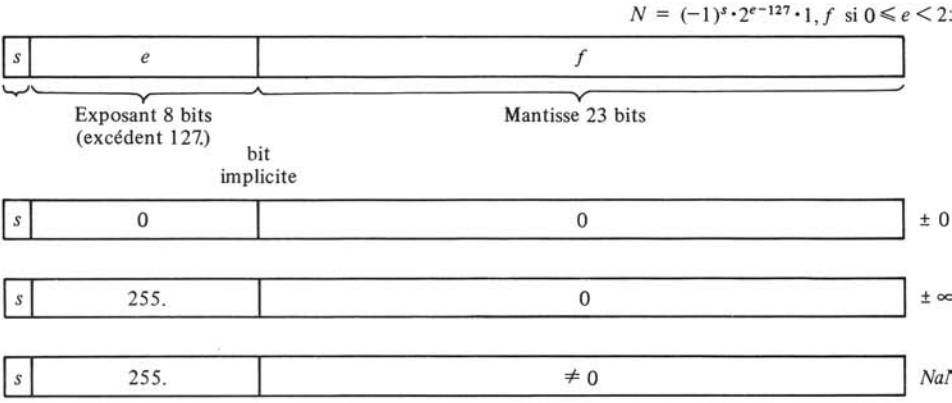


Fig. 2.131

On peut remarquer qu'avec les conventions de ce format flottant, l'ordre des nombres est l'ordre lexicographique, ce qui facilite les comparaisons. Les NaN ne peuvent naturellement pas être comparés, et toute tentative d'effectuer des opérations sur ces objets interrompt l'exécution.

Les différents formats proposés sont donnés dans la figure 2.132 et montrent que dans le format étendu, le bit implicite de la mantisse est rétabli étant donné que ce format est utilisé pour manipuler des nombres non normalisés.

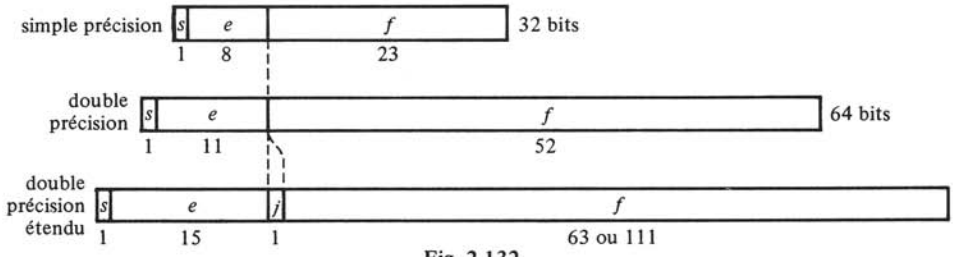


Fig. 2.132

2.8 OPÉRATIONS EN BCD

2.8.1 Avantages du système décimal

Le système décimal restera le système de numérotation utilisé par l'homme. Dans la plupart des ordinateurs, le décimal est converti en binaire et toutes les opérations sont effectuées en binaire. Dans les calculatrices de poche et de bureau, tous les calculs se font en décimal, codé en binaire. Les mini et microordinateurs sont binaires, mais offrent quelques facilités pour effectuer des opérations en décimal directement. Seul le code BCD

(§ 2.2.4) sera étudié dans ce livre. Pour distinguer un nombre abstrait écrit en décimal de sa représentation en BCD, un point est collé à chacun des chiffres du nombre BCD (fig. 2.133). Cette notation est plus condensée que l'écriture binaire et attirera l'attention sur la structure particulière des nombres BCD.

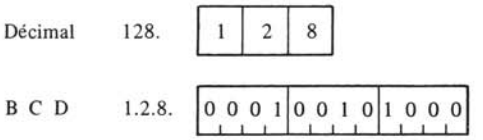


Fig. 2.133

2.8.2 Nombres négatifs

Trois représentations différentes ont été définies pour les nombres négatifs. Elles s'appliquent aux nombres BCD et peuvent conduire à de nombreuses variantes. Dans la représentation signée, le signe peut être soit représenté par un bit de signe indépendant du nombre, soit par un digit de signe. La correspondance entre les signes + et − et deux chiffres ou quartets différents est toutefois arbitraire. Le quartet 0000 peut représenter le signe +, et le quartet 1111 le signe − (fig. 2.134).

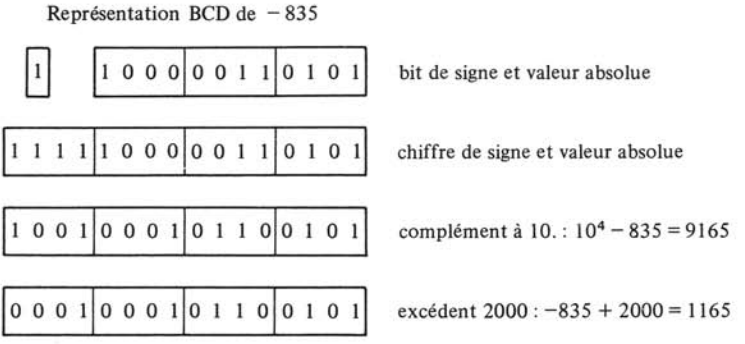


Fig. 2.134

Le chiffre de signe des nombres arithmétiques décimaux est 0 ou 9 (§ 2.4.10). On verra toutefois au prochain paragraphe que cette valeur peut être modifiée. Une représentation en excédent des nombres positifs et négatifs est aussi possible. La valeur de l'excédent dépend de l'application.

2.8.3 Cercle des nombres arithmétiques BCD

Le cercle des nombres arithmétiques, tel qu'il a été défini au paragraphe 2.4.12, est représenté dans la figure 2.135. La définition du signe peut être étendue en convenant que tous les nombres dont le chiffre de signe est 0, 1, 2, 3, 4 sont positifs, et les autres négatifs (fig. 2.136).

En fonction de la convention choisie pour le signe, il faut réaliser le test du signe et des dépassements de capacité (§ 2.4.20). Par la suite, la convention naturelle de la figure 2.135 et le code BCD seront seuls utilisés.

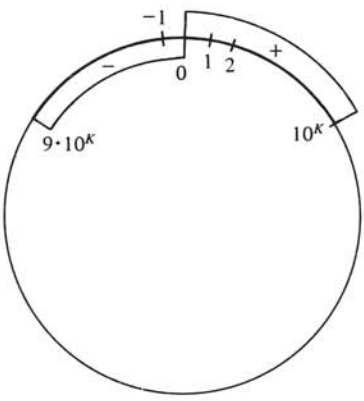


Fig. 2.135

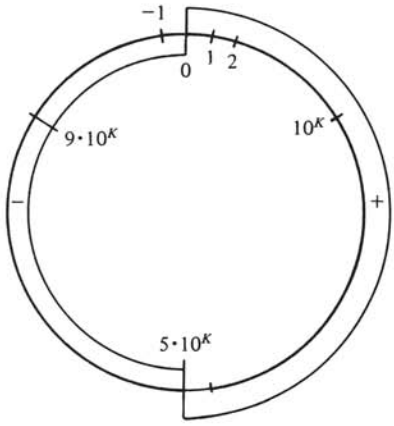


Fig. 2.136

2.8.4 Addition BCD

Un opérateur spécial, appelé *DADD* (la lettre D en préfixe sera utilisée pour caractériser les opérations BCD) permet d'effectuer l'addition de deux nombres BCD, logiques ou arithmétiques (fig. 2.137).

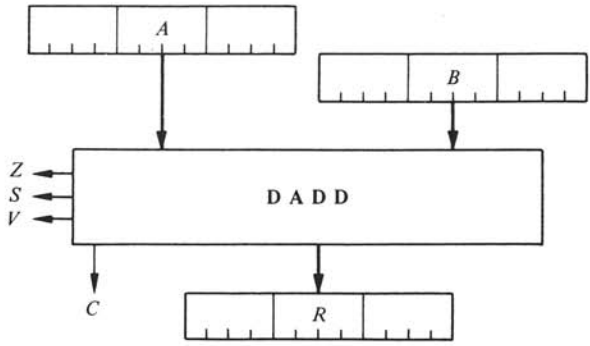


Fig. 2.137

Le rapport *C* signale les dépassements de capacité dans une addition de nombres BCD logiques, et l'indicateur de dépassement *V* doit être utilisé dans les additions de nombres arithmétiques (§ 2.4.20).

■ 2.8.5 Correction décimale

Il est souvent nécessaire d'effectuer une addition BCD avec des opérateurs binaires uniquement. Pour cela, analysons les différences entre la représentation binaire et BCD de l'opération élémentaire d'addition de deux digits.

Une addition de deux chiffres de 0 à 9 et d'un report éventuel fournit un résultat inférieur ou égal à 19. Pour convertir un tel nombre en BCD, il faut, si le résultat est supérieur à 9, ajouter 6 pour tenir compte des 6 configurations de 4 bits qui ne sont pas utilisées pour représenter des chiffres décimaux (fig. 2.138).

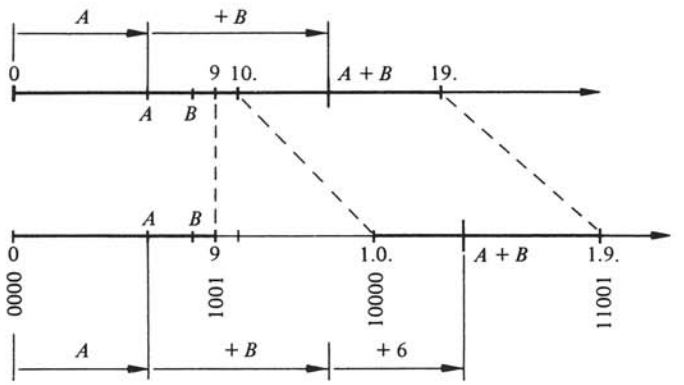


Fig. 2.138

Il est donc possible d'effectuer l'opération en binaire et de corriger le résultat obtenu. Un comparateur et un additionneur conditionnel sont nécessaires dans ce but (fig. 2.139). L'opérateur effectuant cette correction d'un résultat binaire entre 0 et 19. en un nombre BCD formé d'un digit et d'un report de poids 10 s'appelle *DAA* (*decimal adjust after addition*).

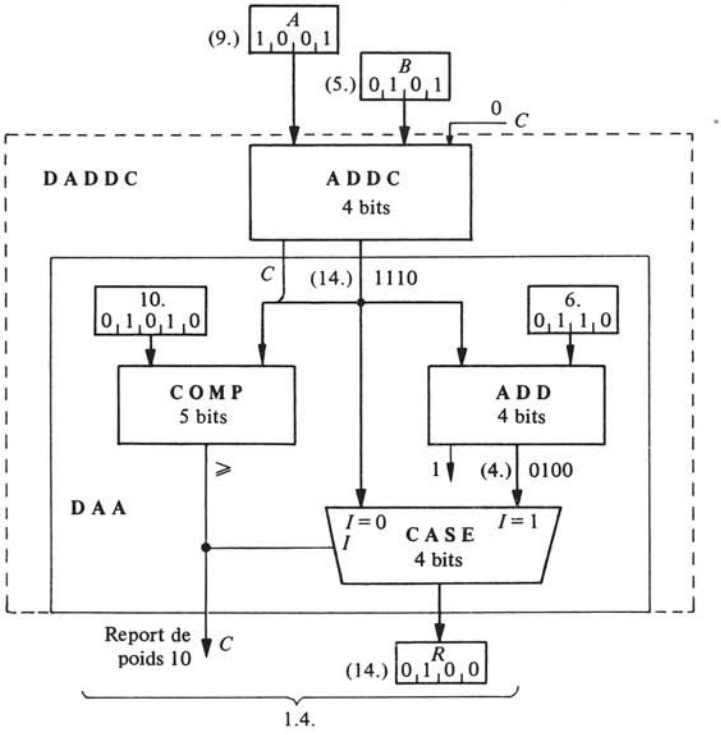


Fig. 2.139

On remarque que cet opérateur utilise le report binaire de l'additionneur 4 bits. Si l'opérateur d'addition binaire additionne 8 ou 16 bits à la fois, la correction décimale

n'est possible que si les reports intermédiaires sont accessibles tous les 4 bits. Dans le cas des microprocesseurs 8 bits, le report intermédiaire est appelé demi-report H (*half-carry*). La figure 2.140 donne le schéma fonctionnel et un exemple d'une opération d'addition BCD 8 bits.

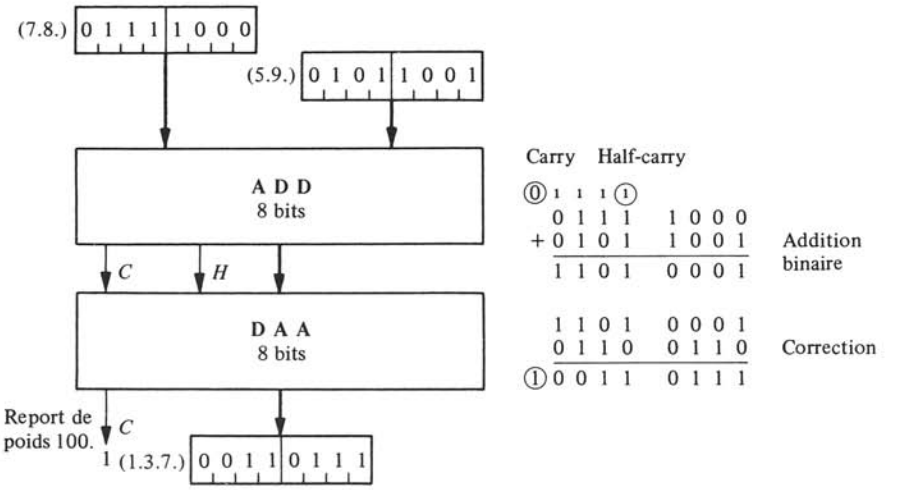


Fig. 2.140

L'indicateur de dépassement V n'est en général pas généré par les opérateurs DAA. Des tests supplémentaires sont nécessaires à la fin de l'opération pour déterminer s'il y a eu dépassement, en fonction de la convention utilisée pour le chiffre de signe.

2.8.6 Exercice

Un cas particulier de l'opérateur d'addition décimale est l'opérateur d'incrémentation décimale *DINC*. Chercher une implémentation de cette opérateur utilisant des opérateurs de remise à zéro CLR (§ 2.5.13).

■ 2.8.7 Soustraction BCD

L'opérateur de soustraction BCD s'appelle *DSUB* et fournit un résultat sous forme de complément à 10 si le 1er nombre est inférieur au second. La réalisation de cette opération à partir d'opérateurs binaires est possible. A nouveau, il faut se ramener à l'opération élémentaire sur un digit. Le résultat de la soustraction de deux chiffres et d'un emprunt éventuel est compris entre -10 et $+9$. Si le résultat binaire est négatif, une correction doit être faite en soustrayant 6. Si le résultat est positif, aucune correction n'est nécessaire (fig. 2.141).

La structure de l'opérateur de correction décimale après soustraction *DAS* (*decimal adjust after subtract*) est donc plus simple que l'opérateur DAA. Le comparateur de la figure 2.139 n'est pas nécessaire. Si le report du soustracteur 4 bits vaut 1, un soustracteur supplémentaire doit retrancher 6. Le report C est le report de poids -10. Au lieu de soustraire 6, il est possible d'ajouter son complément binaire 4 bits, qui vaut 10. La figure 2.142 donne le schéma fonctionnel de ces deux solutions.

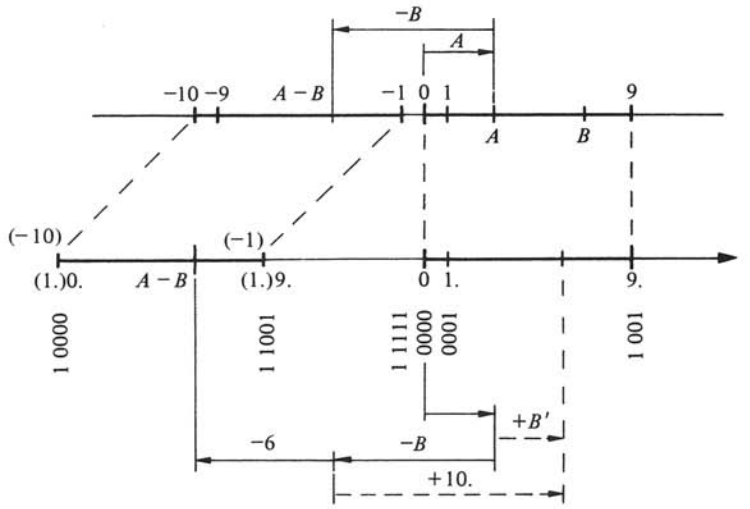


Fig. 2.141

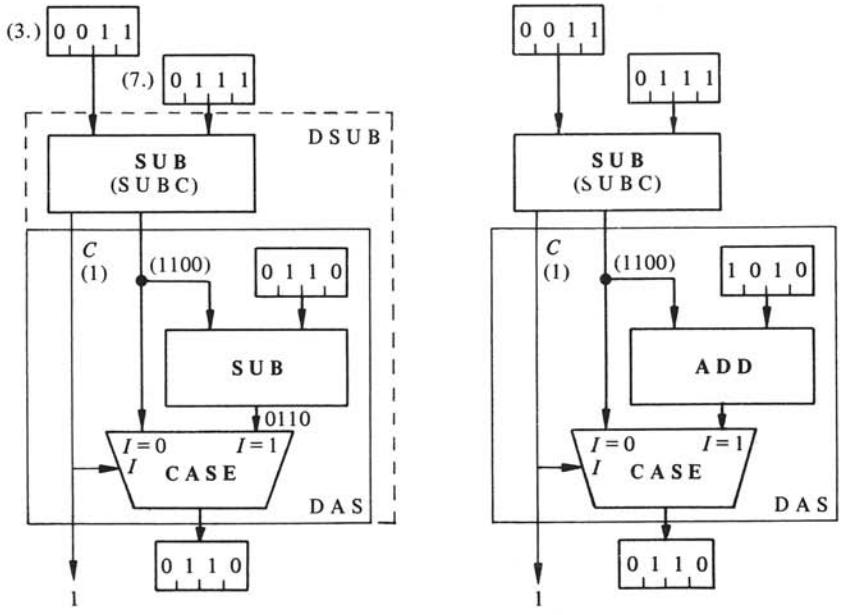


Fig. 2.142

2.8.8 Complément à 9 et à 10

L'opérateur général de complément à 10 (*DNEG*) et deux implémentations découlant des considérations du paragraphe 2.4.24 sont donnés dans la figure 2.143.

L'opérateur *DNOT* calcule le complément à 9. Cet opérateur, qui se décompose en opérateurs élémentaires indépendants agissant sur chaque digit, peut se réaliser de diverses façons. La plus astucieuse consiste à complémenter en binaire et soustraire 6. En effet, le

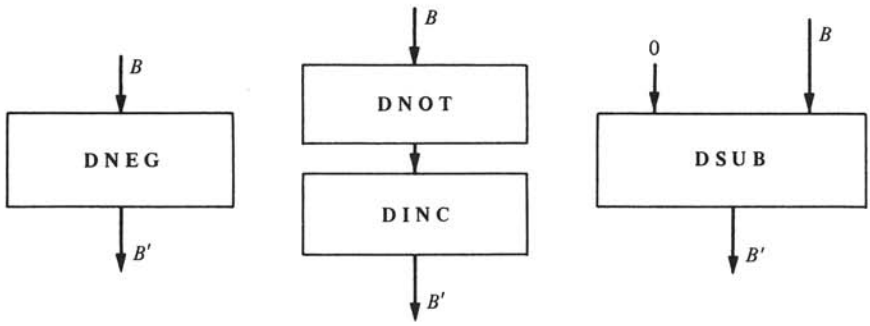


Fig. 2.143

complément d'un digit x est sa différence à 9., et l'on a $9.-x=(15.-x)-6.\ 15.-x$ est le complément binaire 4 bits (fig. 2.144). Au lieu de soustraire 6. on peut ajouter 10., mais dans ce cas les reports de poids 16. ne doivent pas se propager d'un groupe de 4 bits au suivant (fig. 2.145). Avec un soustracteur, aucun report n'est produit car $15.-x \geqslant 6$; un opérateur 8 ou 16 bits peut donc être utilisé.

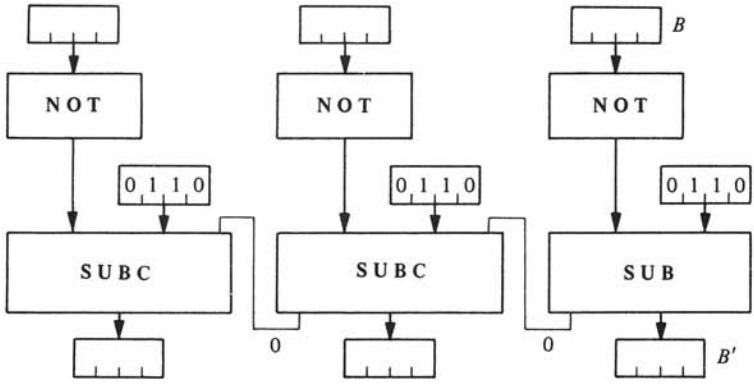


Fig. 2.144

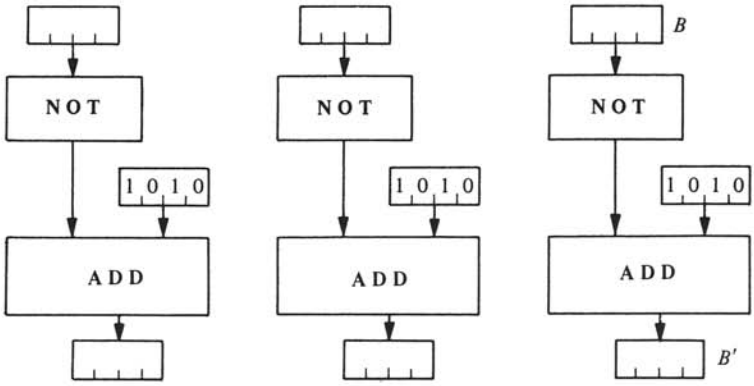


Fig. 2.145

Les figures 2.144 et 2.145 montrent deux décompositions de l'opérateur *DCPL*. La soustraction d'un nombre décimal se ramène à l'addition de son complément à 9 plus 1. La valeur 1 est ajoutée par le report d'entrée de l'additionneur, comme expliqué au paragraphe 2.4.26.

2.8.9 Exemple d'application

Le microprocesseur 8085 dispose de l'instruction DAA, mais pas de DADD, DSUB ni DAS. On remplacera donc

```
DADD A, B   par   ADD   A, B
                  DAA   A

DSUB A, B   par   LOAD  C, A
                  LOAD  A, # 9. * 16. + 9.   ; 9.9. en BCD
                  SUB   A, B                 ; calcule le complément à 9
                  INC   A
                  LOAD  B, A                 ; complément à 10
                  LOAD  A, C
                  ADD   A, B
                  DAA   A

DINC A      par   ADD   A, # 1
                  DAA   A                    ; (ne pas utiliser INC A car
                                             ; la retenue intermédiaire n'est pas
                                             ; initialisée correctement)

DDEC A, B   par   ADD   A, # 9. * 16. + 9.   ; -1 en complément à 10
                  DAA   A
```

2.8.10 Exercice

Utiliser le complément à 9 pour décomposer l'opérateur DSUB de la figure 2.142. Simplifier pour n'avoir finalement que deux opérateurs ACOC par digit.

□ 2.8.11 Double complémentation

La méthode de double complémentation est générale, mais elle présente surtout de l'intérêt pour les nombres BCD. En effet, si A_x est le complément à X de A, X étant un nombre quelconque, on a

$$A - B = X - (X - A + B) = (A_x + B)_x \tag{2.93}$$

Au lieu d'ajouter à A le complément de B, on ajoute à B le complément de A et recomplémente ce résultat.

En BCD, le nombre X est choisi à 15.15.15...15. = H'FFF..F, c'est-à-dire un nombre binaire ne comportant que des 1. X n'est pas un nombre BCD mais le raisonnement ci-dessus est valable pour des nombres quelconques. Le complément à X est simple à calculer, car il est identique au complément binaire. Toutefois, l'opération d'addition du complément ne peut pas se faire avec un additionneur BCD usuel, ni avec un additionneur binaire simple. Le circuit de correction est facile à trouver en considérant l'opération d'addition

sur un digit, et en suivant le raisonnement fait au paragraphe 2.8.5. Le soustracteur élémentaire BCD calcule $a_i - b_i - c_i$ si le résultat est positif (pas de report) et $a_i - b_i - c_i - 6$ si le résultat est négatif. Dans le 1er cas, on peut écrire

$$a_i - b_i - c_i = 15. - (15. - a_i + b_i + c_i) = (a_{ix} + b_i + c_i)_x \qquad (2.94)$$

Dans le second cas

$$a_i - b_i - c_i - 6 = 15. - (15. - a_i + b_i + c_i + 6) = (a_{ix} + b_i + c_i + 6)_x \qquad (2.95)$$

Le report créé par l'addition $a_{ix} + b_i + c_i$ est identique à l'emprunt créé par $a_i - b_i - c_i$,

$$\text{car si } a_i - b_i - c_i < 0,\ a_{ix} + b_i + c_i = 15. - (a_i - b_i - c_i) \geqslant 16. \text{ (report)} \qquad (2.96)$$

$$\text{et si } a_i - b_i - c_i \geqslant 0,\ a_{ix} + b_i + c_i = 15. - (a_i - b_i - c_i) \leqslant 15. \text{ (pas de report)} \qquad (2.97)$$

Les deux cas sont donc caractérisés par le report de l'additionneur calculant $a_{ix} + b_i + c_i$ et le schéma fonctionnel de la figure 2.146 représente bien un soustracteur BCD.

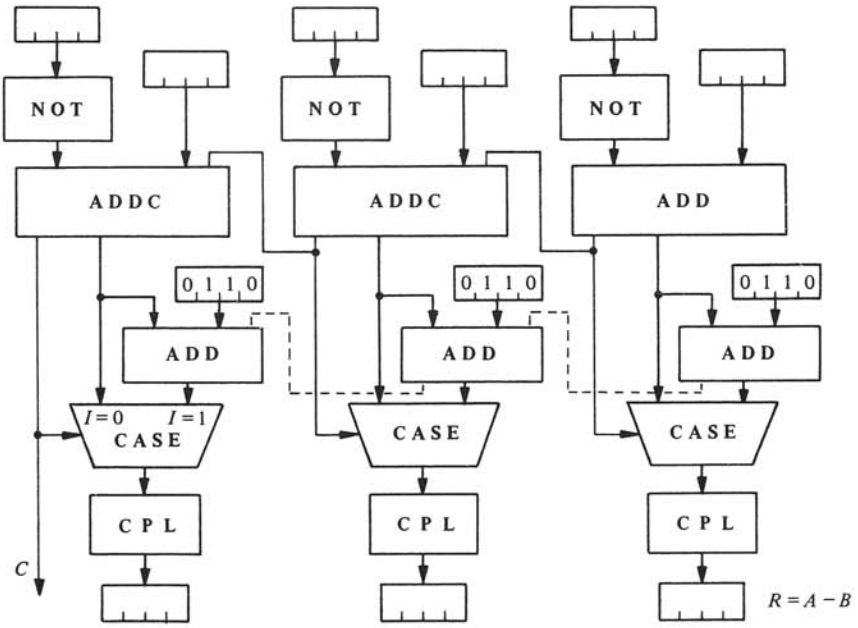


Fig. 2.146

□ 2.8.12 Multiplication BCD

L'opérateur de multiplication vu au paragraphe 2.6.4 peut s'implémenter sans autre en BCD. Les opérateurs de multiplication de 2 digits se laissent facilement réaliser avec une mémoire morte ou une table de référence (§ 4.8.2).

Souvent, un algorithme plus simple est utilisé. Par exemple le produit par un nombre entier petit est souvent obtenu en additionnant le multiplicande un nombre de fois

indiqué par le multiplicateur. L'unité de multiplication partielle par un digit est presque toujours implémentée de cette façon. L'exemple de la figure 2.147 évoque cette façon de procéder et une variante dans laquelle, pour gagner du temps, une suite de plus de 5 additions est remplacée par une suite de moins de 5 soustractions et une correction après décalage. Des tests très complets permettent de remplacer une multiplication par 999 par une seule soustraction et une seule addition.

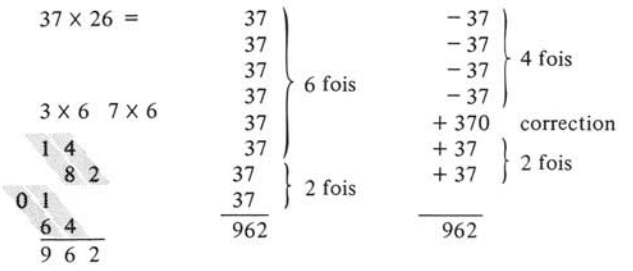


Fig. 2.147

Les tests de dépassement de capacité ne doivent pas être négligés. On trouve dans [38] et [45] différents algorithmes très détaillés.

2.8.13 Exercice

Etudier un algorithme de multiplication BCD tenant compte des poids 1, 2, 4, 8 des bits de chiffres du multiplicateur.

□ 2.8.14 Division BCD

La division BCD se fait toujours par soustraction du diviseur, après alignement du dividende et du diviseur. Une suite de 9 soustractions au plus détermine le chiffre correspondant du quotient. Le test de fin de soustraction ne pouvant se faire que sur un dépassement (résultat négatif), il est nécessaire de rétablir le reste partiel correct avant décalage, ou d'utiliser une méthode plus proche de l'algorithme binaire par comparaison. Plutôt que de rétablir, il est aussi possible d'additionner après décalage (§ 2.6.12).

La figure 2.148 donne un exemple d'opération décimale faite par méthode avec rétablissement et la figure 2.149 sans rétablissement. Le format fixe des nombres n'apparaît pas dans cet exemple; seul le chiffre de signe, testé après chaque opération élémentaire est mis en évidence. La conversion en BCD conserve cette structure et cet algorithme.

□ 2.8.15 Nombres BCD flottants

Dans les calculatrices de poche scientifiques, les nombres sont affichés sous forme flottante. Le format usuel comporte un exposant valant de +99 à −99 et une mantisse de 8-10 chiffres décimaux. La représentation interne de ces nombres peut toutefois être différente, une conversion de format se faisant au moment de l'affichage.

Les opérations sur des nombres BCD flottants se ramènent à des opérations sur les mantisses et les exposants. Les problèmes principaux apparaissent au niveau des détections d'erreurs.

```
41:3 = ?    4 1
          −3
          ---
           1        0
          −3
          ---
(9) 9 8              1
+    3
---------
     1 1   1 −
    − 3
    ---
      8    1 0
    − 3
    ---
      5    1 1
     −3
     --
      2    1 2
    − 3
    ---
(9) 9 9    1 3   Quotient
+    3     =====
---------
     2    Vrai reste
```

Fig. 2.148

```
41:3 = ?  (0) 0 4 1
          −   3
          -------
          (0) 0 1        1
          −   3
          -------
          (9) 9 8 1      2 0
          +     3
          ---------
            (9)8 4       1 9
            +   3
            -------
            (9)8 7       1 8
              + 3
              ---
            (9)9 0       1 7
                3
                -
            (9)9 3       1 6
              + 3
              ---
            (9)9 6       1 5
              + 3
              ---
            (9)9 9       1 4
              + 3
              ---
            (0)0 2       1 3
            Reste        =====
                         Quotient
```

Fig. 2.149

2.9 CHANGEMENTS DE BASE

2.9.1 Problème général

Le problème de la conversion d'un nombre d'une base dans une autre, en particulier de binaire en décimal, n'est pas trivial. Il faut distinguer la base dans laquelle s'effectuent les calculs et la nature des nombres (entiers, fractionnaires, flottants). Ces différents cas sont indépendants et doivent s'étudier séparément. Le passage d'une base donnée à une autre conduit en plus de ces différents cas, à diverses implémentations. Nous ne pourrons pas prétendre être complets dans cette section.

2.9.2 Conventions d'écriture

Considérons de façon générale la conversion d'un nombre abstrait écrit en base p dans son équivalent écrit en base q. Le nombre a une partie entière N et une partie fractionnaire M.

$$N = \sum_{i=0}^{k} a_i p^i = \sum_{i=0}^{k'} b_i q^i \tag{2.98}$$

$$M = \sum_{i=1}^{l} a_{-i} p^{-i} = \sum_{i=1}^{l'} b_{-i} q^{-i} \tag{2.99}$$

Tous les calculs seront effectués en base p. Nous ne considérerons pas le cas où une troisième base est utilisée pour les calculs.

Si toutes les opérations sont faites en base p, il faut définir un moyen pour représenter les chiffres de la base q dans le système de base p. Le code naturel (§ 2.1.9) consistant à représenter ces chiffres par les nombres correspondants en base p sera utilisé. Les chiffres b_j codés en base p, seront notés b_j^* et la base q^*.

Par exemple, si $p = 2$ et $q = 10.$, on retrouve le code BCD et l'on a $q^* = 1010$.

■ 2.9.3 Conversion d'entiers de base q en base p

Partons de l'expression

$$N = \sum_{i=0}^{k} a_i p^i = \sum_{i=0}^{k'} b_i q^i = \sum_{i=0}^{k'} b_i^* q^{*i} \tag{2.100}$$

le nombre étant connu en base q, les chiffres b_i, donc leurs valeurs codées b_i^* sont connues. Il suffit d'effectuer l'opération $\Sigma_{i=0}^{k'} b_i^* q^{*i}$ en base p pour connaître le nombre N en base p, donc avoir tous les coefficients a_i. Une variante de cette méthode permet d'éviter de connaître la valeur des termes q^{*i} et de procéder par récurrence. Ecrivons

$$\begin{aligned} N &= b_{k'}^* \cdot q^{*k'} + \ldots + b_1^* q^* + b_0^* = \\ &= (\ldots(\underbrace{(0 + b_{k'}^*)}_{N_{k'}} \cdot q^* + b_{k'-1}^*) \cdot q^* + \ldots + b_1^*) \cdot q^* + b_0^* \end{aligned} \tag{2.101}$$

$N_{k'}$; $N_{k'-1}$; N_1 ; $N_0 = N$

On a donc

$$N_i = N_{i+1} \cdot q^* + b_i^* \qquad i = k', k'-1, \ldots 0 \tag{2.102}$$

avec

$$N_{k'+1} = 0 \qquad \text{et} \qquad N_0 = N$$

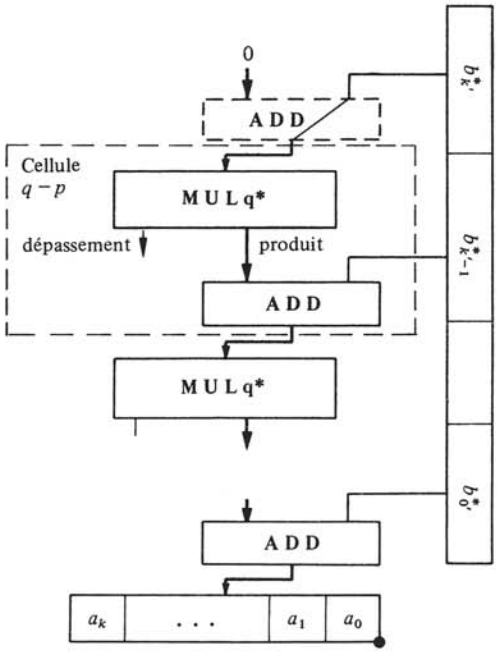


Fig. 2.150

Cette formule, appelée parfois algorithme de Horner conduit au schéma fonctionnel de la figure 2.150. L'opérateur $MULq^*$ est un opérateur à un seul opérande qui multiplie par q^* en base p.

2.9.4 Exercice

Convertir le nombre 0'374 et le nombre hexadécimal H'FC en décimal en effectuant les calculs en décimal.

□ 2.9.5 Conversion de nombres purement fractionnaires de base q en base p

On déduit des formule (2.99) et des considérations du paragraphe précédent que

$$M = b^*_{-1}\cdot q^{*-1} + b^*_{-2}\cdot q^{*-2} + \ldots + b^*_{-l'}\cdot q^{*-l'} = ((\ldots(\underbrace{(0 + b^*_{-l'})}_{N_{-l'}}\cdot q^{*-1} + b^*_{-l'+1})\cdot q^{*-1} + \ldots + b^*_{-1})\cdot q^{*-1} \tag{2.103}$$

$N_{-l'+1}$, $N_{-l'+2}$, $\ldots$, N_0

D'où la formule de récurrence

$$N_{-j+1} = (N_{-j} + b^*_{-j})\cdot q^{*-1} = \frac{N_{-j} + b^*_{-j}}{q^*} \qquad j = l', l'-1, \ldots 1 \tag{2.104}$$

avec

$$N_{-l'} = 0 \qquad \text{et} \qquad N_0 = M$$

Le schéma fonctionnel correspondant est donné dans la figure 2.151. L'opérateur $Div\,q^*$ de division par q^* en base p n'est pas un opérateur simple dans le cas général. Le reste de la divion par q^* est considéré comme erreur d'arrondi et est ignoré.

2.9.6 Exercice

Convertir en décimal le nombre octal 0,52 et le nombre hexadécimal H'0,A8.

■ 2.9.7 Conversion d'entiers de base p en base q

Considérons maintenant le cas où le nombre est connu en base p. Il faut déterminer les b_i de la formule (2.98) dans leur forme codée b^*_i puisque les calculs, donc la représentation des résultats, se fait en base p. Soit donc:

$$N = b^*_{k'}\cdot q^{*k'} + \ldots + b^*_1\cdot q^* + b^*_0 \tag{2.105}$$

Divisons N par q^* selon l'algorithme de division entière (§ 2.6.10); le reste de cette division est inférieur à q^*, et égal à b^*_0, chiffre de poids faible du nombre cherché.

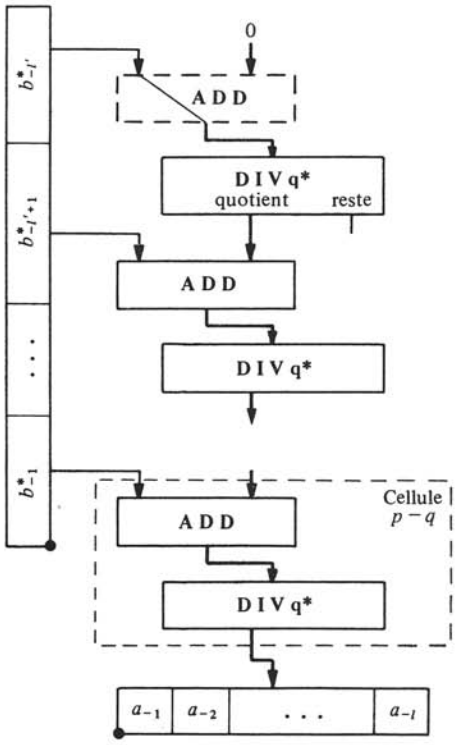


Fig. 2.151

Quotient entier :

$$\left|\frac{N}{q^*}\right| = b^*_{k'} \cdot q^{*k'-1} + \ldots + b^*_1 = N_1 \tag{2.106}$$

Reste : b^*_0.

On peut continuer par récurrence et trouver, par une suite de divisions entières par q^*, tous les chiffres du nombre cherché.

L'opérateur correspondant est représenté dans la figure 2.152.

2.9.8 Exercice

Convertir en base 8 et en base 16. le nombre décimal 250.

□ 2.9.9 Conversion de nombres purement fractionnaires de base p en base q

Nous pouvons écrire

$$M = b^*_{-1} \cdot q^{*-1} + b^*_{-2} \cdot q^{*-2} + \ldots + b^*_{-l'} \cdot q^{*-l'} \tag{2.107}$$

Multiplions en base p le nombre N, connu, par q^*

$$M \cdot q^* = b^*_{-1} + b^*_{-2} \cdot q^{*-1} + \ldots + b^*_{-l'} \cdot q^{*-l'+1} = b^*_{-1} + N_1 \tag{2.108}$$

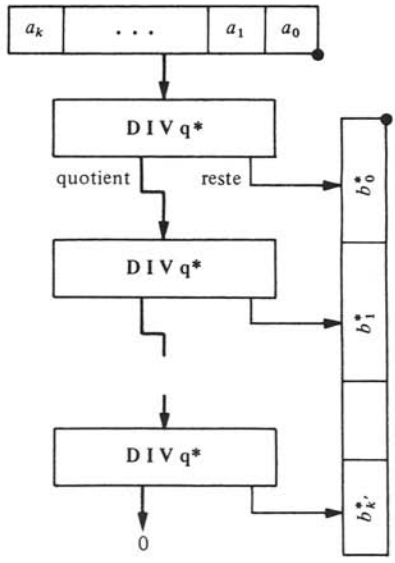


Fig. 2.152

Le résultat est formé d'une partie entière, égale à b^*_{-1}, et d'une partie purement fractionnaire N_1. La partie entière peut être extraite facilement; dans le cas d'un opérateur de multiplication purement fractionnaire, b^*_{-1} est un dépassement de capacité.

Le calcul de $b^*_{-2}, \ldots, b^*_{-l'}$ peut se faire de même par récurrence sur $N_1, \ldots N_{-l'+1}$.

Le schéma fonctionnel de cette décomposition de l'opérateur de conversion est donné dans la figure 2.153.

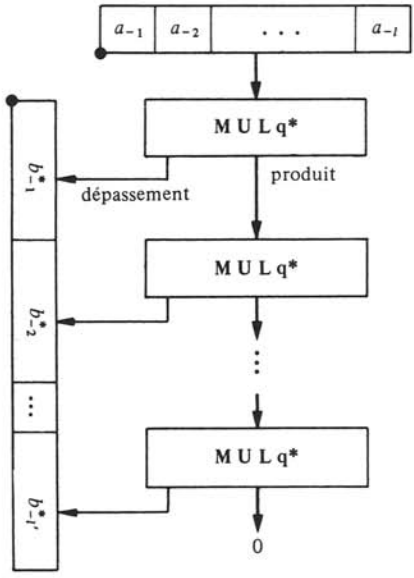


Fig. 2.153

2.9.10 Exercice

Convertir le nombre 0,4 en octal et en hexadécimal en effectuant les calculs en décimal.

2.9.11 Remarque de synthèse

On remarque une grande analogie entre les opérateurs des figures 2.152 et 2.153, ainsi qu'entre 2.150 et 2.151. On peut montrer que deux opérateurs différents permettent d'effectuer les 4 types de conversion [47] et que ces opérateurs se décomposent en cellules élémentaires de conversion. La structure de ces cellules peut être adaptée à des codes différents du code naturel [38].

2.9.12 Conversion entre bases puissances l'une de l'autre

La conversion de nombres est particulièrement simple si la base q est une puissance de la base p.

Si par exemple $q = p^z$ et si y est tel que $y \cdot z = k$, on a

$$N = a_k \cdot p^k + \ldots + a_1 \cdot p + a_0 = (a_{yz+z-1} \cdot p^{z-1} + \ldots + a_{yz}) \cdot p^{yz} + \ldots + (a_{2z-1} \cdot p^{z-1} + \ldots + a_z) \cdot p^z + (a_{z-1} \cdot p^{z-1} + \ldots + a_0) \qquad (2.109)$$

Chaque parenthèse est un nombre inférieur à p^z (§ 2.1.3) donc un chiffre en base q. On peut donc écrire

$$N = b_y \cdot q^y + \ldots + b_1 \cdot q + b_0 \qquad (2.110)$$

La conversion consiste donc simplement dans un regroupement des chiffres du nombre en base p, après extension éventuelle du champ par des zéros non significatifs pour que la longueur du nombre soit multiple de z.

Une démonstration extrêmement similaire peut être faite pour les nombres purement fractionnaires, et pour la conversion en sens inverse. La figure 2.154 évoque cette opération pour $z = 5$. L'opérateur, trivial, n'est pas représenté.

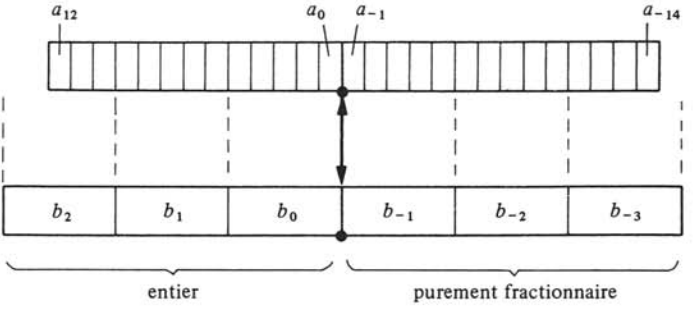


Fig. 2.154

La facilité de ce type de conversion justifie l'emploi de l'octal ou de l'hexadécimal pour représenter du binaire. Un raccourcissement sensible des nombres en résulte, et le plus grand nombre de chiffres différents n'est pas gênant pour l'opérateur humain. Une base 32, avec pour chiffres 0, ..., 9, A, ..., T, U pourrait à ce point de vue être très efficace, mais la conversion de tête serait trop difficile.

2.9.13 Exercice

Vérifier les résultats des exercices 2.9.8 et 2.9.10 en convertissant les nombres octaux et hexadécimaux en binaire.

2.9.14 Conversion de binaire en décimal avec calculs en décimal

La conversion de binaire en décimal est particulièrement facile pour l'opérateur humain, si les calculs sont faits en décimal. La machine, elle, préfère le binaire, et il n'y a guère d'intérêt à demander à un ordinateur de convertir un nombre en binaire, si cet ordinateur dispose de circuits permettant de calculer en décimal.

Il est bon, pour vérifier des opérations faites par la machine, de savoir convertir des nombres binaires rapidement. Quatre algorithmes et des variations découlent des considérations générales des paragraphes précédents [47]. Le tableau de la figure 2.156 illustre les algorithmes principaux par des exemples. La liste des puissances de 2 donnée en annexe (§ 7.1.1) facilite ces conversions.

2.9.15 Exercice

Convertir en binaire ou en décimal en choisissant chaque fois l'algorithme le plus efficace:

0,03125. 1978. 0,1. B'11010111 B'0,1101 B'1101,011.

2.9.16 Conversion de décimal en binaire avec calculs en binaire

Etant donné la symétrie des algorithmes de conversion, les mêmes algorithmes se retrouvent lorsque les calculs sont effectués en binaire, mais peuvent sembler très différents de par la différence apparente entre par exemple une division binaire et une division décimale.

Les chiffres décimaux calculés par les algorithmes binaires apparaissent dans le code naturel, le code BCD. Ainsi, par exemple, la conversion de décimal en binaire d'un nombre fractionnaire nécessite une suite de divisions par 1010 = 10. et correspond à la case en bas à gauche de la figure 2.155, ainsi que les opérations à effectuer pour convertir le nombre décimal 0,24.

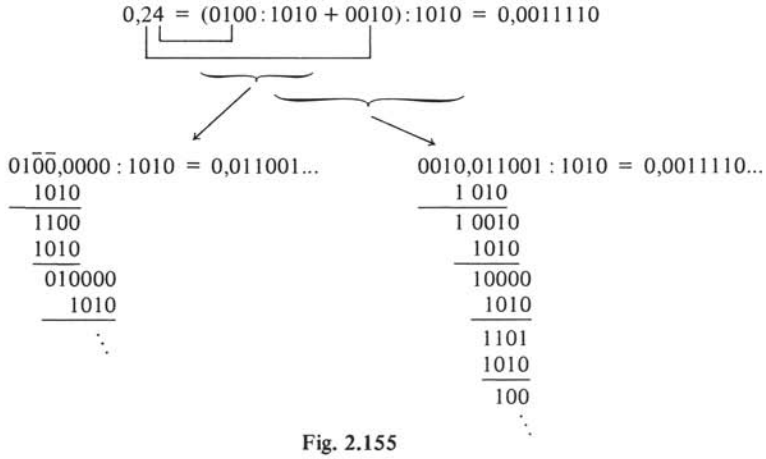


Fig. 2.155

Calculs en décimal

Nombres entiers

binaire → décimal

Somme des poids

1 1 0 1 0 0 1 = 64. + 32. + 8. + 1 = 105.
64 32 16 8 4 2 1

Calcul itératif par addition et dédoublement

1 0 1 1 0 =

((((0 + 1)·2 + 0)·2 + 1)·2 + 1)·2 + 0

1 → 2 → 5 → 11 → = 22

Disposition pratique

Nombre binaire	
1	1
0	2
1	5
1	11.
0	22.

·2 et + ; Résultat décimal

décimal → binaire

Extraction des poids

325. = 1 0 1 0 0 0 1 0 1

256. + 0 + 64. + 0 + 0 + 0 + 4 + 0 + 1

325. − 256. = 69. ; 69. − 64. = 5. ; 5 − 4 = 1 ; 1 − 1 = 0

Calcul itératif par division par 2

105. : 2 = 52. reste 1 (b_0)
52. : 2 = 26. reste 0 (b_1)
26. : 2 = 13. reste 0
13. : 2 = 6 reste 1
6 : 2 = 3 reste 0
3 : 2 = 1 reste 1
1 : 2 = 0 reste 1 (b_6)

Disposition pratique

Nombre décimal — 105. = 1101001

	Résultat binaire
105.	1.
52.	0
26.	0
13.	1
6	0
3	1
1	1
0	

: 2 → quotient ; reste

Nombres fractionnaires

binaire → décimal

Somme des poids

0.1101 = 0,5 + 0,25 + 0,0625 = 0,8125
1 1 1 1
2 4 8 16

Calcul itératif par division par 2

0,1101 = (((1 : 2 + 0) : 2 + 1) : 2 + 1) : 2

0,5 → 0,25 → 1,25 : 2 = 0,625 → 1,625 : 2 = 0,8125

Disposition pratique

Nombre binaire	
1	0,5
0	0,25
1	0,625
1	0,8125
0	

+ et : 2 ; Résultat décimal

décimal → binaire

Extraction des poids

0,4 = 0,011001...

0 + 0,25 + 0,125 + 0 + 0 + 0,015625...

0,4 − 0,5 = 0,15 ; 0,15 − 0,25 = 0,025 ; 0,025 − 0,125 = 0,016... ; 0,016... − 0,0625 = 0,009... ; 0,009... − 0,03125 = ...

Calcul itératif par multiplication par 2

0,4 × 2 = 0,8
0,8 × 2 = 1,6
0,6 × 2 = 1,2
0,2 × 2 = 0,4
0,4 × 2 = 0,8

Disposition pratique

Nombre décimal

Résultat binaire	
0	4
0	8
1	6
1	2
0	4
0	8
1	6
⋮	⋮

·2 → partie entière

Fig. 2.156

2.9.17 Remarque sur la précision des résultats

La conversion d'un nombre fractionnaire est en général un nombre fractionnaire périodique. Même si la conversion "finit", il n'est pas intéressant de donner au résultat une précision supérieure à la donnée. Si la précision absolue décimale est de 10^{-k}, c'est-à-dire que les nombres décimaux ont k décimales, le nombre x de décimales binaires (longueur de la partie purement fractionnaire binaire) doit être telle que $2^{-x} = 10^{-k}$.

D'où $x/k = \log_2 10 \cong 3{,}32$. On retrouve le résultat établi au paragraphe 2.2.1 concernant le rapport des longueurs des grands nombres binaires et décimaux. Il est donc inutile, dans une conversion binaire, de conserver plus de 3,3 fois plus de chiffres significatifs après la virgule qu'en décimal.

□ 2.9.18 Conversion de nombres flottants

La conversion binaire – décimale d'un nombre flottant nécessite d'une part la conversion de la mantisse, puis une correction par multiplications ou divisions successives par l'exposant. Par exemple, pour convertir le nombre décimal $0{,}25.\cdot 10^2$ en binaire, on convertit tout d'abord la mantisse : $0{,}25. = 0{,}01_B = 0{,}1_B \cdot 2^{-1}$, puis on multiplie par 10^2 en binaire flottant : $0{,}1\cdot 2^{-1}\cdot 1010 = 0{,}101\cdot 2^2$, $0{,}101\cdot 2^2\cdot 1010 = 0{,}11001\cdot 2^5$.

Chaque opération est suivie d'une normalisation, et l'accumulation des erreurs d'arrondi au cours de l'opération implique l'utilisation de décimales supplémentaires pendant le calcul [48].

CHAPITRE 3

ARCHITECTURES DES CALCULATRICES ET ORDINATEURS

3.1 CALCULETTES

3.1.1 Introduction

Historiquement, les calculatrices ont eu comme premier objectif de faciliter l'exécution des opérations arithmétiques. Le besoin de répéter celles-ci a conduit à développer les possibilités de programmation, qui elles-mêmes impliquent de bien définir les moyens d'accès à l'information, par un adressage efficace de la mémoire.

Le plan de ce chapitre suit cette démarche et s'appuie sur la structure des calculettes, disponibles depuis 1970 et des microprocesseurs, disponibles depuis 1974, pour introduire les principaux concepts et parvenir progressivement aux caractéristiques les plus avancées que l'on trouve dans les gros ordinateurs et les microprocesseurs les plus récents.

3.1.2 Calculatrice binaire simplifiée

La machine la plus simple que l'on puisse imaginer est une calculatrice binaire avec introduction et affichage des résultats en binaire. Elle peut être réalisée avec deux registres et un additionneur-soustracteur série ou parallèle.

La figure 3.1 donne le schéma bloc d'une architecture à deux registres ne permettant que l'addition et la soustraction.

Les deux registres sont appelés respectivement *accumulateur* (A) et *registre d'entrée* (E). Chaque nombre à additionner ou à soustraire est introduit dans le registre

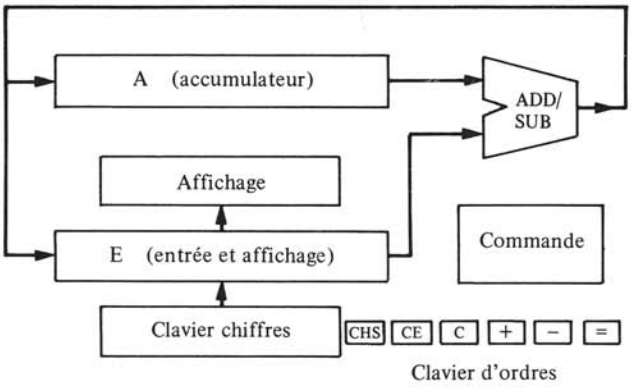


Fig. 3.1

d'entrée, puis combiné avec le contenu de l'accumulateur par passage au travers de l'unité arithmétique. Le résultat est transféré à la fois dans les deux registres, et dès qu'un nouveau chiffre est introduit dans le registre d'affichage, le résultat précédent est effacé de ce registre. Il pourrait paraître plus simple de ne transmettre le résultat qu'à l'accumulateur, et d'effacer le registre E d'entrée et d'affichage à la fin de l'opération, mais ceci coûterait un 2ème affichage sur l'accumulateur, et serait gênant pour les résultats négatifs.

La représentation des nombres négatifs crée le premier problème des calculatrices, et la première ambiguïté due à la différence entre le signe négatif d'un nombre et l'opérateur de soustraction.

Considérons tout d'abord l'exemple d'un résultat négatif. Comme il a été vu au paragraphe 2.4.3, le résultat d'une soustraction binaire est de façon naturelle un nombre en complément à 2. Pour affichage sous forme de nombre signé, ce résultat doit être complémenté, et un signe moins doit être affiché devant la valeur absolue. Pour les calculs ultérieurs, il y a tout avantage à conserver le résultat sous forme complémentaire dans l'accumulateur. Le calcul du complément des résultats négatifs peut se faire par un deuxième passage à travers l'unité arithmétique, effectuant l'opération $B \leftarrow 0 - B$. Des aiguillages appropriés permettent de distinguer l'opération d'addition-soustraction et l'opération de complémentation (exercice 3.1.3).

Le second problème est l'introduction du signe d'un nombre. Le signe peut être considéré comme faisant partie du nombre, et introduit spécialement avec une touche "signe −", qui en fait est généralement une touche de changement de signe [CHS] ou [+/−], afin de permettre la correction du signe. L'opération d'addition ou de soustraction de ce nombre signé est ensuite déclenchée par une touche [+] ou [−].

Une autre approche consiste à considérer que l'on introduit toujours la valeur absolue d'un nombre, et que le signe est défini par l'opération qui suit. Ceci revient à identifier l'addition d'un nombre négatif et la soustraction d'un nombre positif. Cette approche est peu utilisée, car elle ne permet pas la multiplication ou la division par des nombres négatifs.

La touche [=] est traditionnellement utilisée pour déclencher l'opération. Dans le cas de l'addition et de la soustraction, le [+] et le [−] jouent en fait le même rôle; parfois les touches [=] et [+] sont combinées en une seule touche [±]. Ces problèmes d'ordre de touches seront repris au paragraphe 3.1.5.

3.1.3 Exercice

Ajouter au schéma-bloc de la figure 3.1 deux aiguillages et une ligne de test de signe de façon à pouvoir afficher les résultats négatifs en représentation signée, tout en conservant la représentation complémentaire dans l'accumulateur. Définir les opérations à effectuer pour effacer un nombre en cours d'introduction (touche [CE] CLEAR ENTRY) et pour remettre à zéro toute la machine (touche [C] CLEAR).

3.1.4 Calculettes décimales

Le principe d'une calculette décimale n'est pas différent de celui d'une calculatrice binaire. Un code décimal, usuellement le BCD, est les opérateurs associés doivent être utilisés (sect. 2.8). Les 6 codes 4 bits non utilisés pour coder les chiffres peuvent repré-

senter sur l'affichage le point décimal, le signe moins, et des signes spéciaux utilisés en cas d'erreur, décharge des piles, etc.

Le format des nombres est de plus en plus exceptionnellement limité à des nombres en virgule fixe. La virgule est *libre*, *présélectionnable* ou *flottante*. Dans le premier cas, les nombres sont alignés sur la gauche de l'affichage, et la virgule occupe une position qui dépend de la grandeur du nombre; c'est le mode usuel lors de l'introduction des données. La virgule présélectionnable correspond à une virgule fixe, avec possibilité de choisir avec un levier ou des manipulations de touches le nombre de chiffres après la virgule. Les unités, dizaines, etc. occupent toujours les mêmes positions sur l'affichage.

La virgule flottante donne une mantisse normalisée (souvent comprise entre 1 et 10) et un exposant. Une variante dite *virgule flottante d'ingénieur* n'affiche que des exposants multiples de trois, avec une mantisse normalisée comprise entre 1 et 1000.

L'introduction des chiffres se fait dans l'ordre naturel, en commençant par les poids forts et en introduisant la virgule avant la partie fractionnaire. L'introduction de l'exposant doit être précédée de la frappe d'une touche spéciale. En cas d'erreur de frappe, le nombre entier doit être retapé.. Il est facile de prévoir que des modèles futurs offriront de plus grandes facilités pour la récupération des erreurs de l'utilisateur. Une touche d'opération définit la fin de l'introduction d'un nombre. Le résultat est affiché jusqu'à une nouvelle introduction d'un chiffre.

La représentation interne des nombres dans les calculatrices est généralement décimale (code BCD) et flottante normalisée (§ 2.8.15). Une conversion est faite à chaque affichage, et les opérations de calcul sont décomposées en séquences d'opérations sur les chiffres décimaux, sous contrôle d'une unité de commande microprogrammée. Les fabricants de calculettes sont très discrets sur le détail des techniques et représentations qu'ils utilisent, mais il est clair que les modèles les plus récents ont une architecture générale inspirée de celle des microprocesseurs avec un programme qui exécute les fonctions de la calculatrice.

3.1.5 Notation infixée des opérations

L'alternance opérande – opération est une caractéristique de la règle usuelle d'écriture des opérations arithmétiques, dite *notation infixée* ou *notation algébrique.* L'opérateur est écrit entre les 2 opérandes sur lesquels il agit, et un signe égal indique que tout a été préparé et que l'opération peut être effectuée. Lorsqu'il y a plus de deux opérandes, des parenthèses ou des règles de priorités précisent l'ordre dans lequel les opérations doivent être effectuées.

Par exemple, $3 + 2 * 4$ est implicitement équivalent à $3 + (2 * 4)$, et non pas à $(3 + 2) * 4$ qui exige la présence de parenthèses.

La règle usuelle de priorité des opérations algébriques définit trois niveaux d'opérations. La multiplication et la division ont priorité sur l'addition-soustraction. L'exponentiation a priorité sur la multiplication. Ainsi $3 + 2 * 4^5$ est équivalent à $3 + (2 * (4^5))$. Des parenthèses sont nécessaires lorsque l'ordre des opérations doit être différent, par exemple si l'on veut calculer $((3 + 2) * 4)^5$. Les règles de parenthèses et de priorités compliquent passablement la structure des calculettes.

Considérons par exemple une opération aussi simple que $(3 + 2) * (4 + 5)$. Le résultat de la 1ère opération $(3 + 2)$ doit être mémorisé pendant que l'on effectue

(4 + 5). Si la calculatrice n'a que le registre accumulateur et le registre d'affichage mentionnés précédemment, le résultat partiel doit être copié sur un bout de papier et rechargé pour être multiplié avec le deuxième résultat partiel. Il va de soi que la plupart des calculatrices ont un, voire plusieurs registres auxiliaires permettant de sauver et récupérer un résultat intermédiaire par action sur des touches adéquates. L'utilisation de touches parenthèses implique une logique compliquée et un nombre relativement élevé de registres auxiliaires invisibles de l'utilisateur. Leur nombre fixe le nombre de niveaux de parenthèses que l'on peut utiliser dans une expression.

3.1.6 Notation préfixée et suffixée

Le problème des parenthèses peut être élégamment évité grâce à une notation inventée par Lukasiewicz en 1950. Cette notation consiste à donner l'opération avant les opérateurs sur lesquels elle agit (*notation polonaise directe* ou *préfixée*). Le plus souvent, l'opération est donnée après les deux opérateurs sur lesquels elle agit (*notation polonaise inverse* ou *suffixée*). Il en résulte une très grande simplification dans l'implémentation des opérations complexes. L'image d'une *pile* de feuilles sur lesquelles sont inscrits chaque fois un opérande ou un résultat facilite la compréhension de la notation polonaise inverse. Chaque opération remplace les deux nombres au sommet de pile par le résultat. La complexité des opérations traitées ne dépend que de la hauteur maximum de la pile.

La figure 3.2 montre comment les opérations (3 + 2) * 4 et 3 + 2 * 4 sont traitées par la notation suffixée. Le signe Δ sépare les opérandes lorsqu'il n'y a pas d'opérateur. Opérateurs et opérandes peuvent être tous préparés sur la pile avant le début de l'opération; pour l'exécution, la pile est alors explorée par le bas. Dans une calculette, les opérations s'effectuent au fur et à mesure.

Les calculettes qui utilisent la notation suffixée ont un clavier très similaire à celles qui utilisent la notation infixée, mais disposent d'une touche [ENTER] permettant de placer un opérande sur la pile. Cette touche remplace la touche [=], qui n'est plus nécessaire.

Une pile de 4 registres appelés X, Y, Z, T est très souvent utilisée dans les calculettes et dans les circuits de calcul spécialisés. L'introduction d'un nombre dans la pile se fait toujours dans le registre X, qui est le registre d'affichage, avec décalage automatique du contenu des autres registres (empilage). La figure 3.3 représente les deux opérations choisies au début de ce paragraphe avec cette convention. Le *sommet de la pile* (*top of stack*) c'est-à-dire le dernier opérande introduit ou le dernier résultat calculé, se trouve maintenant vers le bas dans cette figure. Si l'on introduit plus de 4 opérandes, les premiers opérandes introduits sont perdus.

A noter que dans la figure 3.3, l'ordre des éléments de la pile est inverse de celui de la figure 3.2. Ces deux représentations sont par ailleurs très différentes, puisque dans la figure 3.3, les éléments se déplacent dans la pile.

3.1.7 Ambiguïté du signal égal

Le signe égal, tout comme le signe moins, a différentes significations qu'il faut bien distinguer.

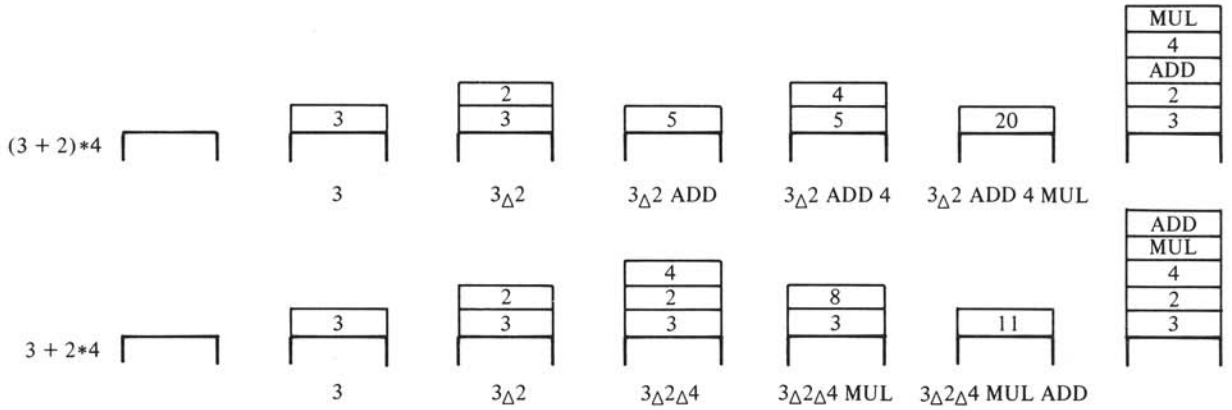

(3 + 2)*4
3
2
3
5
4
5
20
MUL
4
ADD
2
3
3
3△2
3△2 ADD
3△2 ADD 4
3△2 ADD 4 MUL
3 + 2*4
3
2
3
4
2
3
8
3
11
ADD
MUL
4
2
3
3
3△2
3△2△4
3△2△4 MUL
3△2△4 MUL ADD

Fig. 3.2

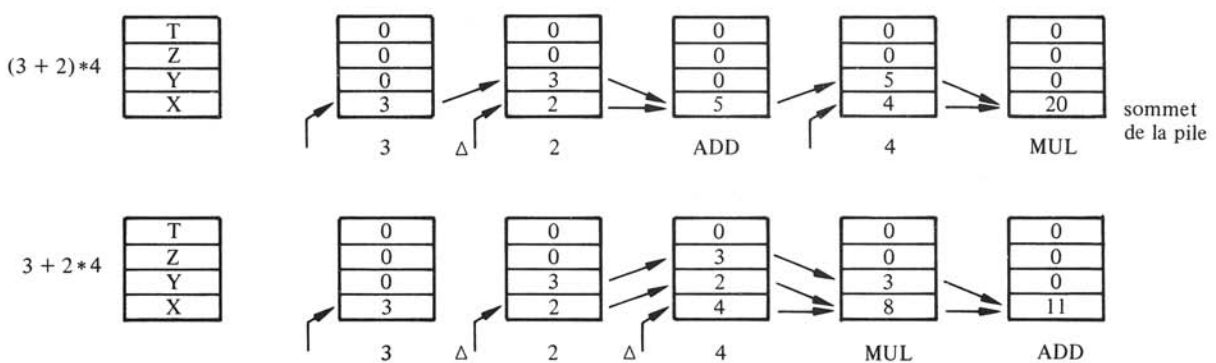

(3 + 2)*4
T
Z
Y
X
0
0
0
3
0
0
3
2
0
0
0
5
0
0
5
4
0
0
0
20
3
△
2
ADD
4
MUL
sommet
de la pile
3 + 2*4
T
Z
Y
X
0
0
0
3
0
0
3
2
0
3
2
4
0
0
3
8
0
0
0
11
3
△
2
△
4
MUL
ADD

Fig. 3.3

- En arithmétique, $3 + 2 = 5$ signifie que le résultat de l'opération $3 + 2$ donne la valeur 5. La touche | = | des calculatrices utilisant la notation algébrique correspond à cette fonction.
- En algèbre, $x = a + b$ signifie que la valeur x s'obtient en additionnant les valeurs de a et b. Une flèche renversée ou le signe "deux points égal" devraient être utilisés dans ce cas ($x \leftarrow a + b$ ou $x := a + b$), mais la pratique n'en est pas encore courante dans l'enseignement secondaire.
- Le signe égal est aussi utilisé pour la comparaison : $x = a + b$? La réponse est binaire : vrai ou faux. Plusieurs types de comparaisons sont possibles (sect. 2.5).
- Lorsqu'un registre reçoit le résultat d'une opération effectuée par un additionneur à partir des valeurs mémorisées dans deux registres A et B, il peut être tentant d'écrire $x = A + B$. Cette notation est à éviter, car elle est choquante dans le cas où le registre recevant le résultat est par exemple le registre contenant le premier opérande, comme c'est souvent le cas avec les calculatrices. Il est choquant d'écrire $A = A + B$, d'où la notation $A := A + B$ utilisée dans la plupart des langages évolués d'ordinateur. La notation ADD A, B est souvent utilisée dans ce livre pour cette opération. C'est en fait une sorte de notation préfixée limitée à deux opérandes qui suppose implicitement que le premier registre opérande mentionné reçoit le résultat. L'utilisation des trois lettres ADD pour caractériser l'addition se justifie pour des raisons de cohérence de notation, le nombre de signes arithmétiques disponibles n'étant pas suffisant pour décrire tous les opérateurs rencontrés.

3.1.8 Registres auxiliaires

L'utilisateur d'une calculatrice simple éprouve rapidement le besoin de disposer de registres auxiliaires pour mémoriser par exemple un résultat partiel sans avoir à le recopier sur une feuille de papier. L'opération de transfert est définie par des touches spéciales. S'il y a plusieurs registres, ils sont numérotés et le numéro du registre suit la touche caractérisant le transfert.

En général, le transfert se fait entre le registre concerné et le registre d'entrée/affichage, celui-ci n'étant pas explicité. On définit les touches [STORE], [RECALL] pour transférer un registre auxiliaire, comme si le centre de l'univers était le registre d'entrée/affichage. Prenons l'exemple de la figure 3.4 avec 10 registres auxiliaires en plus de la pile. Pour transférer le contenu du registre R3 dans le registre R1, deux transferts successifs sont nécessaires, et le registre d'affichage est modifié par cette double opération. Une opération de transfert direct, qui charge R1 par R3 ("LOAD") serait plus efficace; elle exige naturellement de préciser deux opérandes, comme pour une addition. La notation *LOAD R1, R3* exprime ce transfert (§ 2.5.14). Certains fabricants préfèrent noter ce même transfert MOVE R3, R1; d'autres commettent l'erreur d'écrire pour la même opération MOVE R1, R3.

La permutation de deux registres est une opération très pratique. Contrairement au transfert, elle ne détruit pas l'un des deux registres. Cet échange est une opération à 2 opérandes. Ces deux opérandes sont caractérisés par les noms (ou numéros) des registres permutés. Souvent, l'échange ne peut se faire qu'avec le registre d'affichage X et la notation EX R3 est utilisée en place de EX X, R3 pour indiquer la permutation de R3 et du registre d'affichage X.

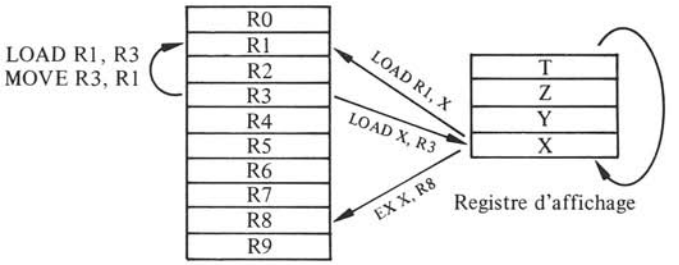


Fig. 3.4

Une *permutation circulaire* des registres, ou *rotation*, est également possible. Cette opération se trouve en particulier dans les calculatrices en notation polonaise utilisant une pile de registres pour les opérandes (fig. 3.4). La rotation permet d'amener successivement tous les registres dans le registre d'affichage, ou d'amener deux registres consécutifs dans les deux positions sur lesquelles les opérations sont effectuées.

L'introduction d'un nombre sur la pile peut également être considérée comme une opération particulière de transferts de registres.

Un avantage important d'une pile est qu'il n'y a pas d'opérande à mentionner. L'accès de certains registres de la pile peut par contre être plus difficile, et nécessiter un nombre d'opérations (échanges, rotations) plus grand que le transfert direct. En fait, les deux modes sont complémentaires, et la possibilité de pouvoir faire les deux types d'opérations est idéale.

3.2. CALCULETTES PROGRAMMABLES

3.2.1 Notion de programme

La séquence d'actions sur les touches d'une calculette peut être mémorisée et répétée automatiquement par la suite. Une touche ou un interrupteur commute dans le mode appelé "programme" ou "apprentissage" qui enregistre le code des touches pressées. Ce code est par exemple un nombre décimal inférieur à 99, permettant de coder autant de touches ou combinaisons de touches, correspondant chacune à une instruction pour la machine. Une *mémoire programme* de 50 à 1000 mots ou *pas* mémorise le programme. Au fur et à mesure de son introduction, le numéro de la position mémoire chargée est affiché. Des touches additionnelles permettent la correction du programme, par avance ou retour en arrière, et réintroduction de la touche correcte.

Le programme peut parfois ne pas contenir des nombres. Les constantes et naturellement les variables doivent être mémorisées dans des registres, et une touche spéciale appelée par exemple RUN/STOP permet de placer dans le programme une instruction d'arrêt laissant à l'utilisateur le loisir d'introduire le nombre dont le programme a besoin pour la suite de son calcul.

Chaque nombre qui doit être introduit en cours de programme doit être suivi par l'action sur une touche "CONTINUE", afin de poursuivre l'exécution du programme là où elle avait été arrêtée par une instruction d'arrêt. Le programme peut être exécuté autant de fois que l'utilisateur le désire, en déclenchant son exécution par une touche [START].

3.2.2 Exercice

Ecrire le programme pour calculer l'équivalent décimal d'un nombre binaire. L'algorithme de multiplication par 2 et additions a été vu au paragraphe 2.9.14. L'introduction de chaque chiffre binaire doit être suivie de l'action sur la touche [RUN/STOP].

3.2.3 Structure interne

Dans son principe général, une calculatrice programmable se construit par adjonction à une calculatrice d'une mémoire de programme et d'un *compteur ordinal* ou *compteur d'adresses* (*program counter* PC) (fig. 3.5) dont le rôle est de définir le numéro de la position mémoire contenant l'instruction à mémoriser ou exécuter. L'interrupteur ou la bascule interne définissant le mode PROGRAM commande des aiguillages (multiplexeurs et démultiplexeurs (sect. V. 1.5)) modifiant l'effet du clavier et de l'affichage. En passant en mode programme, le PC est mis à zéro et les actions du clavier sont mémorisées dans l'ordre de leur introduction. Le PC est un compteur de 2 ou 3 chiffres décimaux et son contenu, de même que le code des touches, est affiché en mode programme.

Pour exécuter le programme, le PC est remis à zéro et la mémoire programme envoie à la calculatrice les codes mémorisés. Le PC est à la fois un compteur et un registre, afin de permettre des boucles et des sauts.

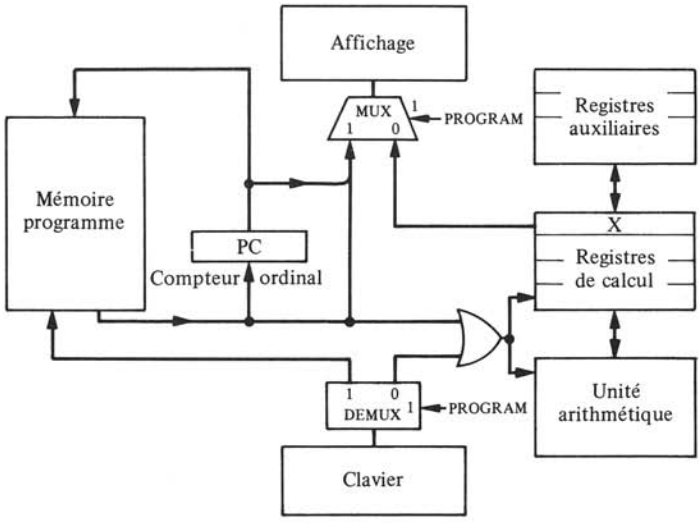


Fig. 3.5

3.2.4 Rupture de séquence

Dans un programme simple, les instructions ou pas de programme sont exécutés dans l'ordre. Le compteur PC est incrémenté à chaque pas. La dernière instruction du programme de l'utilisateur est suivie d'une instruction d'arrêt, interrompant l'exécution automatique et retournant au mode normal. Si l'utilisateur veut que son programme recommence, une instruction de rupture de séquence ou de saut, désignée par les lettres *GOTO* ou *JUMP*, permet de recommencer l'exécution à l'adresse qui suit l'instruction de saut. Cette même instruction permet de passer "par dessus" un groupe d'instructions.

3.2.5 Exemple d'application

L'instruction de saut permet des corrections provisoires du programme, appelées: *rapiéçages* ou *verrues* (*patch*). Si un groupe d'instructions a été oublié, et que la calculatrice n'a pas de touches permettant l'insertion d'instructions (avec décalage des instructions qui suivent et correction des adresses correspondantes), une solution rapide consiste à remplacer une instruction par un saut dans une zone libre de la mémoire programme, mettre dans cette zone l'instruction supprimée suivie des instructions oubliées. Une instruction de saut à la suite du programme termine le raccommodage (fig. 3.6).

Cette méthodologie n'est pas le propre des calculatrices. On la pratique dans tous les systèmes informatiques, pour éviter les opérations complexes liées à la correction d'un gros problème, mais il est évident que ces rapiéçages doivent rester localisés et temporaires si l'on ne veut pas compromettre sérieusement l'intérêt du programme.

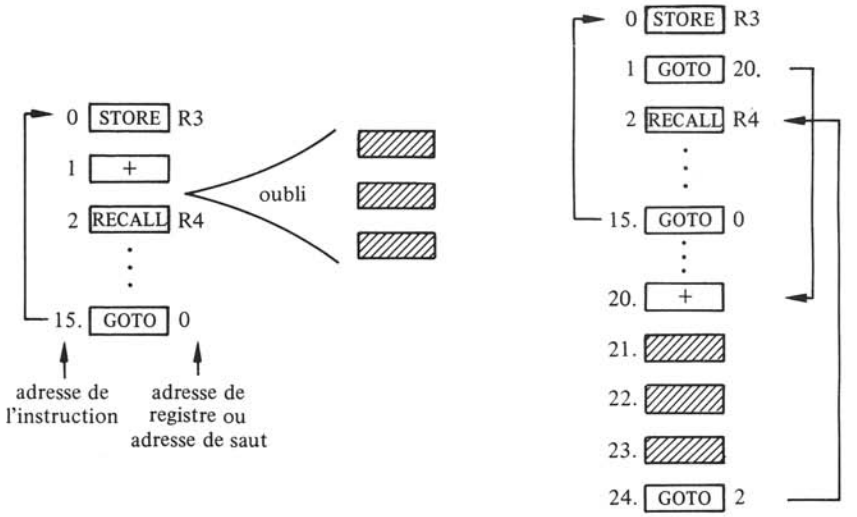


Fig. 3.6

3.2.6 Sauts conditionnels

La plupart des algorithmes de calcul exigent qu'une opération se fasse "à condition que", ou soit répétée "jusqu'à ce que" une certaine condition soit satisfaite. Dans les calculettes, seules quelques conditions de comparaison peuvent être testées : registre d'affichage nul, positif ou négatif, comparaisons entre deux registres, éventuellement test d'un indicateur.

Si la condition est vraie, un saut à une adresse prédéfinie est effectué. Par exemple [x = t] *nn* représente l'action sur la touche [x = t] suivi de l'introduction d'un nombre de deux chiffres et veut dire saut à l'adresse *nn* si $x = t$. Ceci peut s'écrire aussi JUMP, t nn. Dans certaines calculettes, seul un bond (en anglais *SKIP*) par dessus l'instruction suivante est possible. Il y a équivalence fonctionnelle de l'instruction JUMP, t nn et de la paire d'instructions

SKIP, t
JUMP nn

3.2.7 Sous-programmes

Une séquence d'opérations fréquemment répétée peut être considérée comme un bloc séparé, appelé sous-programme ou routine, que l'on appelle par une seule instruction, notée *CALL* ou *GOSUB* et suivie de l'adresse de la routine. L'adresse de retour, c'est-à-dire l'adresse qui suit l'instruction CALL, est sauvée dans un registre spécial. Le sauvetage sur pile (§ 1.2.6) permet l'appel d'une routine dans une routine. Chaque routine est terminée par une instruction de retour, notée *RET*, qui a pour effet de reprendre sur la pile l'adresse de la suite du programme.

La possibilité d'appeler des sous-programmes réduit sensiblement la longueur des programmes complexes, mais ralentit légèrement leur exécution. Il ne faut pas confondre la pile des adresses de retour, invisible pour l'utilisateur, et la pile opérationnelle des calculettes utilisant la notation suffixée.

3.2.8 Perfectionnements

Chaque nouvelle génération de calculettes amène un ensemble de perfectionnements nouveaux dus à un plus grand nombre de touches, un encodage de ces touches plus astucieux, un affichage plus complet, des unités annexes plus variées. Une mémoire magnétique et une imprimante sont des périphériques courants. L'impression de textes alphanumériques est de plus en plus facile.

Du point de vue des fonctions arithmétiques, la gamme des opérations disponibles s'élargit et se complète par des modules incorporés ou enfichables, spécialisant la calculette pour certains types d'applications. La structure de contrôle s'enrichit, le langage devient plus explicite et se rapproche d'un langage d'ordinateur. La différence entre calculette et ordinateur tend à s'amenuiser, la taille et le niveau de performance restant les seules caractéristiques vraiment distinctives.

3.2.9 Modèle de Harvard

La machine construite par Aiken en 1943, le Harvard Mark 1 (§ 1.3.1), avait une mémoire séparée pour les nombres et pour les instructions. En l'honneur de ce pionnier, les calculatrices ayant une mémoire séparée pour les nombres et pour le programme sont appelées *machines de Harvard.* Leur schéma-bloc est donné dans la figure 3.7.

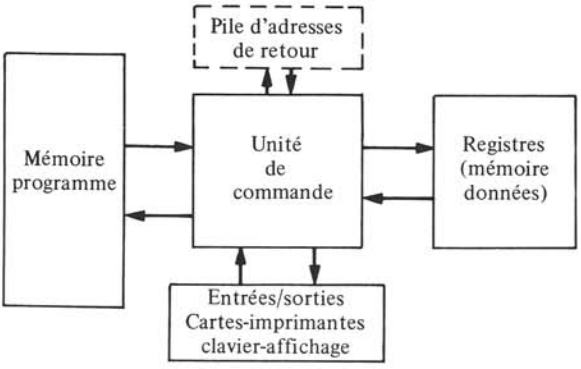


Fig. 3.7

La plupart des calculettes programmables actuelles sont de ce type ou apparaissent comme telles à l'utilisateur; la différence fondamentale entre un nombre et une instruction encourage la mémorisation séparée de ces deux types d'informations, mais des concessions importantes apparaissent sur les modèles récents. Par exemple, de la mémoire "données" peut être convertie en mémoire "programme" et réciproquement à l'aide de touches d'initialisation, et un nombre peut être introduit dans un programme en utilisant la place de plusieurs pas de programme.

3.3 ORDINATEURS

3.3.1 Modèle de von Neumann

La séparation de la mémoire programme et de la mémoire de données est parfaitement adaptée pour les calculs arithmétiques complexes, mais convient mal pour le traitement d'une information non numérique.

L'architecture proposée par von Neumann (§ 1.3.2) n'utilise qu'une seule mémoire, appelée *mémoire banalisée*, pour le programme et les données associées, qu'elles soient numériques ou non. La plupart des processeurs et microprocesseurs sont construits sur ce modèle de von Neumann, qui fait l'objet de cette section. Par la suite ordinateur voudra toujours dire calculatrice de von Neumann.

Le schéma-bloc simplifié d'un ordinateur est donné dans la figure 3.8. La mémoire contient à des emplacements quelconques le programme, les données et des informations de contrôle d'exécution telles que la pile des adresses de retour de sous-programmes. La largeur des mots mémoire ne convient jamais parfaitement bien à chacun de ces types d'information, mais grâce à quelques astuces simples, des compromis très performants résultent de cette organisation.

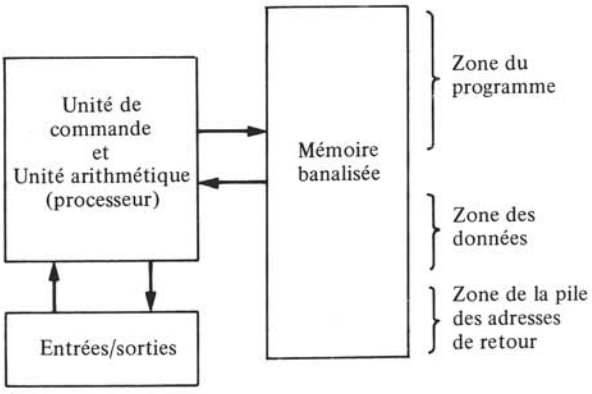


Fig. 3.8

3.3.2 Architecture

Pour réaliser un ordinateur, il faut comme pour une calculette programmable un compteur ordinal (program counter) dont le rôle est de pointer les instructions en mémoire dans l'ordre de leur exécution. De plus, un registre d'instructions I est nécessaire pour mémoriser les instructions pendant leur exécution. L'unité arithmétique peut

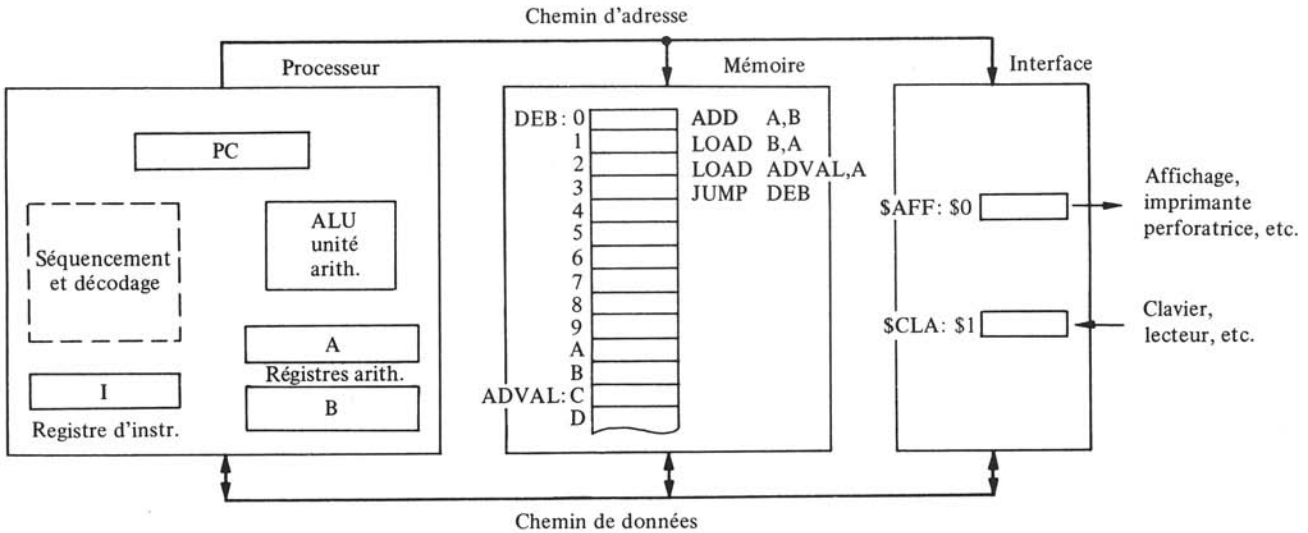

Chemin d'adresse
Processeur
Mémoire
Interface
PC
Séquencement et décodage
ALU unité arith.
A
Régistres arith.
B
I
Registre d'instr.
DEB: 0
1
2
3
4
5
6
7
8
9
A
B
ADVAL: C
D
ADD A,B
LOAD B,A
LOAD ADVAL,A
JUMP DEB
$AFF: $0
Affichage, imprimante perforatrice, etc.
$CLA: $1
Clavier, lecteur, etc.
Chemin de données

Fig. 3.9

avoir une structure très simple. Limitons-nous à quelques instructions agissant sur deux registres internes A et B du processeur et sur la mémoire (fig. 3.9).

La mémoire contient un programme dont l'exécution sera analysée dans un prochain paragraphe. La façon dont la mémoire est préparée est très indirecte et sera détaillée ultérieurement (§ 6.1.13); le transfert se fait à partir d'une mémoire périphérique de plus grande capacité, bande ou disque magnétique, anciennement cartes perforées. Les périphériques sont reliés à l'ordinateur par l'intermédiaire d'interfaces dont l'étude détaillée se fera au chapitre 5.

Le processeur et la mémoire dialoguent par l'intermédiaire d'un chemin d'adresse ou bus d'adresse sélectionnant une cellule mémoire ou le registre d'une interface. L'information qui concerne cette cellule transite par le *chemin de données* (*data bus*) (§ 1.2.5).

Des lignes de commande supplémentaires synchronisent les transferts et l'évolution du programme; les plus importantes seront étudiées au chapitre 5, et peuvent être ignorées dans l'étude fonctionnelle faite dans ce chapitre.

3.3.3 Codage du programme

Le programme que nous supposons chargé en mémoire est le suivant :

```
DEB :  ADD   A, B
       LOAD  B, A
       LOAD  ADVAL, A
       JUMP  DEB
```
(3.1)

Cet exemple arbitraire a pour seul but de montrer quelques instructions types et d'expliquer leur exécution.

A chaque instruction correspond un code binaire caractérisant cette instruction; ce mode est généralement noté par son équivalent octal ou hexadécimal et dans la figure 3.9 il n'a pas été transcrit étant donné que sa valeur est un détail d'implémentation qui n'a rien à voir avec la fonction effectuée. Un programme de traduction, appelé *assembleur*, (sect. 4.3) traduit automatiquement les instructions symboliques. Il procède par assemblage des valeurs binaires correspondant à chacun des éléments de l'instruction, d'où son nom.

Les positions dans lesquelles les instructions sont mémorisées sont numérotées ici en hexadécimal. Cette numérotation sera souvent implicite, et les positions importantes sont caractérisées par un nom, plus exactement par une chaîne de caractères alphanumériques appelé *étiquette* (*label*) représentant le numéro de la position mémoire correspondante.

Par exemple dans la figure 3.9 DEB : désigne l'adresse 0 et ADVAL : la position d'adresse H'C dans laquelle le nombre calculé par le programme est mémorisé (avec perte du résultat précédent puisque la même adresse est toujours concernée). Seules des contraintes de place dans la figure 3.9 ont obligé à abréger l'adresse de début en DEB et l'adresse de la valeur calculée en ADVAL. Plus le programme est complexe et plus il est important que les noms donnés soient explicites et fournissent une aide mnémotechnique efficace.

3.3.4 Exécution

Une impulsion de remise à zéro initialise l'unité de séquencement et le compteur ordinal. La première instruction de notre programme se trouve ainsi pointée et sélectionnée. Le code de cette instruction, qui est le contenu de la position mémoire 0, est alors placé sur le bus d'information et transféré dans le registre d'instruction (fig. 3.10). L'instruction est alors décodée et l'opération d'addition de A et B, avec résultat dans A est effectuée. Simultanément, le compteur ordinal est incrémenté pour préparer l'opération suivante (fig. 3.11).

La seconde instruction est exécutée de façon très similaire à la première, avec tout d'abord recherche de l'instruction en mémoire (fig. 3.12), puis exécution du transfert LOAD B, A qui copie A dans B (fig. 3.13).

La troisième instruction est ensuite recherchée (fig. 3.14) et effectue un transfert similaire au précédent, mais vers la position mémoire d'adresse H'C, à laquelle le nom ADVAL a été donné. A l'instant du transfert, l'adresse H'C se trouve dans le registre d'instruction et est placée sur le bus pour sélectionner la position mémoire avec laquelle le transfert d'information doit se faire (fig. 3.15).

La dernière instruction de notre programme effectue un saut au début. Le transfert dans le registre d'instruction (fig. 316) est suivi par l'exécution du saut, c'est-à-dire par le transfert de l'adresse DEB dans le compteur ordinal (fig. 3.17). Le programme recommence alors à l'instruction d'adresse 0 et s'exécute de la même manière mais les valeurs numériques des registres sont différentes.

L'alternance régulière des *cycles de recherche* de l'instruction et des cycles d'*exécution* s'appelle en anglais "*fetch-execute*" et est caractéristique des architectures de von Neumann simples.

Pour accélérer, la recherche du code des instructions peut être faite dans les temps morts de l'exécution des instructions précédentes. Ces codes sont placés dans un *silo de prérecherche* (*prefetch queue*). L'unité de séquencement prélève les instructions à l'autre extrémité de ce silo, avec un temps d'accès réduit lorsque les instructions se suivent. S'il y a un saut, le contenu du silo est perdu, et il faut recommencer à le remplir.

3.3.5 Types d'instructions

Le programme précédent met en évidence quatre types d'instructions qui ont déjà été rencontrées pour les calculettes programmables (sect. 3.2) :

- les instructions arithmétiques
- les instructions de transfert entre registres internes
- les instructions de transfert entre registre et position mémoire (ou périphérique)
- les instructions de saut.

Ces instructions sont toutes caractérisées par une opération et un groupe d'opérandes et l'on peut distinguer :

- les instructions sans opérande
- les instructions avec un opérande (JUMP DEB)
- les instructions avec deux opérandes (ADD A, B)
- les instructions avec trois opérandes, etc.

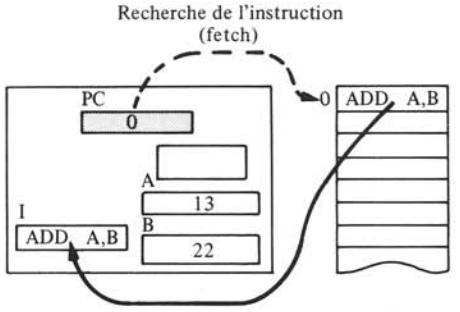


Fig. 3.10

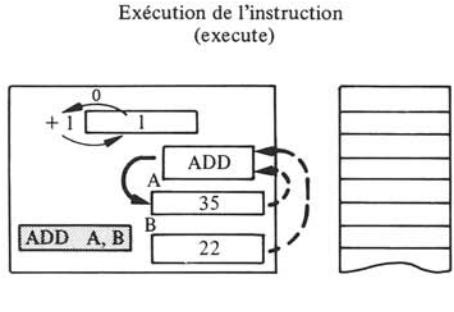


Fig. 3.11

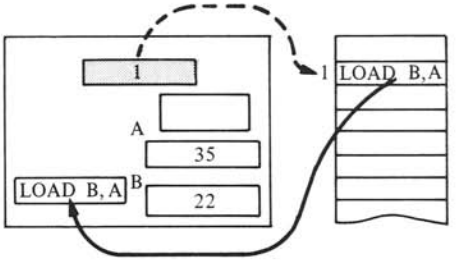


Fig. 3.12

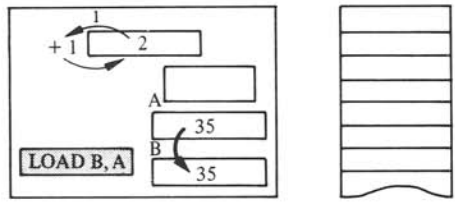


Fig. 3.13

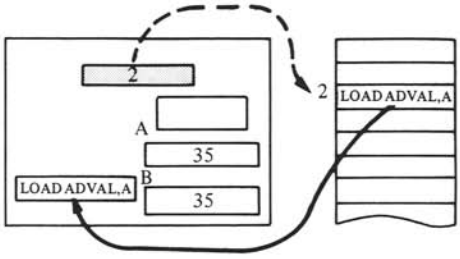


Fig. 3.14

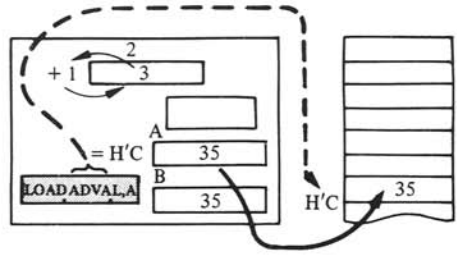


Fig. 3.15

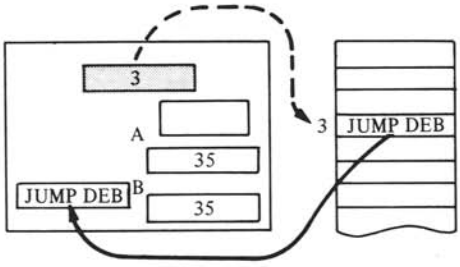


Fig. 3.16

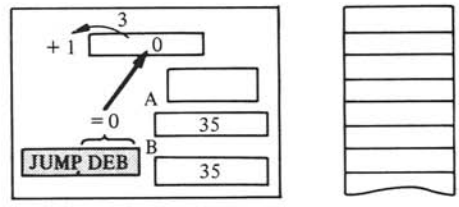


Fig. 3.17

La seule instruction sans opérande s'appelle NOP (no-operation) et est utile pour la mise au point des programmes et pour calibrer la durée d'exécution d'un groupe d'instructions.

D'autres instructions apparaissent comme n'ayant pas d'opérande, mais celui-ci est implicite, c'est-à-dire n'est pas noté de façon explicite.

Les instructions à un opérande modifient le contenu d'une position mémoire, d'un registre, ou d'une bascule de mode du processeur. L'adresse de la position mémoire ou du registre, c'est-à-dire son nom, suit le nom de l'instruction. Par exemple JUMP DEB modifie le compteur ordinal, NOT A remplace le contenu du registre A par son complément à un, et ION met à un la bascule d'interruption (§ 5.3.1).

Les instructions à deux opérandes considèrent deux positions mémoire ou registres simultanément, et modifient l'un, parfois les deux opérandes donnés.

Le traitement simultané de trois opérandes ou plus complique la circuiterie des ordinateurs et peut toujours se remplacer par une suite d'opérations à deux opérandes. Les instructions à trois opérandes ou plus sont rares dans les miniordinateurs et microprocesseurs.

Une différence est souvent faite entre :

- les instructions avec référence mémoire (LOAD A, ADVAL)
- les instructions sans référence mémoire (LOAD A, B).

En fait, la différence entre une position mémoire et un registre est fonctionnellement une simple différence de proximité géographique. Le nombre de positions mémoire étant toutefois nettement plus élevé que le nombre de registres, la structure et le nombre des instructions agissant sur les registres et sur la mémoire sont souvent très différents.

3.3.6 Format des instructions

Une instruction est un mot binaire formé par le *code opératoire* (*operation code, op code*) et par les adresses des opérandes, ou un code caractérisant ces opérandes.

Supposons que le processeur a 16 registres internes, doit adresser au maximum 65536 positions mémoire et dispose de 64 instructions. Quatre bits sont donc nécessaires pour la sélection des registres internes, 16 pour les adresses mémoire, et 6 bits pour le code des instructions. La figure 3.18 donne les *formats* de quelques types d'instructions associées à cette architecture. La permutation et le fractionnement des différents *champs* de l'instruction sont naturellement possibles.

Les différences de longueur de formats, mises en évidence par la figure 3.18, ne sont guère compatibles avec la structure usuelle des mémoires, constituées d'un ensemble de mots de longueur fixe. Des techniques de codage appropriées et des limitations dans le nombre et la nature des instructions permettent de se limiter à quelques longueurs d'instructions en rapport simple avec la longueur des mots mémoire. Un exemple sera donné dans la figure 3.19.

3.3.7 Représentation des nombres et instructions

Les ordinateurs de von Neumann ont une mémoire banalisée contenant à la fois les nombres et les instructions du programme. Les nombres que l'on cherche à représenter sont tout d'abord les nombres entiers et les nombres flottants. La longueur de ces nombres dépend de leur précision, et l'on retrouve avec les nombres le même problème de format variable qu'avec les instructions.

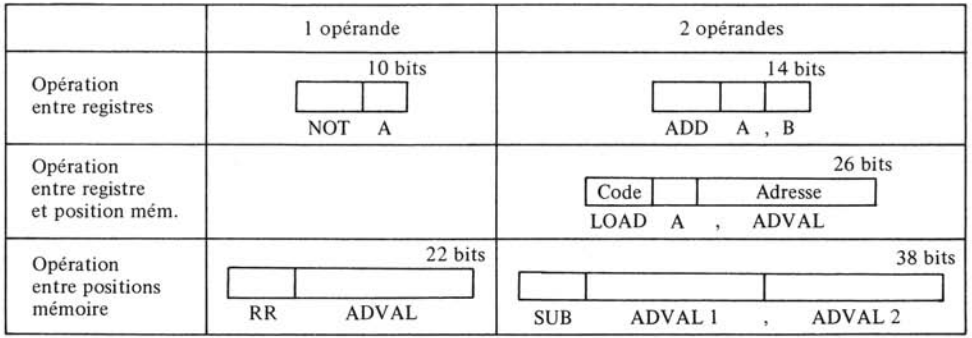


Fig. 3.18

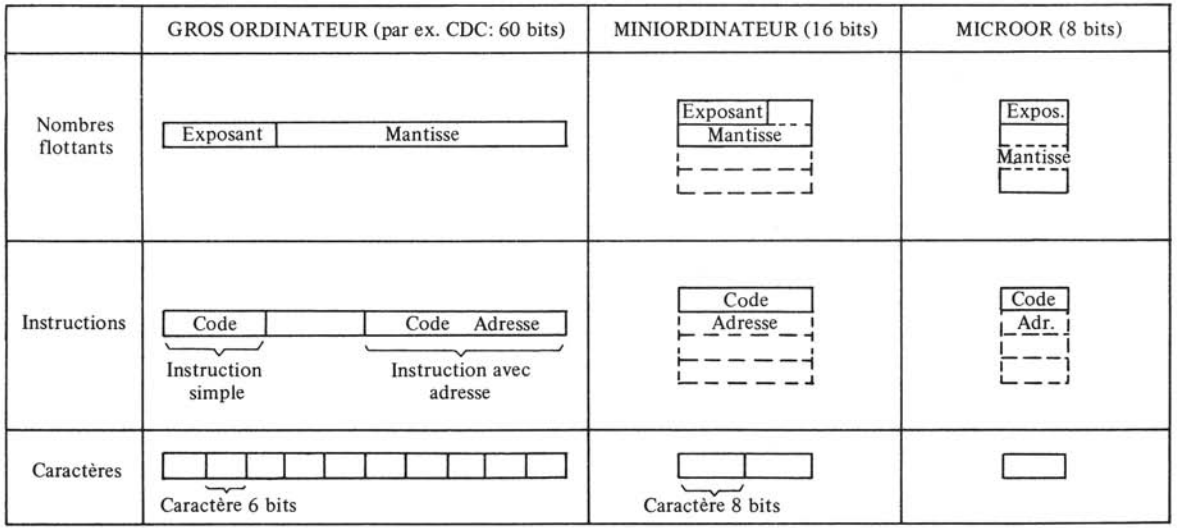


Fig. 3.19

La solution à ce problème est une gamme d'ordinateurs dont les performances sont fixées en grande partie par la dimension des mots mémoire; l'unité arithmétique opère généralement sur des mots dont la longueur est égale à la longueur des mots mémoire. La figure 3.19 résume quelques formats que l'on trouve dans un gros ordinateur, dans un miniordinateur ou microprocesseur 16 bits et dans un microprocesseur simple 8 bits. Le format utilisé pour représenter un texte alphanumérique, c'est-à-dire les codes des caractères constituant ce texte, est également donné.

On remarque que les mots longs conviennent bien pour la représentation des nombres flottants et pour des instructions ayant plusieurs adresses mémoire. Les instructions plus simples peuvent être groupées à plusieurs instructions par mot, ce qui complique passablement leur décodage au moment de l'exécution, et entraîne une sous-utilisation de certains mots. Plusieurs caractères prennent place dans le même mot mémoire. Dans le cas du CDC 7600, ils forment des blocs de 10 caractères même si le texte ou résidu de texte à mémoriser est plus court.

Avec des mots de 16 bits, les nombres flottants doivent être répartis sur 2 à 4 mots mémoire, et les instructions sur 1, 2 ou 3 mots. Chaque mot de 16 bits peut contenir 2 caractères, voire 3 si le nombre de caractères différents est limité à 40 (exercice 2.2.6).

Les microprocesseurs simples ont des mots de 8 bits qui permettent une grande simplicité de l'unité arithmétique et une excellente efficacité dans l'utilisation de la mémoire. Les nombres flottants sont généralement limités à 32 bits et occupent donc 4 positions mémoire consécutives. Le premier but des microprocesseurs n'est pas de concurrencer les gros ordinateurs scientifiques, et leur unité arithmétique est généralement limitée aux nombres entiers. Ceci n'exclut pas, par des séquences de calcul plus complexes, le calcul sur des nombres flottants de précision quelconque. Les instructions ont un nombre de bits multiple de 8 et chaque mot mémoire correspond à un caractère.

Cette structure en mots élémentaires de 8 bits n'est pas l'apanage des seuls microprocesseurs. Les ordinateurs de la famille IBM 360/370 utilisent des instructions de format variable multiple de 8 bits. Le nombre de bits simultanément prélevés en mémoire est de 8, 32 ou 64 bits suivant le modèle.

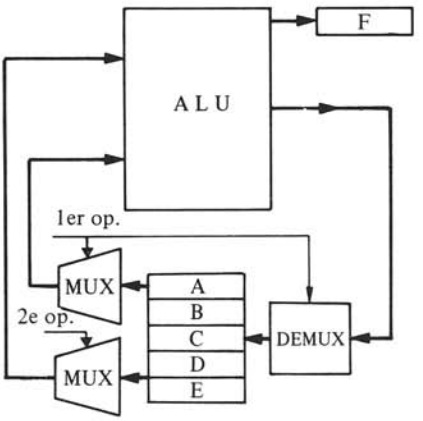


Fig. 3.20

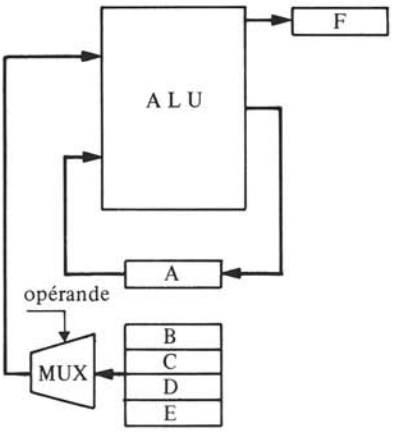


Fig. 3.21

3.3.8 Sélection des opérandes

Considérons le cas d'un processeur incluant plusieurs registres à proximité de l'unité arithmétique et logique (fig. 3.20). De façon générale, les deux opérandes peuvent être choisis parmi l'un quelconque des registres et le résultat placé dans un registre quelconque. Par exemple, on peut soustraire B et D et placer le résultat dans C, ce que nous écrirons

$$\text{SUB}\quad C, B, D \quad \text{ou} \quad C := B - D \tag{3.2}$$

L'intérêt de permettre trois registres distincts est faible en rapport à l'inconvénient apporté par des instructions plus longues, donc une mémoire plus coûteuse. Il est plus avantageux de placer le résultat dans le même registre que le premier opérande, en commandant le multiplexeur du premier opérande et le démultiplexeur de résultat par les mêmes lignes d'adresse (fig. 3.21).

L'instruction (3.2) ne peut plus dans ce cas être exécutée directement et doit être remplacée par le programme (3.3).

$$\left.\begin{array}{l}\text{SUB}\quad B, D\\ \text{LOAD}\quad C, B\end{array}\right\} \quad \text{ou} \quad \left\{\begin{array}{l}B := B - D\\ C := B\end{array}\right. \tag{3.3}$$

dont l'effet n'est toutefois pas le même, puisque dans ce second cas (3.3) le registre B contenant le premier opérande a été détruit et contient le résultat.

Une simplification supplémentaire consiste à utiliser toujours le même registre, appelé accumulateur, comme premier opérande et résultat. Toutes les opérations sont effectuées entre l'accumulateur et un registre, et il suffit de donner dans l'instruction le code de l'opération et l'adresse du registre contenant le deuxième opérande. Le premier opérande est implicite, puisque c'est toujours l'accumulateur. Les constructeurs de ce type de machine ne mentionnent généralement pas les opérateurs implicités dans le libellé de l'instruction et écrivent par exemple dans le cas de l'addition de l'accumulateur et du registre C

$$\text{AD}\quad C \ (\text{ou ADD C, ADC, etc.}) \tag{3.4}$$

Dans cet ouvrage, nous éviterons autant que possible les notations implicites et écrirons au lieu de (3.4)

$$\text{ADD}\ A, C \tag{3.5}$$

Avec la structure de la figure 3.21, l'instruction (3.2) qui soustrait B et D et place le résultat dans C se laisse facilement remplacer par un groupe de trois instructions :

$$\left.\begin{array}{l}\text{LOAD}\quad A, B\\ \text{SUB}\quad A, D\\ \text{LOAD}\quad C, A\end{array}\right\} \tag{3.6}$$

L'effet n'est pas absolument équivalent puisque le registre A est modifié et contient aussi le résultat.

La comparaison entre les différentes structures de registres et d'instructions ne peut toutefois pas se faire valablement sur des opérations aussi partielles. Il est évidemment plus agréable pour le programmeur de disposer d'un processeur à plusieurs accumulateurs évitant des transferts inutiles de registres, mais vu les contraintes de codage des instructions dans des mots de dimension donnée, cette flexibilité peut n'être

atteinte dans les microprocesseurs 8 bits qu'aux dépens d'autres instructions plus intéressantes.

LOAD A, B veut dire "transférer le contenu de B dans A". Ce transfert ne modifie pas B, et le contenu initial de A n'a pas d'influence.

3.3.9 Espace d'adressage

Un ensemble de registres comme les registres A et E de la figure 3.20 sélectionnables par un groupe d'instructions du processeur, forme un *espace d'adressage* (*addressing space*). Avec la mémoire et les entrées/sorties, on a trois espaces d'adressage principaux.

Ces trois espaces sont caractérisés par des notations adéquates : les éléments de l'espace des registres ont des noms réservés (A, HL, R0, etc.) définis par le fabricant. Les éléments de l'espace mémoire ont des noms mnémotechniques définis par l'utilisateur (AVAL, TABLE, etc.) dont la valeur numérique est l'adresse en mémoire. Les éléments de l'espace d'entrée/sortie ont des noms comme pour l'espace mémoire, le signe $ caractérisant cet espace, lorsqu'il est distinct de l'espace mémoire.

Chaque espace a des instructions propres. Un processeur est *orthogonal* si les opérandes d'une instruction peuvent être pris dans un espace quelconque, avec tous les modes possibles.

Un grand espace d'adressage implique un nombre élevé de bits pour sélectionner une position donnée dans cet espace. Ces bits peuvent figurer explicitement dans l'instruction, ou implicitement comme par exemple pour une pile.

3.4. INSTRUCTIONS SIMPLES

3.4.1 Instructions arithmétiques

Contrairement à la calculette qui ne traite que des nombres réels avec une certaine précision, l'ordinateur doit pouvoir travailler avec des entiers de différentes longueurs et dans différentes représentations. Il doit aussi pouvoir effectuer des opérations logiques de comparaison, fractionnement ou fusionnement de mots, modification de bits, etc.

En conséquence, l'unité arithmétique et logique est binaire et effectue les opérations définies dans la section 2.5.

ADD	d, s	INC	d	AND	d, s
ADDC	d, s	DEC	d	OR	d, s
SUB	d, s	NEG	d	XOR	d, s
SUBC	d, s	NOT	d		
COMP	s1, s2	CLR	d		
MUL	d, s	SR	d	RR	d
DIV	d, s	ASR	d	RRC	d
		SL	d	RL	d
				RLC	d

(3.7)

La multiplication et la division, de même que les opérations en virgule flottante sur des nombres d'une certaine longueur, sont le plus souvent programmées. Un gain de vitesse d'un facteur 5 à 20 est obtenu par une solution câblée, au détriment de la simplicité et du prix du processeur.

Les microprocesseurs ont un répertoire d'instructions arithmétiques plus pauvre que les gros miniordinateurs et ordinateurs, mais l'évolution de la technologie permet déjà d'incorporer des opérations décimales en BCD (sect. 2.8), la multiplication et la division binaire (sect. 2.6) et bientôt les opérations en virgule flottante (sect. 2.7).

Pour préciser le type des nombres sur lesquels ces opérations agissent, lorsque plusieurs possibilités sont offertes par le processeur, on peut ajouter au mnémonique de base ou à l'opérande un préfixe et/ou postfixe : D pour décimal, F pour flottant. La taille est souvent caractérisée par cette même lettre (F flottant 32 bits, G flottant 64 bits, etc.), mais d'autres notations seront introduites au paragraphe 3.6.1.

Le type de nombre manipulé par l'unité arithmétique peut dépendre de l'état d'une bascule (bit de mode) qui modifie l'effet d'un groupe d'instructions. Le processeur 6502 par exemple travaille avec les mêmes instructions ADD et SUB soit en décimal (BCD 8 bits), soit en binaire, selon que l'on a exécuté l'instruction SETD (set decimal bit) ou CLRD (clear decimal bit) auparavant.

3.4.2 Indicateurs

Les opérations arithmétiques, on l'a vu dans la section 2.4, génèrent un report C, des indicateurs d'inégalité Z, EQ, LO, etc., un signe S et des indicateurs de dépassement de capacité arithmétique C, V. L'évolution du programme dépend souvent des valeurs de ces indicateurs à la fin d'un groupe d'opérations arithmétiques.

Un *registre d'indicateurs* F (*flag register, processor status word*) réunit tous ces indicateurs en un seul mot qui définit l'état de l'unité arithmétique à un instant donné et peut être sauvé en mémoire ou analysé bit par bit. Si l'opération doit être interrompue, elle peut être reprise au même point après rétablissement des registres de calcul et du registre d'indicateurs, si ceux-ci ont été sauvés en mémoire lors de l'interruption.

3.4.3 Opérations sur des bits

Les différents bits d'un même mot peuvent avoir des significations très différentes et des instructions agissant sur un bit isolé sont très efficaces.

Il n'y a en général pas d'instructions de transfert ou d'addition de bits, mais seulement la possibilité de forcer un bit à un ou à zéro, de changer ou de tester sa valeur. Dans ce dernier cas, la valeur est transférée dans l'indicateur Z. Lorsque ce bit est le bit de signe d'un nombre représenté sous la forme signée, l'intérêt de l'instruction est évident. De même dans les applications industrielles où chaque bit d'un mot commande par exemple un relais.

Par exemple, avec un processeur 8 bits,

SET	C : #0	force le bit de poids faible du registre C à un	
CLR	$RELAIS : #MOT1	arrête un moteur en agissant sur le bit correspondant au périphérique RELAIS	
NOT	A : #SIGN	change le signe de A (SIGN = 7)	(3.8)
TEST	A : #SHIFT	teste si la touche SHIFT est pressée	
SETC		force le CARRY à un (bit C du registre d'indicateurs F) équivalent à SET F: #BITCARRY	

Lorsque ces instructions n'existent pas, des opérations logiques de transfert et de masquage permettent de les simuler (§ 2.5.15). Ainsi on peut remplacer l'instruction

SET C : #0 par OR C, #B'00000001 (3.9)

3.4.4 Instructions de transfert

Une instruction de transfert copie un registre ou position mémoire dans un autre ou échange leur contenu. Il peut y avoir simultanément effet sur certains indicateurs, en particulier Z (égalité) et S (signe négatif), selon les habitudes de chaque constructeur.

Les noms réservés pour ces instructions sont :

MOVE LOAD	s, d d, s	transfert	
EX	d1, d2	échange	
SWAP	d	permute les deux moitiés	(3.10)
CLR	d	initialise avec des zéros	
PUSH	s	transfert sur une pile	
POP	d	reprise de la pile	

Les deux dernières instructions supposent qu'il n'y a qu'une seule pile, ce qui est généralement le cas.

3.4.5 Instructions de saut

Les instructions de saut précisent comme opérande l'adresse de la position mémoire à partir de laquelle l'exécution doit continuer. Ce sont en fait de simples instructions de chargement du compteur ordinal

JUMP d identique à LOAD PC, #d (3.11)

Une instruction de saut conditionnel effectue ou non un saut selon que le résultat d'un certain test est vrai ou faux. Suivant le processeur, on peut ne disposer que du saut à une adresse quelconque (JUMP, t d) ou du saut par-dessus l'instruction suivante (SKIP, t).

Le test se fait le plus généralement sur les états des indicateurs associés à l'unité arithmétique et de commande (§ 3.4.2). Parfois, il est possible de tester directement les contenus de certains registres ou positions mémoire, soit globalement (valeur égale à zéro), soit au niveau de chaque bit individuellement.

Ces différents cas se notent de la façon suivante :

JUMP, CS	Saut si le CARRY est à un (set)	(3.12)
JUMP, CC	Saut si le CARRY est à zéro (clear)	
JUMP, EQ	Saut si l'indicateur Z est à un (si le résultat de l'opération précédente est nul)	
JUMP, NE	Saut si le résultat de l'opération précédente est différent de zéro (Z = 0)	
JUMP, AEQ	Saut si le registre A est égal à zéro (ne tient pas compte de Z et ne le modifie pas)	
JUMP, A : #7NE	Saut si le bit 2^7 du registre A n'est pas égal à zéro (donc est égal à un)	

L'utilisation de notations explicites évite des commentaires supplémentaires et diminue des erreurs de codage.

3.4.6 Appel de sous-programmes

Comme pour les calculettes, la notion de sous-programme est fondamentale pour la programmation des ordinateurs. L'instruction *CALL* avec comme opérande l'adresse du saut diffère de l'instruction JUMP par le fait qu'elle sauve tout d'abord l'adresse de l'instruction suivante (adresse de retour) sur une pile en mémoire.

CALL d identique à PUSH PC
JUMP d (3.13)

Le sous-programme ou routine est terminé par l'instruction

RET identique à POP PC (3.14)

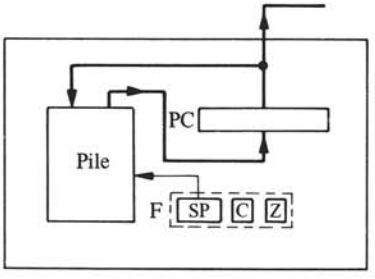


Fig. 3.22

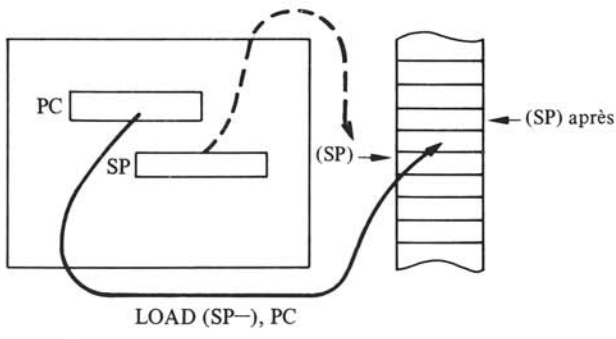


Fig. 3.23

La pile d'adresse de retour est soit une zone spéciale de registre sur le processeur (fig. 3.22), soit une zone en mémoire (fig. 3.23). Dans les deux cas un registre compteur/décompteur spécial, le pointeur de pile SP, définit l'emplacement du sommet de la pile.

3.4.7 Pile commune

Losque la pile est en mémoire (fig. 3.23), sa dimension n'est pas limitée, et on parle parfois de pile infinie. Il est très intéressant de pouvoir placer sur la pile non seulement des adresses de retour, mais encore des résultats intermédiaires de calcul, ou des paramètres quelconques.

Les instructions PUSH A, PUSH B, POP B, POP A, etc. permettent de sauver sur la pile, puis récupérer, le contenu des registres. Ceci permet d'innombrables astuces de programmation, dont certaines ne sont du reste pas recommandées. Par exemple, le programmeur peut, à l'intérieur de la routine, transférer l'adresse de retour dans l'unité arithmétique, la modifier, et la remettre en place pour obtenir un retour à un emplacement différent.

La pile peut être considérée comme une réserve de registres auxiliaires. Une structure de pile arithmétique (§ 3.1.6) peut facilement être simulée; une pile séparée est en général utilisée pour mieux séparer les adresses de retour des opérandes.

La pile est surtout l'endroit idéal pour sauver l'état du processeur lorsque la tâche en cours doit être interrompue au profit d'une tâche plus prioritaire. Les instructions PUSH F, POP F, qui permettent de sauver et récupérer les états des différentes bascules de l'unité arithmétique et de l'unité de commande, sont essentielles pour sauver complètement l'état de la machine.

Lorsque les nombres sont plus courts que les adresses, comme dans le cas des microprocesseurs, les registres arithmétiques sont parfois transférés par paires : on a par exemple pour le Z80 les instructions PUSH AF, PUSH BC, etc.

3.4.8 Exercice

Pour le processeur 8085 ou Z80, remplacer l'instruction manquante EX A, B, qui échange A et B, par un programme ne faisant appel qu'aux instructions PUSH AF, POP AF, PUSH BC, POP BC, LOAD A, B, LOAD B, A.

Une solution simple modifie le registre F ou le registre C. Trouver aussi une solution ne modifiant aucun registre, mais en ayant à disposition toutes les instructions du Z80.

3.5 MODES D'ADRESSAGE

3.5.1 Introduction

De façon générale, une instruction agit sur quelques opérandes implicites ou explicites. Le compteur ordinal définit l'instruction dans l'espace mémoire programme, et les opérandes sont pris dans les espaces définis par l'instruction. Ces espaces peuvent être identiques, mais sont souvent distincts, en tout cas du point de vue logique (fig. 3.24).

Le processeur obtient l'*adresse effective* (*effective address* EA) d'un opérande en interprétant son *adresse exprimée* ou *adresse symbolique* par le codage de l'instruction, c'est-à-dire en effectuant des opérations arithmétiques sur les éléments de l'adresse, et

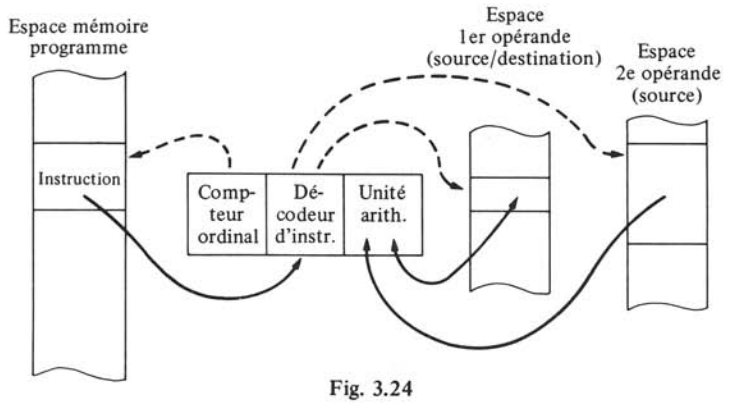


Fig. 3.24

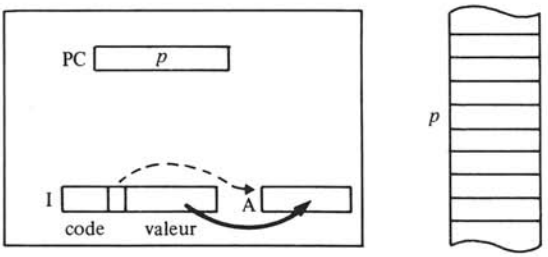
des opérations d'indirection permettant de trouver un élément de l'adresse effective dans une position mémoire adressée par un premier calcul d'adresse. Les calculs d'adresses se font avec des mots de 8, 16 ou 32 bits selon le processeur.

Dans cette section, nous supposerons dans chaque paragraphe une architecture simple avec uniquement les éléments nécessaires pour le mode d'adressage considéré.

L'espace du 1er opérande sera réduit à un registre A (de 8 bits par exemple). La généralisation à des transferts de mots plus longs et à un espace destination plus riche qu'un simple registre sera vue ultérieurement (sect. 3.6).

3.5.2 Adressage immédiat

La façon la plus simple de charger une valeur dans un registre est de donner cette valeur dans l'instruction. Au moment de l'exécution, la partie "opérande" du champ de l'instruction est transférée directement dans A, sans nouvel accès mémoire (fig. 3.25).



LOAD A, ≠ VALEUR

Fig. 3.25

Ce type d'adressage est appelé *adressage immédiat*, car la valeur à transférer est immédiatement disponible dans l'instruction. Nous utiliserons le signe # (*dièze*, *number sign*) pour caractériser les valeurs immédiates et écrirons par exemple :

LOAD A, #VALEUR (3.15)

Plusieurs fabricants écrivent pour cette même instruction une expression du type :

LODI VALEUR (3.16)

plus facile à décoder par le programme de traduction, et caractérisée par la mention du mode d'adressage dans le code mnémonique de l'instruction (lettre I), et par la référence implicite à l'accumulateur A. La complexité des processeurs récents ne permet plus de tels raccourcis de notation.

3.5.3 Adressage absolu

L'*adressage absolu* est le mode d'adressage le plus naturel dans une architecture de von Neumann, et s'écrit par conséquent sans signe spécial. L'adresse de l'opérande dans son espace d'adressage est donnée dans l'instruction (fig. 3.26).

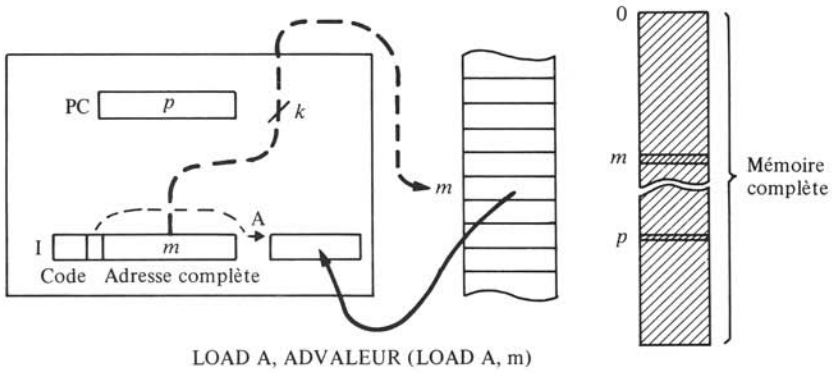


Fig. 3.26

Ce mode d'adressage ne nécessite aucun signe spécial

LOAD A, ADVALEUR (3.17)

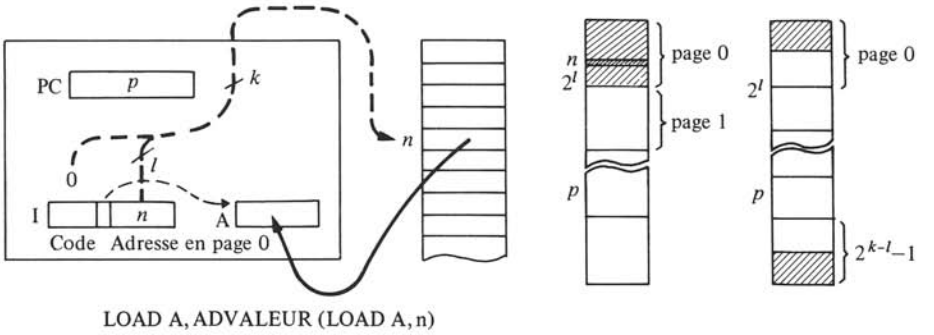
Le symbole ADVALEUR représente la position mémoire contenant le nombre transféré dans A. En d'autres termes, la valeur numérique *m* de ADVALEUR est l'adresse de cette position mémoire. Cette façon de travailler est si courante que l'on identifie généralement adresse et contenu et désigne les adresses mémoire par des noms évoquant directement leur contenu.

L'adressage absolu permet de se référer à une position mémoire quelconque. Par la suite, la lettre m représentera une adresse complète, c'est-à-dire un nombre comportant suffisamment de bits pour repérer une position quelconque. Si la mémoire prévue est grande, ceci implique un nombre important de bits dans l'instruction.

3.5.4 Adressage en page 0

Il y a avantage à avoir des adresses courtes pour les opérandes les plus fréquemment utilisés. Les registres internes ont des adresses très courtes (3 à 4 bits) puisqu'ils sont peu nombreux (8 ou 16). Plusieurs microprocesseurs mettent à part les 256 premières positions de la mémoire pour permettre un adressage court avec un mot de 8 bits. Ces 256 positions forment une *page 0* qui est un sous-espace de l'espace complet d'adressage en mémoire (fig. 3.27).

Avec certains processeurs (Signetics 2650), l'adresse 8 bits est considérée comme une adresse signée en complément à 2, avec extension du signe. Il s'ensuit un fraction-



LOAD A, ADVALEUR (LOAD A, n)

Fig. 3.27

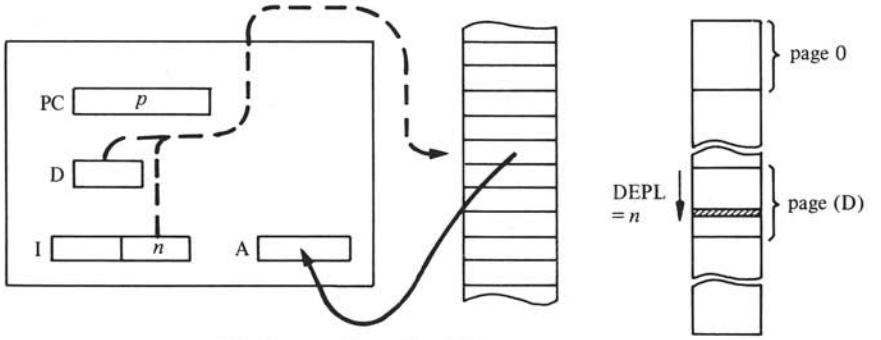
nement du sous-espace d'adressage en deux moitiés situées respectivement aux adresses les plus basses et les plus hautes. Le processeur M68000 réagit de la même façon avec une page 0 ayant 16 bits d'adresse, coupée en deux zones de 32k en haut et en bas de mémoire.

L'adresse exprimée de ce mode d'adressage est la même que pour l'adressage absolu. Si les deux modes absolu étendu et en page 0 sont possibles, le programme de traduction peut choisir lui-même l'adresse effective la plus appropriée.

3.5.5 Adressage en page fixe et mobile

Un processeur peut avoir des instructions d'adressage dans différentes pages. Par exemple, le 8048 a une seule instruction d'adressage en page 3, prévue spécifiquement pour une table de conversion ou un générateur de caractères.

Il est plus pratique de prévoir un registre contenant l'adresse de la page, et de pouvoir changer de page par une instruction. C'est ce que font les processeurs M6802 et M6809 (fig. 3.28)



LOAD A, {D} * 100 + DEPL

Fig. 3.28

La notation pour ce type *d'adresse en page mobile* exprime exactement l'opération qui est effectuée de façon interne : le contenu du registre D (noté entre accolades pour distinguer de l'adresse D) est décalé de 8 bits, donc multiplié par H'100 avant de lui ajouter la valeur n codée dans l'instruction. Cette valeur est appelée *déplacement*

puisqu'elle exprime le déplacement par rapport au début de la page. On écrit:

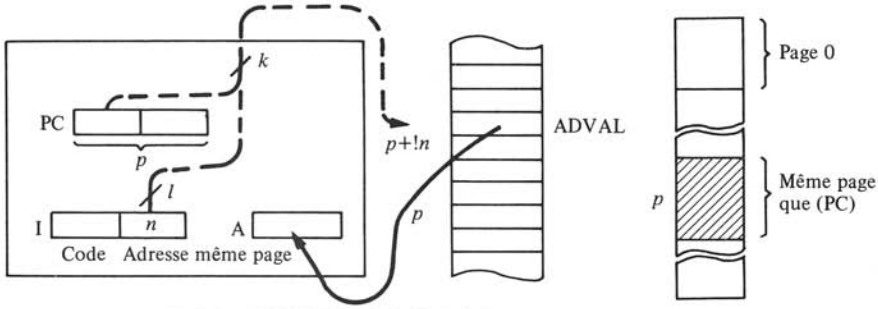
LOAD A, {D} * 100 + DEPL (3.18)

Les fabricants préfèrent une notation plus courte, mais moins précise, avec un signe dans le mnémonique de l'instruction ou dans l'opérande

LOADD A, DEPL ou LOAD A, % DEPL (3.19)

3.5.6 Adressage en page courante

La page mobile peut être définie par les poids forts du compteur ordinal. On parle alors *d'adressage en page courante* (fig. 3.29).



LOAD A, ADVALEUR (LOAD A, !n)

Fig. 3.29

L'adresse exprimée pour ce mode d'adressage peut être la même que pour le mode d'adressage direct. Le programme de traduction peut comparer l'adresse ADVAL et la valeur courante du compteur ordinal PC, en déduire n et choisir ce mode d'adressage plus court si cela est possible.

L'adressage dans la même page a l'avantage d'être simple du point de vue circuits et est utilisé dans les microordinateurs de la famille TMS1000 avec des pages de 64 octets. Du point de vue pratique, il oblige à découper le programme en modules d'une page, diminuant l'efficacité de la mémoire et en compliquant beaucoup la programmation.

Lorsqu'il y a extension mémoire sur un processeur simple (sect. 3.8), on se trouve fréquemment dans une situation d'adressage en page courante, avec heureusement 16 bits d'adresse.

3.5.7 Adressage relatif

Le calcul de l'adresse effective par addition du contenu du compteur ordinal et d'une adresse incomplète contenue dans l'instruction permet de définir un intervalle d'adressage symétrique autour de l'adresse courante. Ce type d'adressage est appelé *relatif* (fig. 3.30).

Lors de l'exécution de l'instruction, le déplacement est ajouté au contenu du compteur d'adresse pour pointer l'opérande. C'est un nombre arithmétique et son signe doit être étendu avant d'être ajouté à l'adresse. Dans certains microprocesseurs, comme

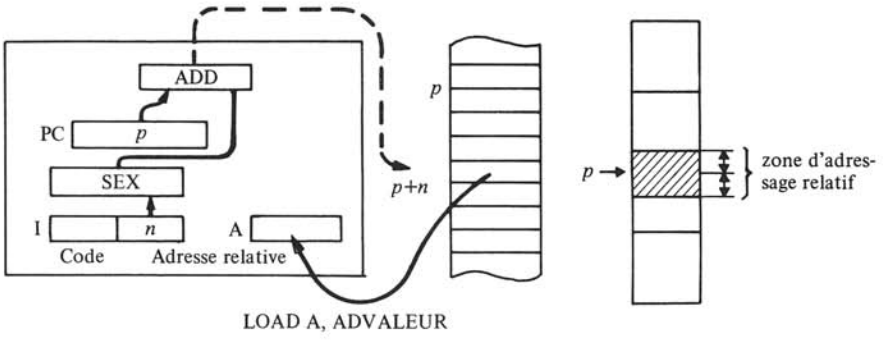


Fig. 3.30

par exemple le 2650, l'additionneur d'adresse est incomplet et entraîne une décomposition de la mémoire en chapitres, la frontière entre chapitres ne pouvant pas être franchie par l'adressage relatif [53].

La notation l'adressage relatif est

$$\{PC\} + n \tag{3.20}$$

mais le programmeur utilise l'adresse exprimée identique à l'adressage direct.

Le programme de traduction peut calculer lui-même la valeur n à partir de l'adresse courante du PC et l'adresse ADVAL qui concerne le programmeur.

Il faut remarquer qu'en général le calcul d'adresse se fait alors que l'instruction a été entièrement lue et que le PC pointe l'instruction suivante et non pas le début de l'instruction en cours. Avec chaque processeur, il faut examiner soigneusement la façon dont l'adresse relative est calculée, et faire attention aux anomalies. Le M68000 par exemple a deux instructions de saut relatives qui utilisent 2 et 4 octets en mémoire, mais prennent toutes deux comme valeur de PC l'adresse de début de l'instruction plus 2.

3.5.8 Remarques

L'adressage relatif est utilisé dans les processeurs simples surtout pour les sauts, afin d'économiser de la place mémoire. Sa raison d'être dans un processeur plus évolué est de permettre l'écriture de modules, de programmes en code *indépendant de la position* ou *translatable* (*relocatable*), c'est-à-dire que l'on peut déplacer en mémoire sans devoir recalculer toutes les adresses. Des déplacements de 16 bits, voire plus, sont alors nécessaires pour permettre une taille suffisante des modules de programmes.

Il est important de rappeler que les notations utilisées pour l'écriture du programme en assembleur sont de deux types:

- d'une part la notation logique LOAD A, ADVAL; le programmeur s'intéresse au contenu de la position mémoire, ADVAL, et peu lui importe comment le processeur va se débrouiller pour l'atteindre;
- d'autre part la notation réelle du mode d'adressage utilisé (en page 0, relatif, etc.).

Un bon assembleur doit permettre au programmeur d'utiliser une notation logique et de choisir à sa place le mode d'adressage le plus court et le plus rapide. Le problème

n'est pas si simple, puisque certains modes d'adressage dépendent de l'exécution du programme (par exemple l'adressage en page mobile) et que le programmeur peut préférer lors de la mise au point les adresses absolues, et lors de l'utilisation finale du programme des adresses relatives.

Le paragraphe 3.6.4 montrera comment affiner les notations pour permettre l'expression précise de chaque mode d'adressage.

3.5.9 Principe des adressages indirects

L'opérande de l'instruction peut être non pas son adresse, mais l'adresse d'une position mémoire ou d'un registre contenant l'adresse de l'opérande. Ce type *d'adressage indirect* (*indirect, deferred*) a de nombreuses variantes et une terminologie complexe, de par le fait que plusieurs niveaux d'indirection peuvent avoir lieu avec certains processeurs, et que les variables ou constantes peuvent être ajoutées lors du calcul de l'adresse finale.

Un exemple type d'application est l'accès de élément i du j-ième tableau du k-ième ensemble de tableaux (fig. 3.31). L'expression qui traduit le calcul d'adresse est tout naturellement $i + \{j + \{k + \text{ADTABLE}\}\}$

Chaque paire d'accolades caractérise le contenu de l'adresse exprimée entre accolades. Nous éviterons d'utiliser des parenthèses dans ce but, comme avec la plupart des langages d'assemblage, pour éviter la confusion avec des opérations arithmétiques traditionnelles, qui peuvent du reste se mélanger avec des calculs d'adresse.

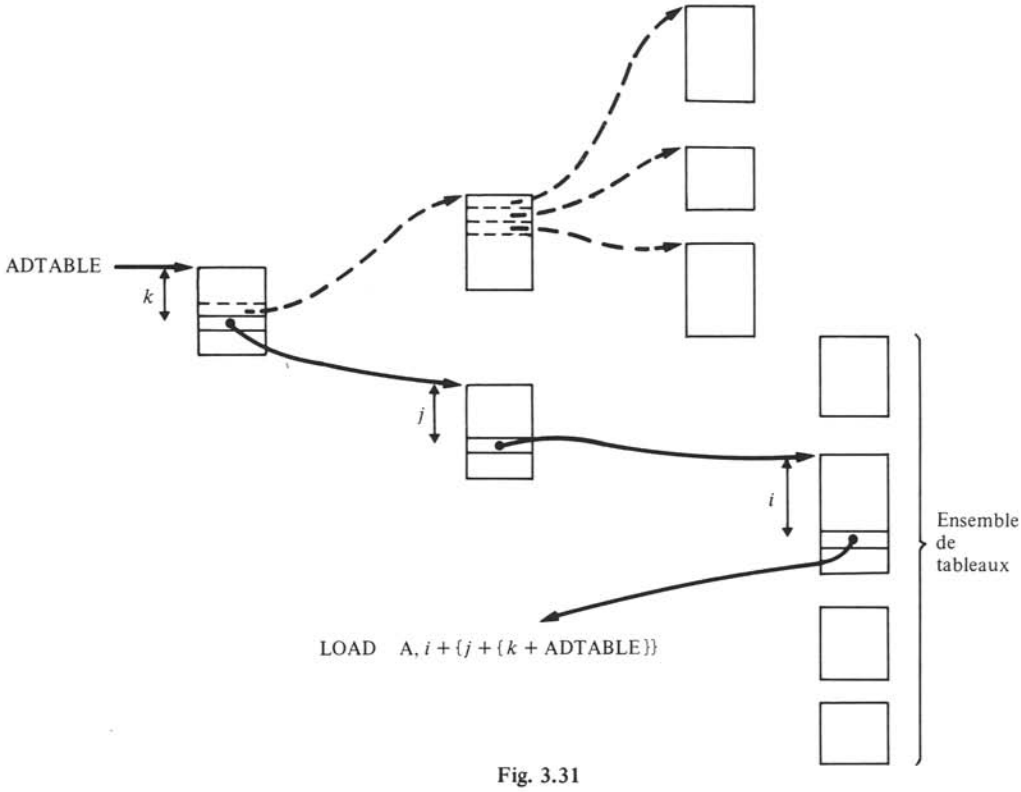


Fig. 3.31

La figure 3.31 ne fait aucune hypothèse particulière concernant les espaces d'adressage contenant les différents tableaux.

Traditionnellement, on fait une distinction entre l'adressage direct, lorsque l'adresse de l'opérande est en mémoire principale, et l'adressage indexé, lorsque l'adresse de l'opérande est dans l'espace des registres du processeur.

3.5.10 Adressage indexé simple

L'*adressage indexé simple* (*register deferred*) se réfère à un ou plusieurs registres du processeur appelés *registres d'index* ou *pointeurs* (fig. 3.32). Les processeurs 8085 et Z80 par exemple ont les registres BC, DE et HL utilisables pour l'adressage indexé, le registre HL dispose de plus d'instruction que BC et DE; ces processeurs ne sont donc pas du tout orthogonaux.

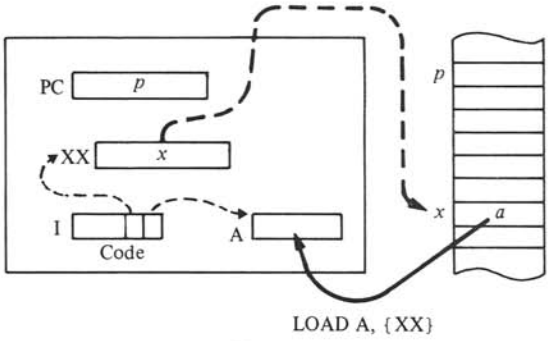


Fig. 3.32

On peut remarquer que le code de l'instruction est très court. Il suffit de spécifier que l'adressage est indexé et non pas direct ou relatif. S'il y a plusieurs registres d'index, il faut encore donner l'adresse du registre d'index.

L'adressage indexé est souvent utilisé à la place de l'adressage direct.

Le processeur 8008, ancêtre du 8085/Z80 et développé en 1971, ne disposait pas d'adressage direct et obligeait à écrire:

$$\left.\begin{array}{l}\text{LOAD HL, \#ADVAL}\\ \text{LOAD A,\{HL\}}\end{array}\right\} \text{ pour LOAD A, ADVAL} \qquad (3.21)$$

L'adressage indexé permet ainsi de simuler un adressage indirect. L'instruction LOAD A, {ADVAL} peut se remplacer par

$$\left.\begin{array}{l}\text{LOAD HL, ADVAL}\\ \text{LOAD A, \{HL\}}\end{array}\right\} \text{ ou } \left.\begin{array}{l}\text{LOAD HL, \#ADVAL}\\ \text{LOAD HL, \{HL\}}\\ \text{LOAD A, \{HL\}}\end{array}\right\} \qquad (3.22)$$

L'instruction LOAD HL, {HL} n'existe pas sur le 8085/Z80, mais peut se remplacer par

$$\left.\begin{array}{l}\text{LOAD A, \{HL\}}\\ \text{INC HL}\\ \text{LOAD H, \{HL\}}\\ \text{LOAD L, A}\end{array}\right\} \qquad (3.23)$$

3.5.11 Application

Illustrons l'intérêt de l'adressage indexé par un programme d'application simple: la somme d'une suite de 4 nombres placés en mémoire à partir de l'adresse TABLE. Avec une architecture à un seul accumulateur, ce programme peut s'écrire

```
LOAD    A, TABLE
ADD     A, TABLE + 1
ADD     A, TABLE + 2
ADD     A, TABLE + 3
```
(3.24)

En utilisant l'adressage indexé, on écrit:

```
LOAD    XX, #TABLE
LOAD    A, {XX}
INC     XX
ADD     A, {XX}
INC     XX
ADD     A, {XX}
INC     XX
ADD     A, {XX}
```
(3.25)

Ce deuxième programme nécessite plus d'instructions, mais elles sont plus courtes car elles ne contiennent qu'une adresse de registre d'index très courte. De plus, la répétition de groupes d'instructions identiques permet de faire une boucle de programmation (sect. 4.4) et de généraliser facilement le programme pour l'addition de tableaux de longueur quelconque.

3.5.12 Remarques

Dans le processeur, la longueur des registres d'index et des registres arithmétiques ne doit pas nécessairement être la même. L'une est fixée par la dimension de la mémoire, l'autre par la précision des nombres et le format des instructions. La plupart des miniordinateurs utilisent des registres de 16 bits pour l'un et l'autre. Les microprocesseurs ont généralement des registres d'index 16 bits et des registres arithmétiques 8 bits. Le CDC 7600 a des registres d'index de 18 bits et des registres arithmétiques de 60 bits et deux unités arithmétiques différentes pour le calcul des adresses (18 bits) et des nombres (60 bits).

3.5.13 Auto-modification

On a vu que l'adressage est souvent utilisé pour parcourir les éléments d'un tableau (§ 3.5.11). Une opération arithmétique simple doit être effectuée pour passer à l'élément suivant du tableau, et certains processeurs effectuent cette opération automatiquement lors de l'adressage plutôt que de forcer le programmeur d'écrire une instruction supplémentaire.

On parle alors *d'adressage post-autoincrémenté* et *pré-autoincrémenté* selon que l'incrémentation se fait avant ou après le traitement mémoire (fig. 3.33)

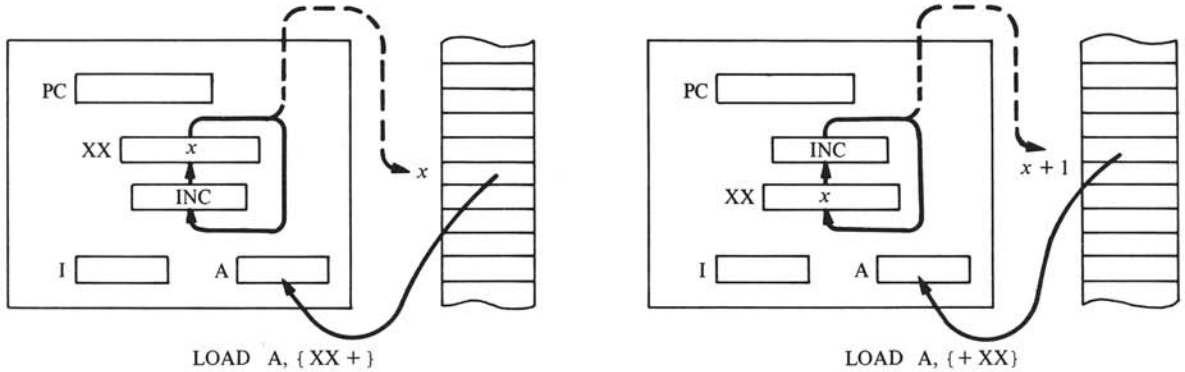


Fig. 3.33

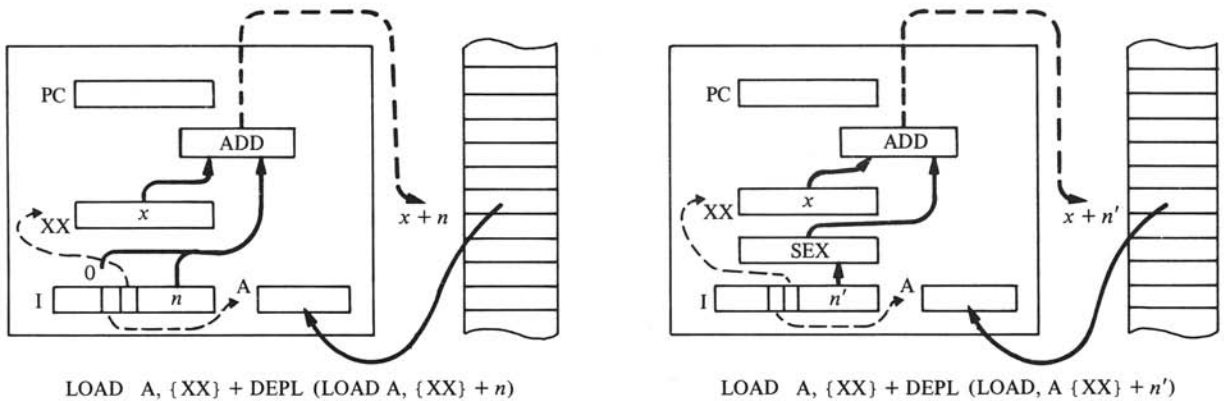


Fig. 3.34

On a de la même façon des adressages autodécrémentés, ce qui s'exprime par les notations

```
LOAD   A, {XX +} post-autoincrémenté
LOAD   A, {+ XX} pré-autoincrémenté
LOAD   A, {XX −} post-autodécrémenté
LOAD   A, {− XX} pré-autodécrémenté
```
(3.26)

Les processeurs bien conçus disposent en général d'un adressage post-autoincrémenté et pré-autodécrémenté, pour permettre la réalisation d'une pile.

En effet, si l'on se réfère à la figure 3.23, placer un élément sur la pile revient à écrire de façon indexée par rapport au registre pointeur de pile SP, avec décrémentation préalable pour pointer la prochaine place vide en mémoire. Lorsque l'on reprend cet élément, le pointeur est déplacé après lecture pour pointer au repos le sommet de la pile. Il y a naturellement des fabricants qui remplissent la pile dans l'autre sens, ou pointent avec SP en dessus du sommet.

Dans le cas des appels et retours de sous-programmes, on peut écrire de façon équivalente:

```
PUSH   PC   ou   LOAD   {− SP}, PC
POP    PC   ou   LOAD   PC, {SP +}
```
(3.27)

L'incrémentation peut se faire de un ou deux (cas du M6809), ou dépendre de la taille des mots transférés (M68000), de façon à pointer automatiquement l'élément suivant en tenant compte de sa taille (1, 2, 4 octets). Les notations d'assembleur doivent être explicites dans ce sens.

3.5.14 Adressage indexé avec déplacement

Un déplacement fixe ou immédiat, codé dans le champ de l'instruction, peut être ajouté au contenu du registre d'index. La longueur de ce déplacement est en général plus courte que celle du registre d'index. Un déplacement 8 bits est suffisant pour accéder aux éléments d'une table de 256 éléments. Suivant les cas, le déplacement est ajouté avec ou sans extension du signe, c'est-à-dire que pour un déplacement 8 bits, le déplacement est de 0 à 255 décimal, ou de −128 à +127 (fig. 3.34).

Certains microprocesseurs ont un déplacement long (16 bits) et un registre d'index court (8 bits). Le rôle des deux éléments de l'adresse est alors en général inversé, la valeur immédiate dans l'instruction pointant le début de la table, et le contenu du registre définissant le déplacement dans la table.

L'adressage indexé le plus flexible permet à la fois un déplacement fixe et un déplacement dans un registre. C'est le cas du M68000, qui permet les deux modes d'adressage représentés dans la figure 3.35. La notation utilisée en légende de cette figure sera expliquée au paragraphe 3.6.4.

La recherche du n-ième élément d'une table implique la connaissance de la taille des éléments de cette table, supposés tous pareils. Si ces éléments ont 4 octets, il faut multiplier le déplacement par 4. Dans des processeurs 16 et 32 bits récents, cette multiplication est automatique, et dépend du type de nombre transféré. Une parenthèse carrée à la place de l'accolade est utilisée pour le registre dont le déplacement est multiplié automatiquement.

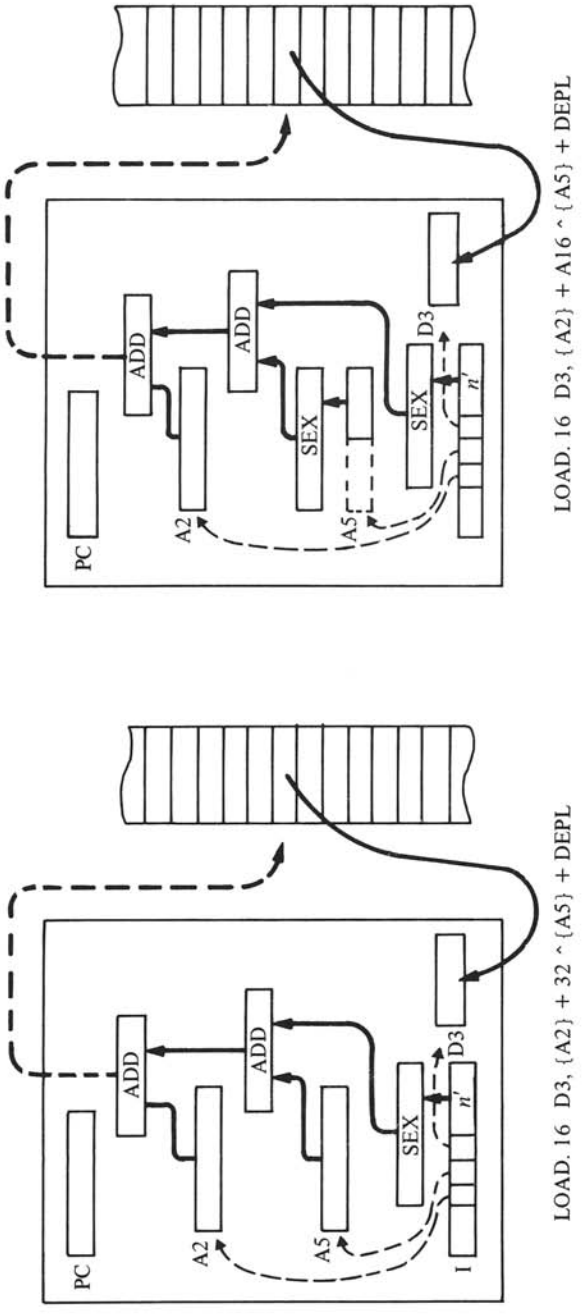

PC
ADD
ADD
A2
SEX
A5
SEX
D3
n'
LOAD. 16 D3, {A2} + A16 ^ {A5} + DEPL
PC
ADD
ADD
A2
A5
SEX
D3
n'
I
LOAD. 16 D3, {A2} + 32 ^ {A5} + DEPL

Fig. 3.35

Par exemple, le VAX a le mode d'adressage

```
LOAD. 32   R0, {R1} + [R2] + BASE      équivalent à
LOAD. 32   R0, {R1} + 4 * {R2} + BASE                    (3.28)
```

La complexité plus grande des opérations d'adressage sur les nouveaux processeurs comme le M68000 implique des notations plus précises qui définissent le nombre de bits utilisés pour le calcul d'adresse, la nature arithmétique ou logique des adresses (§ 3.6.4) et la taille des opérandes (§ 3.6.1).

3.5.15 Adressage indirect

Rappelons que l'adressage indirect au sens étroit usuel se réfère à une position mémoire et non pas à un registre comme l'adressage indexé. Pour pouvoir utiliser le contenu de cette position mémoire comme adresse de sélection, un transfert préalable dans un registre temporaire TT est nécessaire. Ce registre joue alors le rôle de registre d'index (fig. 3.36).

En général, l'adressage indirect offre moins de variantes que l'adressage indexé, car les instructions correspondantes sont très longues et d'une exécution ralentie par les accès mémoire supplémentaires.

L'adressage indirect peut être limité à la page 0 ou relatif suivant que la première adresse donnée dans l'instruction, est elle-même en page 0 ou relative. Dans certains ordinateurs, par exemple les NOVA-ECLIPSE, la position mémoire contient en plus de l'adresse un bit indiquant que l'adresse est elle-même une adresse indirecte: la position "im" contient l'adresse de l'information à transférer et le nombre de niveaux d'indirection n'est pas limité.

3.5.16 Adressages indirects indexés

Adressages indirect et indexé peuvent se combiner, mais on se ramène dans presque tous les cas à un cas particulier du modèle général de la figure 3.31, qui correspond à la fonction importante que chaque fabricant de processeurs cherche à faciliter au maximum. Toutefois, les contraintes de vitesse et de complexité du processeur obligent chaque fabricant à se limiter à quelques cas particuliers, auxquels des noms ambigus et des notations peu intuitives sont donnés.

Par exemple, si on dispose d'un niveau d'indexation et d'indirection, avec une valeur immédiate dans l'instruction et un additionneur d'adresses, on peut envisager trois modes d'adressage:

```
a)   {VAL + {XX}}
b)   {XX} + {VAL}                                        (3.29)
c)   VAL + {{XX}}
```

Le premier mode est parfois appelé *préindexé* ou *indirect indexé* (*indexed indirect*) et les deux suivants *postindexé* ou *indexé indirect* (*indirect indexed*). La figure 3.37 montre l'effet en mémoire de ces modes d'adressage et met en évidence la symétrie des deux derniers modes. Il ne revient toutefois pas tout-à-fait au même pour le programmeur d'avoir comme paramètre d'assemblage {VAL} une adresse de base ou un déplacement dans une table.

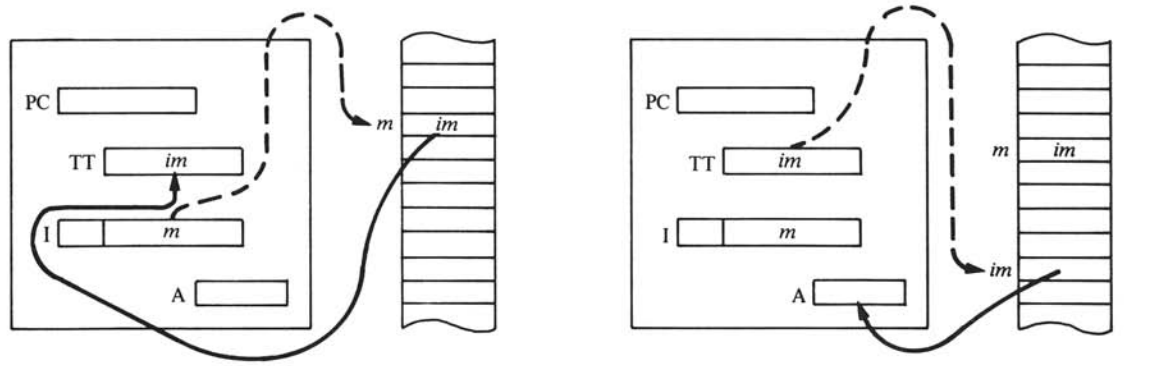


Fig. 3.36

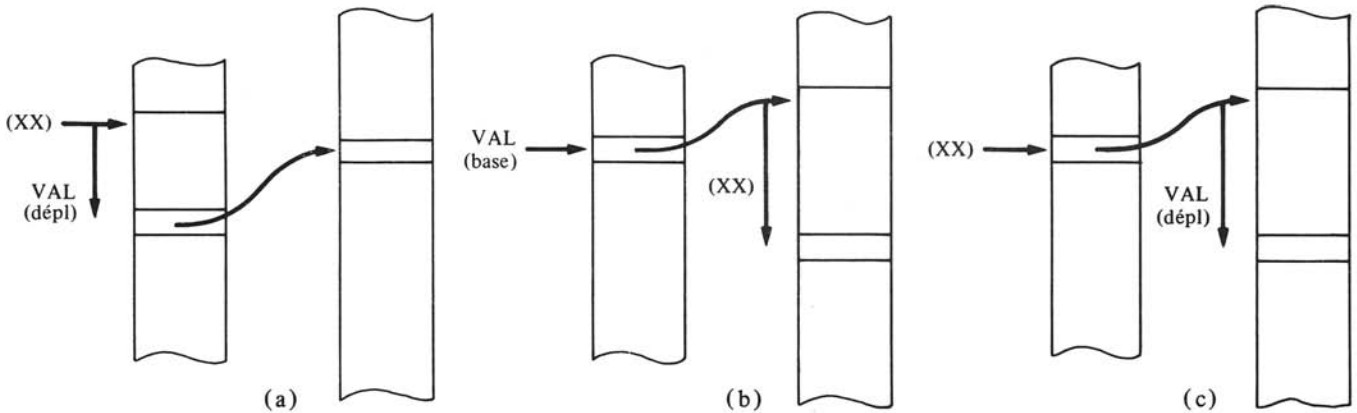


Fig. 3.37

3.5.17 Note sur l'adressage immédiat

L'adressage immédiat simple vu au paragraphe 3.5.2 se généralise à toute adresse calculée par le processeur suite à un mode d'adressage plus ou moins complexe, mais qui, au lieu d'être utilisée finalement pour prélever un opérande, est transférée comme étant l'opérande.

Le M68000 dispose de cette facilité, qui est considérée par le fabricant comme une instruction spéciale (Move Address) plutôt que comme un mode d'adressage.

3.5.18 Autres modes d'adressage

Les noms donnés aux modes d'adressage, et les notations utilisées sont très nombreux mais tous ces modes rentrent dans l'un des types vus ci-dessus. Les notations utilisées par les fabricants sont souvent simplifiées, le seul but étant de permettre la distinction entre les modes d'adressage possibles sur un seul processeur.

3.6 TYPES DE DONNÉES

3.6.1 Taille des opérandes

Dans un microprocesseur simple, les opérandes ont 8 bits ou 16 bits, et le nom du registre avec lequel le transfert s'effectue suffit à préciser quelle est la taille de l'opérande. Par exemple, avec les processeurs de la famille 8080,

```
LOAD  A, ADVAL      transfère 8 bits        (3.30)
LOAD  HL, ADVAL     transfère 16 bits
```

Pour un processeur plus perfectionné, surtout si des transferts directs mémoire-mémoire sont possibles, ces notations ne sont pas assez précises. La taille de l'opérande est alors notée en postfixe du code mnémonique.

Par exemple, avec les microprocesseurs de la famille M68000, on écrit

```
LOAD. 8   D0, ADVAL.  ou  LOAD. B   D0, ADVAL
LOAD. 16  D0, ADVAL   ou  LOAD. W   D0, ADVAL      (3.31)
LOAD. 32  D0, ADVAL   ou  LOAD. L   D0, ADVAL
```

La notation MOVE est souvent utilisée à la place du LOAD pour ce processeur.

Il peut arriver que l'opération n'utilise pas la même taille pour l'opérande source et l'opérande destination. Un postfixe peut alors être ajouté à chacun des opérandes, de façon à aider le programmeur à bien distinguer des instructions complexes.

Simultanément à la taille, la nature du nombre manipulé (nombre logique, arithmétique, BCD, flottant) peut être précisée. Ceci pourrait être donné dans tous les cas, mais entraînerait un alourdissement inutile de la notation. La plupart des fabricants précisent cette nature dans le mnémonique de l'instruction, ce qui est plus simple lorsque le répertoire est pauvre et peu orthogonal.

Par exemple, le M68000 a deux instructions de multiplication, pour les nombres arithmétiques logiques (non signés) et signés. On a le choix pour les notations.

```
MUL     D0, D1              MULA     D0, D1              (3.32)
MUL.16  D0, D1              MUL. A16 D0, D1              (3.33)
MUL     D0.32, D0.16, D1.16 MUL      D0.A32,D0. A16,D1.A16 (3.34)
```

La première notation (3.32), qui est proche de celle du fabricant, est possible par le fait que le M68000 n'a pas d'autre instruction de multiplication (de deux mots de 8 bits ou de 32 bits par exemple). La troisième (3.34) montre explicitement que le produit (1er opérande) a 32 bits, et qu'il n'a donc jamais de dépassement de capacité. La lettre A est réservée pour les nombres arithmétiques en complément à 2, la lettre S pour les nombres avec signe et valeur absolue. Les nombres logiques (non signés) sont caractérisés seulement par leur nombre de bits.

Dans la pratique on utilise plutôt des notations courtes dans le programme, mais un langage clair de description des instructions est utile lorsqu'il faut expliciter un comportement particulier ou plus complexe.

3.6.2 Adresses et sous-adresses

Les éléments d'un espace mémoire ont tous la même taille et sont numérotés pour qu'à chaque élément corresponde une adresse unique. La plupart des fabricants considèrent que la taille naturelle est l'octet de 8 bits. Les éléments plus grands (doublets, quadlets) sont caractérisés par l'adresse de leur premier octet. Un doublet de 16 bits comporte deux octets dont l'ordre est important. Le premier est souvent l'octet de gauche, en supposant que l'on ait écrit le mot de 16 bits avec les poids faibles à droite comme cela a été fait souvent dans le chapitre 2. A peu près toutes les conventions se rencontrent et le programme doit commencer à bien identifier les options prises par le fabricant.

Le même problème se pose s'il s'agit de définir une sous-adresse de quartet ou de bit dans un octet. Une majorité de fabricants opte heureusement pour une numérotation identique au rang binaire, c'est-à-dire de droite à gauche.

La figure 3.38 compare les conventions qui se présentent avec les processeurs M68000 et VAX et montre différentes représentations de la même information (nombre binaire, nombre BCD, texte) dans l'espace des registres (éléments de 32 bits) et dans l'espace mémoire (éléments de 8 bits). La solution du VAX peut paraître surprenante pour les nombres, mais elle correspond à la logique des opérations binaires qui commencent toujours par les poids faibles.

3.6.3 Adressage de bits

Une adresse de bits se caractérise par l'adresse du mot qui contient ce bit, et l'adresse de ce bit dans le mot. Les modes d'adressage de sous-adresse de bits sont simples (immédiat ou direct). Le signe : est utilisé pour séparer les deux parties de l'adresse.

On écrit par exemple pour le M68000:

```
SET  D0 : #3     Met à 1 le bit numéro 3
SET  D0 : D1     Met à 1 le bit dont le rang est dans D1
SET  D0 : 3      Met à 1 le bit dont le rang est dans
                 la position mémoire 3
```
(3.35)

Abusivement, une ancienne notation CALM pour le Z80 écrivait:

```
SET    A : 3        au lieu de SET A : #3
```
(3.36)

Etant donné qu'il n'est pas possible avec le Z80 de lire le rang de bit en mémoire (dans la position 3 par exemple), cette notation était plus simple, mais manquait de rigueur. Ce genre d'abus de notation est malheureusement pratiqué par tous les fabricants.

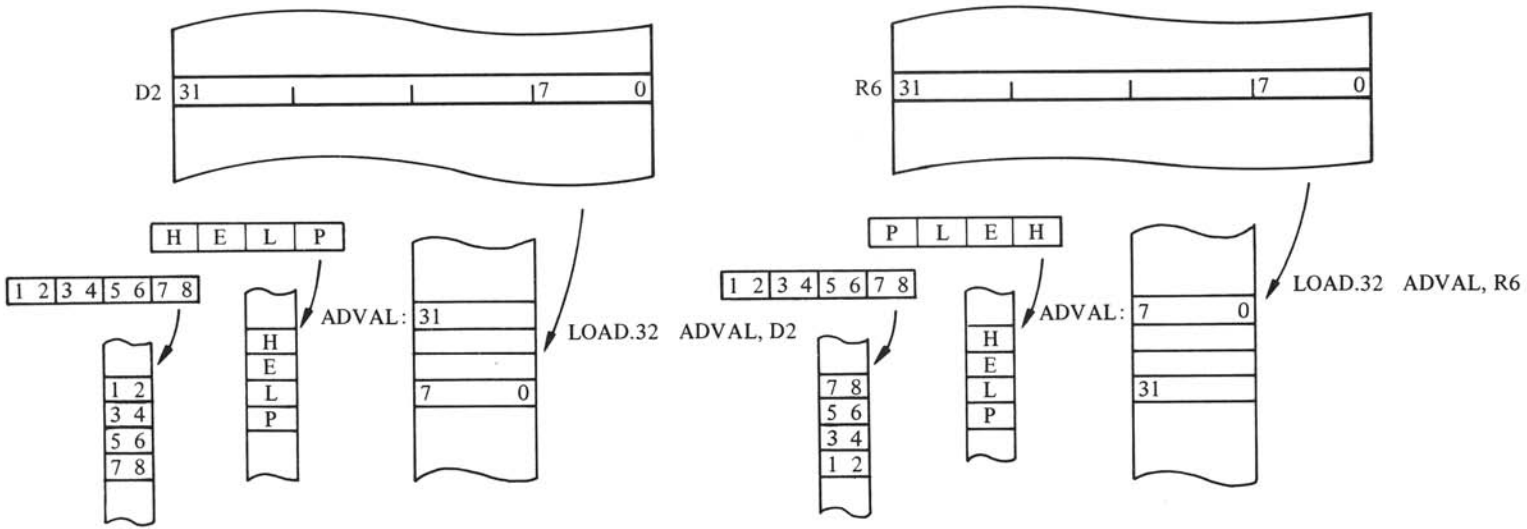

D2
31
7
0
H E L P
1 2 3 4 5 6 7 8
ADVAL:
LOAD.32 ADVAL, D2
1 2
3 4
5 6
7 8
H
E
L
P
31
7 0
R6
P L E H
LOAD.32 ADVAL, R6

Fig. 3.38

Une chaîne de bits peut être adressée par une expression un peu plus complexe. L'adresse du premier bit et la longueur de la chaîne suffisent à caractériser une telle chaîne.

3.6.4 Sous espace d'adressage

Un grand espace d'adressage implique des adresses longues, qui prennent de la place en mémoire. Plusieurs modes d'adressage, par exemple l'adressage en page zéro (§ 3.5.4), utilisent un sous-espace qu'il est intéressant de savoir noter précisément.

De façon générale, on peut convenir qu'un espace d'adressage est infini, et préciser en préfixe de l'opérande la taille de l'adresse. On écrit

```
LOAD. 8     D0, 16 ^ ADVAL ou
LOAD        D0.8, 16 ^ ADVAL.8                    (3.37)
```

La notation devient lourde puisque la taille de l'opérande peut être notée en postfixe, mais avec deux éléments on caractérise complètement un opérande et son mode d'accès (fig. 3.39(a)).

Cette notation permet de bien caractériser le mode d'adressage en page 0 (§ 3.5.4). Lorsque les adresses sont considérées comme des nombres arithmétiques avec extension du signe, la lettre A précède la taille (fig. 3.39(b)). Par exemple, avec le M68000, une adresse en page 0 est caractérisée par la notation A16 ^ ADVAL. Lorsqu'il y a le choix entre adresse courte et adresse longue, le programme de traduction connaît la valeur ADVAL et choisit toujours l'adresse la plus courte. En précisant l'espace avec, dans le cas du 68000 la notation A16 ^ ADVAL, il y a vérification de la valeur de l'adresse, et mention d'une erreur si cette valeur ne peut pas s'exprimer sous forme d'un nombre arithmétique 16 bits.

En précisant 32 ^ ADVAL, une adresse longue de 32 bits est utilisée pour l'instruction, même si la valeur de l'adresse est inférieure à 16 bits.

Un cas similaire se pose dans les adressages indexés avec déplacement, pour caractériser le sous-espace dans lequel ce déplacement peut se faire (fig. 3.39(c) et (d)).

Par exemple, le processeur 6800 a un mode d'adressage indexé avec déplacement 8 bits positif, et se décrit par l'expression d'adressage (IX) + 8 ^ DEPL. Le processeur Z80 admet par contre des déplacements 8 bits en complément à 2, et il faut écrire {IX} + A8^DEPL.

Le cas de l'adressage relatif est plus délicat à comprendre, car le pointeur à l'adresse courante n'est pas mentionné explicitement et a une valeur d'exécution qui dépend de l'architecture interne du processeur.

La notation Rn ^ ADVAL caractérise une adresse relative n bits, c'est-à-dire codée par rapport au PC avec un déplacement n bits (§ 3.5.7). On a donc:

$$Rn \wedge ADVAL = (PC) + An \wedge DEPL$$

La figure 3.39(e) met en évidence le sous-espace relatif. Comme indiqué au paragraphe 3.5.7, il faut bien distinguer l'adresse de l'instruction contenant l'adresse relative d'un opérande et la valeur du compteur ordinal PC lorsqu'il exécute le transfert de l'opérande.

3.6.5 Caractérisation des espaces d'adressage

Chaque processeur a de 2 à 4 espaces d'adressage. L'espace des registres est caractérisé par des noms réservés (A, B, IX, RO), l'espace mémoire par des nombres (adresses)

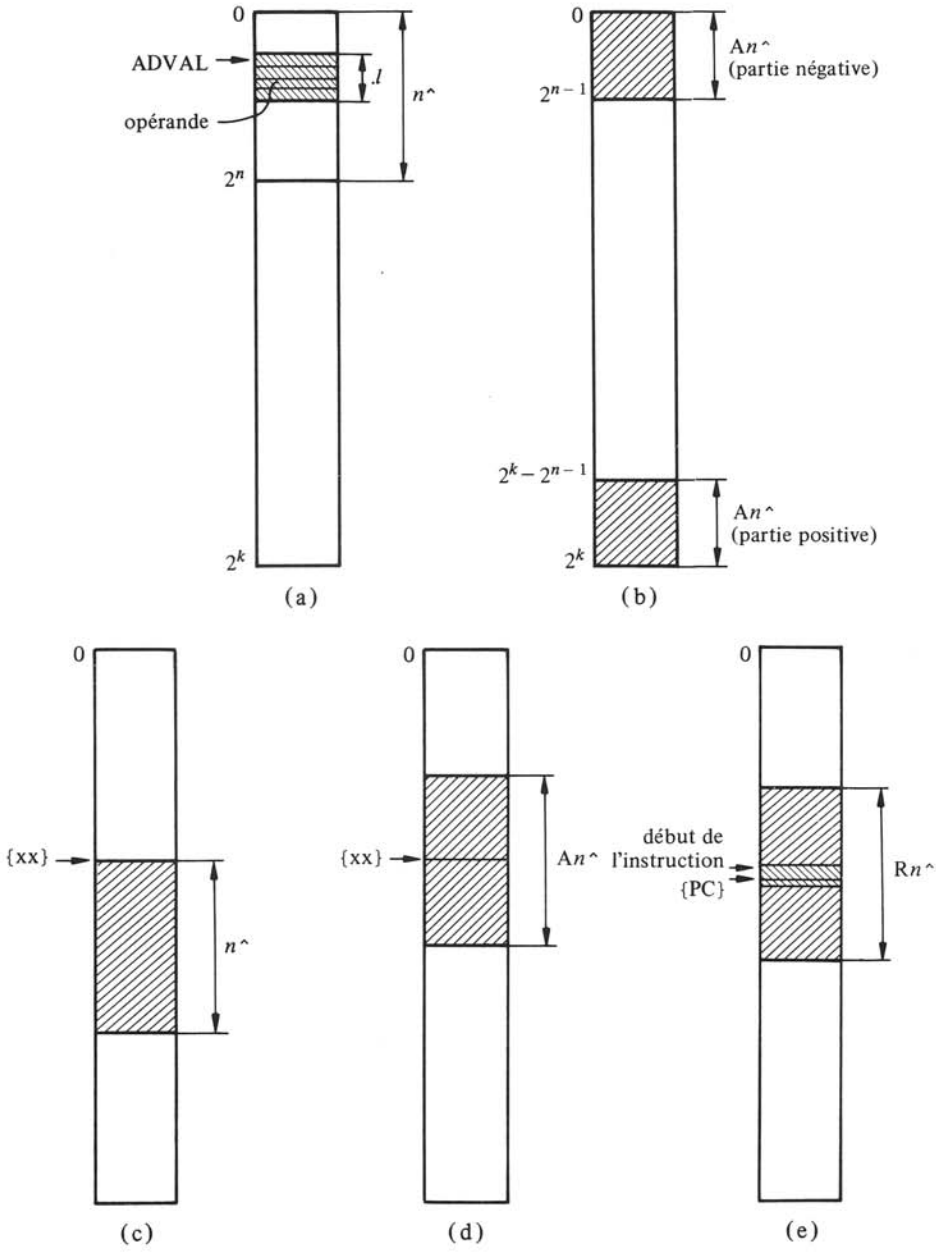


Fig. 3.39

notés par des symboles mnémotechniques. Les espaces supplémentaires nécessitent des symboles caractérisés par un signe spécial: % pour un espace de données supplémentaire (processeur 8048), $ pour l'espace des entrées-sorties (processeur 8085/Z80). Des codes mnémoniques spéciaux sont souvent utilisés par les fabricants:

```
LD      A, ADVAL    lecture en mémoire
INP     A, ADPER    lecture d'un périphérique
CALM :                                          (3.38)
LOAD    A, ADVAL
LOAD    A, $ADPER
```

3.6.6 Types d'opérandes

Les opérandes sont dans les processeurs simples des entiers de 8 ou 16 bits, arithmétiques ou logiques (chap. 2). Le type décimal (codé BCD) n'existe généralement que sous forme logique, avec 2 ou 4 digits.

A titre d'exemple la figure 3.40 donne les types de données manipulés par des instructions des processeurs Z80, M68000 et IAPX 432 [62, 63]. Il est évident que par programmation ou avec des circuits supplémentaires, des types de données quelconques peuvent être réalisés. Les types de données flottants de l'IAPX 432 sont du reste manipulés par un processeur arithmétique associé.

3.6.7 Modèle pour le programmeur

Un processeur est une structure très riche et variée avec ses espaces et modes d'adressage, ses types de données et son répertoire d'instructions de plus en plus complet.

La première chose que le programmeur doit avoir présente à l'esprit est l'espace des registres, avec les indicateurs, les différents espaces d'adressage avec leurs sous-espaces et zones réservées, et les types de données.

Ce modèle prend pour le processeur M68000 [61] l'allure de la figure 3.41.

Lors de l'écriture du programme en langage d'assemblage (chap. 4), l'affectation des registres et de la mémoire et l'effet des instructions sur les indicateurs doivent rester constamment présents à l'esprit du programmeur.

3.6.8 Architectures de microprocesseurs

Le modèle du programmeur fait ressortir l'architecture de chaque système, ce terme général d'architecture évoquant l'équilibre entre les différents espaces d'adressage, leurs facilités d'accès et les opérations que l'on peut y effectuer.

L'architecture primitive de von Neumann était basée sur un registre accumulateur et un espace mémoire unique. La plupart des microprocesseurs ont repris cette philosophie, en augmentant le nombre de registres proches de l'unité arithmétique, pour accélérer l'exécution. Le nombre de registres, comme on le voit dans le M68000 (fig. 3.41), peut devenir très important, ce qui est très efficace pour un processeur donné, mais très lent lorsque le processeur doit quitter temporairement une tâche pour en effectuer une autre. Dans un tel cas, les contenus de tous les registres utilisés doivent être sauvés en mémoire avant de pouvoir affecter ces registres à une autre tâche.

Une solution pour pallier ce défaut est de dédoubler les banques de registres. Cette solution a été utilisée dans le Z80 [51], mais ne peut être utilisée que pour une 2ème tâche unique, et non pas pour plusieurs tâches imbriquées. La communication entre espaces dédoublés est par ailleurs souvent très difficile.

Fig. 3.40

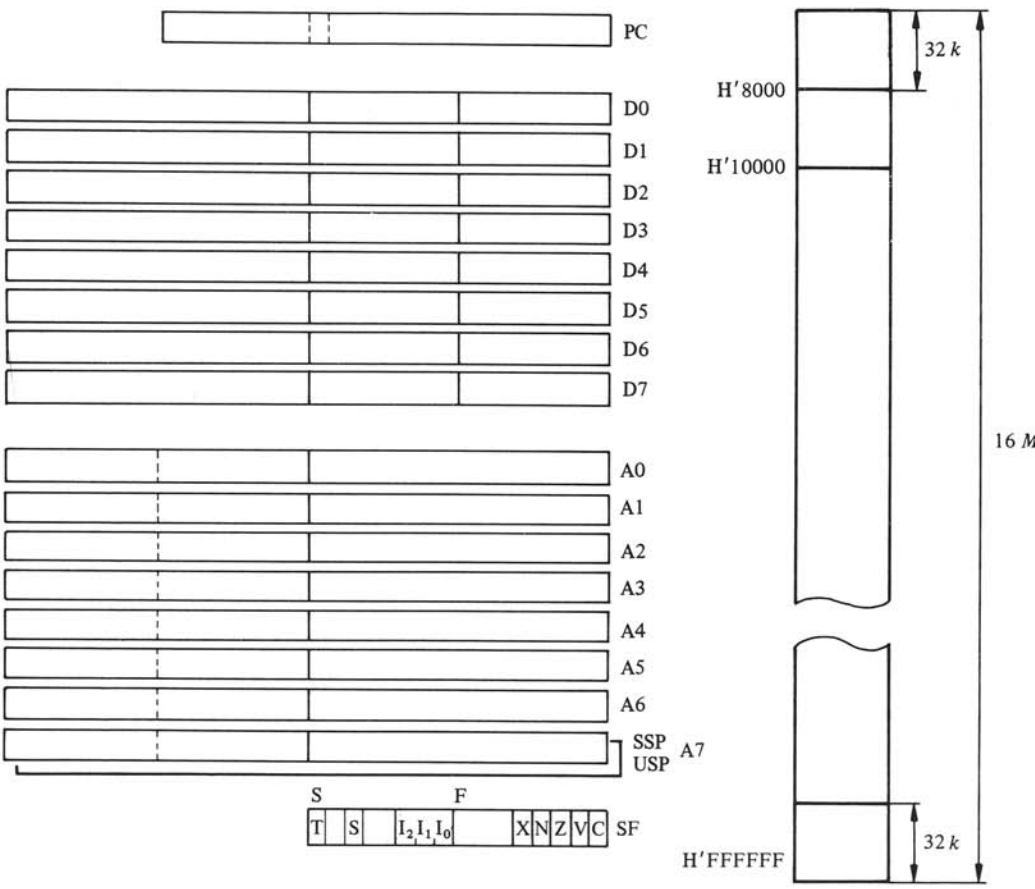


Fig. 3.41

Une architecture assez différente consiste à n'avoir qu'un seul espace mémoire, et un pointeur définissant un *sous-espace de travail* (*workspace*) correspondant en fait à un groupe de registres. Cette architecture (fig. 3.42) est utilisée par les processeurs de la famille Texas 9900, et a l'inconvénient d'une plus grande lenteur due à l'accès de chaque information dans une mémoire plus grande, généralement extérieure au processeur.

Ce type d'architecture peut avoir trois niveaux, comme dans le cas du processeur RCA 1802 [53] (fig. 3.43). Ceci réduit évidemment encore la vitesse. Ce processeur très simple par ailleurs, dispose de pointeurs 4 bits à des registres 16 bits qui peuvent eux-mêmes pointer de l'information en mémoire.

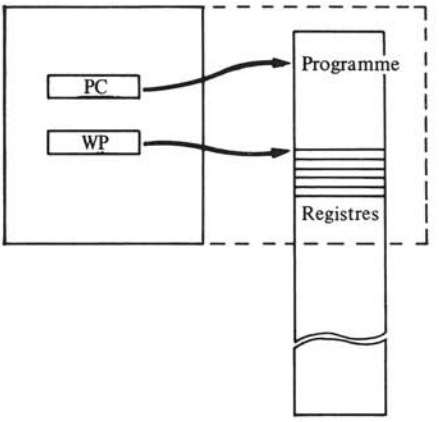


Fig. 3.42

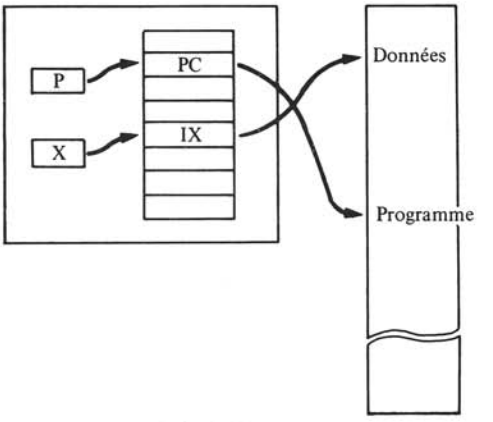


Fig. 3.43

3.6.9 Largeur d'un processeur

Les processeurs ont une *largeur* de (ou sont dits à) 4 bits, 8 bits, 16 bits, etc., mais cette terminologie dépend de plusieurs paramètres.

- L'unité arithmétique et logique opère sur des mots de 4, 8, 16, 32 bits ou plus. Plusieurs types de données peuvent être possibles dans le microprocesseur (fig. 3.40).
- Les instructions ont un format qui arrange les champs par groupes de 8, 16 bits ou plus. La longueur totale varie selon l'instruction et l'adressage.
- Les adresses mémoire sont définies à partir d'éléments de 8, 16, 24 bits. Le nombre de lignes d'adresse est fixe pour un processeur donné.
- Les mots mémoire peuvent être transférés avec le processeur sur un chemin de données de 8, 16, 32 bits et des variantes d'un même processeur peuvent ne différer que sur ce point (par exemple le 8086 et le 8088).

Par exemple, une même instruction de transfert d'une valeur immédiate dans l'un des registres principaux peut prendre pour 5 processeurs différents les formes données dans la figure 3.44.

Le TMS 1000 est un processeur dit 4 bits (taille de l'unité arithmétique) avec des instructions 8 bits. Le Z80 est un processeur dit 8 bits (taille du chemin de données) avec des instructions arithmétiques 8 et 16 bits. Le M68000 est un processeur de 16 bits (taille du chemin de données) avec des instructions arithmétiques 8, 16 et 32 bits et un adres-

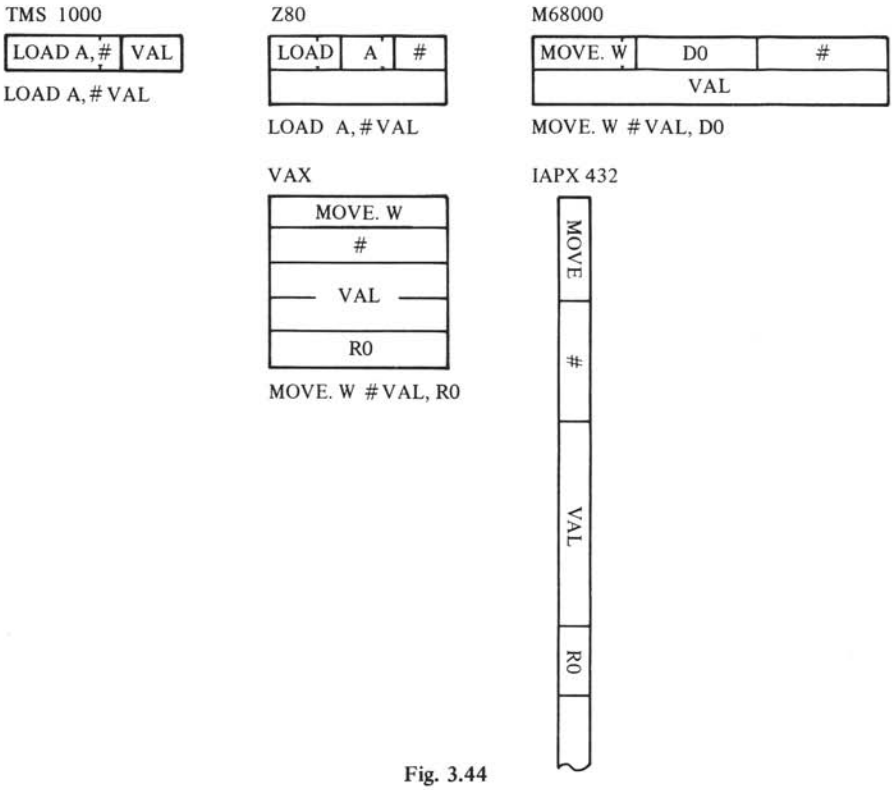


Fig. 3.44

sage par octets, avec des contraintes d'alignement pour les mots de 16 bits et 32 bits. Le VAX 780 est un processeur dit 32 bits (taille du chemin de données) avec des instructions 8, 16, 32 et 64 et 128 bits, et un adressage par octets sans contraintes d'alignement. L'IAPX 432 est un processeur dit 32 bits (décision du groupe de marketing) avec des instructions arithmétiques de 8 à 80 bits, un adressage par bit sans contraintes d'alignement, et un chemin de données 16 bits.

On voit en résumé qu'une terminologie correcte serait de toute façon difficile, étant donné la richesse des solutions liées aux éléments d'un processeur.

3.7 INSTRUCTIONS PARTICULIÈRES

3.7.1 Instructions d'entrée sortie

En toute rigueur, il n'y a pas d'*instructions d'entrée-sortie*, mais des instructions sur un espace d'entrée-sortie, sur lequel les interfaces d'entrée-sortie sont usuellement connectées. Plusieurs processeurs n'ont pas d'espace d'entrée-sortie séparé et adressent les interfaces comme de la mémoire. On dit alors que les entrées-sorties sont *cartographiées en mémoire* (*memory-mapped* I/0).

Lorsqu'il y a un espace d'entrée-sortie séparé cet espace est souvent plus petit (8 bits d'adresse) et permet des instructions plus rapides, mais avec des modes d'adressage très limités.

3.7.2 Déplacement de blocs

Un bloc d'information doit souvent être déplacé d'une zone mémoire à une autre. Une instruction unique peut être implémentée dans ce but. Cette instruction a trois paramètres:

- l'adresse source du début du bloc à déplacer
- l'adresse destination du début de la zone devant recevoir le bloc
- la longueur du bloc.

En général, ces trois informations sont préparées dans des registres de l'unité arithmétique et le transfert se fait en répétant automatiquement le transfert indexé autoincrémenté avec un pointeur source et un pointeur destination.

Un problème apparaît toutefois si le début du bloc destination recouvre le bloc source (fig. 3.45). Le transfert indexé autoincrémenté détruit la fin du registre source avant qu'il soit recopié. Il faut donc dans ce cas commencer par transférer la fin du bloc source et effectuer un adressage indexé autodécrémenté. Des instructions différentes de déplacement de blocs sont donc nécessaires.

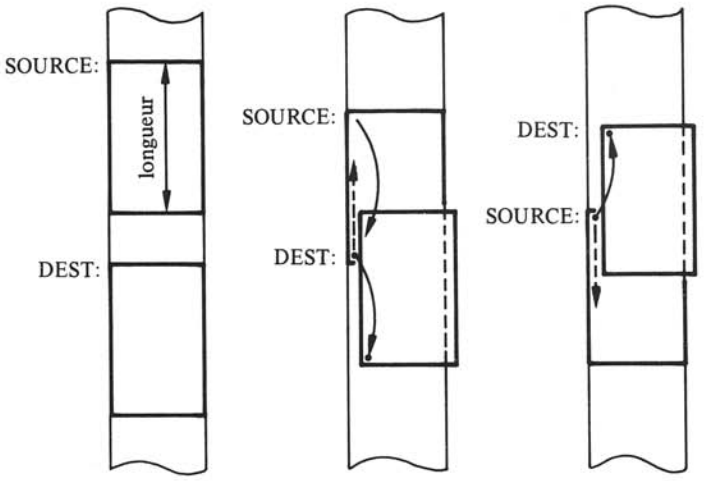


Fig. 3.45

3.7.3 Recherche de caractères

La recherche d'un caractère particulier, ou d'une chaîne de caractères prédéterminée, est une opération courante dans l'analyse d'un ordre ou d'un texte.

Ces opérations ont été pendant longtemps effectuées par des sous-programmes de 20 à 50 instructions, mais l'évolution de la technologie permet d'incorporer, même dans un microprocesseur, des instructions facilitant le traitement de blocs d'information. Les instructions les plus fréquemment implémentées sont les suivantes:

- recherche d'un caractère dans un bloc; les paramètres sont l'adresse et la longueur du bloc, et le caractère; un indicateur permet de savoir si le caractère a été trouvé ou non;
- copie d'une chaîne de caractères jusqu'à ce qu'un caractère d'un certain type soit rencontré;

- copie sélective d'une chaîne de caractères avec suppression ou adjonction automatique de certains caractères;
- comparaison de deux chaînes de caractères.

Dans des applications de traitement de texte, les caractères traités sont les codes ASCII des lettres et signes utilisés dans le texte. Dans le calcul décimal, les caractères sont des chiffres BCD et les instructions ci-dessus peuvent faciliter considérablement la suppression du zéro non significatif ou l'alignement de deux nombres flottants.

3.7.4 Opérations sur des tableaux et listes

De nombreuses instructions peuvent être imaginées pour faciliter l'accès ou la manipulation de tableaux à plusieurs dimensions ou de listes chaînées. Des instructions de ce type sont disponibles dans les gros miniordinateurs, et le deviendront rapidement dans les microprocesseurs.

3.7.5 Coprocesseur

Les opérations en virgule sont complexes et n'intéressent qu'une partie des acheteurs d'un microprocesseur. Une solution est alors d'associer au processeur un processeur arithmétique spécialisé qui reçoit du processeur principal des opérandes et ordres d'exécution d'opérations. On effectue en attendant le résultat des opérations annexes [62]. Si le coprocesseur n'est pas présent, un sous-programme simulant le coprocesseur peut être appelé automatiquement.

3.7.6 Microprogrammation

Les instructions du processeur sont décomposées au niveau du processeur en opérations élémentaires de tranfert et de test, appelées *microinstructions*. Le répertoire de microinstructions est très primitif, mais s'exécute très rapidement. La mémoire de microprogramme est une mémoire morte, qui peut être complétée par une mémoire vive accessible par le programmeur de façon à permettre des instructions spéciales [63].

3.7.7 Pile opérationnelle

Une façon élégante de résoudre le problème d'adressage des registres et de leur limitation en nombre, est d'opérer directement sur une pile comme cela se fait sur les calculettes (§ 3.1.6).

Tous les opérandes deviennent implicites et les bits d'adressage économisés peuvent permettre de définir un répertoire d'instructions plus riche. Les opérations sont également plus rapides, à cause du nombre généralement plus restreint d'opérations de transfert de registres.

Des difficultés apparaissent toutefois si un processeur à pile doit être utilisé pour faire des opérations multiprécision, ou si une unité ne disposant que d'opérations entières doit être programmée pour effectuer des opérations en virgule flottante. L'utilisation de plusieurs piles peut aider à résoudre ces problèmes, mais il semble néanmoins que les *ordinateurs à pile* (*stack computers*) comme le Burroughs 5500 ne connaissent qu'un faible développement. Aucun miniordinateur et microprocesseur ne copie actuellement ce type d'architecture au niveau le plus bas, quand bien même il ait été montré à plusieurs

reprises que ce genre de machine est plus efficace pour les calculs complexes et se prête mieux à l'implémentation des compilateurs [56].

3.7.8 Changement de contexte

Le passage d'une tâche à l'autre, avant que la première tâche ne soit terminée, est souvent nécessaire lorsque la 2ème tâche ne peut attendre, ce qui est souvent le cas avec les périphériques. Le contexte de la tâche, c'est-à-dire les contenus des registres et positions mémoire qui risquent d'être modifiés par la nouvelle tâche, doit être sauvé pour pouvoir être rétabli ultérieurement. Un sauvetage en zones fixes prédéfinies est simple, mais ne permet pas la *rentrance*, c'est-à-dire l'appel d'une tâche alors que l'on se trouve encore dans cette même tâche. Ceci peut se produire dans une routine par exemple si le changement de tâche se produit dans une routine arithmétique, et que la 2ème tâche doit également utiliser cette même routine arithmétique.

L'utilisation d'une ou plusieurs piles est indispensable pour résoudre correctement les problèmes de tâches multiples simultanées. En plus des instructions PUSH ET POP précédemment rencontrées, des instructions peuvent aider à définir des zones de sauvetage et transfert de paramètres appelés *cadre* (*frame*).

3.7.9 Interruptions et trappes

L'instruction CALL permet d'appeler une routine à un instant voulu par le programmeur. L'interruption (sect. 5.3) permet de forcer l'appel d'une routine suite à une action extérieure prioritaire, donc de changer de tâche. Le changement de contexte se fait automatiquement ou par des instructions spécifiques prévues dans la routine d'interruption.

Dans les processeurs prévus pour un seul utilisateur (cf. chap. 5), une bascule dans le processeur indique si l'utilisateur accepte d'être interrompu ou non; l'interruption est transparente, mais son temps d'exécution peut être gênant. Les instructions ION (interrupt on) et IOF (interrupt off) agissent sur cette bascule d'interruptions. Plusieurs niveaux peuvent être prévus dans les interruptions, de façon à ne permettre à chaque instant que des interruptions par des tâches plus importantes que la tâche en cours.

Le besoin d'interrompre la tâche en cours peut provenir de l'exécution du programme lui-même. Par exemple, un dépassement de capacité dans une opération arithmétique, ou un accès dans une portion non-autorisée de la mémoire. Les routines qui sont appelées par ces événements sont appelées *déroutements* ou *trappes* et le programmeur définit leur action. Des instructions notées TRAP permettent de forcer l'appel de routines utilisées lors de la *mise au point* (*debug*) pour détecter le passage par certains points du programme. Ces instructions ne diffèrent en fait des appels de sous-programmes que par un mode d'adressage particulier qui permet des instructions plus courtes.

Une possibilité d'interruption supplémentaire que l'utilisateur ne peut pas inhiber est prévue, en particulier pour que l'annonce d'une panne de courant déclenche des opérations de sauvetage de l'état complet des processus en cours, de façon à permettre la continuation après rétablissement du courant. On parle d'interruptions non masquables (NMI non maskable interrupt).

3.7.10 Instructions indivisibles

L'exécution d'une instruction peut impliquer plusieurs transferts avec la mémoire; dans un système complexe dans lequel plusieurs processeurs se partagent une mémoire

commune, il est possible qu'un processeur modifie le contenu d'une portion mémoire entre la lecture et la réécriture de cette information par un autre processeur.

Une instruction ou un groupe d'instructions est dit *indivisible* si aucune autre action modifiant les registres ou la mémoire ne peut arriver pendant leur exécution. Dans le cas où un processeur exécute plusieurs tâches en parallèle, les instructions indivisibles sont obtenues en plaçant le processeur dans le mode où les interruptions ne sont plus acceptées avec l'instruction IOF.

S'il y a plusieurs processeurs, le blocage du chemin de données en faveur d'un seul processeur doit pouvoir se faire soit par des instructions spéciales dites indivisibles (test et activation de bits en mémoire), soit par activation d'une *ligne de blocage* (*lock*) reliant le processeur et la mémoire. Des instructions spéciales commandent cette ligne.

3.7.11 Instructions privilégiées

Dans certains processeurs, des instructions dites *privilégiées* ou *instructions système* ne sont plus décodées à partir de l'instant où une instruction particulière a été exécutée. Ceci permet de mettre en place des protections (accès à une zone mémoire ou à des périphériques), ce qui est nécessaire lorsque plusieurs utilisateurs se partagent la même machine.

Les interruptions forcent le mode privilégié, et permettent au système de vérifier la bonne utilisation des ressources communes par les différents programmes.

3.8 EXTENSION ET PROTECTION MÉMOIRE

3.8.1 Introduction

La longueur du compteur d'adresse et des mots mémoire fixe la dimension maximum de la mémoire qui peut être adressée par un processeur donné. Deux cas peuvent alors se produire.

Le plus souvent, des contraintes de prix et d'architecture ont conduit à limiter la dimension mémoire adressable par le processeur à une valeur trop faible pour certains utilisateurs. Des techniques particulières permettent alors d'étendre physiquement cette mémoire et d'adresser les zones supplémentaires.

Le problème est inverse dans certains gros ordinateurs et dans les nouveaux microprocesseurs 16 bits où la mémoire achetée par l'utilisateur peut être nettement plus faible que le maximum possible. Plutôt que de limiter l'utilisateur aux zones implémentées, on peut lui donner l'impression de posséder virtuellement une mémoire complète.

Dans ces techniques d'extension mémoire, une mémoire de masse sur un disque magnétique est généralement associée à la mémoire principale. Des circuits adéquats permettent la permutation de blocs de programme entre mémoire principale et mémoire de masse, de façon invisible pour l'utilisateur (sect. 5.4).

Par ailleurs, la puissance des ordinateurs et leur rapidité, relativement à l'homme, permettent et encouragent l'utilisation simultanée du même ordinateur par plusieurs personnes ou programmes. Anciennement, le prix de l'unité arithmétique obligeait ce partage.

Chaque utilisateur ou tâche se voit attribuer une zone mémoire et un programme superviseur donne le contrôle successivement à chacun des programmes coexistants dans la mémoire. Le temps accordé à chaque programme est par exemple de l'ordre de quelques millisecondes, permettant néanmoins l'exécution d'un millier d'instructions.

3.8.2 Extension par blocs

La façon la plus simple d'étendre la mémoire consiste à juxtaposer plusieurs blocs mémoire identiques et à sélectionner l'un ou l'autre en fonction d'un *registre de blocs* BR (*bloc register*) qui fait partie du processeur ou est physiquement considéré comme un périphérique (fig. 3.46). Le chargement du contenu de ce registre donne accès à un bloc mémoire différent, sans changement de l'adresse dans le bloc.

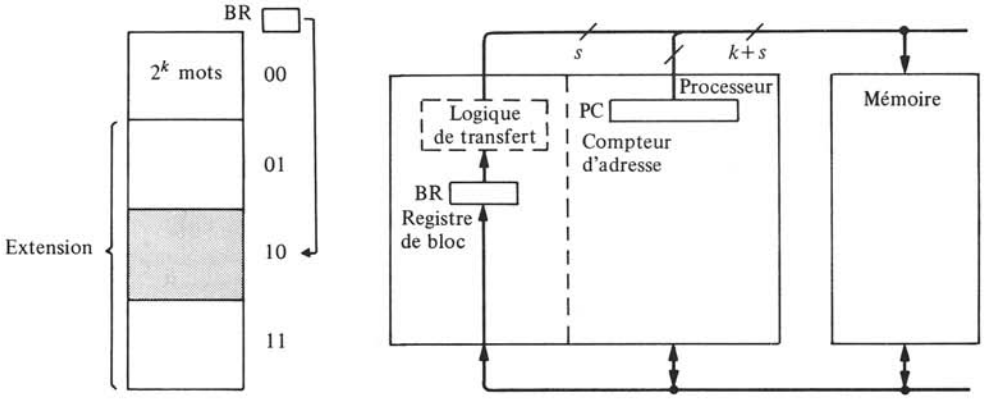


Fig. 3.46

Pour pouvoir sauter à une adresse quelconque dans un bloc quelconque, une logique supplémentaire est nécessaire pour retarder le transfert du contenu du registre de bloc sur le chemin de données jusqu'à ce que le compteur ordinal ait lui aussi changé son contenu.

La paire d'instructions

```
LOAD    BR, #NEXTBLOC        (3.39)
JUMP    ADRNEXTBLOC
```

doit alors toujours être exécutée pour changer le bloc.

La logique de transfert peut être différente si les blocs supplémentaires ne sont jamais occupés par des programmes, mais sont réservés pour des tables de nombres ou des chaînes de caractères adressées de façon indirecte uniquement. Le bloc dont l'adresse a été préparée dans un registre de bloc BR est transféré sur le bus seulement lorsqu'une instruction de transfert indexée est exécutée.

Un cas simple limitant la dimension mémoire pour une même dimension de registre utilise le poids fort de la ligne d'adresse sortant du processeur pour commuter entre l'adressage normal (avec seulement la moitié des positions) et l'adressage de bloc sélectionné par le registre de bloc (fig. 3.47).

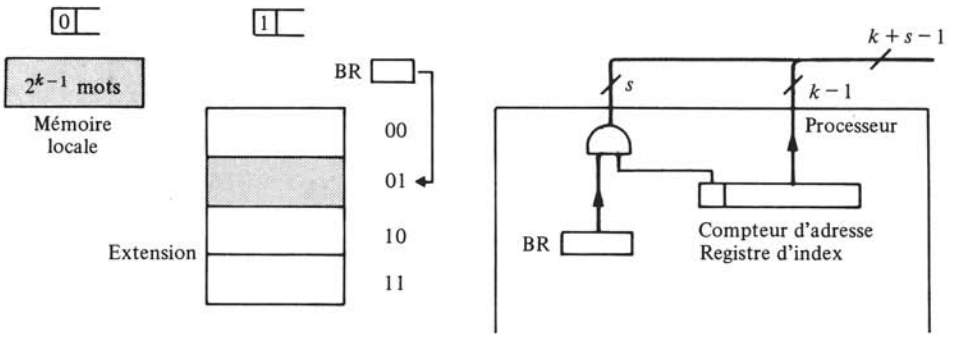


Fig. 3.47

3.8.3 Extension par registre de base

Une technique plus souple que le découpage en blocs avec tous les problèmes associés à la communication entre blocs s'inspire de l'adressage relatif vu au paragraphe 3.5.7. Le contenu du compteur ordinal est considéré comme une valeur relative par rapport à un *registre de base*, qui contient une *adresse de base*. Souvent le registre de base a la même longueur que le compteur ordinal, mais est décalé du nombre de bits correspondant à l'extension. Le registre de base permet de déplacer le programme de l'utilisateur dans l'extension mémoire, de façon quasi continue, d'où le nom de *mémoire continue* donné parfois par opposition à la mémoire organisée en blocs (fig. 3.48). Les processeurs de la famille Intel IAPX-86 utilisent abondamment ce type d'adressage.

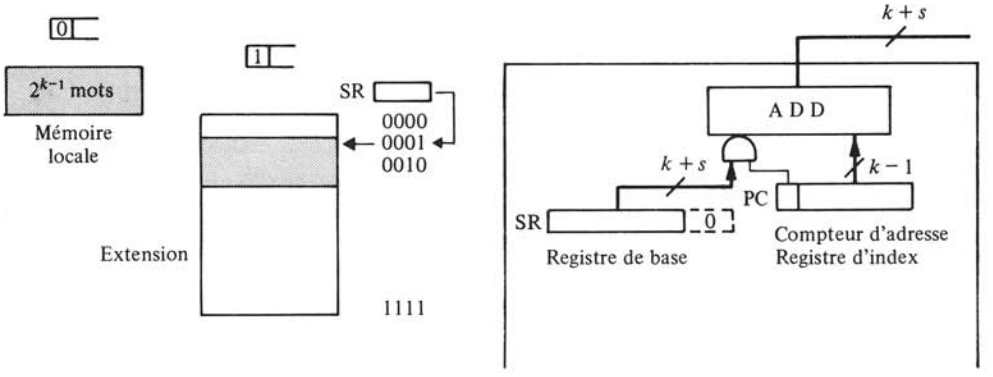


Fig. 3.48

Remarquons en particulier que le registre de base permet à l'utilisateur d'écrire toutes ses parties de programme à partir de l'adresse zéro et de les juxtaposer dans la mémoire sans changer les adresses. Le *relogement* (*relocation*), ou correction des adresses, est fait automatiquement par le registre de base. La présence de celui-ci facilite donc la programmation et justifie sa présence dans plusieurs ordinateurs, même lorsque l'extension mémoire n'est pas nécessaire.

3.8.4 Adresses logiques et physiques

Les architectures des précédents paragraphes montrent qu'à une même adresse dans le programme peuvent correspondre une ou plusieurs adresses différentes dans la mémoire.

L'adresse définie par le programme est appelée *logique* et l'adresse effective en mémoire est une adresse *physique*.

Dans les paragraphes précédents, la correspondance entre adresse logique et physique est définie par un "OU" logique ou par une addition binaire. Dans les systèmes multiutilisateurs, elle est souvent définie en plus par une *table de correspondance* (*map*) gérée par des instructions et un programme de supervision inaccessible aux utilisateurs. La table de correspondance permet d'établir des frontières infranchissables entre les tranches d'adresses physiques attribuées aux différents utilisateurs, garantissant ainsi la protection de chaque programme.

La figure 3.49 résume le principe de base généralement utilisé: à chaque groupe d'adresses logiques mises à disposition de l'utilisateur correspond un bloc d'adresses physiques, sans que l'attribution ne soit nécessairement complète.

Dans le cas de la décomposition de la mémoire en pages de longueur fixe, on parle souvent de mémoire *paginée* (*paged memory*).

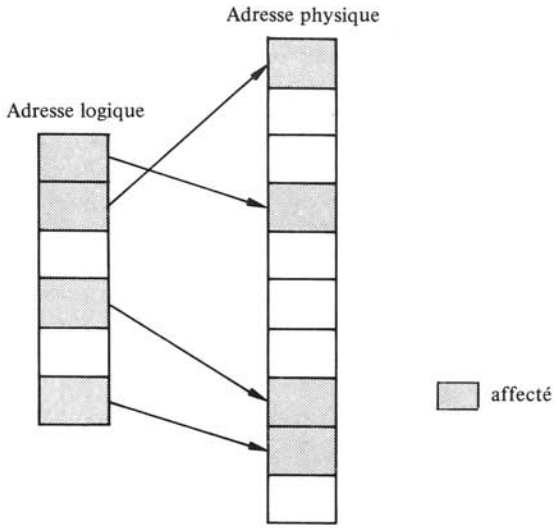


Fig. 3.49

Si le programme utilisateur se réfère à une adresse logique qui n'a pas de correspondant, le bloc adéquat est transféré de la mémoire de masse (disque magnétique) vers un bloc mémoire libre (ou préalablement libéré par un transfert de son contenu sur disque).

Le schéma bloc de la figure 3.50 évoque l'architecture générale permettant ce comportement. Si $s > r$, l'utilisateur dispose d'une mémoire physique plus grande que la mémoire logique, on travaille en *commutation de banque* (*bankswitching*).

Si $s < r$, c'est la mémoire logique qui est la plus grande, mais grâce au transfert immédiat de l'information demandée par une adresse logique, les limitations de la mémoire physique n'apparaissent pas à l'utilisateur; on dit que la mémoire est *virtuelle* (*virtual memory*).

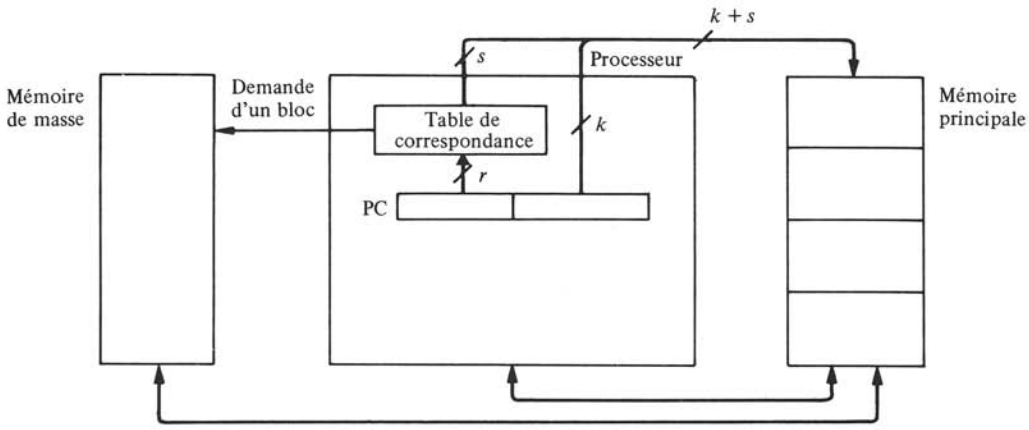


Fig. 3.50

3.8.5 Structure des tâches en mémoire

Une tâche ou processus consiste de façon typique en un programme principal et un ensemble de routines agissant sur une structure de données. Ces modules doivent être linéarisés en mémoire (fig. 3.51).

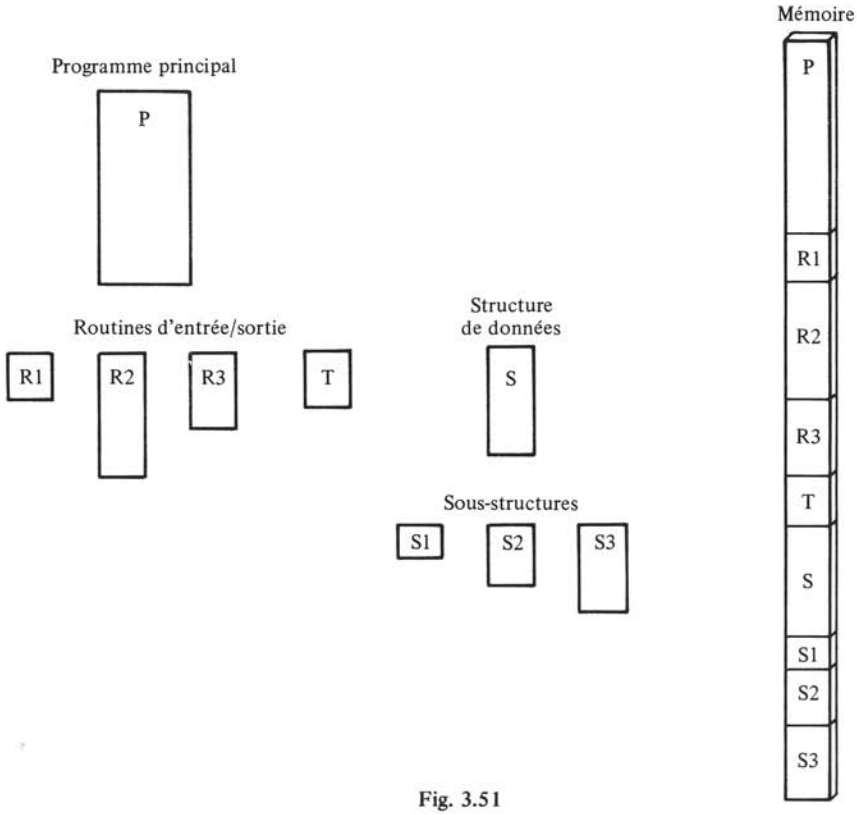


Fig. 3.51

La linéarisation peut être faite par le programmeur au moment de l'édition de son programme, ce qui se fait de façon courante lorsqu'on programme en assembleur. La linéarisation peut être faite au chargement par un programme dit éditeur de lien (§ 4.3.8) qui translate des modules préalablement assemblés ou compilés comme s'ils devaient résider seuls à partir de l'adresse 0, à une adresse physique différente.

La linéarisation peut enfin être effectuée au moment de l'exécution, afin de permettre par exemple l'agrandissement d'une table ou le déplacement d'une partie du programme pour permettre la coexistence avec un autre programme. L'utilisation d'une logique spéciale dans le processeur ou entre le processeur et la mémoire est alors nécessaire.

3.8.6 Segmentation

La *segmentation* de la mémoire logique selon des blocs de longueur prédéfinie, associables en segments de différentes longueurs, permet non seulement une correspondance quelconque avec des adresses physiques, mais encore une protection face aux erreurs de programmation. A chaque bloc logique ou page se trouve associé un attribut (formé par un mot binaire de quelques bits) définissant si le bloc physique peut être lu ou écrit. L'adresse logique peut contenir l'adresse d'un processus, ce qui permet une protection spécifique à chaque processus. Le programme superviseur est le seul qui peut changer les protections et donner le contrôle au processus suivant.

Dans l'exemple de la figure 3.52, trois processus de 4 blocs logiques au plus sont cartographiés dans une mémoire physique de 7 blocs. La table de correspondance (map) est

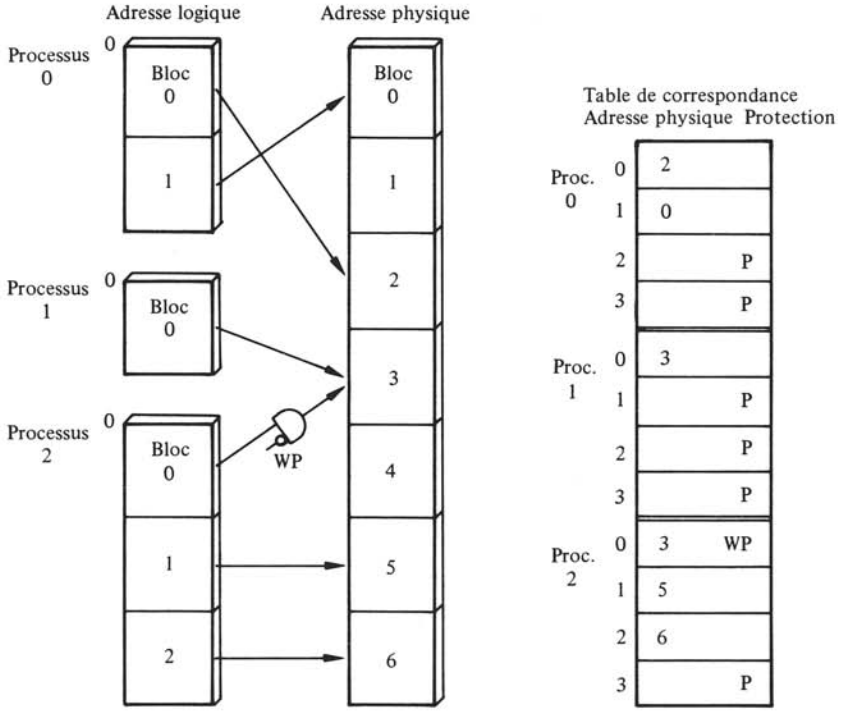


Fig. 3.52

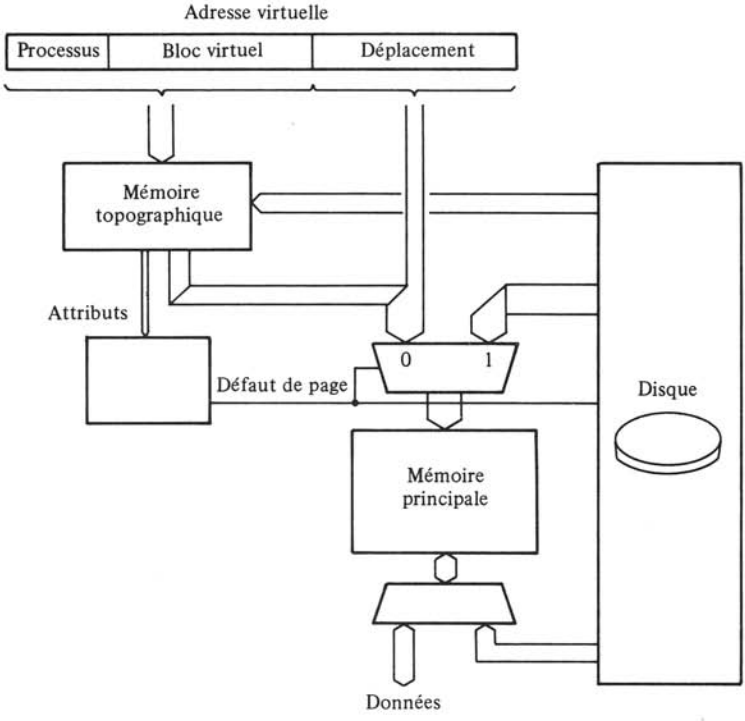


Fig. 3.53

en général une mémoire rapide dans l'interface mémoire, comme évoqué dans la figure 3.53 mais la table peut aussi se trouver dans une zone de la mémoire principale, l'adresse physique étant chargée indirectement.

La segmentation permet de disposer de plusieurs espaces logiques indépendants, dont la longueur peut être différente, et modifiable pendant l'exécution. Tout se passe comme si la mémoire avait deux dimensions, la dimension nouvelle permettant de séparer les types d'information et de processus traités par le programme. Ceci simplifie la manipulation des structures du programme, par le compilateur et par le chargeur. Chaque segment ne contenant qu'un type d'information, le niveau de protection est facile à définir.

3.8.7 Mémoire virtuelle

La quantité de mémoire logique demandée par un utilisateur peut être grande. Pour éviter de forcer le programmeur à couper son programme en tranches compatibles avec des nombres de blocs optimisés pour les travaux courants, on peut concevoir un système avec un espace logique très grand. Seule une partie de cet espace est cartographiée sur la mémoire physique, forcément limitée. Si l'utilisateur accède à un bloc pour lequel il n'y a pas de correspondant physique, un bloc est tout d'abord attribué par le système, et préparé à partir de l'information stockée sur disque (la taille de l'espace disque est supposée supérieure à celle de l'espace virtuel) avant que l'accès ne puisse se terminer.

La figure 3.53 donne un schéma-bloc d'une *mémoire virtuelle* simple, qui peut compléter la mémoire segmentée du paragraphe précédent.

La table de correspondance contient en plus des attributs de protection une indication de la fréquence des accès à chaque bloc, de façon à réaffecter d'abord des blocs ayant la plus faible probabilité d'être accédés à nouveau dans l'immédiat.

Si lors d'un accès la page correspondante n'est pas présente, le système doit prendre le contrôle et

- identifier la page virtuelle en faute
- sélectionner une page physique propre à recevoir la page virtuelle demandée
- sauver éventuellement cette page sur disque
- charger la page demandée.

Pendant le temps des transferts disque, on cherchera naturellement à effectuer une autre tâche.

On peut remarquer que si l'espace virtuel est grand et les blocs petits, la mémoire topographique est très grande. On peut alors la remplacer par une mémoire associative, dont la taille dépend du nombre de blocs dans la mémoire physique. L'accès à cette mémoire associative permet de savoir si le bloc virtuel existe en mémoire, et quel est son correspondant physique.

Pour réduire la taille de la mémoire topographique, il est également possible de décomposer l'adresse virtuelle en 3 champs d'adresse de segments (de 64K chacun par exemple), d'adresse de pages dans un segment (de 1K chacun par exemple) et d'adresse dans la page. La table de segments est alors en mémoire topographique rapide, et la table de segments en mémoire principale. Une très grande flexibilité est atteinte par ce moyen, permettant la coexistence d'un grand nombre de processus de taille très variable en mémoire, au prix d'un temps d'accès plus élevé et d'une gestion adéquate de la table de segments et des tables de pages [24, 62].

3.9 ARCHITECTURES SPÉCIALES

3.9.1 Augmentations des performances

Des architectures particulières permettent l'amélioration des performances des processeurs. Les unités de mesure utilisées pour évaluer ces performances sont le MIPS (Million Instruction Per Second) et le Mégaflop/sec (Million Floating point Operation Per second). La durée des différentes instructions et opérations dans un même processeur étant variable, ces unités sont définies selon une valeur moyenne, ou dans le cas le plus favorable.

On a d'une part un flot d'instructions correspondant au programme et d'autre part un flot de données à modifier ou créer. Dans le cas simple vu au début de ce chapitre ou ces deux flots sont uniques (*SISD: Single Instructions Single Data*), ils passent par un bus unique en suivant l'alternance recherche/exécution.

Pour augmenter les performances, on peut augmenter la vitesse des transferts, faire en sorte que les flots d'instructions et de données se déroulent en parallèle, et soient eux-mêmes décomposés en flots parallèles. On a alors des machines dites *SIMD* (*Single Instruction Multiple Data*) ou *MIMD* (*Multiple Instruction Multiple Data*) [57].

3.9.2 Techniques courantes

De façon plus précise, on rencontre les techniques suivantes pour augmenter les performances:

- Les instructions sont cherchées en mémoire à l'avance et stockées temporairement dans une *queue de préchargement* (*prefetch queue*), pour être immédiatement disponibles lorsque l'instruction précédente se termine. Cette technique est déjà abondamment utilisée dans les processeurs 16 et 32 bits [62, 63].
- Les transferts mémoire sont accélérés par l'utilisation d'une *mémoire cache* servant de tampon entre la mémoire principale et l'unité arithmétique [57]. Cette mémoire cache plus petite est beaucoup plus rapide, et contient dans 60 à 90% des cas l'information demandée.
- L'unité arithmétique est démultipliée, avec des unités spécialisées très rapides par types d'opérations ou types d'opérandes. Si une unité est spécialisée pour le traitement des vecteurs ou de nombres flottants, on parle de *processeur vectoriel* (*array processor*) [58, 59].
- Des processeurs satellites peuvent s'occuper de toute la gestion détaillée des périphériques [57, 59].
- Plusieurs décodeurs d'instructions indépendants peuvent exécuter un seul flot d'instructions selon une technique *à la chaîne* (*pileline*) [57] ou plusieurs flots d'instructions en parallèle [58]. Dans ce cas, on a un système multiprocesseur qui peut prendre de nombreuses formes selon la grille des interconnexions entre les décodeurs d'instructions, unités arithmétiques, mémoires et périphériques [56, 58].

Pour obtenir des performances élevées, les ordinateurs doivent se miniaturiser afin de réduire les temps de propagation de l'information sur les fils de liaison; l'évolution des performances dépend plus de la solution des problèmes de connectique et de dissipation que de la possibilité de multiplier les mémoire et unités arithmétiques.

3.9.3 Autres solutions

Des solutions basées sur des principes différents de l'architecture de von Neumann sont activement recherchées depuis de nombreuses années. En particulier, les *processeurs associatifs* ont des architectures très différentes utilisant des mémoires associatives [57, 60]. Le coût de ces mémoires a pour l'instant limité leur emploi à certaines parties de l'unité de commande, lorsque l'augmentation de performances est significative et la dimension de la mémoire associative raisonnable [58].

3.9.4 Conclusion

L'architecture des processeurs ne se définit jamais d'après des considérations de performances d'instructions isolées. La programmation efficace complexe et la compilation rapide de langages évolués motive toutes les améliorations architecturales, qui sont multiples et variées.

L'investissement dans les langages et programmes d'application actuels ralentit l'évolution des superordinateurs introduisant de nouveaux concepts, encourage le développement des microprocesseurs dont le niveau de performance est équivalent aux gros ordinateurs d'i y a dix ans et permet de multiplier les applications des architectures et logiciels éprouvés.

CHAPITRE 4

PROGRAMMATION EN ASSEMBLEUR

4.1 INTRODUCTION

4.1.1 Langage machine

Programmer un ordinateur, c'est effectuer un ensemble d'opérations permettant d'initialiser sa mémoire et d'obtenir le comportement voulu. La suite des mots binaires chargés en mémoire est appelée *programme en langage machine*. Ce programme est généralement représenté en hexadécimal pour faciliter son écriture et son introduction par un clavier. Inversement, le contenu de la mémoire peut être copié sur un support magnétique ou imprimé. On parle de *listage binaire* (*binary listing, memory dump, core image*). La figure 4.1 donne l'exemple d'un listage binaire (ici en hexadécimal) d'un petit programme pour microprocesseur, dans le format utilisé par Intel.

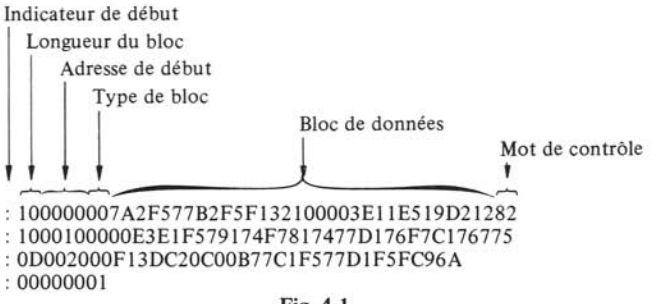


Fig. 4.1

Les listages binaires sont fort peu lisibles, même pour une personne connaissant bien la machine. La moindre erreur de transcription entraîne une erreur dans l'exécution Pour cette raison, le listage de la figure 4.1 est formatté en blocs de 16. mots mémoire au plus (2 caractères chacun), précédés et suivis d'indications complémentaires telles que l'adresse du premier mot, la longueur du bloc, et une indication de contrôle permettant de vérifier la cohérence du message.

4.1.2 Langage d'assemblage

L'utilisation de codes et symboles mnémoniques au lieu des nombres représentant les instructions, adresses et paramètres facilite grandement la tâche du programmeur. Une notation condensée aussi explicite que possible permet de décrire l'effet de chaque instruction. Par exemple l'expression ADD A,B peut représenter l'instruction d'addition de deux registres. Les éléments de cette expression correspondent à un code de commande de l'unité arithmétique et à des adresses d'opérandes; ces éléments doivent être assemblés

pour former une instruction binaire, d'où le terme de *langage d'assemblage* donné aux programmes écrits avec des symboles représentant les codes et les opérandes.

En d'autres termes, le langage d'assemblage est une variante alphanumérique du langage machine. Généralement, à chaque instruction écrite en langage d'assemblage (on dit souvent écrite en assembleur) correspond une instruction machine et réciproquement.

La figure 4.2 donne un exemple de programme en assembleur CALM. Ce programme cherche l'octet le plus petit (nombres logiques) dans une zone mémoire et a été écrit et traduit pour deux processeurs différents.

```
82/07/29 11:00:32  GETMIN.SR                                        01-01

1                              .TITLE   GETMIN.SR   ; Exemple de programme en assembleur

2                              ; Ce programme cherche l'élément le plus petit d'une zone mémoire
3                              ; Structure du programme:
4                              ;  Initialiser les pointeurs et la valeur d'entrée du paramètre che
5                              ;   Répéter jusqu'à la fin de la table
6                              ;     Lire la valeur dans la table
7                              ;     Si cette valeur est inférieure au paramètre courant
8                              ;       Prendre pour paramètre la valeur lue
9                              ;   Fin de la partie répétée
10                             ;   Rendre le contrôle au moniteur de mise au pointx

11                  2300       ADZONE   = H'2300    ; Adresse de la zone mémoire
12                  00A0       LZONE    = 160.              ; Longueur de la zone

13                             ; Variante Z80, limitée à une zone de 256. positions au maximum

14                             .PROC    Z80
15   8000                      .LOC     H'8000      ; Adresse pour le programme

16                             ZGETMIN:
17   8000 210023                       LOAD        HL,#ADZONE
18   8003 06A0                         LOAD        B,#LZONE
19   8005 3EFF                         LOAD        A,#H'FF    ; Valeur initiale égale au maximum
20
21   8007 BE                   2$:     COMP        A,(HL)
22   8008 D20C80                       JUMP,HS     4$
23   800B 7E                           LOAD        A,(HL)
24   800C 23                   4$:     INC         HL
25   800D 05                           DEC         B
26   800E 20F7                         JUMP,NE     2$

27   8010 76                           TRAP                   ; Le plus petit nombre est dans A

28                             ; Variante 68000, permettant une zone de 64k

29                             .PROC    M68000
30   8200                      .LOC     H'8200

31                             MGETMIN:
32   8200 41F82300                     LOAD.32     A0,#ADZONE
33   8204 303C00A0                     LOAD.16     D0,#LZONE
34   8208 123C00FF                     LOAD.8      D1,#H'FF

35   820C B218                 2$:     COMP.8      D1,(A0+)
36   820E 64000006                     JUMP,HS     4$
37   8212 1228FFFF                     LOAD.8      D1,(A0)-1
38   8216 5340                 4$:     DEC.16      D0
39   8218 66FC                         JUMP,NE     2$

40   821A 4E40                         TRAP        #0         ; Le plus petit nombre est dans D1

41                  0001       .END

Pass 2 complete          7 seconds
18 references
0/0/0 if/else/endif
0 inserted files
Source file 30 usefull lines long
Binary file 021C bytes long
Assembly time: 11 seconds  163 lines/min
```

Fig. 4.2

La partie gauche représente les adresses et contenus des positions mémoire, tels qu'ils sont calculés par le programme assembleur (sect. 4.3).

La partie droite a été préparée avec un éditeur de textes (§ 6.3.6) et commence par une indication de titre et des commentaires (ligne 1). La zone mémoire à explorer est ensuite caractérisée en utilisant l'hexadécimal pour son adresse et le décimal pour sa longueur (lignes 11 et 12), selon ce qui est le plus naturel au programmeur. Le programme Z80 commence par des directives définissant le processeur et l'emplacement en mémoire du code traduit (lignes 14 et 15).

Le programme proprement dit utilise les opérateurs et modes d'adressage vus précédemment. Les étiquettes GETMIN:, 2$:, 4$: caractérisent des positions mémoire contenant le premier octet de l'instruction suivant l'étiquette.

Les registres du Z80 ont des noms réservés à une ou deux lettres selon qu'ils ont une longueur de 8 ou 16 bits.

Les registres du M68000 (§ 3.6.7) ont une longueur utilisée de 8, 16 ou 32 bits, précisée en postfixe du mnémonique de l'instruction.

La suite de caractères formant un programme en langage d'assemblage est appelée *programme source* (ou fichier, listage source). L'utilisation d'une syntaxe permet une traduction automatique par un programme appelé *assembleur*. L'assembleur génère un programme binaire exécutable ou *programme objet* qui prend, si on l'imprime, l'allure de la figure 4.1, ainsi qu'un *listage complet* ou *fichier listing* ayant l'allure de la figure 4.2.

4.2 SYNTAXE D'UN LANGAGE D'ASSEMBLAGE

4.2.1 Introduction

Le but de cette section est de montrer les caractéristiques et possibilités des langages d'assemblage. Chaque ordinateur a son langage d'assemblage propre, dépendant de son architecture et des habitudes du fabricant. De très grandes similitudes existent toutefois entre les langages d'assemblage des mini et microordinateurs actuels; un langage unique avec des variations aux niveaux des codes et des opérandes, peut donc être utilisé. La syntaxe du langage CALM (Common Assembly Language for Microprocessors) sera présentée dans cette section et utilisée dans les exemples et exercices. Une présentation relativement rigoureuse familiarisera le lecteur avec les notations permettant de définir précisément la syntaxe d'un langage de programmation quelconque et d'aborder l'analyse d'un problème complexe.

4.2.2 Jeu de caractères

L'écriture d'un programme, quel que soit le langage, utilise les caractères courants disponibles sur les machines à écrire et terminaux d'ordinateur. Un seul langage, l'APL, nécessite des caractères spéciaux [64].

Un jeu de 128 caractères, codés par des mots de 7 bits, a été normalisé avec un nombre relativement restreint de différences d'un fabricant à l'autre. Ce code, appelé *code ASCII* (*American Standard Code for Information Interchange*) ou *code ISO* (*International Standard Organisation*) est bien connu (§ 7.2.1). IBM est le seul fabricant à utiliser un code très différent, le *code EBCDIC*, avec un jeu de caractères très semblable codé sur 8 bits (§ 7.2.2).

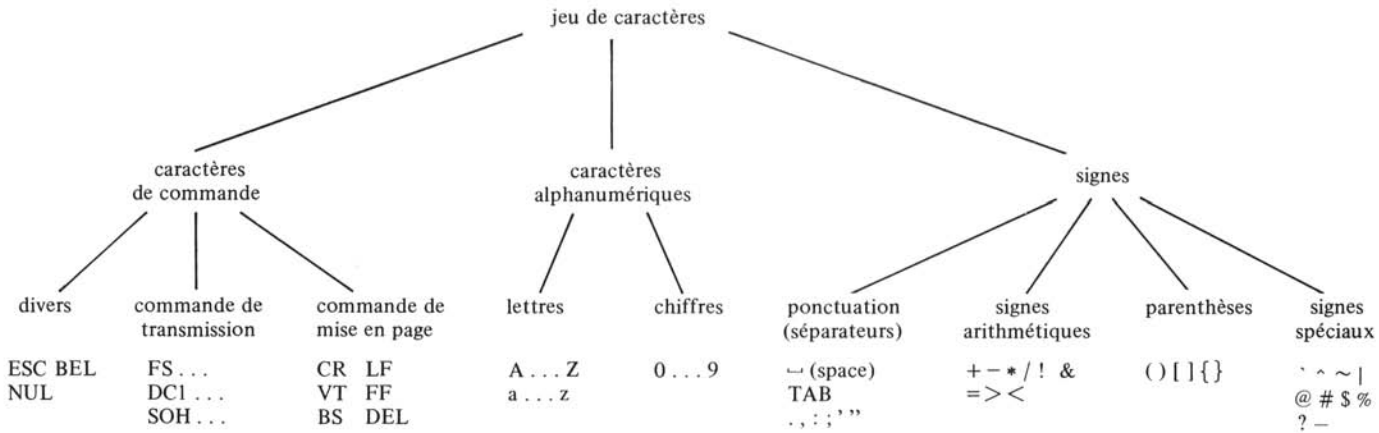

jeu de caractères
caractères de commande
caractères alphanumériques
signes
divers
commande de transmission
commande de mise en page
lettres
chiffres
ponctuation (séparateurs)
signes arithmétiques
parenthèses
signes spéciaux
ESC BEL NUL
FS . . . DC1 . . . SOH . . .
CR LF VT FF BS DEL
A . . . Z a . . . z
0 . . . 9
␣ (space) TAB . , : ; ' "
+ − * / ! & = > <
( ) [ ] { }
` ^ ~ | @ # $ % ? _

Fig. 4.3

Chaque langage fait un usage différent des caractères de ponctuation et des caractères spéciaux. Les catégories que l'on est usuellement amené à distinguer dans un jeu de caractères sont représentées par l'arbre de la figure 4.3. Les lettres accentuées ne sont pas utilisées en informatique, mais doivent naturellement apparaître dans les commentaires. Leur codage dans un jeu de 128 caractères nécessite des séquences spéciales, ou le sacrifice de certains caractères de commande.

On distingue trois grandes catégories: les caractères de *commande*, les caractères *alphanumériques* (lettres et chiffres) et les *signes*. Les caractères de commande sont désignés par des groupes de lettres mnémotechniques; par exemple, la touche "retour de chariot" génère un code désigné par le symbole CR (carriage return) parfois représenté ici par le signe ↙.

Avec le code ASCII, sur les 128 codes, 95 ont un signe graphique caractéristique (lettres, chiffres, signes) et sont appelés *signes graphiques*. Le tabulateur (TAB) est considéré comme un signe graphique, occupant la place de 1 à 8 caractères, dans le cas fréquent où les tabulateurs sont posés de 8 en 8.

Une liste exhaustive du jeu de caractères usuel est donnée en annexe. Les codes EBCDIC (§ 7.2.2) et Telex (§ 7.2.3) sont nettement moins utilisés que le code ASCII (§ 7.2.1).

Ce jeu de caractères peut difficilement être étendu par adjonction de nouveaux codes, car la plupart des équipements sont prévus pour travailler avec 7 bits, voire seulement 6 bits. L'utilisation de deux caractères ou plus permet d'étendre les possibilités de codage. La notation:= remplace la flèche renversée ← comme caractère d'assignation dans la plupart des langages évolués d'ordinateurs. Pour exprimer toutes les opérations arithmétiques, on rencontre des groupements de signes comme par exemple ** pour l'exponentiation, SIN, ABS etc. pour les nombreuses fonctions spéciales.

4.2.3 Chiffres

La notion de chiffre dans un langage de programmation nécessite une définition précise. Dans le langage des ensembles, on dira que x est un chiffre si

$$x \in \{0, 1, ..., 9\} \qquad (4.1)$$

(x appartient à l'ensemble des nombres 0, 1, ... 9)

Dans la notation de Backus-Naur-Wirth [73], fréquemment utilisée pour décrire la syntaxe d'un langage de programmation, on écrit

chiffre = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9" (4.2)

Le signe = caractérise une définition, des guillemets encadrent les caractères primitifs du jeu de caractères, et les barres correspondent au OU logique. La formule (4.2) signifie donc simplement qu'un chiffre est l'un des symboles 0 ou 1 ou 2 ...; les pointillés ne sont pas utilisés dans la formule (4.2), si l'on veut faciliter la lecture et l'interprétation par un ordinateur, et éviter des notions implicites d'ordre et de codes consécutifs.

Une autre représentation fréquemment utilisée est celle des *diagrammes de syntaxe*. La figure 4.4 donne le diagramme définissant un chiffre. Le diagramme doit être parcouru de gauche à droite en suivant l'un quelconque des trajets possibles.

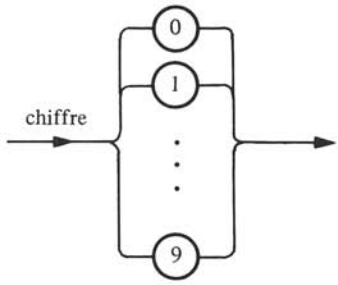


Fig. 4.4

4.2.4 Nombre décimal entier positif

La convention faite au chapitre 2 (§ 2.1.5) a été de faire suivre les nombres décimaux par un point.
Il en résulte la forme de Backus-Naur-Wirth suivante:

nombre décimal = chiffre {chiffre} "." (4.3)

L'accolade indique une possibilité d'omission ou d'itération. La juxtaposition sans signe spécial indique la concaténation, c'est-à-dire la juxtaposation des termes connus. Cette définition permet de construire 3.,000620., mais pas 26 ou 32.4.

Le diagramme de syntaxe définissant un nombre décimal est donné dans la figure 4.5. Un rectangle est utilisé pour un symbole déjà défini. Le parcours du diagramme montre très explicitement qu'un nombre décimal est défini comme au moins un chiffre suivi d'un point.

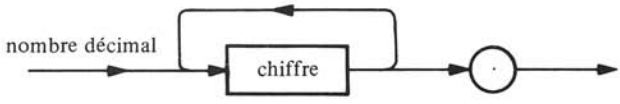


Fig. 4.5

Pour des raisons historiques et pratiques, deux autres possibilités d'écriture existent pour les nombres décimaux dans les assembleurs CALM3:

- la notation sans point final (si la base par défaut est initialisée comme étant la base 10);
- la notation préfixe par D' (§ 4.2.8).

4.2.5 Exercice

Définir complètement un nombre octal entier signé (positif ou négatif) selon Backus-Naur-Wirth et selon les diagrammes de syntaxe.

4.2.6 *p*-chiffre

Un chiffre en base p a été défini comme un nombre entier positif inférieur à p (§ 2.1.2). La représentation d'un chiffre dans un langage ne doit pas être ambiguë. Pour

éviter l'utilisation de parenthèses, un signe unique doit correspondre à chaque chiffre. Si la base p est supérieure à 10., la liste des chiffres usuels est complétée par les lettres de l'alphabet dans leur ordre usuel. Ceci résout le problème jusqu'à la base 36.. La base 40. étant parfois utilisée (§ 2.2.6), on peut compléter l'alphabet avec des signes [\ ] ^ − ‘ qui suivent immédiatement la lettre Z dans le code ASCII, puis utiliser les minuscules. Ceci facilite la conversion, puisque les codes se suivent continûment, sauf entre 9 et A.

4.2.7 Exercice

L'exercice 2.1.19 a défini un système de numération de base 3. Donner le diagramme de syntaxe et la représentation de Backus-Naur-Wirth correspondant aux nombres positifs entiers écrits dans cette base.

4.2.8 *p*-nombre

La définition d'un nombre en base $p > 10.$ ne peut pas être la simple concaténation de ses chiffres. Il faut prévoir un moyen de distinguer le nombre BAD en base 16. du nom BAD utilisable pour une étiquette ou une variable.

L'utilisation d'un préfixe spécial évite toute confusion; en hexadécimal on écrit H'BAD. Dans une base par défaut quelconque supérieure à 10., un zéro non significatif doit être placé devant le nombre: on écrit 0BAD, le zéro étant traditionnellement biffé en mini et microinformatique (en FORTRAN, le O est biffé: FØRTRAN, une recommandation internationale est de biffer partiellement le O: FȮRTRAN).

Avec ces conventions, la définition complète des nombres en CALM3 est

```
nombre décimal          : chiffre {chiffre }"." | "D"'"'"chiffre {chiffre }
nombre binaire          : "B"'"'" chiffre binaire {chiffre binaire }
nombre octal            : "O"'"'" chiffre octal {chiffre octal }
nombre décimal          : "H"'"'" chiffre hexa {chiffre hexa }
nombre dans une base par défaut supérieure à 9                    (4.4)
                        : chiffre {chiffre dans la base par défaut }
nombre dans une base par défaut inférieure à 10.
                        : chiffre dans la base par défaut {chiffre dans la base par
                          défaut }
```

Le diagramme de syntaxe correspondant est donné dans la figure 4.6. On remarque qu'un nombre décimal peut avoir jusqu'à trois représentations équivalentes si la base par défaut est 10 (§ 4.2.5).

4.2.9 Remarque

Les assembleurs des différents fabricants ont des notations différentes pour les nombre dans les bases courantes.

Par exemple, les nombres hexadécimaux sont terminés par un H: 0BADH, les nombres octaux par un Q: 7324Q, les nombres binaires par un B: 101101B et les nombres décimaux par un T: 999T.

Une autre convention emploie des signes ASCII peu utilisés et note % 101101 (binaire), $ BAD (hexadécimal) et @ 3742 (octal).

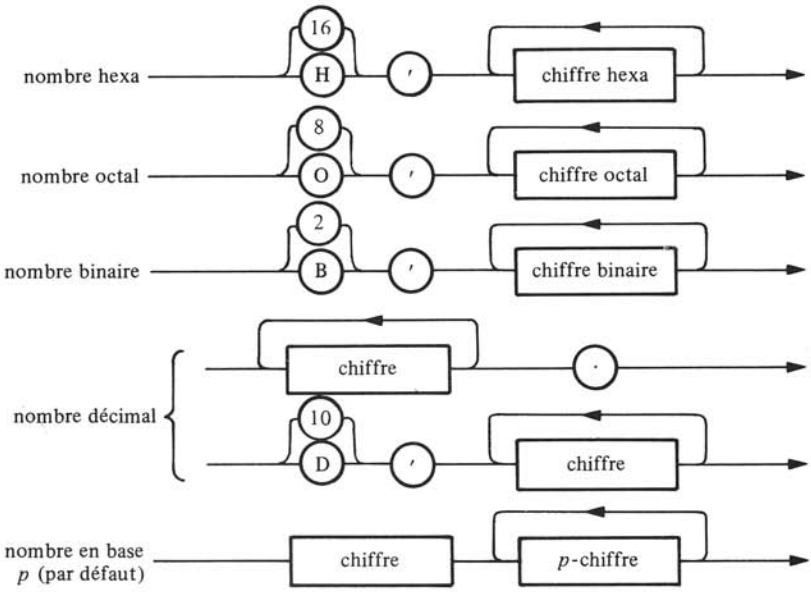


Fig. 4.6

4.2.10 Méthologie

Dans les paragraphes qui précèdent, l'approche qui a été faite procède du simple au compliqué, des concepts de base vers des concepts plus élaborés. Cette approche est dite *montante* (*bottom-up*).

Une approche *descendante* (*top-down*) est généralement recommandée. Elle consiste à partir du concept le plus général, l'objectif même de la tâche à accomplir, et de décomposer ce concept en concepts de plus en plus simples jusqu'à ce que les éléments de base (axiomes, jeu de caractères, etc.) soient atteints.

Ceci s'applique en particulier à la définition d'un langage. La compréhension de l'approche descendante suppose une certaine familiarisation avec le langage, pour justifier la nécessité et la suffisance des décompositions successives. Si le lecteur n'a pas encore de connaissance de langage d'assemblage, l'exemple de la figure 4.2 peut être relu pour prendre conscience des types d'informations que l'on doit pouvoir exprimer dans le fichier source d'un programme à assembler.

4.2.11 Fichier source

Un *fichier à assembler* est une suite de caractères comportant toute l'information permettant à l'assembleur de créer automatiquement un fichier binaire exécutable.

Le fichier est formé de lignes terminées par un caractère de retour de chariot CR (fig. 4.7).

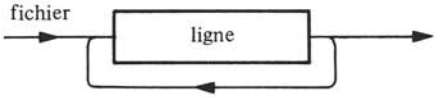


Fig. 4.7

Une première décomposition montre que certaines lignes sont des *directives* servant de guide pour le programmeur et définissant des symboles et positions mémoire. D'autres lignes contiennent les instructions proprement dites et sont dites *lignes d'instruction*. Une *ligne de fin* termine le fichier. Elle peut éventuellement être suivie d'autres lignes, mais celles-ci seront ignorées par le programme de traduction.

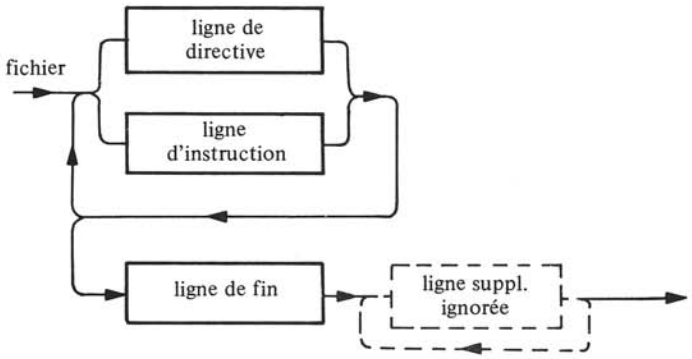


Fig. 4.8

Le diagramme de syntaxe de cette première décomposition est donné dans la figure 4.8 et montre que l'alternance des lignes de directive et des lignes d'instruction est quelconque dans le programme.

Du point de vue syntaxique, il est préférable de poursuivre la décomposition à partir de la notion de ligne.

4.2.12 Notion de ligne

Une ligne d'assembleur est un ensemble de caractères de longueur limitée (132 ou 256 caractères), composée d'une première partie significative (éventuellement vide) appelée *étiquette*, d'une seconde partie (éventuellement vide) qui est une *instruction* ou *pseudo-instruction* et d'une troisième partie (éventuellement vide) appelée *commentaire*. L'instruction ne contient jamais le caractère " ; " qui est réservé comme en-tête de la partie commentaire. Un terminateur de ligne (caractère CR) existe dans tous les cas (fig. 4.9).

L'étiquette est optionnelle et prend la valeur de la position mémoire courante. Le diagramme de syntaxe de la figure 4.9 montre que, formellement, une instruction peut avoir plusieurs étiquettes, sur la même ligne ou non.

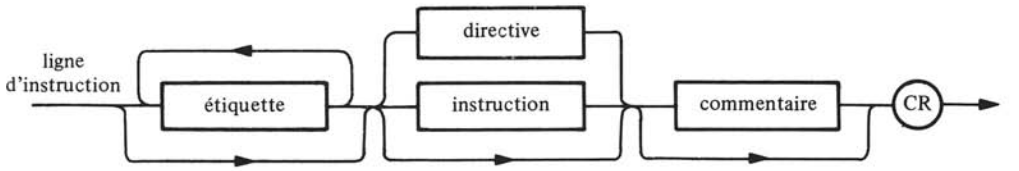


Fig. 4.9

4.2.13 Etiquette

Descendons encore d'un niveau dans la décomposition en définissant les termes utilisés au niveau précédent. Le diagramme de syntaxe d'une étiquette est donné dans la figure 4.10 et montre qu'une étiquette est constituée d'un *symbole*, c'est-à-dire d'un mot représentant un nombre (en l'occurence une adresse) suivi d'un signe "deux points". Des séparateurs peuvent être ajoutés de part et d'autre du symbole, mais ne sont pas nécessaires.

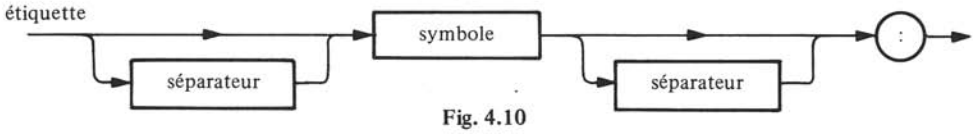


Fig. 4.10

Les diagrammes de syntaxe ont été dessinés dans ce livre de façon à mettre en évidence certaines règles usuelles d'écriture. Pour des raisons de lisibilité, une étiquette n'est généralement pas précédée d'un séparateur et le deux points n'est pas séparé du symbole par un espace ou une virgule.

4.2.14 Commentaire

Un commentaire est caractérisé par le signe "point-virgule" ou le signe "boa (barre oblique arrière)". Un texte quelconque peut être écrit jusqu'à la fin de la ligne (fig. 4.11). Un commentaire trop long peut suivant les cas déborder l'écran ou l'imprimante et n'être partiellement plus visible. Il peut aussi déborder la place en mémoire pour une ligne, et être partiellement perdu.

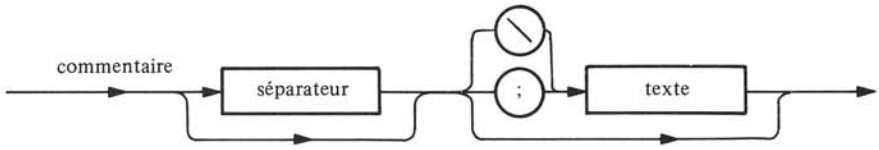


Fig. 4.11

4.2.15 Instruction

Le code d'une instruction comporte un *code mnémotechnique* ou *code mnémonique* caractérisant l'opération effectuée par le processeur. Un séparateur précède généralement ce code et doit le suivre si une condition de test (dans un saut conditionnel) et/ou des opérandes (sauts et opérations arithmétiques) précisent l'opération (fig. 4.12)

Les symboles réservés utilisés comme mnémoniques, conditions de test et opérandes sont définis par la carte de référence au processeur considéré. La représentation des modes d'adressage est définie par une syntaxe précise, mais en général tous les modes d'adressage ne sont pas disponibles pour une instruction donnée, étant donné le manque d'orthogonalité des processeurs.

La syntaxe CALM laisse une très grande liberté de mise en page; le format est dit *libre* par opposition au format *fixe* utilisé par d'autres assembleurs dans lesquels le code mnémotechnique doit être placé en colonne 7, les opérandes en colonne 18 et 24, etc. Toutefois, même avec un format libre, des règles de lisibilité imposent une certaine discipline. L'exemple du programme de la figure 4.2 montre qu'un tabulateur est utilisé devant

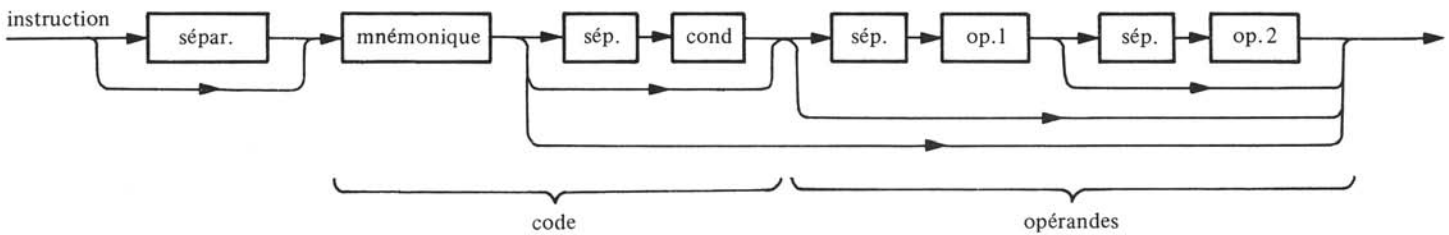

instruction
sépar.
mnémonique
sép.
cond
sép.
op.1
sép.
op.2
code
opérandes

Fig. 4.12

le mnémonique de l'instruction et devant le premier opérande. Une virgule sépare un mnémonique de saut conditionnel et la condition associée, ainsi que les opérandes. Des espaces supplémentaires peuvent permettre des effets tels que l'on en voit dans les langages évolués pour mettre en évidence la structuration en modules.

4.2.16 Directive

Le diagramme de syntaxe des directives (fig. 4.13) montre que l'on distingue les *pseudo-instructions d'affectations* qui définissent les symboles, les *pseudo-instructions de commande*, qui sont des instructions générales pour l'assembleur et les *pseudo-instructions de génération*, qui réservent ou agissent des positions mémoire.

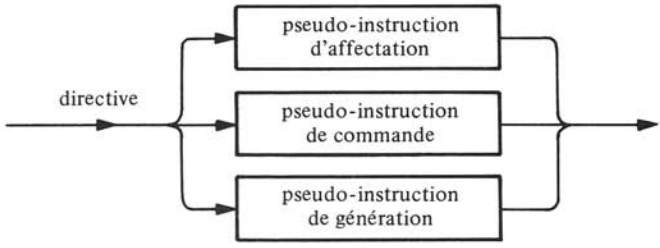


Fig. 4.13

4.2.17 Pseudo-instruction d'affectation

La pseudo-instruction d'affectation, caractérisée dans la plupart des assembleurs par l'expression EQU (abréviation de "equate") est caractérisée dans CALM par le signe "=" (égal). Cette pseudo-instruction assigne une valeur à un symbole. Cette valeur est le résultat du calcul d'une expression, définie plus loin. Le diagramme de syntaxe de l'affectation est donné dans la figure 4.14.

Dans le programme de la figure 4.2 les lignes 11 et 12 sont des affectations.

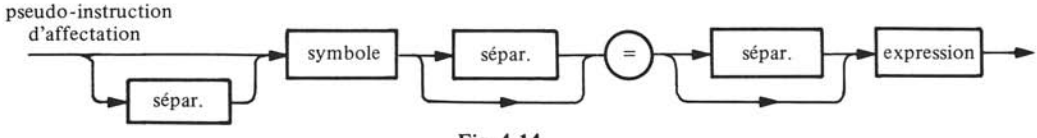


Fig. 4.14

4.2.18 Pseudo-instructions de commande

Les pseudo-instructions ont été classées en deux catégories, selon qu'elles génèrent ou non des positions mémoire. Les pseudo-instructions de commande (fig. 4.15) ne génèrent pas de code binaire et ne réservent pas de place en mémoire.

Elles agissent sur l'assembleur pour:

- définir la mise en page (pseudo-instructions .TITLE, .CHAP, .LIST);
- définir le compteur d'adresses de l'assembleur (pseudo-instruction .LOC, exemple ligne 15 de la figure 4.2);
- de réserver une zone mémoire d'une certaine longueur (pseudo-instructions .BLK.8, .BLK.16, .BLK.32);

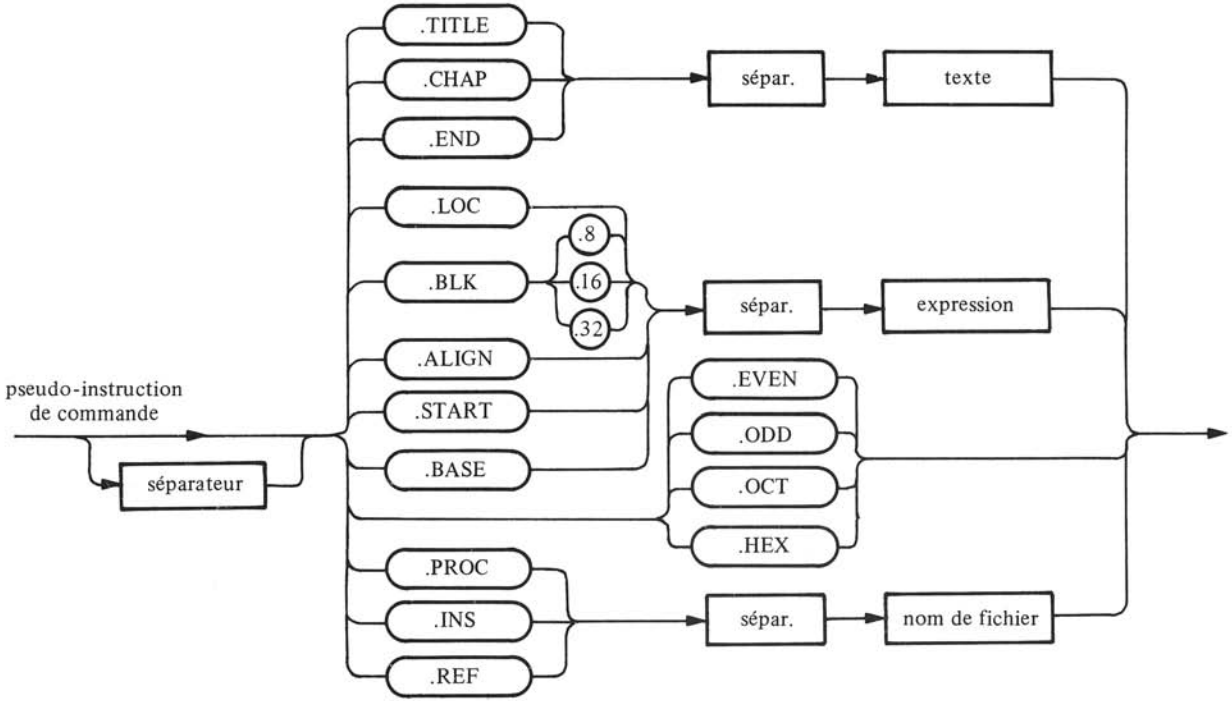

pseudo-instruction
de commande
séparateur
.TITLE
.CHAP
.END
.LOC
.BLK
.8
.16
.32
.ALIGN
.START
.BASE
.PROC
.INS
.REF
sépar.
texte
sépar.
expression
.EVEN
.ODD
.OCT
.HEX
sépar.
nom de fichier

Fig. 4.15

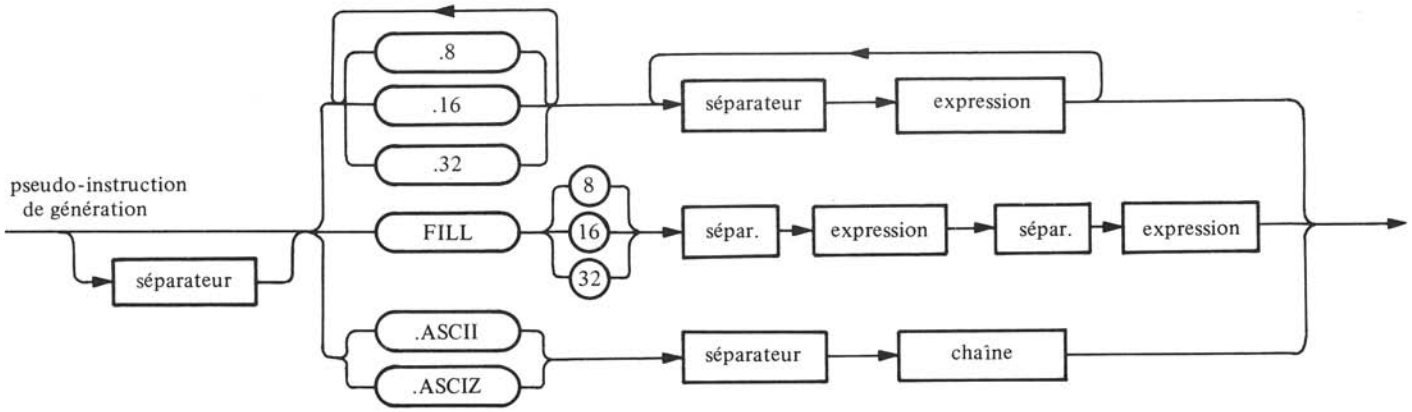

pseudo-instruction
de génération
séparateur
.8
.16
.32
FILL
8
16
32
.ASCII
.ASCIZ
séparateur
expression
sépar.
expression
sépar.
expression
séparateur
chaîne

Fig. 4.16

- d'aligner le compteur d'adresses de l'assembleur à la prochaine valeur puissance de 2 ou à une valeur impaire (.ALIGN, .EVEN, .ODD);
- terminer le processus d'assemblage (pseudo-instruction .END, exemple en ligne 41 de la figure 4.2);
- de préciser l'adresse de la première instruction à exécuter après chargement du programme (.START);
- modifier la base du système de numération utilisé par défaut (pseudo-instruction .BASE);
- modifier la base utilisée pour l'impression du fichier listing (pseudo-instructions .OCT, .HEX);
- définir le processeur pour lequel le code doit être généré (pseudo-instruction .PROC, exemple en ligne 14 et 29 de la figure 4.2);
- appeler un module à insérer (pseudo-instructions .INS, .REF);

La plupart de ces pseudo-instructions ont un opérande qui est un texte (§ 4.2.21), un nombre ou une expression (§ 4.2.23) ou un nom de fichier (§ 6.1.7).

4.2.19 Pseudo-instructions de génération

Le diagramme de syntaxe des pseudo-instructions générant du code est donné dans la figure 4.16.

Le code peut être généré:

- en plaçant en mémoire une suite d'octets, doublets, quadlets dont on donne la liste (pseudo-instructions .8, .16, .32);
- en plaçant en mémoire une suite mélangée d'octets, doublets ou quadlets, dont l'ordre de succession est défini par la pseudo-instruction (par exemple .16.8.32.8.8);
- en remplissant une zone mémoire avec une valeur prédéfinie;
- en plaçant en mémoire les codes ASCII correspondant à un texte (pseudo-instructions .ASCII, .ASCIZ). La pseudo-instruction .ASCIZ ajoute automatiquement un caractère NULL (code égal à zéro) à la fin du texte.

D'autres pseudo-instructions associées aux possibilités supplémentaires de textes longs (§ 4.2.21), de macroinstructions (§ 4.3.12), de lien à des modules translatables (§ 4.3.8) seront revues par la suite. Des exemples de pseudo-instructions spéciales seront rencontrés par la suite.

4.2.20 Séparateurs

Un séparateur est au choix une suite non vide d'espaces (noté SPACE ou ␣), de virgules (,) ou de tabulateurs (TAB) (fig. 4.17). Le tabulateur définit l'emplacement du prochain caractère comme étant au moins une position plus loin et à une distance multiple de 8 du premier caractère de la ligne. Dans la pratique, le choix entre virgule ou tabulateur est fixé par le contexte. L'espace est rarement utilisé comme séparateur.

4.2.21 Texte et chaîne

Des distinctions précises doivent être faites pour les différents types de textes que l'on peut rencontrer dans les titres, en commentaire ou que l'on veut encoder pour qu'ils

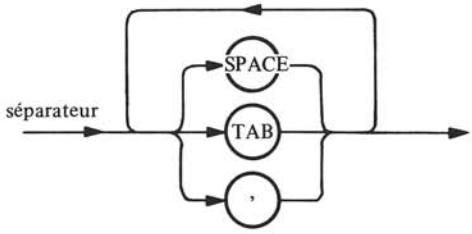


Fig. 4.17

soient restitués lors de l'exécution du programme. Dans chaque cas, l'assembleur réagit différemment. Pour un titre (dans un .TITLE ou .CHAP), il sauve le titre dans une mémoire tampon, pour le répéter en haut des pages du listage. Ce qui déborde de cette mémoire tampon (de 80 caractères par exemple) est perdu, et tous les caractères sont acceptés sauf naturellement le CR qui termine la ligne.

Dans un commentaire, le texte est tout simplement ignoré par l'assembleur, qui ne surveille que le CR de fin de ligne. Si la ligne est trop longue, il y a toutefois erreur. Le diagramme de syntaxe d'un texte est donné dans la figure 4.18.

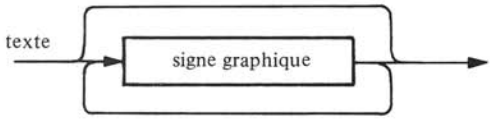


Fig. 4.18

Dans un texte codable, l'assembleur doit convertir chaque signe en code ASCII, et l'utilisateur doit pouvoir générer des codes ASCII spéciaux ou non tolérés par l'assembleur comme le CR. Ces signes spéciaux sont définis par leur code mnémonique placé entre crochets. Ceci interdit l'utilisation simple du signe ⟨ dans les textes codables. Pour générer le code du signe ⟨ il faut écrire ⟨H'3C⟩ ou mieux ⟨'⟨⟩ (§ 4.2.24).

Pour faire apparaître le code du retour de chariot, afin qu'un texte écrit sur une ligne en assembleur soit affiché ultérieurement sur deux lignes lors de l'exécution du programme affichant ce texte, il faut écrire ⟨CR⟩.

Il est important de bien délimiter les textes codés par rapport aux instructions assembleur. Un caractère spécial de parenthésage est utilisé dans ce but, comme le "ou". Ce caractère ne doit pas se trouver dans le texte puisqu'il termine la chaîne. Le choix du caractère de parenthésage est indifférent à l'assembleur: il met à part le premier caractère

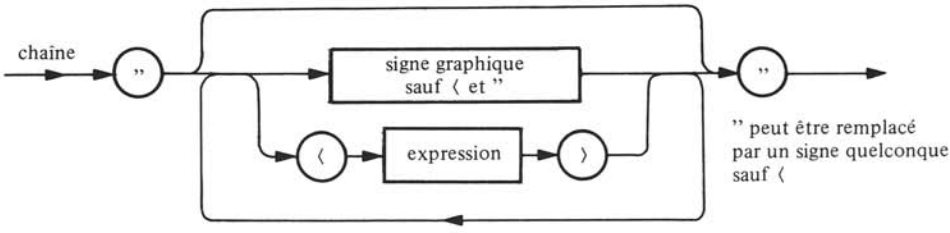


Fig. 4.19

rencontré et le compare avec les suivants pour savoir quand interrompre le codage. Le diagramme de syntaxe correspondant est donné dans la figure 4.19.

Ceci permet la représentation de textes au kilomètre, dont la mise en page se fait à l'affichage.

4.2.22 Exercice

Ecrire la ligne permettant de placer en mémoire les caractères qui, une fois transmis à une imprimante, donnent le message suivant sur deux lignes:

$A < B$ si
B, C et $D > 0$

4.2.23 Expression

Une expression est formée de nombres, symboles, parenthèses et opérations arithmétiques. Lorsque la valeur des symboles utilisés est connue, la valeur de l'expression

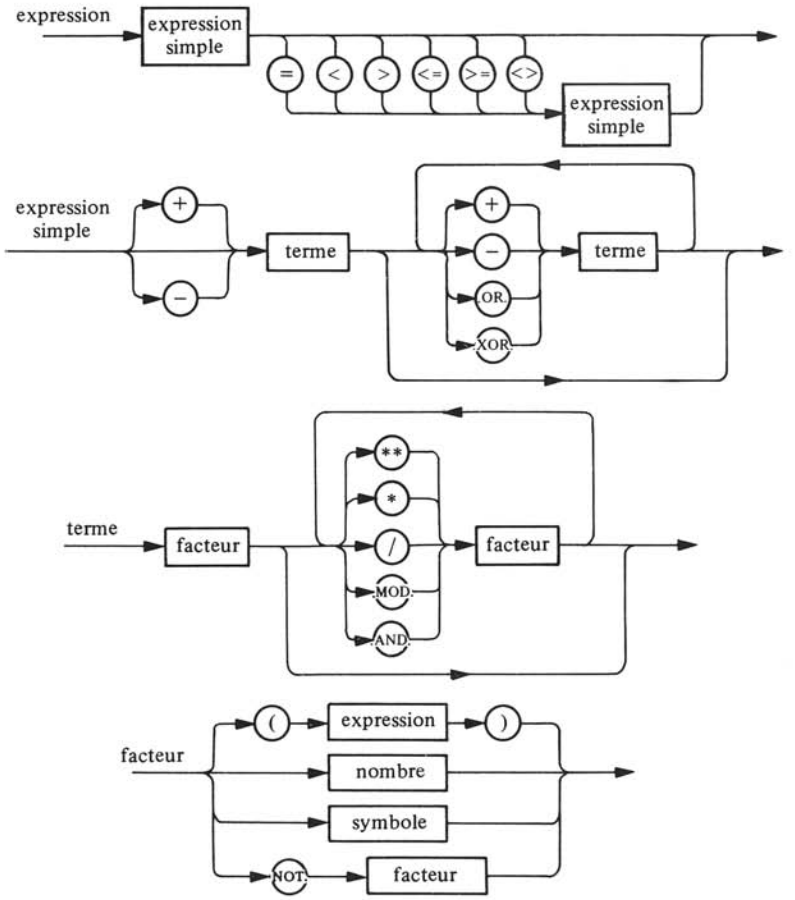


Fig. 4.20

peut être calculée, avec un reśultat entier. Suite à une division, la partie fractionnaire est tronquée, sans arrondi. La représentation en mémoire est un nombre arithmétique dont la longueur dépend de l'instruction. L'assembleur signale les dépassements de capacité et divisions par zéro lorsqu'il calcule une expression.

Le diagramme de syntaxe d'une expression est donné dans la figure 4.20. La définition est récursive étant donné que le facteur d'une expression peut être une expression, avec les règles usuelles de parenthésage.

Les opérateurs multiplicatifs: exposant (**), fois (*), divise (/) et ET logique ont la priorité sur les opérations additives: plus (+), moins (−) et les deux types de OU logiques. Dans certains assembleurs, cette règle usuelle de priorité n'est toutefois pas respectée, l'expression s'évaluant dans l'ordre de lecture.

Les expression formées avec les signes d'égalité ou d'inégalité sont booléennes. Elles valent 0 ou −1 selon que la relation est vraie ou fausse.

La figure 4.21 donne une expression assez complexe et montre sa décomposition en termes, facteurs et expressions.

La valeur de cette expression peut être calculée si l'on connaît la valeur PAGE et BMAP. Si ces deux symboles valent 3, l'expression vaut 21..

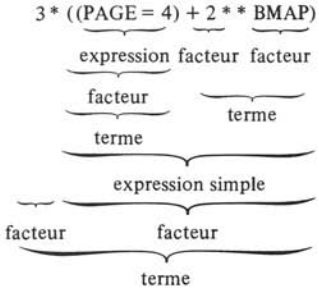


Fig. 4.21

4.2.24 Symboles et nombres

Avec les notions de symbole et de nombre, on atteint le dernier niveau de définition. Les diagrammes de syntaxe ne contiennent plus de rectangles impliquant de nouvelles définitions.

Un symbole commence toujours par une lettre, pour le distinguer d'un nombre. Souvent quelques signes non affectés à d'autres buts sont permis dans la formation d'un

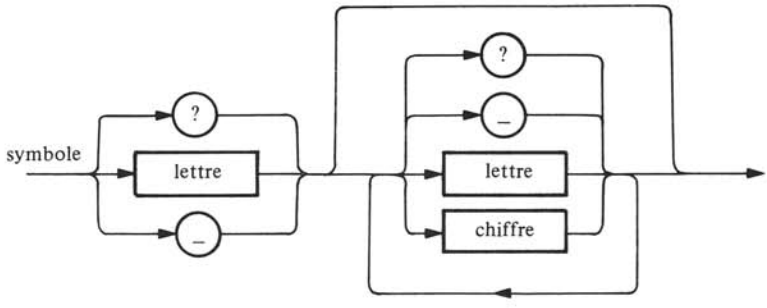


Fig. 4.22

symbole. C'est le cas en particulier des signes ? et _ (souligné) qui peuvent être utilisés librement dans un symbole CALM (fig. 4.22).

La représentation des nombres a été vue aux paragraphes 4.2.4 et 4.2.8. La définition complète de la figure 4.23 correspond à la définition des codes des signes graphiques, qui font intervenir le signe réservé ". Par exemple, "A" est le code ASCII de la lettre A, et a pour valeur H'41. Le symbole APC est un nombre particulier dont la valeur est celle du compteur d'adresses au début de l'instruction courante.

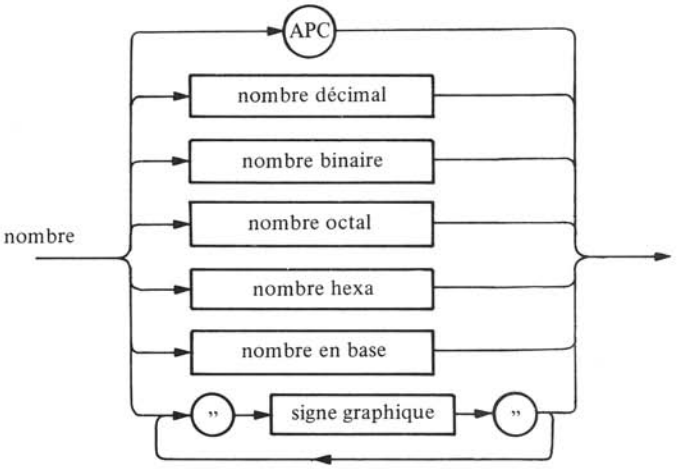


Fig. 4.23

4.2.25 Exercice

Dans la liste qui suit, dire chaque fois s'il s'agit d'un expression, d'une chaîne, d'un symbole, d'un nombre ou d'une écriture non autorisée.

A_ET_B
"TRUC"
−36
0A3F
/APPELLE "DIEZE"/
ADRESSEDEDEBUTDEROUTINE
" "
3 * (TRUC + 2) .OR. 125

4.2.26 Exercice

Calculer en hexa la valeur des expressions suivantes, sachant que TOTO = H'102, TRUC = 6, CHOSE = 4

TOTO * TRUC/H'100
TOTO * TRUC. AND. H'FF
(TRUC + CHOSE) * 2
TOTO. AND. TRUC * CHOSE
TOTO. OR. TRUC * CHOSE

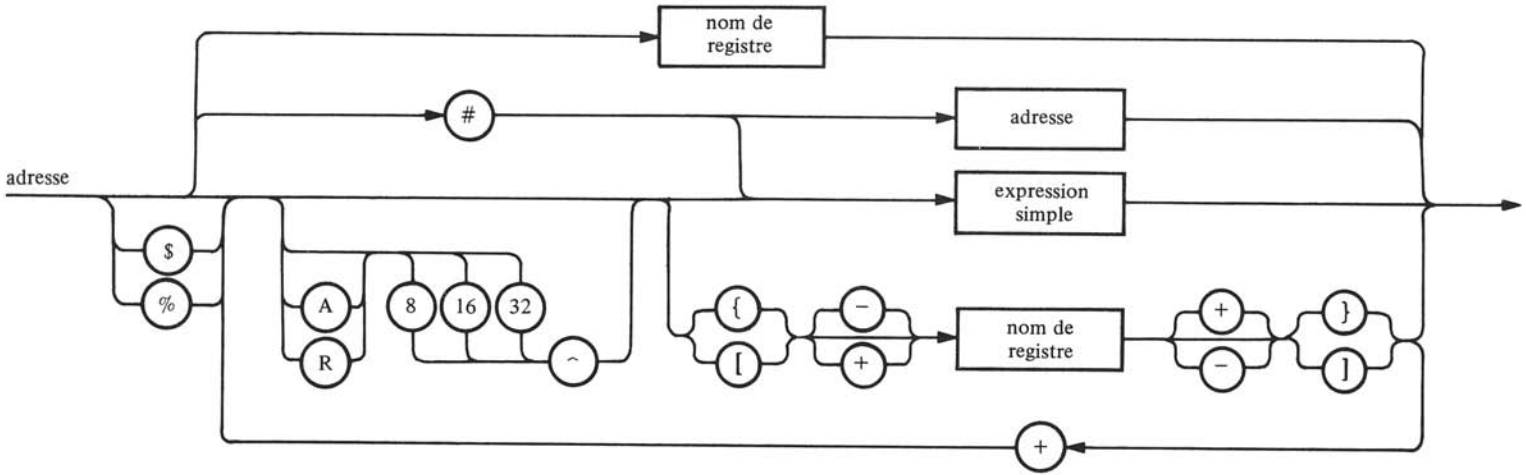

adresse
nom de registre
#
adresse
expression simple
$
%
A
R
8
16
32
~
{
[
−
+
nom de registre
+
−
}
]
+

Fig. 4.24

4.2.27 Expressions d'adressage

Les expressions d'adressage décrivent les possibilités de l'unité de calcul d'adresse du processeur et dépendent du processeur. La syntaxe de ces expressions évoluera en fonction des nouvelles possibilités de calcul d'adresse, par exemple le produit de deux registres facilitant l'accès à des tableaux à 2 dimensions.

Le diagramme de la figure 4.24 résume les expressions d'adressage supportées actuellement par CALM3. On peut y retrouver les modes d'adressage vus à la section 3.5.

Par exemple, le microprocesseur M68000 reconnaît les modes d'adressage suivants:

```
immédiat:          #VAL
                   #{A0}+{D3}+DEPL (l'un des modes d'adressage qui suit)

absolu et          ADVAL
relatif               A16*ADVAL
                      32*ADVAL
                    R*ADVAL
                      R16*ADVAL
                      R8*ADNEXT  (seulement pour les sauts)

indexés:           {A0}
                   {A0+}
                   {-A0}
                   {A0}+DEPL
                   {A0}+{D3}+DEPL
                       {A0}+A16*{D3}+DEPL
                       {A0}+32*{D3}+DEPL

                   ADVAL+{D3}
                       ADVAL+A16*{D3}
                       ADVAL+32*{D3}
```

Le Z80 est nettement moins riche en modes d'adressage mais dispose d'un espace d'entrée-sortie avec l'adressage indexé par rapport à l'un des registres (fig. 4.25). Chaque mode d'adressage n'est disponible qu'avec un nombre restreint d'instructions; le répertoire complet figure en annexe (§ 7.3.1).

Le 8085, dont le Z80 est une extension, est encore plus pauvre en modes d'adressage (fig. 4.26).

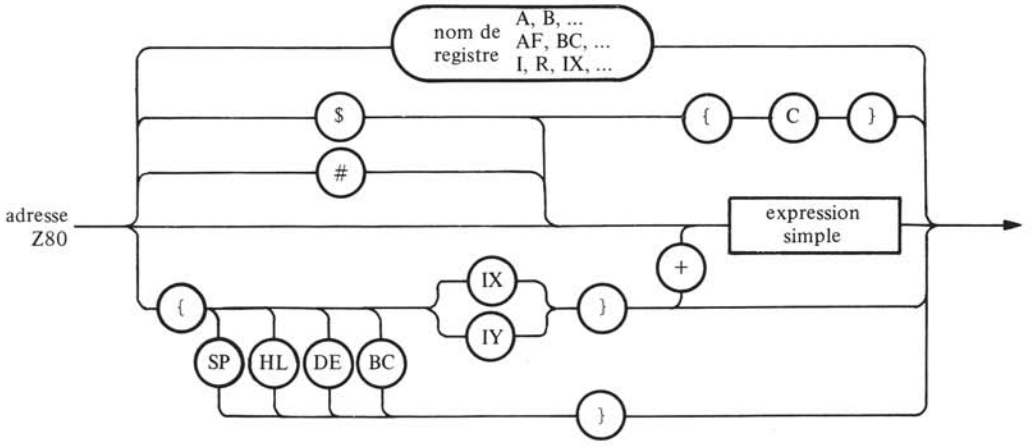


Fig. 4.25

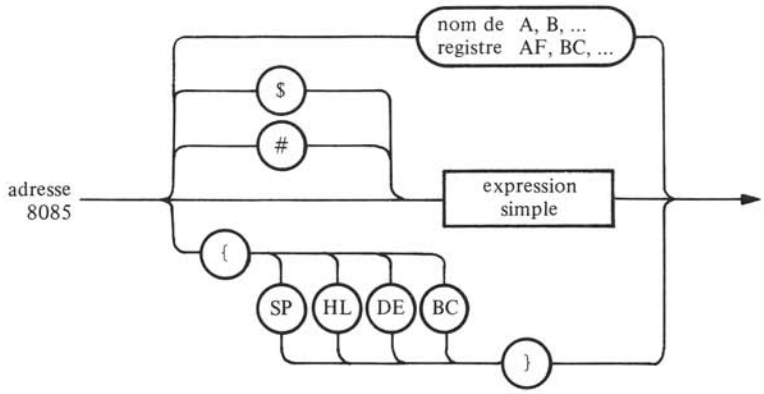


Fig. 4.26

4.3 TRAVAIL DE L'ASSEMBLEUR

4.3.1 Définitions

Un assembleur est un programme qui traduit un programme ou fichier source écrit en langage d'assemblage en programme objet binaire, avec en option un fichier listing et une indication des erreurs. En cas d'erreurs, le fichier objet est généralement supprimé, car il n'a plus de signification.

Lorsque l'assembleur produit du code pour la machine même sur laquelle il s'exécute, on parle d'*assembleur résident*. Si la machine est différente, on parle d'*assembleur croisé* ou *cross- assembleur*. C'est souvent le cas avec les microprocesseurs, pour lesquels le développement des programmes se fait en général avantageusement sur un ordinateur plus rapide et plus complet.

Le langage utilisé pour écrire un assembleur peut être quelconque. S'il est écrit en langage d'assemblage de la machine hôte, une efficacité maximale en résulte. L'évolution technologique rapide et la multiplicité des types de microprocesseurs encouragent l'écriture d'*assembleurs paramétrés* dans lesquels les instructions de génération des codes sont déterminées par une table dépendant du processeur.

4.3.2 Traduction des mnémoniques

La détermination des codes de chaque instruction est l'opération la plus facile. Dans quelques langages de microprocesseurs simples, il y a correspondance directe entre chaque mnémonique et les valeurs binaires des instructions. Il suffit alors d'assembler ce code avec l'adresse des opérandes. Les mnémoniques sont plus compréhensibles s'ils expriment uniquement la nature de l'opération avec une notation claire pour les opérandes et modes d'adressage, comme cela est fait dans le langage CALM. Une analyse beaucoup plus complète de l'instruction est alors nécessaire pour déterminer le code correspondant de l'instruction.

La plupart des instructions sont formées, en plus du code de l'instruction, d'une adresse ou d'une valeur exprimée dans le langage source sous forme de symboles ou ex-

pressions. La valeur binaire ne peut être déterminée que si la valeur de tous les symboles utilisés dans l'instruction est connue. Ceci est rarement le cas, car le fichier source est parcouru linéairement par le programme assembleur, et un symbole rencontré dans une instruction peut n'être déclaré par une affectation ou une étiquette qu'ultérieurement. Ainsi donc, le fichier source doit être parcouru au moins deux fois pour permettre l'achèvement du processus d'assemblage.

4.3.3 Table des symboles

Lors de la première lecture du fichier source, une *table des symboles* est créée par l'assembleur. Cette table associe à chaque symbole utilisé dans le programme sa valeur (fig. 4.27). La place réservée pour chaque symbole peut être fixe ou variable; en général la longueur mémorisée est limitée à 12 caractères, mais les symboles peuvent être plus longs tant que leurs 12 premiers caractères sont différents. Quelques bits d'état doivent être associés à chaque symbole, pour caractériser la nature de ce symbole, certaines conditions particulières, normales ou anormales.

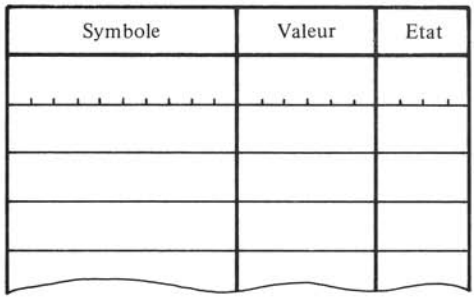


Fig. 4.27

Tout nouveau symbole est placé dans la table. Si sa valeur est connue, par exemple si c'est une étiquette ou le premier terme d'une affectation dont le second terme est défini, la partie valeur de la table peut être complétée et un bit d'état mis à jour. Une double définition peut être marquée par un autre bit d'état, et apparaître ultérieurement dans la liste des erreurs.

Pour chaque symbole trouvé dans le programme source, il faut parcourir la table des symboles pour connaître sa valeur. Cette recherche ralentit l'exécution et différentes méthodes permettent d'accélérer la recherche. Une solution naturelle serait de classer les symboles dans 28 sous-tableaux selon la première lettre du symbole (A, B, ..., Z, ?, –). Ces sous-tableaux ne se rempliraient toutefois pas de façon régulière, et l'on préfère calculer à partir de chaque symbole un nombre valant de 0 à $n-1$, n valant par exemple 16. La règle de correspondance qui fournit ce nombre est appelée *fonction de hachage* [46].

Une fonction de hachage simple consiste à additionner les codes ASCII des caractères du symbole et à calculer le reste de la division par n. Ceci fournit une distribution statistique uniforme des symboles dans les classes définies par la fonction de hachage. Le fractionnement de la table des symboles accélère la recherche, mais complique les opérations lorsque l'une des tables est pleine.

4.3.4 Passes

Chaque lecture du programme source par l'assembleur est appelée *passe*. Dans la première passe, la table des symboles est créée. Ceci implique un décodage partiel des instructions et des pseudo-instructions afin de maintenir à jour le compteur d'adresses de l'assembleur et définir correctement les étiquettes. La plupart des erreurs de syntaxe et les doubles définitions sont reconnues en première passe, mais elles ne sont pas nécessairement signalées à ce moment là.

Dans la seconde passe, le code est généré et accumulé dans un fichier objet; simultanément un fichier listing est créé. Les erreurs sont marquées dans le fichier listing, parfois également dans un fichier d'erreurs spécial ou dans le fichier source. Souvent le fichier listing est complété par la liste des symboles et les adresses de définition et d'occurrence de ces symboles.

La figure 4.28 résume schématiquement les opérations faites par les deux passes de l'assembleur.

4.3.5 Traitement des erreurs

Un programme ne peut être assemblé correctement que si les règles lexicales, syntaxiques et sémantiques sont respectées.

En d'autres termes, les noms réservés et les symboles ne doivent pas avoir de fautes d'orthographe, les règles données par les diagrammes de syntaxe doivent être respectées et les instructions doivent avoir un sens pour le processeur considéré.

La plupart des erreurs lexicographiques, des erreurs de syntaxe et les doubles définitions sont reconnues en première passe. En seconde passe, l'assembleur évalue les expressions et trouve les erreurs associées et les valeurs incorrectes, par exemple, un adressage relatif avec une amplitude excessive ou un octet dont la valeur est supérieure à 255.

L'assembleur ne trouve naturellement pas les erreurs de sémantique d'un groupe d'instructions, ni les fautes de frappe ayant conduit à un identificateur reconnu par l'assembleur. Ces fautes-là résultent du manque de redondance au niveau du langage d'assemblage et sont les plus difficiles à retrouver. Par exemple, si le programmeur a défini deux étiquettes COD et COO, et qu'il tape JUMP COD au lieu de JUMP COO, l'assembleur ne peut pas s'en rendre compte, et le programmeur peut avoir de la peine à retrouver son erreur étant donné la relativement mauvaise qualité des écrans et des imprimantes. De bonnes habitudes de travail permettent de minimiser le nombre d'erreurs et d'accélérer la mise au point.

Les erreurs peuvent être signalées de différentes façons. Si les lignes du programme source sont numérotées, la liste des lignes contenant une erreur et une indication de la nature de l'erreur est suffisante pour la correction.

Parfois, les erreurs ne sont indiquées que dans le fichier listing. Une impression complète, donc longue, de ce fichier est alors nécessaire pour connaître toutes les erreurs.

La solution idéale est d'avoir les erreurs indiquées directement dans le fichier source. La recherche de la correction des erreurs se fait alors avec le programme éditeur et le risque d'effectuer la correction au mauvais endroit est nul.

4.3.6 Fichier objet

Le fichier objet créé par la deuxième passe de l'assembleur doit pouvoir être chargé ultérieurement dans la zone mémoire spécifiée par le programme. Ce fichier doit donc

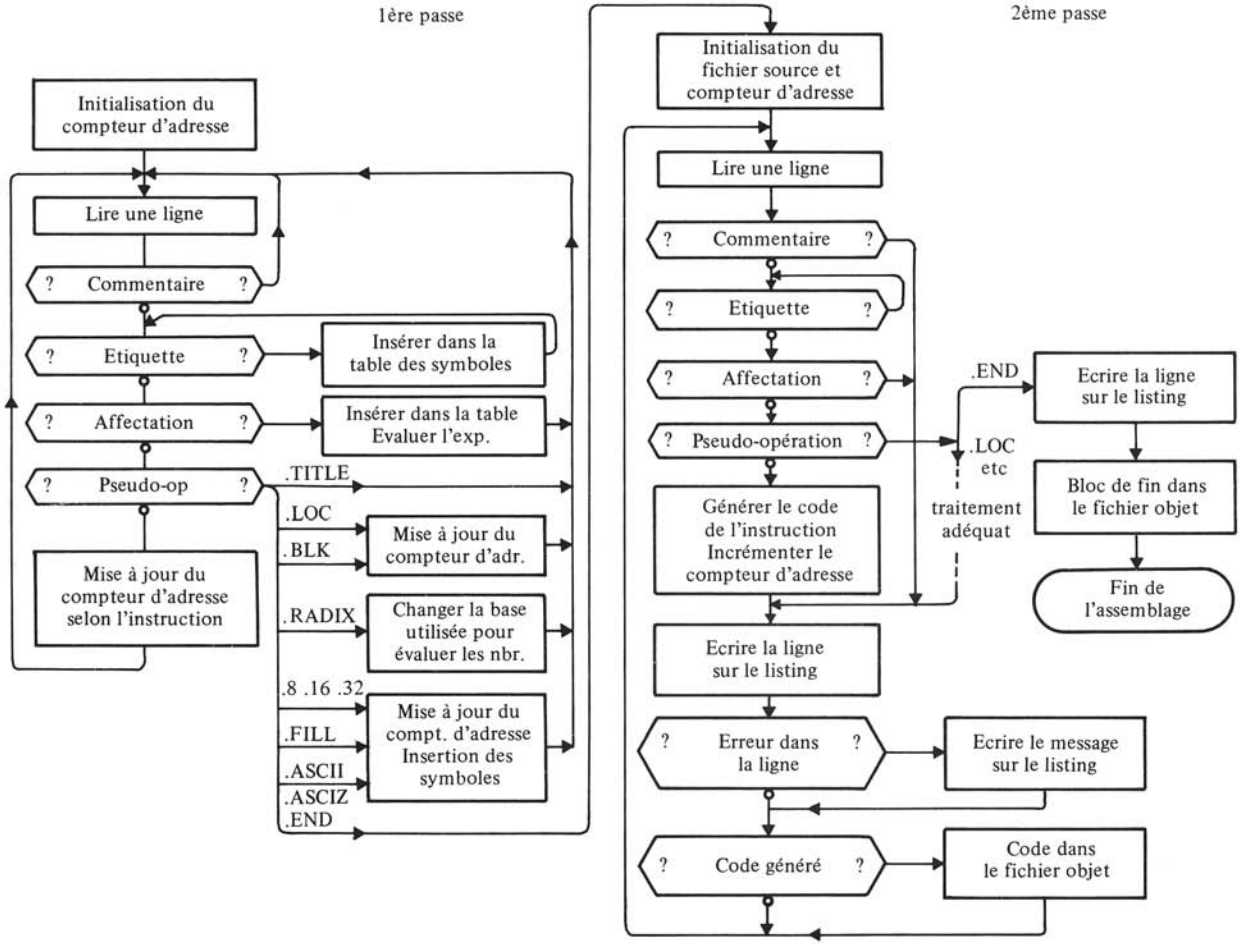

1ère passe
Initialisation du compteur d'adresse
Lire une ligne
? Commentaire ?
? Etiquette ?
Insérer dans la table des symboles
? Affectation ?
Insérer dans la table Evaluer l'exp.
? Pseudo-op ?
.TITLE
.LOC
.BLK
Mise à jour du compteur d'adr.
.RADIX
Changer la base utilisée pour évaluer les nbr.
.8 .16 .32
.FILL
.ASCII
.ASCIZ
.END
Mise à jour du compteur d'adresse selon l'instruction
Mise à jour du compt. d'adresse Insertion des symboles
2ème passe
Initialisation du fichier source et compteur d'adresse
Lire une ligne
? Commentaire ?
? Etiquette ?
? Affectation ?
? Pseudo-opération ?
.END
.LOC etc
traitement adéquat
Ecrire la ligne sur le listing
Bloc de fin dans le fichier objet
Fin de l'assemblage
Générer le code de l'instruction Incrémenter le compteur d'adresse
Ecrire la ligne sur le listing
? Erreur dans la ligne ?
Ecrire le message sur le listing
? Code généré ?
Code dans le fichier objet

Fig. 4.28

contenir en plus des codes des instructions, les adresses de changement et des indications de commande. Une structure en blocs, semblable à celle de la figure 4.1, est généralement utilisée. Chaque fabricant a son propre format; souvent, comme dans la figure 4.1, chaque mot mémoire de 8 bits est transmis comme deux mots de 4 bits successifs, avec pour codage des mots de 4 bits le code ASCII des chiffres hexadécimaux correspondants. Quelques lettres ou signes supplémentaires permettent de distinguer les blocs. Un tel format a l'avantage d'être compatible avec tous les systèmes de transmission entre ordinateur, imprimante, perforateur, modem, etc. Il a toutefois l'inconvénient d'être deux fois plus encombrant, donc plus lent qu'un format transmettant 8 bits à la fois.

4.3.7 Binaire translatable

Le fichier objet défini précédemment doit toujours être chargé au même endroit dans la mémoire, car le programme peut contenir des adresses de tables ou de saut absolues. Si le programme doit être déplacé, par exemple pour le charger derrière un autre programme, il doit être réassemblé, ce qui est un problème relativement mineur avec un assembleur rapide, mais empêche la constitution d'une bibliothèque de modules préassemblés et juxtaposés au moment du chargement en mémoire.

A cause de cela, les assembleurs génèrent un code relatif à l'adresse zéro, appelé *binaire translatable* (*relocatable code*), dans lequel tous les codes correspondant à des différences mémoire absolues sont marqués spécialement. Un chargeur spécial, appelée *chargeur translatable* (*relocatable loader*), ajoute l'adresse effective de début de programme à toutes les adresses absolues au moment du chargement.

4.3.8 Assemblage séparé

L'assemblage séparé de modules accélère le développement de programmes supplémentaires.

Dans un cas simple, les programmes existants sont figés dans une zone mémoire connue (par exemple de la mémoire morte) et il est facile de s'y référer puisque les adresses sont fixes. Un fichier de référence est créé avec les valeurs des symboles de ces programmes. La pseudo-instruction .REF permet de charger la table des symboles correspondante.

Dans le cas général, plusieurs modules se référant les uns aux autres sont associés pour former un programme unique. Chaque module doit clairement indiquer les symboles définis dans d'autres modules (pseudo-instruction .IMPORT) et les symboles qui peuvent être utilisés par d'autres modules (pseudo-instruction .EXPORT). L'assembleur procède pour chaque module à un assemblage partiel créé un *fichier binaire symbolique* commençant çant à l'adresse zéro, et contenant dans une table de relogement les indications permettant d'achever le calcul des codes des instructions incomplètement assemblées. Lorsque le code est indépendant de la position, grâce à l'utilisation de l'adressage relatif, il n'a pas besoin d'être transformé pour être relogé (§ 3.5.8).

4.3.9 Editeur de lien

A partir des fichiers binaires symboliques et d'un *plan de chargement* (*load map*) assignant les emplacements des modules en mémoire, l'*éditeur de lien* (*linking loader*) établit la liaison entre les modules, complète les valeurs des symboles manquantes, et finit le

calcul des adresses absolues. Ce programme, presqu'aussi complexe et lent qu'un assembleur, produit un fichier binaire unique, qui peut alors être chargé et exécuté [46].

Dans certains cas, le relogement peut se faire au chargement et, naturellement, l'utilisation d'une mémoire segmentée (§ 3.8.6) simplifie le relogement des groupes de modules indépendants.

4.3.10 Bibliothèque de programmes

La constitution d'une bibliothèque de programme est bien connue dans les gros ordinateurs et les miniordinateurs. Chaque programme de bibliothèque peut être écrit dans différents langages plus ou moins évolués. La compatibilité doit toutefois être assurée au niveau des programmes binaires translatables associés à chaque programme de librairie.

Pour les microprocesseurs, la notion de bibliothèque de programmes se développe lentement. Il est moins nécessaire de disposer de cette facilité, étant donné la plus grande variété des applications et l'importance de l'optimisation des programmes tant du point de vue de la taille mémoire que de la vitesse d'exécution. Par exemple, une bibliothèque de programmes pour microprocesseurs doit contenir plus de dix types de multiplication; la précision peut varier, les nombres peuvent être entiers, fractionnaires, flottants, binaires ou décimaux. Le même programme de multiplication de deux nombres entiers de 16 bits peut s'écrire en 10 instructions et être très lent, ou au contraire prendre 100 instructions et être très rapide. Une nouvelle possibilité est de confier à des circuits intégrés spéciaux l'exécution des routines "de librairie", comme par exemple les opérations en virgule flottante.

4.3.11 Assemblage conditionnel

Les assembleurs sont généralement complétés par des possibilités supplémentaires, exposées dans la fin de cette section.

L'assemblage conditionnel permet de n'assembler une partie de programme que si une condition est satisfaite. Différentes variantes d'un même programme peuvent alors exister dans un même fichier source. Les pseudo-instructions . IF expression et . ENDIF encadrent la partie de programme qui est assemblée si la valeur de l'expression est différente de zéro. La pseudo-instruction . ELSE peut précéder les instructions qui doivent être assemblées lorsque la condition n'est pas vraie, c'est-à-dire lorsque la valeur de l'expression est nulle.

4.3.12 Macro-instructions

Une *macro-instruction* permet d'appeler une séquence d'instructions par un nom unique. L'utilisation de paramètres de conditions, d'opérations logiques, permet la génération de séquences d'instructions offrant d'innombrables possibilités.

Les macro-instructions les plus simples sont définies par les pseudo-instructions .MACRO symbole et .ENDMACRO. Elles peuvent ensuite être appelées dans le programme en donnant simplement leur nom. Par exemple, les lignes (4.5) définissent la macro LOADHLDE remplaçant l'instruction LOAD HL, DE qui manque dans les microprocesseurs 8085 ou Z80 pour transférer le contenu du registre 16 bits DE dans le registre HL, en utilisant les instructions disponibles sur les registres 8 bits.

```
.MACRO      LOADHLDE
            LOAD  H, D
            LOAD  L, E                    (4.5)
.ENDMACRO
```

Une fois cette macro définie, l'instruction LOADHLDE peut être utilisée dans le programme (4.6):

```
...
ADD         A, B
LOADHLDE                                  (4.6)
JUMP        TRUC
...
```

L'assembleur gérant des macro-instructions s'appelle *macroassembleur*. Il remplace les macro-instructions par les instructions correspondantes lors de la création des fichiers listing et objet. On parle d'expansion des macro-instructions et le listage du programme (4.6) prend par exemple la forme (4.7):

```
1034   80      ADD       A,B
               LOADHLE   macro subsistant en commentaires
1035   62      LOAD      H,D  }  expansion     (4.7)
1036   6B      LOAD      L, E }
1037   18 34   JUMP      TRUC
```

4.3.13 Macro-instructions à paramètre

Les macro-instructions peuvent contenir des paramètres. Par exemple si l'on doit souvent, dans un système 8085/Z80, déplacer un mot mémoire dans une autre position mémoire en mettant à zéro la position source, on peut écrire la macro (4.8):

```
.MACRO      LOADZ    %1,%2 ; %1 destination, %2 source
LOAD        HL, %2
LOAD        %1, HL                        (4.8)
LOAD        HL, #0
LOAD        %2, HL
. ENDMARCO
```

Ceci permet d'utiliser dans le programme des instructions du type:

```
...
LOADZ       TABLE, OBJETS * 2
...                                       (4.9)
LOADZ       TRUC + 2, NOMBRE
...
```

A l'assemblage, le macro-assembleur remplace les *paramètres formels* utilisés dans la définition de la macro-instruction par les expressions choisies par le programmeur. Après expansion, le début du programme (4.9) devient:

```
LOADZ     TABLE, OBJETS * 2 ; macro
LOAD      HL, OBJETS * 2
LOAD      TABLE, HL
LOAD      HL, # 0
LOAD      OBJETS * 2, HL
...
```
(4.10)

Une macro-instruction ne doit pas être confondue avec un sous-programme; elle joue le même rôle, mais la macro-instruction est remplacée chaque fois qu'elle est mentionnée par le groupe d'instructions correspondant.

Les macro-instructions peuvent être imbriquées et offrent, grâce aux possibilités d'appel récursif et d'assemblage conditionnel, des facilités de programmation pour simuler des fonctions qui n'existent pas [14].

4.3.14 Directives de listage

Les directives de listage sont des pseudo-instructions permettant au programmeur de choisir ce qui sera imprimé dans le fichier listing. Les noms de ces pseudo-instructions sont par exemple .LIST, .PAGE, etc., avec des indications supplémentaires pour préciser par exemple que la définition des macro-instructions doit être supprimée, ou leur expansion, ou encore par exemple la table des symboles. Ceci permet de générer des listages ne contenant que l'information souhaitée, dont plus faciles à lire.

Ces directives de listage et d'assemblage peuvent être communiquées à l'assembleur au moment de l'appel de l'assembleur, et influencer la création des fichiers assemblés au moyen d'indicateurs complémentaires.

Des directives supplémentaires peuvent permettre au programmeur d'influencer la forme des fichiers créés. Des pseudo-instructions précisent si le code est absolu ou translatable, si le listage doit inclure l'expansion des macro-instructions. le nombre de lignes imprimées sur une page, etc.

Lorsque ces directives agissent sur l'ensemble du programme, elles peuvent être définies par le programmeur au moment de l'appel de l'assembleur, et apparaître par exemple au moyen de *clés* (*switch*) associées aux noms des fichiers créés. Par exemple l'ordre AS TEST TEST/L/X TEST/B assemble le fichier TEST.SR (source), crée un fichier TEST.LS (listing) avec table de référence croisée et crée un fichier TEST.SM (binaire). L'écriture abrégée AS TEST correspond à la procédure standard qui crée le binaire (s'il est correct) et signale toutes les erreurs.

4.4 STRUCTURATION DES PROGRAMMES

4.4.1 Méthodologie

Il peut sembler facile d'aligner des instructions pour obtenir un programme ayant un comportement intéressant. Pour plus d'une vingtaine d'instructions, l'expérience montre toutefois que l'absence d'une méthode de travail éprouvée entraîne une perte de temps considérable à la mise au point et lors des adaptations ultérieures du programme.

L'écriture d'un programme implique les étapes suivantes:

- analyse et compréhension du problème;
- choix d'un algorithme adapté à sa résolution;

- écriture (description) de l'algorithme dans un langage naturel structuré (pseudo-code);
- choix d'un langage et d'une ou plusieurs méthodes pour la traduction, la mise au point et l'exécution du programme;
- codage de l'algorithme dans le langage choisi, en suivant et corrigeant la description en pseudo-code;
- test et mise au point du programme;
- itération de certaines étapes jusqu'à ce que le tout soit satisfaisant.

A chaque niveau, les choix sont multiples et les contraintes de vitesse et de taill mémoire imposent aux systèmes microinformatiques des contraintes nouvelles par rapport aux systèmes informatiques traditionnels.

L'expérience des programmeurs des gros systèmes est toutefois précieuse. Elle a conduit à la définition des langages évolués dits bien structurés, tels que le PASCAL [67]. Les méthodes de la programmation structurée peuvent toutefois s'appliquer à n'importe quel langage, y compris le langage d'assemblage, et sont souvent trop peu connues des ingénieurs appelés à programmer les systèmes mini et microordinateurs.

En particulier la trosième étape, la description du programme dans un langage naturel inspiré d'un langage informatique est extrêmement importante:

- elle permet une plus grande maîtrise de la logique du processus à codifier;
- elle facilite considérablement le codage dans n'importe quel langage, et garantit dans le cas du codage en assembleur une bonne structure générale;
- elle facilite la définition des tests permettant la vérification du programme;
- elle constitue une partie importante de la documentation associée au programme et doit être incorporée directement au fichier source du programme.

L'objectif de cette section est simultanément de montrer comment tirer parti des avantages de la programmation structurée et de mettre en évidence certaines techniques avancées de programmation, valables aussi bien en langage d'assemblage qu'en langage évolué.

L'orientation générale est toutefois axée sur les langages d'assemblage de microprocesseurs. Les exemples utiliseront les processeurs Z80, et M68000, dont les cartes de référence sont données en annexe (sect. 7.3). L'utilisation du langage CALM facilite la conversion à d'autres processeurs plus ou moins performants et à l'utilisation d'autres notations.

4.4.2 Analyse descendante

Le principe général de la programmation structurée consiste à partir du système donné et, par une suite de décompositions en blocs tous bien définis, à parvenir à une étape qui se laisse naturellement, et sans risque d'erreurs, traduire dans le langage voulu. On parle d'*analyse descendante* (*top down design*). Une condition essentielle est que chaque bloc soit bien défini, avec un point d'entrée et un point de sortie uniques.

Chaque bloc comporte un seul point d'entrée et un seul point de sortie. Il est caractérisé par:

- ce qu'il reçoit (un point d'entrée seulement);
- ce qu'il fait;
- ce qu'il fournit (un point de sortie seulement).

La dimension de chaque bloc est telle que sa description s'exprime clairement. Si le programme est comparé à un livre, le bloc est une section ou un paragraphe. Chaque bloc contient d'autres blocs de niveau inférieur et les blocs de plus bas niveau sont écrits dans le langage choisi, donc décomposés en instructions.

La figure 4.29 illustre le principe de cette méthode.

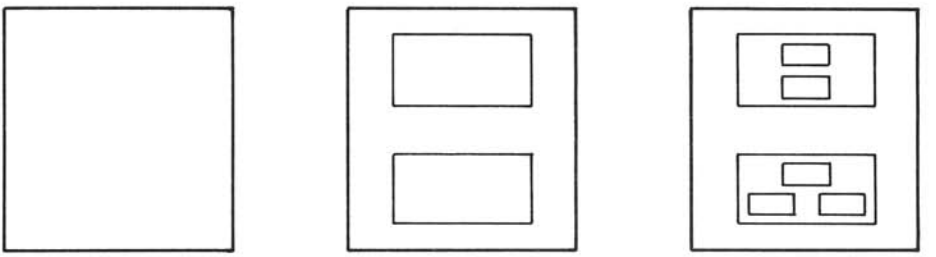


Fig. 4.29

Les avantages que l'on retire de cette méthode sont les suivants:

- meilleure maîtrise d'un problème complexe;
- risque d'erreur réduit;
- temps de test plus courts;
- correction et modification facilitées;
- fiabilité accrue;
- clarté accrue (en particulier pour des tiers).

Une occupation mémoire plus grande et un temps d'exécution plus élevé résultent parfois de la structuration des programmes, mais une bonne structure garantit aussi la compacité et l'efficacité des programmes.

Le temps de réflexion important nécessaire à cette structuration et le nombre de documents intermédiaires apparemment inutiles qu'il faut produire ont pour effet, surtout chez les débutants, que tout-à-coup, le problème semblant être devenu suffisamment clair, les instructions nécessaires pour résoudre le problème sont écrites à la suite les unes des autres en laissant de côté les bons principes: le temps ainsi économisé est généralement perdu ultérieurement.

4.4.3 Programmation montante

Les étapes d'analyse descendante précédemment décrites sont suivies d'une *programmation montante* (*bottom-up programming*) partant des spécifications du langage et, dans le cas d'un langage d'assemblage, de l'architecture de la machine et de son répertoire d'instructions.

Les tests de chaque bloc sont effectués en les appelant depuis un programme de test ad hoc.

La programmation montante permet d'optimiser au mieux le programme en évitant de perdre les avantages de la structuration. Les blocs de niveau le plus bas peuvent être optimisés en vitesse et en taille. Les registres et positions mémoire qu'ils utilisent peuvent

être naturellement utilisés pour transférer l'information avec les blocs de niveau supérieur, évitant les transferts inutiles que l'on rencontre sans cesse en analysant le code produit par un compilateur.

En fait le programmeur expérimenté utilise une approche multiple qui, suivant le cas, se concentre tout d'abord sur les blocs de plus bas niveau, ou au contraire sur le programme principal. Une approche descendante de la programmation et des tests est conseillée lorsqu'elle peut s'appliquer, c'est-à-dire lorsque l'on peut court-circuiter l'accès aux blocs inférieurs et vérifier valablement le comportement.

Il est rare que la solution optimale soit trouvée du premier coup. La programmation structurée garantit une solution qui fonctionne dans un temps minimum. L'optimisation coûte du temps; les indications économiques du programme fixent le niveau d'optimisation que l'on peut raisonnablement atteindre.

4.4.4 Blocs

La notion de bloc vue au paragraphe précédent est très générale. Un *bloc* est une suite d'instructions constituant une entité fonctionnelle. Une condition de la programmation structurée est que chaque bloc ait un seul point d'entrée et un seul point de sortie (fig. 4.30). Cette condition semble difficile à satisfaire lorsque le bloc a, de façon naturelle, un débranchement au milieu (fig. 4.31).

Fig. 4.30

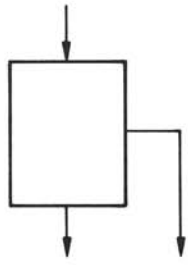


Fig. 4.31

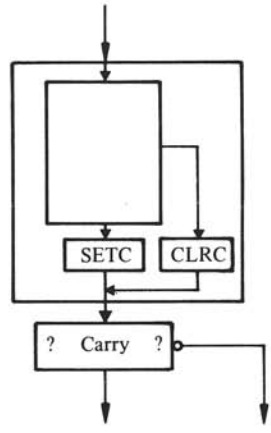


Fig. 4.32

Dans ce cas, l'utilisation d'une variable de condition, par exemple un fanion de l'unité arithmétique ou une variable dans un registre, permet, en rajoutant quelques instructions, de se ramener à un bloc bien structuré ne comportant qu'un seul point de sortie. Une instruction de test de la variable utilisée permet alors d'effectuer le débranchement (fig. 4.32).

Cette façon de procéder semble toujours inutilement compliquée au débutant. Elle est nécessaire lorsque la pile est utilisée pour sauver les variables à l'entrée d'un sous-programme, afin de garantir un rétablissement correct de l'état. Elle est conseillée dans tous les autres cas pour diminuer les risques d'erreur et faciliter une documentation correcte de chaque bloc.

Souvent, un bloc est implémenté sous forme d'un sous-programme ou routine. Il est appelé par une instruction CALL et est terminé par une instruction RETURN unique. Il est évident que dans ce cas il y a un seul point d'entrée et un seul point de sortie. La tentation peut toutefois être grande de placer un saut conditionnel au milieu d'un sous-programme pour retourner au programme principal ou passer dans une autre routine; des erreurs de pile, en plus des difficultés de compréhension du programme, en résultent généralement.

Un bloc peut également être défini sous forme d'une macro-instruction (§ 4.3.12). Les règles d'écriture et d'utilisation sont les mêmes que pour une routine. Les macroinstructions se justifient lorsque le bloc est court et que son appel prend du temps, ou lorsque l'utilisation de paramètres formels et de conditions d'assemblage permet de générer des groupes d'instructions différents au moment de l'assemblage.

4.4.5 Transferts de paramètres

Un bloc ayant une entrée et une sortie, on parlera tout naturellement de paramètres d'entrée et de paramètres de sortie. Un *paramètre* est une information correspondant à une variable booléenne, un nombre, un caractère, etc. Ce paramètre est contenu dans un registre, une position mémoire ou sur la pile. Il peut aussi être donné par un indicateur de l'unité arithmétique (C, Z, V, S), une entrée directe sur le processeur ou une variable dans une interface.

Par exemple, le bloc effectuant le produit de deux nombres, noté MUL *a*, *b* a pour paramètres d'entrée les variables *a* et *b* et pour paramètres de sortie la variable *a* et un indicateur de dépassement de capacité. Parfois MUL *a*, *b* fournit les poids faibles du résultat dans *a* et les poids forts dans *b*. L'indicateur de dépassement de capacité n'existe pas dans ce cas; son rôle est joué par *b* et l'utilisateur est libre de choisir la précision qu'il veut, et de générer l'indicateur de dépassement correspondant.

Dans l'exemple ci-dessus, *a* et *b* représentent les variables contenant les nombres. Ce sont des registres ou positions mémoire et une très grande variété de façons de transmettre les paramètres est possible.

Par exemple, avec le processeur Z80, un bloc calculant le produit de deux nombres entiers positifs de 16 bits utilise fréquemment les registres HL et DE pour contenir les paramètres d'entrée. HL contient le produit et l'indicateur C (carry) est programmé pour indiquer le dépassement de capacité (fig. 4.33).

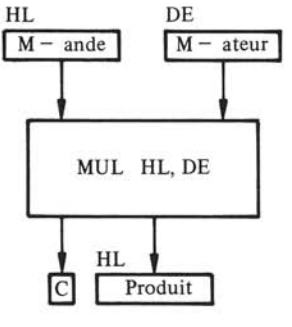


Fig. 4.33

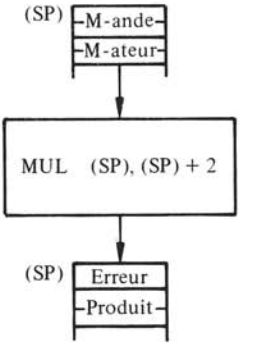


Fig. 4.34

Une variante consiste à remplacer HL et DE par des positions mémoire. Ces positions mémoires peuvent être sur une pile pointée par le pointeur de pile usuel, ou par un registre d'index. L'exemple de la figure 4.34 correspond à ce cas et montre qu'en plus un paramètre de sortie spécial est placé sur la pile, pour indiquer les diverses conditions d'erreurs qui ont pu avoir lieu lors de l'opération. La pile contenant également l'adresse de retour, la programmation de ce cas est délicate.

Les paramètres ne sont pas toujours les opérandes eux-mêmes. Ils peuvent être des pointeurs aux opérandes. C'est par exemple le cas de la figure 4.35 qui multiplie des nombres en multiprécision ou des vecteurs dont la longueur est prédéfinie et ne figure donc pas comme paramètre explicite.

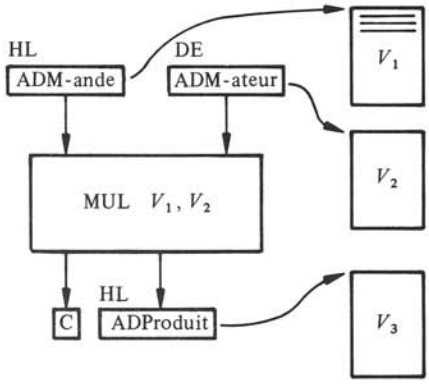


Fig. 4.35

4.4.6 Variables auxiliaires

A l'intérieur d'un bloc, des variables auxiliaires sont généralement nécessaires pour pouvoir effectuer les calculs, tests, transferts.

Ces variables sont prises en général uniquement dans des registres, pour faciliter la rentrance et ne doivent pas apparaître comme modifiées vu de l'extérieur du bloc, si l'on ne veut pas devoir faire attention dans tout le reste du programme au fait que certains registres peuvent être modifiés si le bloc est appelé.

Les registres utilisés comme variables auxiliaires sont sauvés sur la pile à l'entrée du bloc et rétablis en sortie.

Souvent, malheureusement, des soucis d'optimisation en place mémoire ou en vitesse ne permettent pas de suivre cette recommandation. La plupart des problèmes associés à la mise au point des programmes écrits en langage d'assemblage proviennent d'une mauvaise utilisation des variables; une documentation claire de chaque bloc permet de minimiser ces problèmes.

4.5 EXEMPLE: MÉMOIRE TAMPON CIRCULAIRE

4.5.1 Donnée du problème

Etudions la réalisation par programme d'une mémoire silo (§ 1.2.7). En programmation, une telle possibilité de mémorisation est souvent appelée *mémoire tampon circulaire* (*circular buffer*), étant donné l'utilisation d'une zone mémoire pour laquelle les adres-

ses d'extrémité sont identifiées. Dans cette zone mémoire, deux pointeurs se suivent, et caractèrisent respectivement la tête et la queue de la zone mémorisée. Le pointeur de queue indique l'emplacement où sera mémorisé le prochain mot (SIN) et le pointeur de queue indique l'emplacement du prochain caractère qui sortira (SOUT) (fig. 4.36). Deux conditions limites sont intéressantes: lorsque la mémoire tampon est vide, les deux pointeurs indiquent la même position (fig. 4.37); lorsque la mémoire est pleine, le pointeur de queue (SOUT) est une position "avant" le pointeur de tête (SIN), en tenant compte de la relation de circularité (fig. 4.38). Une position reste libre, pour simplifier la distinction entre ces deux cas.

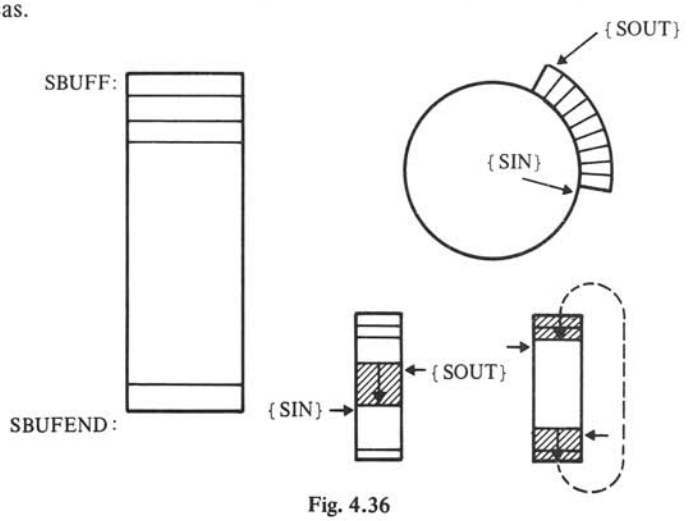


Fig. 4.36

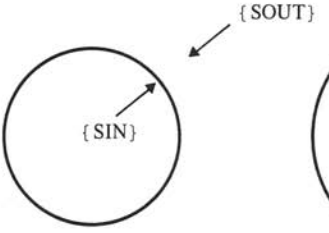


Fig. 4.37

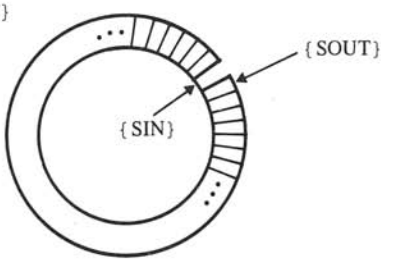


Fig. 4.38

Il importe de ne pas lire un mot lorsque la mémoire tampon est vide et de ne pas en écrire un lorsqu'elle est pleine. Ceci nous amène naturellement à séparer les opérations de test et de transfert.

4.5.2 Décomposition

La décomposition proposée consiste en quatre blocs:

- Test si le silo est plein (STFULL)
- Ecriture d'un mot dans le silo (SWRITE)
- Test si le silo est vide (STEMPTY)
- Lecture d'un mot du silo (SREAD)

On remarque la similitude entre ces 4 blocs et les 4 signaux qui permettent de commander un circuit silo [23].

Les paramètres d'entrée et de sortie de ces blocs sont représentés dans la figure 4.39.

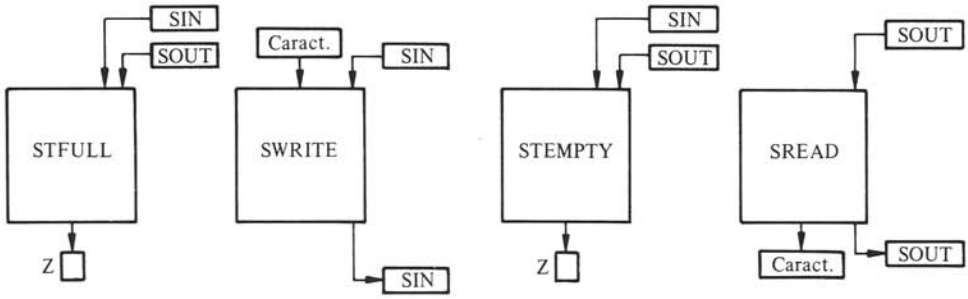


Fig. 4.39

Le premier bloc, STFULL, peut se détailler comme suit:

```
copier la valeur du pointeur SIN dans un registre auxiliaire
incrémenter circulairement le registre auxiliaire
comparer la valeur obtenue avec la valeur du pointeur SOUT          (4.11)
initialiser l'indicateur EQ/NE selon qu'il y a égalité ou non
```

Le second bloc, SWRITE, s'écrit:

```
copier le caractère dans la position pointée par SIN
incrémenter circulairement la valeur du pointeur SIN                (4.12)
```

Le troisième bloc, STEMPY, s'écrit:

```
comparer les contenus de SIN et SOUT
initialiser l'indicateur EQ/NE selon qu'il y a égalité ou non       (4.13)
```

Le quatrième bloc enfin, SREAD, s'écrit:

```
lire le caractère pointé par SOUT
incrémenter circulairement la valeur du pointeur SOUT               (4.14)
```

On remarque que trois de ces blocs font appel à une "incrémentation circulaire" qui n'est pas une notion usuelle et n'existe pas dans les processeurs actuels sous forme d'une instruction simple. Il y a donc avantage à considérer cette opération comme un bloc séparé que l'on peut écrire:

```
ajouter 1 à la valeur de la variable
si la valeur résultante est égale à la valeur SBUFEND
    alors initialiser la variable avec la valeur SBUFF              (4.15)
fin de condition si
```

L'analyse descendante des blocs ne peut pas être poussée plus loin. Le choix du processeur et des registres ou positions mémoire utilisés pour les variables doit être effectué avant de pouvoir commencer la programmation montante.

4.5.3 Codage des blocs.

Choisissons comme processeur le M68000, dont la structure des registres a été vue au paragraphe 3.6.7, et utilisons les notations "MOVE" (§ 7.3.2).

Les variables SIN et SOUT sont conservées en mémoire et la variable correspondant aux caractères lus ou écrits est assignée ici au registre D3, pour faciliter le test de la routine avec les routines de lecture du clavier et d'affichage sur l'écran du micro-ordinateur, qui utilisent ce registre comme paramètre. L'adressage indirect par rapport aux positions mémoire SIN et SOUT n'étant pas possible avec le M68000, le registre A0 sera utilisé temporairement pour remplacer l'adressage indirect manquant par un adressage indexé. SIN, SOUT et A0 sont du type adresse 32 bits. D3 est du type caractère 8 bits.

L'analyse descendante ayant mis en évidence le problème élémentaire de l'incrémentation circulaire, commençons par écrire le bloc correspondant:

```
INCPOIN:    INC. 32     A0
            COMP. 32    # SBUFEND, A0
            JUMP, NE    9 $                          (4.16)
            MOVE.32     # SBUFF, A0
9 $:        RET
```

Ce bloc n'est pas suffisamment bien documenté. Il est nécessaire de bien préciser les paramètres d'entrée et de sortie et les variables éventuellement modifiées par ces instructions. Ceci peut se noter:

```
; -----  INCPOIN  Routine auxiliaire d'incrémentation
; in     A0. 32   pointeur
; out    A0. 32   pointeur incrémenté circulairement      (4.17)
                  dans SBUFF-SBUFEND
; mod    F A0. 32
```

Les blocs de test, de lecture et d'écriture s'écrivent facilement, en suivant les décompositions (4.11) à (4.14):

```
STFULL:     MOVE.32     SIN, A0
            CALL        INCPOIN                      (4.18)
            COMP.32     SOUT, A0
            RET

SWRITE:     MOVE.32     SIN, A0
            MOVE.8      D3, {A0}
            CALL        INCPOIN                      (4.19)
            MOVE.32     A0, SIN
            RET

STEMPTY:    MOVE.32     SOUT, A0
            COMP.32     SIN, A0                      (4.20)
            RET
```

```
SREAD:    MOVE.32   SOUT, A0
          MOVE.8    {A0}, D3
          CALL      INCPOIN                    (4.21)
          MOVE.32   A0, SOUT
          RET
```

La documentation de ces blocs d'instructions est donnée dans le listage complet de la figure 4.40.

```
                          .TITLE  TSILO.SR  (EX440)
                          .PROC   M68000

                          .REF    SM8             ; Références aux routines du systeme
                          .REF    M8SYS

                          .START  TSILO
1000                              .LOC    1000

                          ; **** Programme de test des routines silo

                          TSILO:
1000 23FC0000104C0000108C         MOVE.32 #SBUFF,SIN
100A 23FC0000104C00001090         MOVE.32 #SBUFF,SOUT

1014 A002A020             LOOP:   .32     ?GETCAR ?AFCAR  ; Lecture du clavier et echo
1018 B63C000D                     COMP.8  #CR,D3
101C 67000014                     JUMP,EQ GET

1020 4EB900001094         PUT:    CALL    STFULL          ; Avant d'écrire on verifie
1026 67000020                     JUMP,EQ ERFULL
102A 4EB9000010A4                 CALL    SWRITE
1030 60E2                         JUMP    LOOP

1032 4EB9000010B6         GET:    CALL    STEMPTY         ; Avant de lire on verifie
1038 6700000E                     JUMP,EQ EREMPTY
103C 4EB90000010C0                CALL    SREAD
1042 A024A020                     .32     ?AFSPACE ?AFCAR ; Un espace precède les car relus
1046 60CC                         JUMP    LOOP

                          ERFULL:                         ; Procédures d'erreur simplifiees: un bruit
                          EREMPTY:
1048 A044                         .32     ?BUZZ
104A 60C8                         JUMP    LOOP

                          ; **** Variables en mémoire vive

104C              0040    SBUFF:  .BLK.8  64.     ; Silo
                          SBUFEND:

108C              0004    SIN:    .BLK.32 1       ; Pointeurs circulaires
1090              0004    SOUT:   .BLK.32 1

                          ; **** Routines SILO

                          ;-------\
                          ; STFULL >      Teste si le silo est plein
                          ;-------/
                          ; in    SIN pointeur écriture  SOUT pointeur lecture
                          ; out   EQ si plein
                          ; mod   F  A0.32

1094 2078108C             STFULL: MOVE.32 SIN,A0
1098 4EB9000010D2                 CALL    INCPOINT
109E B1F81090                     COMP.32 SOUT,A0
10A2 4E75                         RET

                          ;-------\
                          ; SWRITE >      Ecrit dans le silo
                          ;-------/
                          ; in    D3.8 caractère à écrire
                          ;       SIN pointeur écriture
                          ; out   SIN pointeur incrémenté circulairement de 1 byte
                          ; mod   F  SIN  A0

10A4 2078108C             SWRITE: MOVE.32 SIN,A0
10A8 1083                         MOVE.8  D3,{A0}
10AA 4EB9000010D2                 CALL    INCPOIN
10B0 21C8108C                     MOVE.32 A0,SIN
10B4 4E75                         RET
```

Fig. 4.40 (*continue en face*)

```
                        ;-------\
                        ; STEMPTY>       Teste si le silo est vide
                        ;-------/
                        ; in    SIN  SOUT
                        ; out   EQ si vide
                        ; mod   F  A0.32

10B6 20781090           STEMPTY:MOVE.32 SOUT,A0
10BA B1F8108C                   COMP.32 SIN,A0
10BE 4E75                       RET

                        ;-------\
                        ; SREAD  >      Lit dans le silo
                        ;-------/
                        ; in    SOUT pointeur de lecture
                        ; out   SOUT incrémenté circulairement
                        ;       D3.8  caractère lu
                        ; mod   F  D3.8  A0.32  SOUT

10C0 20781090           SREAD:  MOVE.32 SOUT,A0
10C4 1610                       MOVE.8  {A0},D3
10C6 4EB9000010D2               CALL    INCPOIN
10CC 21C81090                   MOVE.32 A0,SOUT
10D0 4E75                       RET


                        ;----   INCPOIN  Routine auxiliaire d'incrémentation
                        ; in    A0.32  pointeur
                        ; out   A0.32  pointeur incrémenté circulairement dans SBUFF-SBUFEND
                        ; mod   F  A0

10D2 5288               INCPOIN: INC.32 A0
10D4 B1FC0000108C               COMP.32 #SBUFEND,A0
10DA 67000006                   JUMP,NE 9$
10DE 41F8104C                   MOVE.32 #SBUFF,A0
10E2 4E75               9$:     RET

                1000    .END
```

Fig. 4.40 (*suite*)

4.5.4 Programme de test

Pour tester les routines silo, le programme doit réaliser toutes les conditions d'utilisation et visualiser de façon aussi claire que possible le fonctionnement général de façon à ce que les erreurs éventuelles soient apparentes pour le programmeur.

Le programme de la figure 4.40 attend des caractères du clavier et les place dans le silo. La touche de retour de chariot est décodée spécialement et extrait un caractère du silo. Avec le choix de cette touche, les caractères lus apparaissent verticalement sur l'écran. En cas de silo plein ou vide, un son se fait automatiquement entendre et le silo n'est pas modifié.

Les routines ?GETCAR et ?AFCAR sont à des adresses fixes dans le système utilisé pour exécuter le programme, et sont définies par le fichier de références SM8 déclaré au début du programme.

Dans ce programme, on notera bien la différence entre l'adressage immédiat, utilisé pour initialiser les positions mémoire SIN et SOUT avec les pointeurs du silo, l'adressage absolu utilisé pour transférer ces pointeurs avec le registre A0, et l'adressage indexé pour transférer l'information pointée par A0.

4.5.5 Exercice

Ecrire le programme ci-dessus pour un autre processeur, par exemple le Z80.

4.6 EXEMPLE: PROGRAMME ARITHMÉTIQUE

4.6.1 Donnée et analyse du problème

Soit à convertir un nombre entier décimal (codé BCD dans un registre de 16 bits) en son équivalent binaire. L'algorithme de conversion choisi effectue les opérations en binaire, puisque le processeur est binaire. Il a été expliqué au paragraphe 2.9.9 et consiste en des multiplications successives par 1010, avec addition à chaque fois du chiffre de poids fort (fig. 4.41).

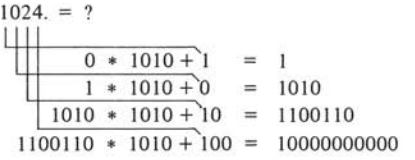


Fig. 4.41

La longueur des registres nombres peut être choisie la même pour le résultat binaire et pour le nombre BCD décimal, car le résultat binaire est plus court que la donnée BCD.

Appelons DECI la variable contenant le nombre décimal et NBDIGIT le nombre de chiffres maximum. Un registre de longueur 4 * NBDIGIT doit être utilisé pour mémoriser DECI. La variable BIN devant recevoir le résultat binaire a la même longueur (fig. 4.42).

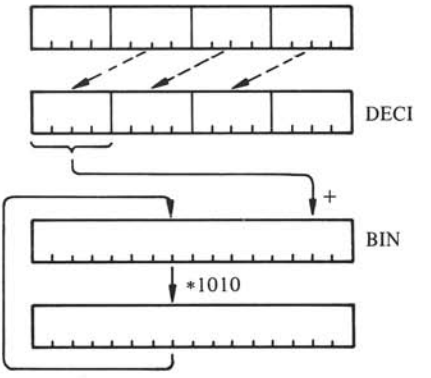


Fig. 4.42

L'algorithme de conversion consiste à prendre successivement les chiffres de DECI, en commençant par les poids forts et à les additionner au résultat partiel précédent multiplié par 1010 en binaire. La figure 4.41 montre les opérations élémentaires à effectuer dans notre cas de conversion. Pour permettre l'écriture d'une boucle de programmation, un décalage du nombre décimal permet de prélever le chiffre à additionner dans un emplacement fixe. La décomposition de l'algorithme de conversion peut alors s'écrire en langage naturel:

```
initialiser BIN avec la valeur zéro
répéter NBDIGIT fois la séquence
    multiplier BIN par 1010 en binaire                    (4.22)
    additionner le chiffre de poids fort de DECI
    décaler à gauche DECI d'un chiffre (4 bits)
fin de répétition
```

Précisons maintenant la description. Notre objectif étant de faire un programme en langage d'assemblage, nous utiliserons les opérateurs binaires usuels vus au chapitre 2.

```
CLR                BIN
repeat             NBDIGIT times
   MUL                BIN, # 10.                    (4.23)
   ADD                BIN, DECI. highdigit
   SL4                DECI
endrepeat
```

La décomposition ne peut pas se continuer rentablement sans tenir compte de la précision visée, de la vitesse de calcul, et de l'architecture du processeur utilisé. L'analyse descendante est terminée.

4.6.2 Programmation

Choisissons de faire la programmation pour le Z80 et limitons les grandeurs des nombres à 4 chiffres, soit 16 bits, afin de bénéficier des instructions d'addition 16 bits du Z80. Un extrait de la table de codage est donné dans la figure 4.43 avec seulement les instructions utiles pour cet exemple. La table complète se trouve en annexe (§ 7.3.1).

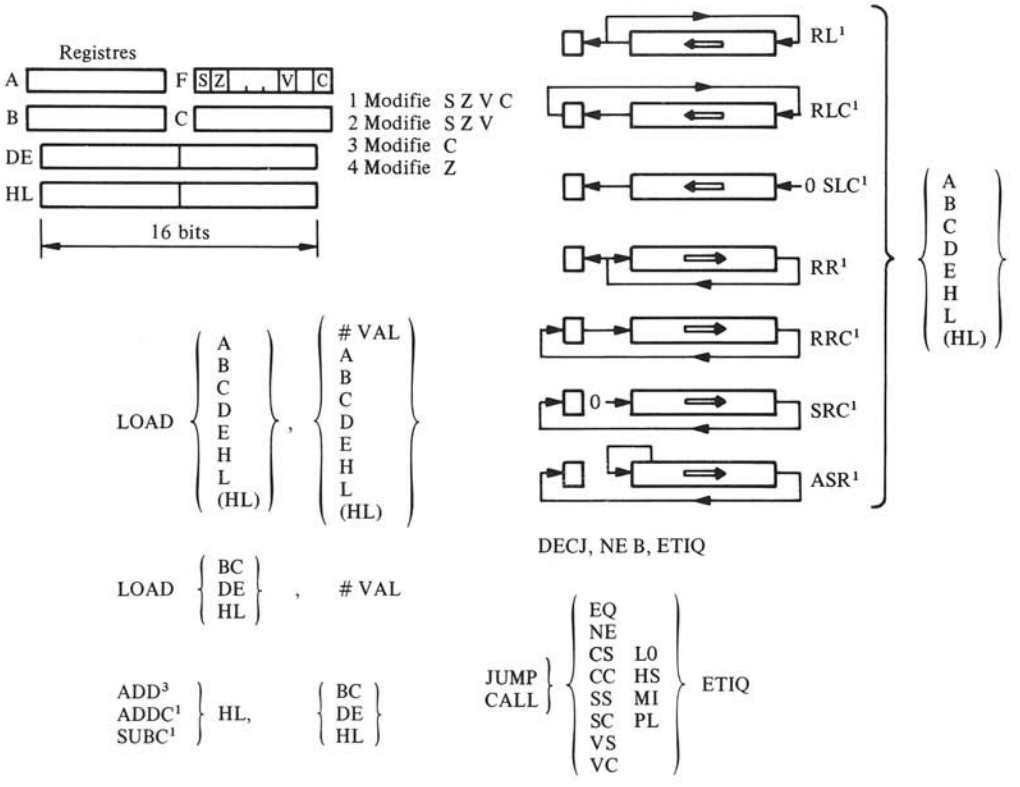


Fig. 4.43

Choisissons tout d'abord l'instruction

```
MUL     BIN, # 10.                                    (4.24)
```

Il serait absurde d'appeler une routine de multiplication générale pour effectuer cette opération particulière de multiplication par 1010 binaire. Des décalages et une addition conduisent au résultat, mais pour utiliser efficacement les possibilités du Z80, le registre HL doit être utilisé pour la variable BIN. Le décalage à gauche de HL s'obtient en en additionnant HL à lui-même (multiplication par 2). L'instruction (4.24) peut alors être remplacée par les instructions

```
ADD     HL, HL          ; HL = BIN
LOAD    saveHL, HL      ; sauve 2 * HL
ADD     HL, HL                                        (4.25)
ADD     HL, HL          ; crée 8 * HL initial
ADD     HL, saveHL      ; additionne 2 * HL et 8 * HL
```

Cette partie de programme a besoin d'un registre ou d'une position mémoire 16 bits appelée pour l'instant " saveHL ". L'addition n'étant possible qu'entre HL et la paire de registres DE ou BC, il y a tout avantage à prendre l'un de ces deux registres. Choisissons BC plutôt que DE, en étant prêts à remettre en cause ce choix si la suite de l'analyse fait apparaître un inconvénient majeur. Le fait que l'instruction LOAD BC, HL n'existe pas n'en est pas un; deux transferts de 8 bits pallient ce défaut et le bloc de multiplication par 10. s'écrit:

```
MUL10:  ADD     HL, HL
        LOAD    B, H
        LOAD    C, L                                  (4.26)
        ADD     HL, HL
        ADD     HL, HL
        ADD     HL, BC
```

Pour utiliser ce bloc sans risque d'erreur, il faut savoir:

- qu'il multiplie par 10. en binaire;
- que le nombre à multiplier doit être donné dans HL;
- que le résultat peut être trouvé dans HL;
- que l'ancien contenu de HL a été détruit;
- que le dépassement éventuel de capacité n'est pas reconnaissable;
- que le contenu de la paire de registres BC est modifié.

Les deux derniers points mentionnés sont très importants car ils sont en dehors du centre du problème posé: leur oubli facile entraînera des erreurs. Une programmation saine supprimerait ces deux points en rajoutant des instructions. Dans le cas particulier, le nombre d'instructions doublerait et nous considérerons ce garde-fous comme trop coûteux d'autant plus que le risque de dépassement de capacité est nul par les conditions mêmes du problème.

Passons à l'instruction suivante:

```
ADD     BIN, DECI. highdigit                          (4.27)
```

Puisque BIN = HL, il faut nous arranger pour utiliser l'une des instructions ADD HL, BC ADD HL, DE. Il faut aussi choisir le registre ou la position mémoire contenant DECI; prenons le registre DE. Le registre BC et le registre A sont libres. L'addition recherchée peut se faire en transférant D (8 bits de poids forts de DECI) dans C, en décalant C à droite de 4 positions, en forçant B à 0, et en ajoutant les contenus de HL et BC.

```
ADDBI:  LOAD    C, D
        SRC     C
        SRC     C
        SRC     C                                   (4.28)
        SRC     C
        LOAD    B, #0
        ADD     HL, BC
```

La répétition de 4 fois la même instruction doit être étudiée de plus près. Une boucle exécutée quatre fois prend moins de place en mémoire, mais ralentit l'exécution et utilise un registre supplémentaire comme compteur.

```
ADDBI:  LOAD    C, D
        LOAD    B, #4
ADB2:   SRC     C                                   (4.29)
        DECJ, NE B, ADB2
        ADD     HL, BC      ; B = 0
```

On remarque que l'instruction initialisant B à zéro n'est plus nécessaire, car le décomptage s'est terminé avec B = 0. Une astuce de ce type n'est pas dangereuse à l'intérieur d'un petit bloc, mais un commentaire doit attirer l'attention sur ce fait.

La comparaison détaillée de (4.28) et (4.29) montre que le premier bloc utilise 12 octets et s'exécute en 21,6 μs (400 ns par état). Le second bloc utilise 8 octets et s'exécute en 40,4 μs. Hors contexte, il n'est pas possible de dire laquelle de ces deux solutions est la meilleure.

La dernière instruction du programme (4.23) qu'il nous reste à analyser est

```
SL4     DECI                                        (4.30)
```

Elle décale à gauche de 4 positions le nombre décimal, mémorisé dans le registre DE.

Avec le Z80, ce décalage peut se faire avec les instructions

```
SLC     E
RLC     D                                           (4.31)
```

répétées 4 fois, ce qui peut avantageusement se faire dans une boucle.

Le registre B, libre, peut être utilisé comme compteur.

```
SL4DE:  LOAD    B, # 4
SL4D2:  SLC     E                                   (4.32)
        RLC     D
        DEC J, NE B, SL4D2
```

Le problème est résolu, l'implémentation de la remise à zéro initiale du nombre binaire et de la structure de commande "repeat" étant triviales. Le programme résultant est donné dans la figure 4.44.

```
.TITLE EX444     ;Exemple de programme arithmetique structuré
.PROC   Z80

.LOC    1000     ;Pour le test du module isolé

;Programme de conversion décimal en binaire
;Convertit un nombre BCD de 4 digits en son equivalent binaire

;in     DE = DECI nombre BCD de 4 digits
;out    HL = BIN  equivalent binaire
;mod    A,B,C  registres modifiés

NBDIGIT = 4      ;Nombre de chiffres BCD par mot

CONVDB: LOAD     HL,#0            ;CLR    BIN

        LOAD     A,#NBDIGIT       ;repeat NBDIGIT times

2$:     ADD      HL,HL            ;  MUL    BIN,#10.
        LOAD     B,H
        LOAD     C,L      ;modifie BC
        ADD      HL,HL
        ADD      HL,HL
        ADD      HL,BC

        LOAD     C,D              ;  ADD  BIN,DECI.highdigit
        LOAD     B,#4     ;modifie B
4$:     SRC      C
        DECJ,NE  B,4$
        ADD      HL,BC

        LOAD     B,#4             ;  SL4  DECI
6$:     SLC      E        ;modifie B
        RLC      D
        DECJ,NE  B,6$

        DEC      A
        JUMP,NE  2$               ;endrepeat

        TRAP             ; Pour le test

.END
```

Fig. 4.44

4.6.3 Exercice

Ecrire le programme de conversion de décimal en binaire pour le processeur M68000.

4.6.4 Variante

Le programme de la figure 4.44 ne satisfait pas le programmeur ayant un peu d'expérience du Z80. En effet, le décalage du registre DE et l'extraction du chiffre de poids fort

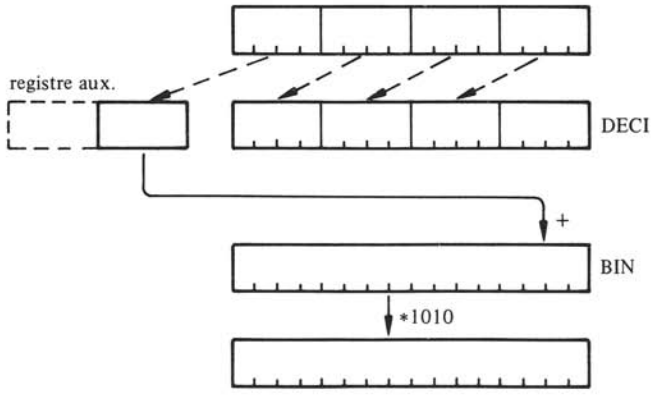


Fig. 4.45

de DE peuvent être faits simultanément, en concaténant un registre auxiliaire devant le registre BIN.

Les deux instructions

```
ADD     BIN, DECI. highdigit                    (4.33)
SL4     DECI
```

sont remplacées dans ce cas par

```
SL4     aux DECI    ; registre 20 bits ou plus  (4.34)
ADD     BIN, aux
```

et la figure 4.42 doit être refaite pour illustrer ce cas (fig. 4.45).

```
.TITLE EX446            ;Variante pour l'exemple de programme
                        ; arithmetique structuré
.PROC     Z80

.LOC      1000          ;Pour le test du module isolé

;Programme de conversion décimal en binaire
;Convertit un nombre BCD de 4 digits en son equivalent binaire

;in       DE = DECI nombre BCD de 4 digits
;out      HL = BIN  equivalent binaire
;mod      A,B,C  registres modifiés

NBDIGIT   = 4           ;Nombre de chiffres BCD par mot

CONVDB:   LOAD    HL,#0                 ;CLR      BIN

          LOAD    B,#NBDIGIT            ;repeat NBDIGIT times

2$:       PUSH    BC

          ADD     HL,HL                 ;  MUL      BIN,#10.
          LOAD    B,H
          LOAD    C,L           ; modifie BC
          ADD     HL,HL
          ADD     HL,HL
          ADD     HL,BC

          LOAD    A,#0                  ;  SL4      ADE
          EX      HL,DE
          LOAD    B,#4          ;modifie B
4$:       ADD     HL,HL
          RLC     A
          DECJ,NE B,4$
          EX      HL,DE

          ADD     A,L                   ;  ADD      HL,A
          LOAD    L,A
          LOAD    A,H
          ADDC    A,#0
          LOAD    H,A

          POP     BC
          DECJ,NE B,2$                  ;endrepeat

          TRAP

.END
```

Fig. 4.46

Le registre auxiliaire tout désigné avec le Z80 est le registre A et les instructions correspondantes ont été utilisées dans le programme de la figure 4.46. On remarque dans ce programme que le registre A n'étant plus utilisable comme compteur de chiffres, une position mémoire de la pile est utilisée.

Ce nouveau programme peut être comparé au précédent du point de vue vitesse et taille. A première vue, la différence ne semble pas suffisamment significative pour justifier le travail qui vient d'être fait à double. Ce genre d'exercice est toutefois très utile jusqu'à ce que les caractéristiques d'un processeur soient toutes bien assimilées.

4.7 STRUCTURES DE COMMANDE

4.7.1 Introduction

L'analyse d'un problème assez complexe conduit souvent à une première décomposition en blocs assez nombreux et à des relations assez compliquées entre ces blocs. La figure 4.47 montre une décomposition de ce type, qui doit à tout prix être évitée, car elle est difficile à maîtriser.

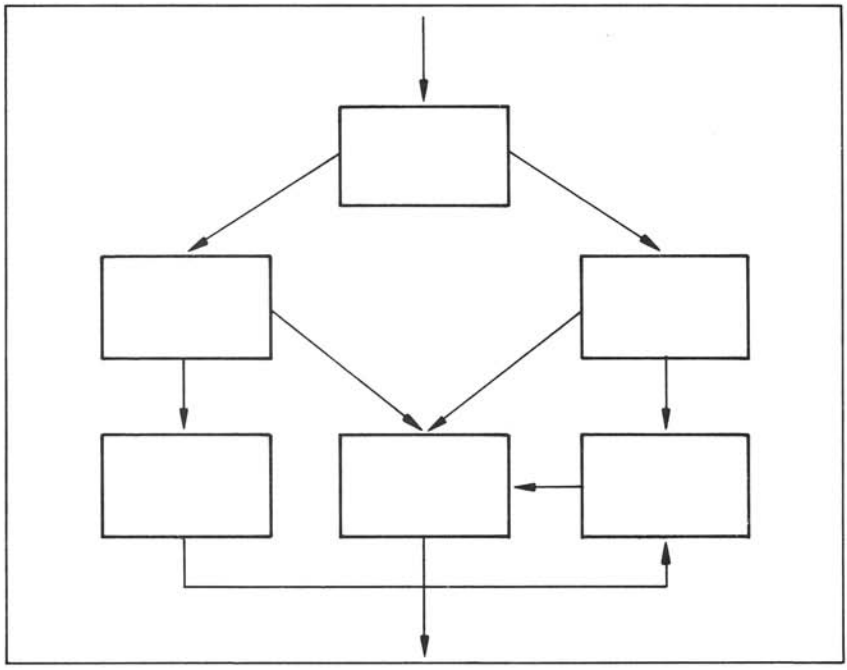


Fig. 4.47

Cette façon naturelle de programmer est directement compatible avec l'architecture des ordinateurs, et tire parti des instructions de saut, conditionnel ou non, implémentées dans la machine. Le langage d'assemblage et les langages évolués anciens tels que le FORTRAN [65] et le BASIC [66] permettent avec leur JUMP et GOTO de programmer une structure aussi imbriquée que l'on veut. Une très grande efficacité peut en résulter, mais le programmeur même doué ne parvient pas à imaginer lors de chacun des sauts quel est l'état exact de la machine en fonction des sauts précédents, donc à garantir un comportement correct de son programme.

La programmation structurée montre que quelques structures de contrôle, dont le GOTO est exclu, suffisent pour relier les modules définis au cours des étapes successives de l'analyse descendante. Ces structures sont définies dans la figure 4.48, avec leur représentation sous forme d'organigramme, de structogramme, d'instructions en langage évolué, d'instructions en langage d'assemblage et de langage naturel.

Aucun langage évolué ne contient exactement ces 8 structures notées de cette façon. PASCAL [67] reconnaît par exemple (a) (b) (c) (d) (g) (annexe 7.3.3).

La figure 4.48, met en évidence cinq représentations équivalentes, adaptées chacune à des besoins spécifiques.

Chaque programmeur a, en fonction de sa formation et de son expérience, une préférence marquée pour l'une ou pour l'autre de ces représentations. Il est bon de bien se familiariser à ces équivalences, et de remarquer en particulier qu'il est aussi facile d'écrire de façon structurée en assembleur qu'en langage évolué.

4.7.2 Exemple: simulation d'une machine à écrire

Comme troisième exemple de programme, considérons la simulation d'une machine à écrire (imprimante, télétype) sur un écran lié à un processeur. L'écran montre une portion de mémoire et, pour simplifier le programme, les nouveaux caractères sont introduits au bas de l'écran, sur la dernière ligne. Le retour de chariot fait monter d'une ligne tout le texte, crée une ligne vide et place un pointeur (une étoile) montrant l'emplacement du prochain caractère tapé au début de la dernière ligne. Les caractères proviennent d'un clavier ou d'un autre système; le code ASCII est utilisé dans les deux cas, qui ne sont en fait pas différents.

D'une façon plus précise, le programme peut s'écrire

```
effacer tout l'écran
répéter    (écriture d'une ligne)
  initialiser le pointeur au début de la dernière ligne
  répéter (écriture d'un caractère)
    attendre le prochain caractère
    si c'est le caractère de retour de chariot CR, sortir de la boucle
               d'écriture d'un caractère
    considérer à nouveau le caractère
      si c'est le caractère de correction DELETE, effacer le caractère
               précédent si c'est possible
      si ... (autres cas éventuels)
      pour tous les autres caractères, afficher le caractère à l'emplacement
               du pointeur, et déplacer le pointeur d'une position
    fin des cas possibles
    si le pointeur a avancé en dehors de l'écran, sortir de la boucle d'écriture
               d'un caractère
  fin de la boucle, retourner attendre le caractère suivant
  déplacer le contenu de l'écran vers le haut, effacer la dernière ligne
fin de la boucle de répétition, retourner initialiser le pointeur et lire une ligne
```

(4.35)

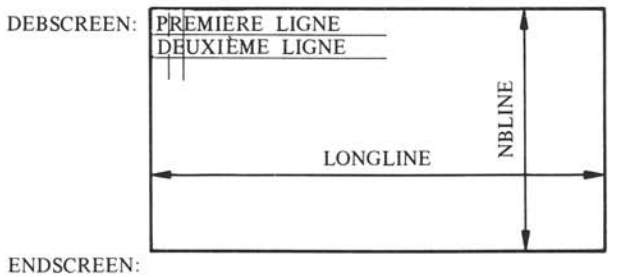
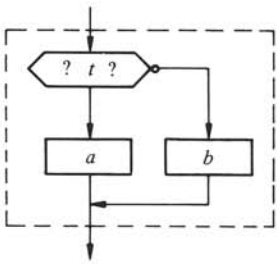
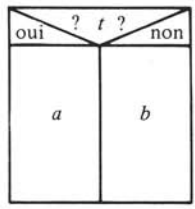


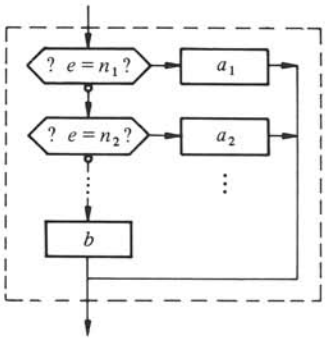
Fig. 4.49

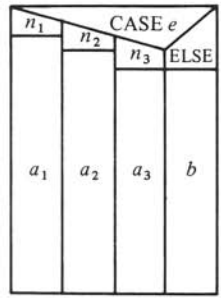





```
IF t                    JUMP,t x
  THEN DO a               b
  ELSE DO b             JUMP y
ENDIF                x:   a
                     y:
```

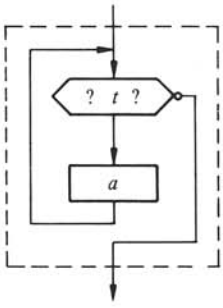
(a) si la condition booléenne t est vraie, faire a, autrement faire b

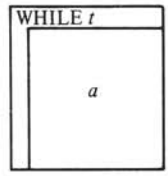





```
CASE e                  COMP e,#n1
  OF n1  DO a1          JUMP,EQ x1
  OF n2  DO a2          COMP e,#n2
      :                 JUMP,EQ x2
      :                     :
  ELSE  DO b                :
ENDCASE                     b
                     y:
                     x:   a1
                          JUMP y
                     x:   a2
                          JUMP y
```

(b) si e vaut n_1, faire a_1, si e vaut n_2, faire a_2, autrement faire b

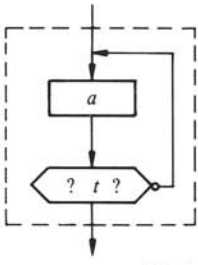


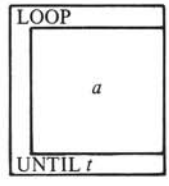



```
WHILE t              x: JUMP,t̄ y
  DO a                  a
ENDWHILE                JUMP x
                     y:

  t̄ signifie t non vrai
```

(c) tant que la condition t est vraie, faire a

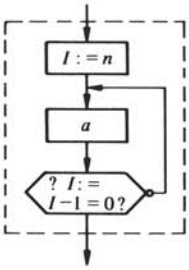


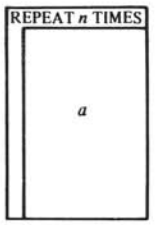



```
REPEAT               x:   a
  DO a                    JUMP,t̄ x
UNTIL t
```

(d) faire a aussi longtemps que la condition t n'est pas vraie

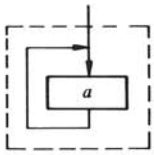
Fig. 4.48

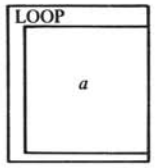





```
REPEAT n TIMES          LOAD I,#n
  DO a                x:   a
ENDREPEAT               DECJ.NE I,x
```

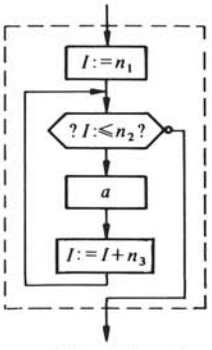
(e) répéter n fois l'opération a






```
LOOP                    x:   a
  DO a                      JUMP x
ENDLOOP
```

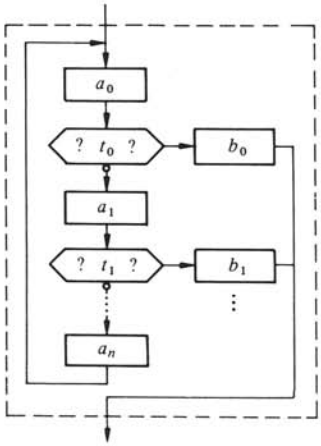
(f) répéter a sans cesse

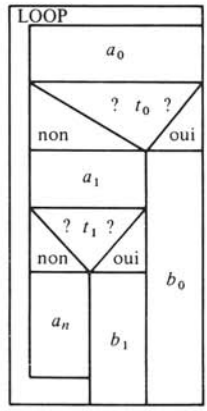



```
FOR I=n1 TO n2 STEP n3
  DO  a
ENDFOR

                        LOAD I,#n1
                    x:  COMP I,#n2
                        JUMP,HI y
                          a
                        ADD I,#n3
                        JUMP x
                    y:
```

(g) répéter l'opération a avec tout d'abord $I = n_1$, puis I diminué de n_3, etc. tant que I est inférieur ou égal à n_2






```
LOOP                    x:   a0
  DO a0                      JUMP,t0 y0
  ON t0 DO b0 EXIT           a1
  DO a1                      JUMP,t1 y1
  ON t1 DO b1 EXIT            :
     :                        an
     :                     JUMP x
  DO an                 y0: b0
ENDLOOP                     JUMP z
                        y1: b1
                            JUMP z
                            ...
                        z:
```

(h) faire l'opération a_0 si la condition t_0 est vraie, faire b_0 et c'est tout
⋮
faire a_n et recommencer

La figure 4.49 précise la signification de quelques-uns des termes définis dans le programme (4.36). Souvent ces termes sont inspirés de l'anglais pour permettre des symboles plus courts.

Notons que le programme (4.36) n'est pas écrit dans un langage évolué existant; sa syntaxe est inspirée du Pascal, mais n'en respecte pas les formes précises.

```
constants DEBSCREEN=H'4000,  LONGLINE=64., NBLINE=16.
          ENDSCREEN=DEBSCREEN+NBLINE*LONGLINE
          DELETE=O'177, CR=O'15,  SPACE=O'40,  STAR='*

declare   CHARACTER:8 bits  POINTER:16 bits

program  (*TTY simulator*)
  do CLEARSCREEN
  loop  (*write a line*)
    do POINTER:=ENDSCREEN-LONGLINE
      loop  (*get a character in a line*)
        do GETCAR (CHARACTER)
        if (CHARACTER=CR) or (POINTER=ENDSCREEN)
          then do EXIT
          else do
            case CHARACTER                                    (4.36)
              of DELETE  do  ERASE
              of ...
            endcase
        endif
      endloop
    do MOVUPSCREEN
    do CLEARLASTLINE
  endloop
endprogram

procedure ERASE  (*efface le dernier caractère*)
  if POINTER-ENDSCREEN<>ENDSCREEN-LONGLINE
    then do
      {POINTER}:-SPACE, POINTER:=POINTER-1, {POINTER}:=STAR
  endif
endprocedure

procedure CLEARSCREEN
.....
```

Le programme (4.36) n'est pas complet: certaines procédures simples n'ont pas été détaillées. Elles pourraient du reste être directement détaillées dans le programme principal, comme cela a été fait pour l'écriture du caractère sur l'écran et le déplacement du pointeur. La procédure GETCAR dépend de l'interface avec le clavier ou la ligne de transmission.

On remarque que les identificateurs ont été soigneusement choisis, de façon à faciliter la compréhension. Tout programme doit être tôt ou tard relu, ne serait-ce que par son créateur; la clarté de l'écriture est un investissement rentable.

Le programme (4.36) admet une représentation équivalente sous la forme du structogramme de la figure 4.50. La représentation sous forme de structogramme peut faciliter la compréhension et accélérer la mise au point du programme.

L'analyse du programme est maintenant suffisamment poussée pour permettre le codage en assembleur, ou dans un autre langage. Le programme en assembleur est donné dans la figure 4.51 et ne doit pas nécessiter de commentaire supplémentaire, autre que les

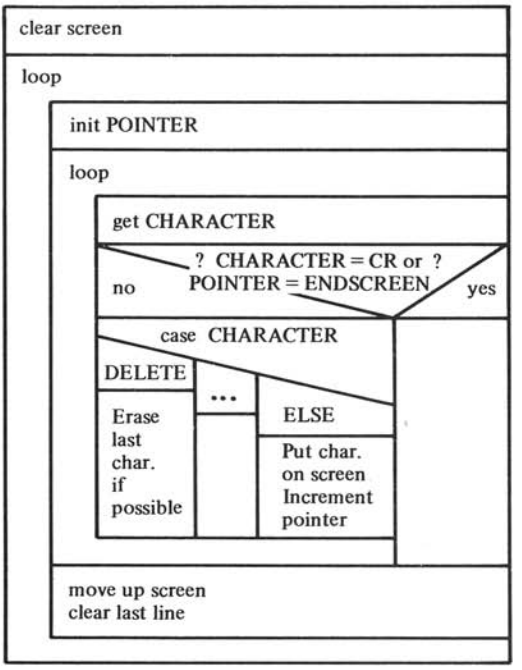


Fig. 4.50

descriptions (4.35) et/ou (4.36) qui peuvent être éventuellement incorporées dans le même fichier d'ordinateur sous forme de commentaire. L'expérience montre que toute information non intimement associée au programme traduit par l'ordinateur est rapidement perdue; le programme reste et évolue, les organigrammes associés n'évoluent généralement pas simultanément, car ils ne peuvent en général pas être modifiés sur le même terminal d'édition étant donné leur structure graphique.

4.8 ACCÈS DANS UN TABLEAU

4.8.1 Table de conversion

Les données numériques sur lesquelles un programme doit travailler sont généralement groupées sous forme de *tableaux* de nombres, caractérisés par une adresse de début et une longueur. A chaque nombre peut correspondre une ou plusieurs positions mémoire et la notion de nombre doit être entendue dans le sens large de groupements de bits pouvant avoir des significations très variées. Les caractéristiques d'un tableau doivent être *statiques*, c'est-à- dire définies au moment de l'écriture du programme avec une adresse et une longueur fixes, ou *dynamiques*. Dans ce cas, les paramètres caractérisant le tableau sont des variables qui peuvent être modifiées en cours d'exécution du programme.

Les tableaux sont fréquemment utilisés pour des *tables de conversion* (*look-up tables*) faisant correspondre à une valeur d'entrée une valeur de sortie ayant un format et une précision donnés. Pour accéder à la table, la valeur d'entrée doit être convertie en un nombre entier représentant le numéro d'ordre de la valeur de sortie correspondante.

```
.TITLE  EX451   ;Programme de simulation de télétype
.PROC   Z80
.REF    SM6     ;Définit ?GETCAR

;Constantes

DEBSCREEN= H'4000       ;Adresse de début d'écran
LONGLINE = 64.          ;Nombre de caractères par ligne (<256.)
NBLINE   = 16.
LONGSCREEN= NBLINE*LONGLINE
ENDSCREEN= DEBSCREEN+LONGSCREEN

DELETE  = O'177
CR      = O'15          ;Retour de chariot
SPACE   = O'40
STAR    = '*

.LOC    H'8800          ;Adresse du début de programme

TTY:
        LOAD    HL,#DEBSCREEN           ;Effacement de l'écran (CLEARSCREEN)
        LOAD    BC,#ENDSCREEN-DEBSCREEN
2$:     LOAD    {HL},#SPACE
        INC     HL
        DEC     BC
        LOAD    A,B
        OR      A,C
        JUMP,NE 2$

; L'insertion se fait dans la dernière ligne

4$:     LOAD    HL,#ENDSCREEN-LONGLINE
        LOAD    {HL},#STAR

;Insertion d'un caractère

6$:     CALL    ?GETCAR         ; Lecture clavier (routine système)
        COMP    A,#CR
        JUMP,EQ 10$

        PUSH    AF
        LOAD    A,H
        COMP    A,#ENDSCREEN/H'100      ;COMP   HL,#ENDSCREEN
        JUMP,NE 7$
        LOAD    A,L
        COMP    A,#ENDSCREEN.AND.H'FF
        JUMP,NE 7$
        POP     AF
        JUMP    10$                     ;EXIT if =
7$:     POP     AF

        COMP    A,#DELETE
        JUMP,EQ 8$
        ;...  test d'autres touches spéciales (TAB, BELL, etc)

        LOAD    {HL},A
        INC     HL
        LOAD    {HL},#STAR
        JUMP    6$

8$:     CALL    DODEL
        JUMP    6$

;Décalage du contenu de l'écran vers le haut (MOVUPSCREEN)

10$:    LOAD    HL,#DEBSCREEN+LONGLINE          ;source
        LOAD    DE,#DEBSCREEN                   ;destination
        LOAD    BC,#LONGSCREEN-LONGLINE         ;longueur
        LDIR                                    ;Move bloc

;Effacement de la dernière ligne (CLEARLASTLINE)

        LOAD    HL,#ENDSCREEN-LONGLINE
        LOAD    B,#LONGLINE
12$:    LOAD    {HL},#SPACE
        INC     HL
        DECJ,NE B,12$
        JUMP    4$

;Routine d'effacement du dernier caractère
; avec test si le début de la ligne est atteint  (ERASE)

DODEL:  PUSH    AF
        LOAD    A,H
        COMP    A,#(ENDSCREEN-LONGLINE}/H'100
        JUMP,NE 2$
        LOAD    A,L
        COMP    A,#(ENDSCREEN-LONGLINE).AND.H'FF
        JUMP,EQ 4$
2$:     LOAD    {HL},#SPACE
        DEC     HL
        LOAD    {HL},#STAR
4$:     POP     AF
        RET

.END
```

Fig. 4.51

Il est alors possible de calculer l'adresse physique de cette valeur en mémoire, de l'extraire et d'effectuer une conversion de format éventuelle (masquage, décompactification) avant de fournir le résultat. La figure 4.52 résume ce processus de conversion.

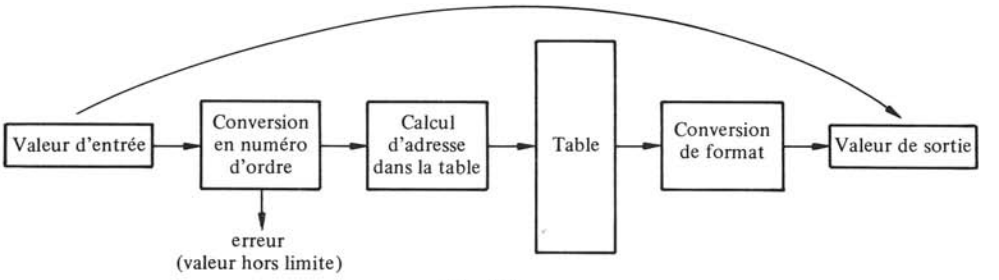


Fig. 4.52

Dans le cas simple d'une table de sinus par exemple, la table peut contenir les valeurs de 0 à 90°, sous forme de 91. points d'entrée. Les valeurs de sortie sont par exemple des nombres binaires purement fractionnaires de 16 bits (précision $2^{-16} \cong 10^{-5}$), occupant deux mots de 8 bits successifs.

La conversion d'une valeur angulaire entre 0° et 180° revient à chercher l'angle de 0° à 90° correspondant, puis à doubler la valeur pour obtenir la distance en octets depuis le début de la table, et ajouter cette valeur à l'adresse de début de la table.

La lecture de la mémoire à l'adresse calculée fournit la valeur cherchée.

4.8.2 Adressage dans un tableau

Continuons à nous appuyer sur l'exemple précédent et supposons que la table de sinus commence à l'adresse ADTABLESIN et contient dans l'ordre les sinus des angles 0, 1, 2, ... 90°, avec deux octets pour chaque valeur. La longueur de la table est donc de 2 * 91 octets.

Une forme type de cette table, contenant les sinus, multipliés par 1024. (précision 10 bits) est:

```
ADTABLESIN:
        .16     0       18.     36.     54.     71.     ; 0°-4°
        .16     89.     107.    125.    143.    160.    ; 5°-9°
        :                                                                (4.37)
        .16     1020.   1021.   1022.   1023.   1024.   ; 85°-89°
        .16     1024.                                   ; 90°
```

Si l'assembleur connaît la fonction sinus, ce qui est rare, ces valeurs peuvent être générées par une macroinstruction, plutôt que d'être péniblement calculées à la main.

A partir de cette table, le programmeur qui a besoin de la valeur d'un sinus particulier, par exemple de 45°, s'écrira dans son programme:

```
LOAD.16     D4, ADTABLESIN + 2 * 45                                  (4.38)
```

La multiplication par 2 correspond au fait que chaque sinus occupe deux octets, et que les adresses mémoire sont des adresses d'octets, avec le processeur choisi (M68000 ou

Z80). L'opération de calcul d'adresse est effectuée par l'assembleur qui peut évaluer l'expression et coder l'instruction.

En général, l'angle est dépendant de l'application et est une variable d'exécution. Si le registre D0 contient cet angle, le programmeur aimerait pouvoir écrire

```
LOAD.16     D4, ADTABLESIN + 2 * {D0}          (4.39)
```

La multiplication par 2 doit maintenant être effectuée à l'exécution. Peu de processeurs actuels permettent ce mode d'adressage (§ 3.5.14) et il faut remplacer cette instruction par

```
SL.32       D0
LOAD.16     D4, ADTABLESIN + {D0}              (4.40)
```

L'instruction de décalage à gauche SL (§ 2.5.7) multiplie par 2. L'instruction ADD.32 DO, DO ou l'instruction MUL.32 D0, # 2 peuvent être utilisées à la place, mais sont moins efficaces.

Notons encore que les calculs d'adresse se font ici sur 32 bits. Il arrive que le processeur ajoute à l'adresse de la table (32 bits) le contenu d'un registre D0 de 16 bits seulement, avec extension de D0 par des zéros ou par son signe (§ 3.5.14).

L'adresse de début de la table est en général une variable, mémorisée dans un registre d'index, par exemple A0. Ce registre est initialisé par l'instruction

```
LOAD.32     A0, # ADTABLESIN                   (4.41)
```

et les instructions (4.38) et (4.40) s'écrivent

```
LOAD.16     D4, {A0}+ 2 * 45.                  (4.42)
```

```
SL.32       D0
LOAD.16     D4, {A0} + {D0}                    (4.43)
```

Ces exemples ont permis de retrouver les modes d'adressage immédiat, direct et indexé (sect. 3.5). Les instructions (4.40) à (4.43) sont compatibles avec le M68000, parfois avec des limitations dans le champ des opérandes.

Par contre avec le Z80, le calcul d'adresse doit se faire par des opérations arithmétiques séparées plutôt que par le mode d'adressage. Convertie dans des registres de Z80, l'opération (4.43) par exemple s'écrit:

```
SL          DE
LOAD        BC, {HL} + {DE}                    (4.44)
```

Elle doit se coder avec 6 instructions du Z80 et modifie le registre HL pointant le début de la table:

```
SL      E
RLC     D
ADD     HL, DE
LOAD    C, {HL}                                (4.45)
INC     HL
LOAD    B, {HL}
```

4.8.3 Accès à un générateur de caractères

Plusieurs imprimantes génèrent les caractères selon une matrice de points. Considérons que le cas où ces points sont imprimés par colonne. Une table de conversion fait correspondre à chaque caractère, dont le code ASCII est donné dans un registre, une suite de mots binaires correspondant aux colonnes du caractère. Un texte imprimé type est montré dans la figure 4.53. Chaque caractère est inscrit dans une matrice de 5 x 7 points, avec un espace suffisant de part et d'autre. Cet espace peut être mémorisé dans la table, pour faciliter la programmation. Par exemple, une matrice de 8 x 8 points est facile à adresser puisque le rang du caractère doit être multiplié par 8, c'est-à-dire décalé trois fois.

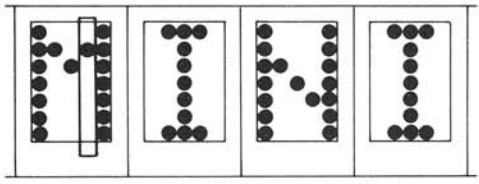


Fig. 4.53

Considérons le cas un peu plus délicat de l'écriture proportionnelle dans lequel la place accordée à chaque caractère est variable (fig. 4.54).

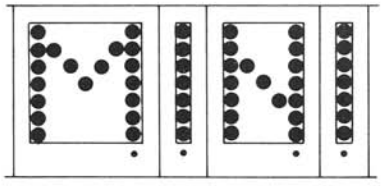


Fig. 4.54

Etant donné que, dans cet exemple simple, les caractères n'ont que 7 points de haut, utilisons le 8-ième bit libre comme terminateur de caractère (marqueur). Deux espaces sont ajoutés automatiquement entre deux caractères successifs.

A cause de cette taille variable des caractères, un calcul arithmétique simple ne peut pas être effectué pour trouver le n-ième caractère. Un parcours de la table doit être effectué, en comptant les occurences du marqueur placé sur le 8-ième bit dans la dernière colonne de chaque caractère. Le programme correspondant peut s'écrire:

```
pointer le début de la table de caractères
répéter n fois (recherche du n-ième caractère)
  répéter (avance jusqu'au marqueur de fin de caractère)
    lire un octet dans la table
    pointer la position suivante de la table
    si le marqueur est dans l'octet, sortir de la boucle intérieure
  fin répétition
fin répétition                                                    (4.46)
répéter (transfert du caractère vers l'imprimante)
  lire un octet dans la table
  pointer la position suivante de la table
  envoyer à l'imprimante les 7 bits significatifs de l'octet
  si le marqueur est dans l'octet, sortir de la boucle
fin répétition
```

Avec un processeur Z80, si le rang du caractère à imprimer est dans le registre A, le programme correspondant s'écrit:

```
SEARCH: LOAD    HL,=ADTABCAR
        OR      A,A             ; initialise l'indicateur de nullité
        JUMP,EQ FOUND           ; nécessaire si n = 0
2$:     TEST    {HL}:#7         ; marqueur atteint ?
        INC     HL              ; ne modifie pas l'indicateur EQ
        JUMP,EQ 2$              ; saut si le marqueur est = 0
        DEC     A               ; déompte à chaque occurence du marqueur
        JUMP,NE 2$                                                    (4.47)
FOUND:  LOAD    A,{HL}          ; lecture d'une colonne
        INC     HL
        CALL    PRINT           ; impression d'une colonne
        TEST    {HL}:#7
        JUMP,EQ FOUND
;... impression espace entre caractères, caractère suivant, etc.
```

La table de caractères doit tout naturellement s'écrire en binaire. Un changement de base permet un contrôle facile de la forme des lettres (fig. 4.55).

Avec l'algorithme de recherche utilisé, le temps d'accès à un caractère est relative-

```
.TITLE EX455          ;Table de génerateur de caractère

.RADIX    2

ADTABCAR:
LETAT:      .8        00111001
            .8        01000101
            .8        00111001
            .8        01000010
            .8        10111100

LETA:       .8        01111100
            .8        00010010
            .8        00010001
            .8        00010010
            .8        01111100

LETB:       .8        01111111
            .8        01001001
            .8        01001001
            .8        01001001
            .8        10110110

;...

LETI:       .8        11111111

LETJ:       .8        01000001
            .8        01000001
            .8        10111111

;...

.RADIX    10.

;....
.END
```

Fig. 4.55

ment long. S'il y a 64 caractères, il peut être évalué à 3 ms au maximum. Ceci peut limiter la vitesse à 300 caractères par seconde, mais une méthode d'accès différente permet d'augmenter notablement cette vitesse (§ 4.8.5).

4.8.4 Table d'adresse de table

Une application peut comporter de nombreuses tables, groupées par familles, une table peut contenir l'adresse de début de chacune des tables (fig. 4.56). La table peut également contenir d'autres informations concernant la longueur et la nature de la table.

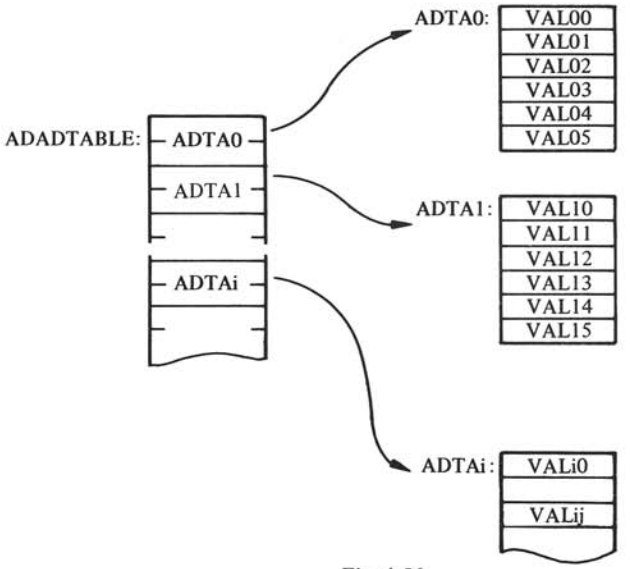


Fig. 4.56

Il faut tenir compte de la taille des éléments dans chaque tableau et choisir en fonction du processeur ce qui peut être calculé par un mode d'adressage prévu par le fabricant, et ce qui doit être calculé par des opérations arithmétiques sur les variables représentant les adresses.

Par exemple, il est équivalent d'écrire:

```
LOAD.32     A1, # {A0} + {D0}          (4.48)
```

ou

```
LOAD.32     A1, A0                     (4.49)
ADD.32      A1, D0
```

Dans le premier cas, le processeur dispose d'un mode d'adressage qui calcule la somme des contenus de A0 et D0 et transfère la valeur calculée dans A1. En toute rigueur, une parenthèse arithmétique devrait suivre le signe #, mais une règle de précédence évite cette longueur d'écriture.

L'écriture LOAD. 32 A1, A0 + D0 serait incorrecte puisqu'elle exprimerait que l'on transfère dans A1 le contenu d'un registre dont l'adresse est la somme des adresses des registres A0 et D0. Il n'est toutefois pas usuel de raisonner avec des adresses de regis-

tres; les registres forment un espace à part avec des noms réservés, contrairement aux noms de positions mémoire qui sont des symboles que l'on peut additionner : LOAD.32 A1, BASE + DEPL a un sens si BASE et DEPL sont des adresses mémoire.

Si un registre ou une position mémoire contient une adresse, l'accolade d'indirection met en évidence ce fait. Les écritures LOAD.32 A1, {A0} + DEPL et LOAD.32 A1, ADBASE + DEPL sont correctes, mais pas LOAD.32 A1,A0 + DEPL (adresses d'objets dans des espaces différents).

Le processeur Z80 dispose, en plus du sous-espace mémoire et de son espace de registres, d'un espace d'entrée-sortie caractérisé par le signe $. L'expression qui suit et qui exprime l'adresse est implicitement entre parenthèses. Les modes d'adressage LOAD A, $ {C} (adresse du périphérique dans le registre C) existent dans le Z80. Les modes LOAD A, $ {C} + DEPL ou LOAD A, $ {C} + {D} ont un sens, mais n'existent pas.

4.8.5 Exercice

Ecrire le bloc d'instructions permettant d'accéder à l'élément VAL ij (8 bits) de la figure 4.56, pour le processeur et les emplacements de paramètres suivants :

processeur	Z80	M68000
taille de l'espace d'adressage	16 bits	32 bits
taille des éléments des tables	8 bits	16 bits
nombre d'éléments par table	NBEL $< 2^8$	NBEL $< 2^{16}$
indice i	C	D3.16
indice j	E	D4.16

4.8.6 Exemple

Revenons à l'exemple du générateur de caractères proportionnels de la figure 4.54. Si la recherche séquentielle dans la table est trop lente, une table contenant les adresses de chaque début de caractère peut être construite, et la référence à un caractère quelconque peut se faire en passant par cette table d'adresses.

En assembleur cette table se forme simplement avec les étiquettes des débuts des portions de tables correspondant à chaque caractère (voir fig. 4.55)

```
SEARCH:   LOAD    HL,#ADTABCAR
          SL      A              ; double le déplacement (mots dans la table)
          LOAD    E,A
          LOAD    D,#0
          ADD     HL,DE          ; ajoute le déplacement
          LOAD    E,{HL}                                          (4.50)
          INC     HL
          LOAD    D,{HL}
          EX      HL,DE          ; place l'adresse lue dans HL
FOUND :           ...            ; comme dans (4.47)
```

L'exécution est beaucoup plus rapide et le temps d'accès à un caractère est constant. L'occupation mémoire est toutefois supérieure.

4.8.7 Exercice

Ecrire le programme Z80 qui génère la table d'adresse de caractères (§ 4.8.6) à partir de la lecture du générateur de caractères de la figure 4.55.

4.8.8 Routines système

Un ensemble de routines, appelées *routines système*, est mis à disposition de l'utilisateur d'un système, pour lui faciliter son travail. Elles lui permettent d'utiliser non pas les périphériques réels du système, avec toutes leurs contraintes technologiques, mais des périphériques virtuels idéaux, faciles à utiliser. Par exemple, une routine système fait apparaître l'imprimante comme un objet à qui l'on donne l'adresse du premier caractère à imprimer et la longueur du texte. Cette routine s'occupe d'attendre les quittances, faire des sauts de page au bon moment, etc.

L'intérêt principal de cette approche, en plus de la plus grande facilité pour l'utilisateur, est la plus grande flexibilité lors de changements. Si l'imprimante est remplacée par un modèle différent, la routine système doit être récrite, mais pas les programmes de l'utilisateur. L'appel des routines système peut se faire à leur adresse de début (fig. 4.57), mais celle-ci est susceptible de changer à chaque révision; on évite donc cette solution, qui obligerait l'utilisateur à modifier des adresses dans son programme.

Des adresses fixes contenant des instructions de saut aux différentes routines peuvent être utilisées (fig. 4.58). L'emplacement de cette table de sauts peut être figé en mémoire pour garantir à l'utilisateur la compatibilité de ses programmes.

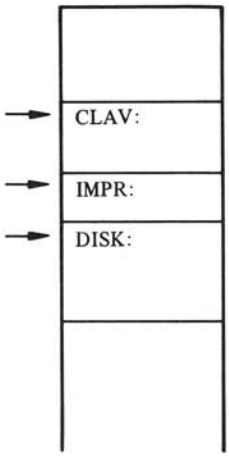


Fig. 4.57

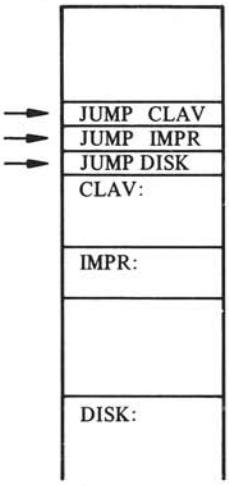


Fig. 4.58

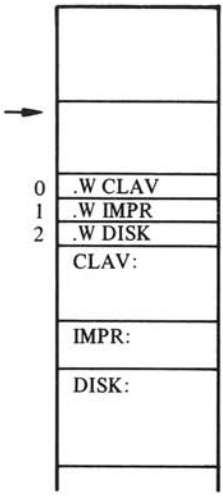


Fig. 4.59

Le plus flexible est de caractériser chaque routine par un numéro d'ordre (numéro de l'appel), et de transférer ce numéro comme paramètre lors de l'appel d'une routine système unique (fig. 4.59). Les opérations supplémentaires qui doivent être effectuées ralentissent l'exécution; pour des opérations simples et fréquentes (affichage de points sur l'écran par exemple), il faut parfois revenir à l'une des solutions précédentes.

4.8.9 Exercice

Ecrire pour le processeur Z80 ou le M68000 la routine permettant de sauter à une routine dont le numéro d'ordre se trouve dans l'octet suivant l'appel. Les adresses des routines sont dans une table ordonnée. Les routines de numéro 0, 1, 2, ..., 12 existent toutes. Si une routine de numéro d'ordre supérieur est appelée, le programme doit sauter dans un programme ERREUR.

4.8.10 Listes

Une liste est une suite chaînée de tableaux. Chaque tableau contient en plus des informations implicites ou explicites le caractérisant (longueur, nature du contenu), l'adresse du tableau suivant. Un indicateur spécial, par exemple l'adresse 0 qui est facile à tester, permet de savoir quand la liste est terminée.

La figure 4.60 montre une telle liste, formée de tableaux de longueur fixe et la figure 4.61 montre une liste bidirectionnelle, qui peut être parcourue dans les deux sens grâce à un double jeu de pointeurs.

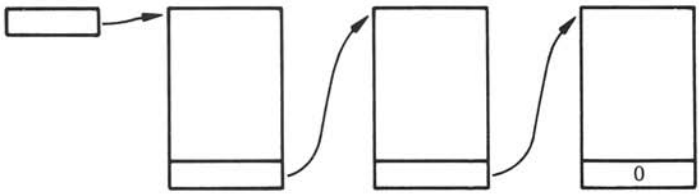


Fig. 4.60

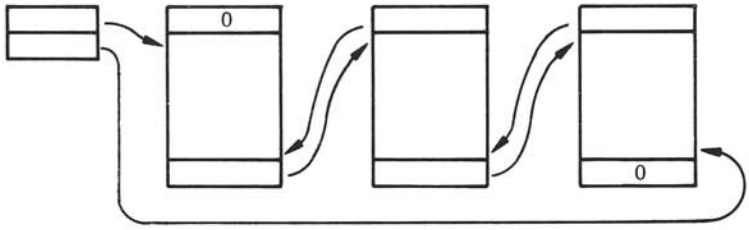


Fig. 4.61

Les listes permettent d'organiser des quantités importantes d'information de façon très variable. Par exemple, pour trier et ordonner des tableaux de nombres, il est plus efficace de modifier des pointeurs plutôt que de déplacer physiquement l'information elle-même.

4.8.11 Changement de contexte

Un programme en cours d'exécution, c'est-à-dire une tâche ou un processus, doit parfois être interrompu à un instant quelconque pour donner le contrôle à un autre processus. Le processus initial doit pouvoir être poursuivi ultérieurement sans perte d'information. Le contexte dans lequel le programme travaille, c'est-à-dire les états des registres, indicateurs et positions mémoire à l'instant de l'interruption, doivent être sauvés avant de donner le contrôle à l'autre processus. Plus exactement, toutes les variables

qui risquent d'être modifiées par le second processus doivent être sauvées, sur une pile ou dans une zone mémoire réservée pour ce but.

L'architecture et les instructions à disposition pour un processeur donné facilitent plus ou moins la programmation d'un tel *changement de contexte* (*context switching*). La nature et le nombre des processus qui peuvent être amenés à s'exécuter "simultanément" impliquent le recours à des techniques de programmation variées et multiples. Le prix très faible des microprocesseurs encourage l'utilisation d'un processeur par processus.

4.8.12 Exemple

Un cas relativement particulier de changement de contexte se trouve dans tous les systèmes microprocesseurs utilisés pour la mise au point de programmes. L'insertion d'un point d'arrêt dans le programme de l'utilisateur, ou l'apparition d'une condition particulière décodée par des circuits appropriés, interrompt le processus en cours et donne le contrôle à un *moniteur de mise au point*. L'état du processus interrompu ne doit pas être modifié d'une part pour pouvoir le continuer, et d'autre part pour pouvoir l'examiner à l'aide du programme moniteur.

Usuellement, l'état des registres, y compris les indicateurs et les pointeurs de pile, sont sauvés par le moniteur dans une zone mémoire réservée. Le moniteur se définit sa propre pile, pour éviter de charger la pile du programme testé, dont la longueur peut être insuffisante.

Le sauvetage sur une nouvelle pile y compris le sauvetage de l'ancien pointeur de pile nécessite une bonne compréhension du processeur. Par exemple, pour le 8085, l'instruction permettant de sauver le pointeur de pile SP en mémoire manque et le sauvetage de l'état doit se commencer par les instructions suivantes:

```
 LOAD   SAVHL,HL     ; sauvetage du registre HL
 POP    HL           ; récupération de l'adresse de retour au processus

 LOAD   SAVPC,HL     ;
*PUSH   AF           ; sauvetage des indicateurs, en particulier du Carry qui
                              peut être modifié par les instr. suivantes

*LOAD   HL,#2
*ADD    HL,SP                ; transfert dans HL la valeur SP corrigée du fait
                                  qu'un PUSH AF a été nécessaire

*POP    AF
 LOAD   SP,#NEWSTACK
*PUSH   HL ou LOAD  SAVSP,HL ; sauvetage de l'ancien pointeur de pile
                             ; sur la nouvelle pile ou en mémoire
 ...                         ; sauvetage des autres registres
```

(4.51)

Avec le Z80, les instructions marquées d'une * peuvent être remplacées par l'instruction unique: LOAD SAVSP, SP. Cette instruction supplémentaire facilite donc grandement la programmation.

4.8.13 Exercice

Ecrire le programme inverse du programme précédent, permettant de récupérer l'état des registres SP, HL et AF, et donc de continuer l'exécution à l'endroit où elle a été interrompue.

4.8.14 Programmes automodifiables

La flexibilité de l'architecture de von Neumann et des modules d'adressage permet d'écrire un programme qui se modifie lui-même. Une instruction peut être remplacée par une autre, ou seul l'un des paramètres d'une instruction peut être modifié.

Ces programmes ne peuvent naturellement pas s'exécuter en mémoire morte. Ils sont appelés *automodifiables* ou *impurs* et leur emploi n'est pas recommandé.

4.8.15 Rentrance

Une même routine dont le code ne se trouve qu'une seule fois en mémoire et qui peut être utilisée par plusieurs processus est appelée *rentrante*. Une telle routine ne doit utiliser que des registres et positions mémoire qui sont sauvés lors du changement de contexte. En général, aucune position mémoire fixe n'est sauvée lors du changement de contexte, mais la pile peut jouer un rôle important pour étendre un espace de registres insuffisant.

Les routines de gestion d'une mémoire silo vues dans la section 4.5 ne sont pas rentrantes, puisque le silo est ses pointeurs sont à des emplacements prédéfinis en mémoire. Par contre la routine de conversion décimal en binaire que l'on peut déduire du programme de la figure 4.45 l'est.

4.8.16 Récursivité

Une routine qui peut s'appeler elle-même est appelée *récursive*. Une routine récursive ne doit modifier aucun registre autre que ceux qui rendent les paramètres. La pile permet de sauver les paramètres temporaires et l'on peut être amené à définir plusieurs piles caractérisées par différents pointeurs initialisés en dehors de la routine récursive.

Comme exemple de routine récursive, considérons le calcul de la série arithmétique d'un nombre:

$$SA(n) = 1 + 2 + \ldots + (n-1) + n \tag{4.52}$$

que l'on peut calculer par récurrence

$$SA(n) = SA(n-1) + n \quad \text{avec } SA(0) = 0 \tag{4.53}$$

La routine correspondante s'écrit pour le Z80

```
;----\
; SA   >
;----/
;in       A paramètre n
;out      A valeur calculée SA(n-1)+n

SA:       COMP     A,#0
          RET,EQ                ;SA(0)=0
          PUSH     AF           ;sauve n
          DEC      A            ;n:=n-1
          CALL     SA           ;SA(n-1)
          PUSH     BC
          LOAD     B,A
          POP      AF
          ADD      A,B          ;n+SA(n-1)
          POP      BC
          RET
```
(4.54)

4.8.17 Exercice

Ecrire la routine récursive SA du paragraphe 4.8.16 pour le processeur M68000. Séparer la pile d'adresse de retour de la pile des valeurs n. Ajouter un test pour éviter le débordement des piles.

4.8.18 Coroutines

La notion de coroutine est définie dans quelques langages évolués pour permettre à deux tâches d'avancer alternativement, et de donner à l'utilisateur une impression de simultanéité des deux tâches.

Le programmeur prévoit explicitement les endroits de son programme où la coroutine est appelée et où la coroutine rend la commande au programme principal, en faisant attention à ce que la durée d'exécution entre deux transferts ne soit pas trop grande.

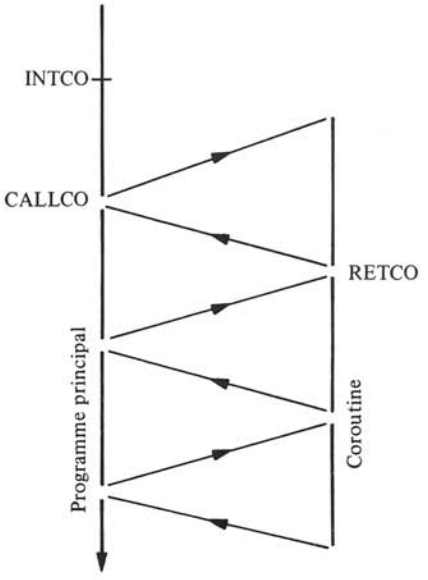


Fig. 4.62

La figure 4.62 illustre le principe des coroutines, qui sauvent leur état, avant de rendre la commande au programme principal.

Le retour de coroutine peut se programmer assez facilement en assembleur, surtout si le processeur dispose de modes d'adressage souples [14], mais aucun processeur actuel n'a de facilités au niveau de l'architecture matérielle pour sauver l'adresse de retour de la coroutine.

La notion de coroutine n'est qu'un pas vers la gestion de plusieurs tâches simultanées, qui sort du cadre de ce livre car elle recourt à des techniques plutôt logicielles [74].

CHAPITRE 5

INTERFACES ET PÉRIPHÉRIQUES

5.1 TRANSFERTS D'INFORMATION

5.1.1 Configuration type

Un système ordinateur comporte en plus du processeur proprement dit de la mémoire et des interfaces liées aux périphériques. Une configuration simple est donnée dans la figure 5.1, et montre que des câbles et un chemin de données appelé plus simplement bus (sect. 1.2) relient les unités et permettent les transferts d'information.

Ce chapitre étudie les modes de transfert sur les câbles et sur le bus et présente les types d'interactions entre le processeur et les périphériques.

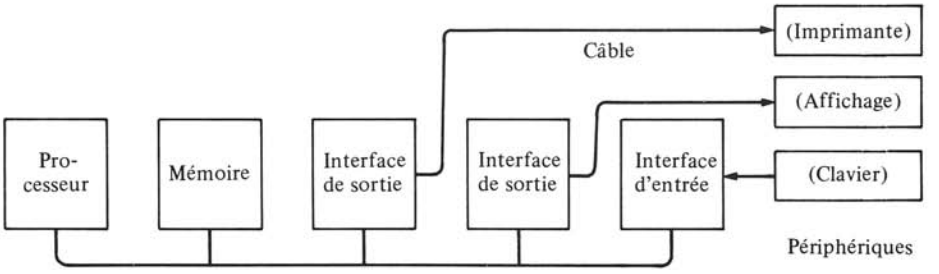


Fig. 5.1

Dans tout transfert d'information, il faut distinguer la *source* et la *destination.* Les termes *d'émetteur* (*talker*) et de *récepteur* (*listener*) sont aussi souvent utilisés.

Il faut également distinguer un *commandant*, qui prend l'initiative du transfert en agissant sur des lignes du chemin de commande qui lui sont réservées, et un *répondant* qui observe ces lignes et obéit de façon inconditionnelle.

Certaines unités du système sont susceptibles de jouer un rôle de commandant à un instant donné; on les appelle *maître* (*master*). Les unités qui ne peuvent que jouer le rôle de répondant sont des *esclaves* (*slave*).

Dans le système de la figure 5.1, le processeur est un maître et commande le système pendant l'exécution du programme. Une interface peut très bien être maître, mais alors sa structure est plus complexe (sect. 5.4).

Les transferts entre processeur, mémoire et interface se font de façon bidirectionnelle sur le bus, sous contrôle du commandant. Les transferts entre interfaces et périphériques sont généralement unidirectionnels. Selon les cas, l'interface (sur ordre du processeur) ou le périphérique prennent l'intiative des transferts et jouent donc un rôle de commandant.

Les transferts par câble et par bus présentent de nombreux points communs, et la suite de ce chapitre présente les différents types de transferts en partant des cas les plus simples. Les cas des transferts unidirectionnels, bidirectionnels et avec sélection du destinataire seront succesivement étudiés.

5.1.2 Transferts unidirectionnels

Le transfert unidirectionnel entre une unité source et une unité destination implique un câble pour le transfert de l'information et des lignes de commande du transfert (fig. 5.2). L'information est transférée le plus souvent par mots de 8 ou 16 bits à la fois. Le cas du transfert série sera étudié dans la section 5.5.

Parfois, par exemple pour commander les lampes d'affichage ou des électro-aimants, il suffit de transmettre l'information sans se préoccuper de ce qui se passe lorsque l'information apparaît ou disparaît (état de transition). En général, un signal de commande doit indiquer à l'unité destination qu'une nouvelle *information* ou *donnée* (*data*) a été placée sur le câble et que l'action correspondante peut être entreprise. Ce signal, par exemple, démarre l'impression d'un caractère sur une machine à écrire.

Un signal en retour doit indiquer quand l'action précédemment déclenchée est terminée, c'est-à-dire quand une nouvelle action peut être entreprise.

Cet échange de signaux de commande entre la source et la destination est appelé *synchronisation réciproque des échanges* (*handshaking*) et peut prendre des formes plus ou moins complexes.

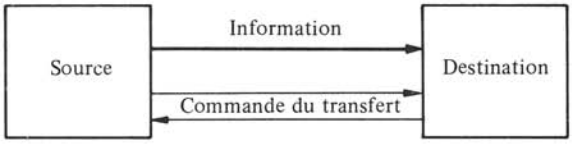


Fig. 5.2

Dans le cas où la source prend l'initiative du transfert, la procédure la plus simple est donnée dans la figure 5.3. Lors du premier transfert, la source doit supposer l'existence et la disponibilité de l'unité destination. La figure 5.4 correspond au cas inverse où l'unité destination prend l'initiative en demandant l'information à la source.

En général, l' "acceptation d'un envoi" est équivalente à la "demande de l'information suivante", et les deux procédures, une fois lancées, sont très similaires, donc souvent

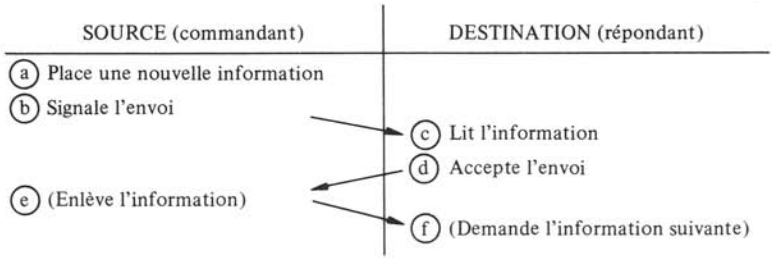


Fig. 5.3

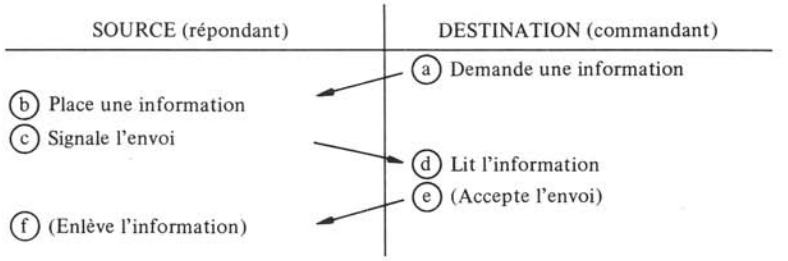


Fig. 5.4

identifiées. L'étape d'"enlèvement de l'information" est inutile dans un transfert unidirectionnel; le placement d'une nouvelle information détruit l'ancienne.

5.1.3 Signaux type

Limitons-nous à l'étude plus détaillée des transferts dont l'intiative est prise par la source. Une représentation sous forme de diagramme des temps permet de mettre en évidence la séquence détaillée et les conditions de bon fonctionnement (fig. 5.5).

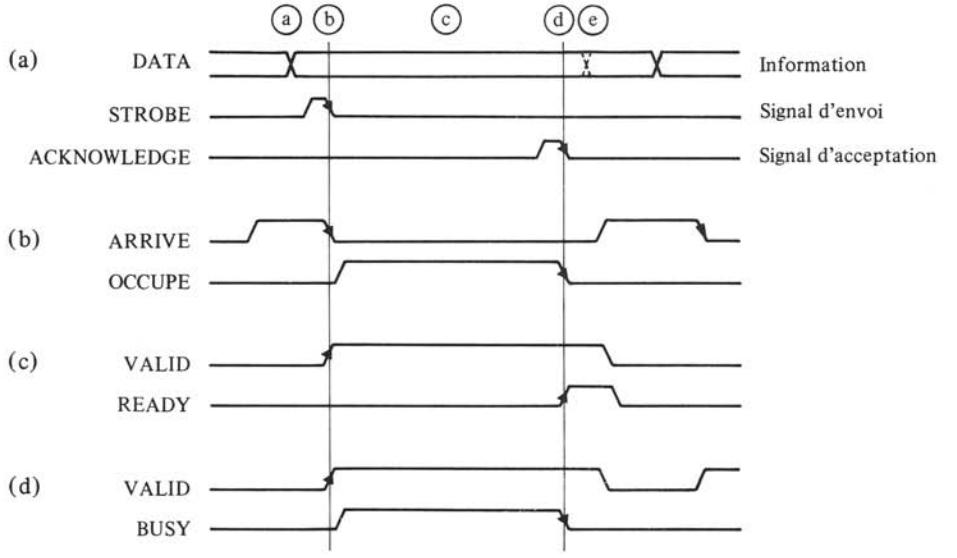


Fig. 5.5

Les deux signaux de synchronisation des échanges peuvent être des impulsions (STROBE, ACKNOWLEDGE, fig. 5.5 (a)) dont le flanc descendant caractérise l'instant de l'action. Les impulsions brèves n'étant pas favorables, une paire de signaux dont les flancs sont actifs est préférable (fig. 5.5 (b)). Par exemple, le signal ARRIVE est actif pendant la période transitoire lorsque l'information change. Le signal OCCUPE est actif pendant que l'unité destination traite l'information.

On rencontre souvent d'autres paires de signaux, telles que VALID, READY ou VALID, BUSY (fig. 5.5 (c) et (d)). Le flanc actif de ces signaux (points ⓑ et ⓓ a tou jours la même signification, mais le second flanc et l'état peuvent avoir en plus une signification propre. Par exemple, le flanc montant du signal OCCUPE signale que l'information a été reconnue. L'état OCCUPE = 1 indique que le récepteur est en train de traiter l'information (en supposant que OCCUPE passe instantanément à 1 en réponse au signal ARRIV et l'état OCCUPE = 0 indique qu'il peut recevoir une nouvelle information.

Il est évident que dans les protocoles de la figure 5.5, les niveaux logiques des signaux de synchronisation et les données peuvent être inversés lors de la transmission. On raisonnera ici en termes de fonctions réalisées, en évoquant occasionnellement les contraintes des implémentations technologiques.

Par exemple, dans le cas d'une transmission avec impulsion d'envoi (STROBE) et impulsion d'acceptation (ACK), comme décrit dans la figure 5.5 (a), il faut spécifier le temps pendant lequel l'information est stable de part et d'autre de l'impulsion, et la lon gueur de l'impulsion.

Pour garantir le bon fonctionnement, il suffit de définir un ensemble complet de valeurs temporelles minimales. Dans le développement d'interfaces, des marges de sécurité importantes sont souvent prises pour tenir compte de l'imperfection de réalisation, des parasites éventuels et permettre une interface plus simple.

Les spécifications temporelles de la norme CENTRONICS utilisée avec des imprimantes sont données dans la figure 5.6 [80]. On remarque que les signaux sont inversés que les temps ont été arrondis à 1 microseconde et que des temps qui sont par natu positifs n'ont pas été spécifiés. Le signal BUSY est redondant et n'est généré que lorsque l'attente est longue.

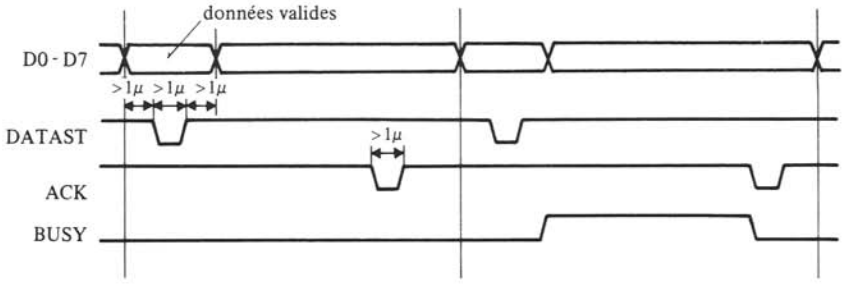


Fig. 5.6

5.1.4 Transferts bidirectionnels

Une ligne supplémentaire, caractérisant le sens de la transmission, doit exister lorsque le transfert est bidirectionnel (fig. 5.7). Ce sens est défini par le commandant qui prend l'initiative de tous les transferts.

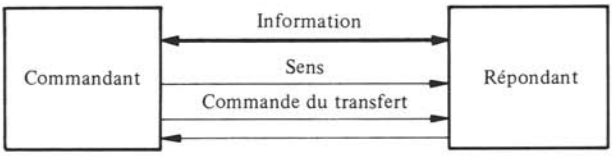


Fig. 5.7

L'utilisation des lignes d'informations dans les deux sens implique le recours à des circuits à trois états (§ 1.2.4) ou à collecteur ouvert.

On distingue deux types de transferts bidirectionnels. Les *transferts asynchrones* utilisent des lignes de synchronisation des échanges similaires à celles rencontrées dans les transferts unidirectionnels. Les *transferts synchrones* supposent au contraire que le répondant peut suivre le rythme imposé par le commandant.

5.1.5 Transferts asynchrones

La figure 5.8 détaille les signaux nécessaires à un transfert asynchrone type. La ligne WRITE définit le sens de transfert: la notion de la lecture/écriture est toujours relative au commandant, qui dans un ordinateur est en général le processeur. La ligne WRITE se note souvent $W/\overline{R}$, pour montrer que lorsque WRITE = 0, la fonction réalisée est une lecture (READ = 1).

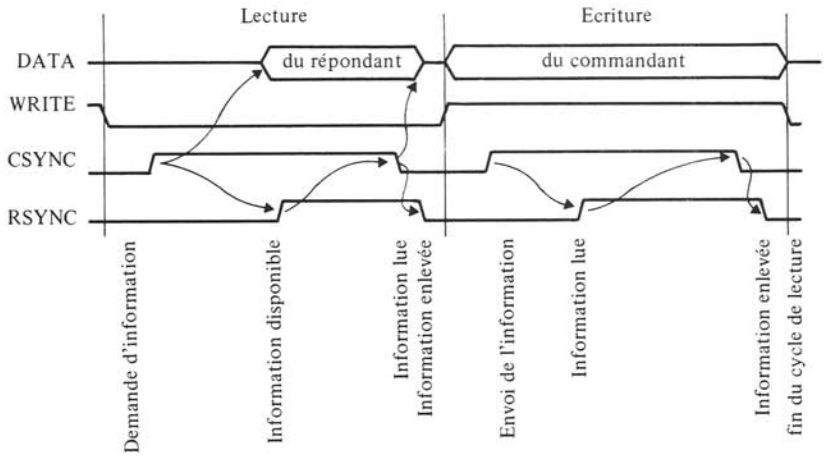


Fig. 5.8

Dans un cycle de lecture, le commandant n'impose aucun état sur les lignes d'information et signale la demande d'une information en activant un signal appelé ici CSYNC (Commander Synchronisation). L'état de cette ligne et de la ligne WRITE = 0 indique au répondant qu'il peut placer l'information. Lorsque c'est fait, le répondant active une ligne appelée RSYNC (Responder Synchronisation), le commandant peut alors transférer l'information et désactiver son signal CSYNC. Le répondant répond immédiatement en désactivant RSYNC.

Le déroulement d'un cycle d'écriture est similaire. En général, le commandant prend le contrôle des lignes d'information dès le début du cycle; le signal CSYNC avec WRITE = 1 indique que l'information est stable.

Il peut arriver que le commandant transfère de l'information à un répondant inexistant ou défectueux, qui n'émet pas de RSYNC. Pour éviter que la transmission ne se bloque dans une attente infinie, un *temps limite* (*timeout*) est accordé, après lequel une procédure d'erreur spéciale est déclenchée.

5.1.6 Transferts synchrones

Une durée fixe est assignée aux cycles de lecture et d'écriture dans un transfert synchrone (fig. 5.9). En lecture, un *temps d'accès* maximum est accordé au répondant pour qu'il fournisse l'information demandée; le commandant utilise le reste de la durée de sélection pour son transfert interne (*temps d'enregistrement* ou *set-up time*). Souvent, il est exigé que l'information du répondant ne disparaisse pas instantanément lorsque le signal de sélection est désactivé. Le *temps de maintien* (*hold-time*) qui est exigé est souvent inférieur à ce qui est réalisé naturellement par les retards des circuits: le respect de ces temps est donc automatique. En écriture, la durée de sélection est le temps accordé du répondant pour enregistrer l'information. Ces conditions sont symétriques du cas précédent et il suffit pour un bon fonctionnement que l'information arrive assez vite pour permettre son enregistrement. Dans la pratique, le commandant sait généralement dès le début du cycle quelle information il doit écrire et il la place avant d'activer la ligne de sélection avec un temps de maintien préalable qui peut être une condition de bon fonctionnement du répondant.

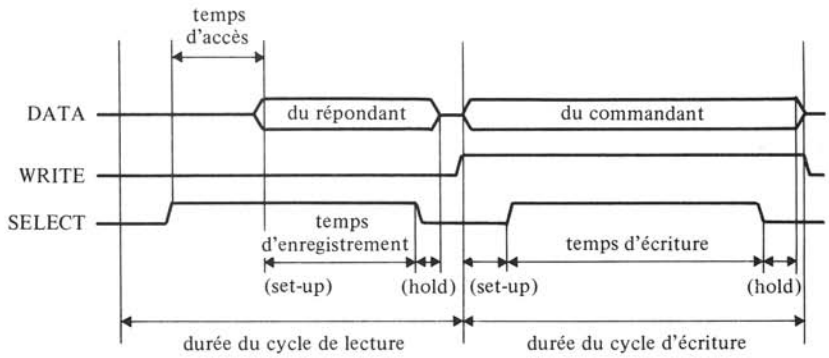


Fig. 5.9

Les transferts synchrones peuvent toujours être utilisés, mais la vitesse doit être fixée selon le répondant le plus lent, dans le cas où le commandant ne connaît pas à l'avance la vitesse du répondant avec lequel il va travailler.

5.1.7 Transferts semi-synchrones

Les transferts synchrones sont préférables à cause de leur simplicité. Usuellement les répondants (mémoires) sont plus rapides que le commandant (processeur). Si ce n'est pas le cas, le répondant doit signaler sur une ligne allant vers le commandant qu'il est prêt (READY). Il est donc équivalent de signaler qu'il n'est pas encore prêt (WAIT ou NOTREADY) et que la durée du cycle doit être prolongée (fig. 5.10).

Les signaux READY et NOTREADY n'ont de signification que pendant la durée de la sélection, voire à certains instants précis d'échantillonnage et sont dans ce sens inverses l'un de l'autre. L'intérêt du signal NOTREADY est que, par défaut, un transfert purement synchrone est effectué.

La figure 5.11 montre un exemple type dans lequel un cycle sur deux est ralenti. Le signal NOTREADY est échantillonné par le commandant au début de la période de sélection.

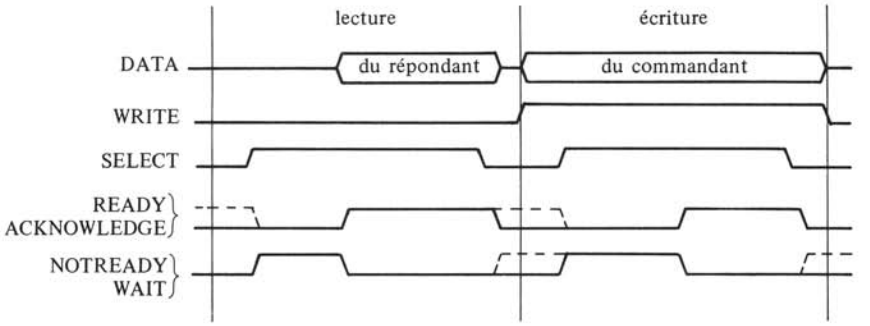


Fig. 5.10

Si NOTREADY est actif, la durée de sélection est prolongée d'un multiple de la période de base des opérations de transfert dans le commandant, c'est-à-dire de son horloge de synchronisation.

Cette solution est plus simple que la solution purement asynchrone par le fait que les processus se déroulent toujours selon une phase précise. L'efficacité est plus faible par le fait que les durées des cycles sont augmentées par quanta relativement importants.

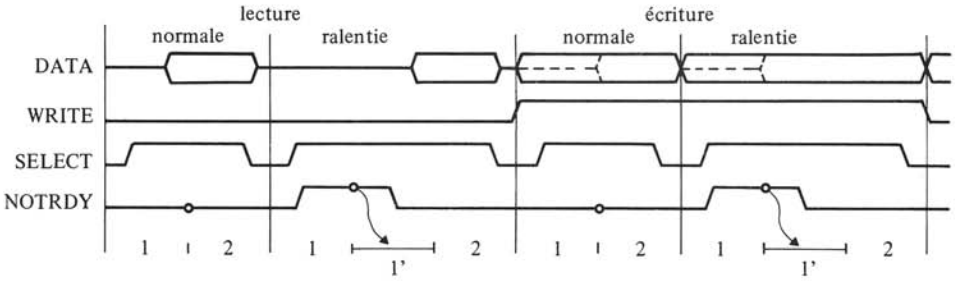


Fig. 5.11

5.1.8 Variante

Au lieu de donner sur une ligne le sens et sur une autre l'impulsion de transfert, plusieurs fabricants préfèrent considérer une impulsion de lecture et une impulsion d'écriture (fig. 5.12). Cette approche est plus dans l'optique de l'implémentation techno-

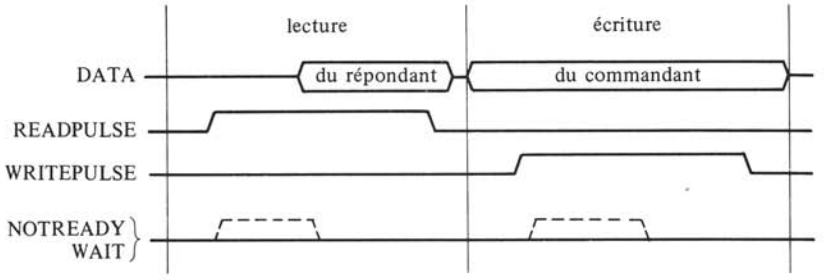


Fig. 5.12

logique, mais contient moins d'information sur le plan fonctionnel. L'utilisation d'un signal NOTREADY pour ralentir les transferts est possible comme dans le cas précédent.

La conversion des signaux WRITE et SELECT en READPULSE et WRITEPULSE est triviale (fig. 5.13). La conversion inverse peut sembler plus simple, mais pour retrouver toute l'information contenue dans les relations temporelles de WRITE et SELECT, des retards sont nécessaires (fig. 5.14) [80]. En effet, le signal WRITE est défini dans la figure 3.9 comme ayant un temps de préparation (set-up) positif. Le décodage simple donnerait un temps négatif.

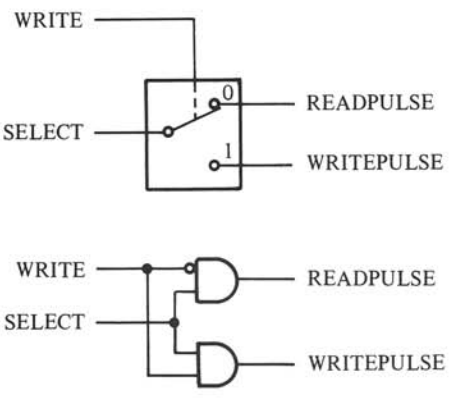


Fig. 5.13

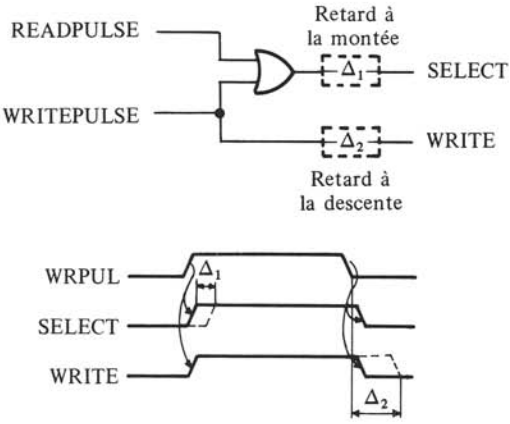


Fig. 5.14

5.1.9 Commentaires

Les systèmes de transfert précédemment décrits sont tous utilisés par différents fabricants. Digital Equipement, avec sa série célèbre de PDP11 [83], utilise une transmission asynchrone et des signaux de base correspondant à ceux de la figure 5.8. Les microprocesseurs 16 bits tendent à utiliser une solution du même type, synchronisée par l'horloge du processeur (cas Motorola 68000) [62]. Les microprocesseurs simples [53, 90] utilisent une transmission synchrone ou semi-synchrone. Motorola (microprocesseurs 6802, 6809) et d'autres fabricants ont un signal de lecture/écriture et une impulsion de sélection. Intel (8080, 8086), Zilog (Z80) et d'autres ont une impulsion de lecture et une impulsion d'écriture. Ces différences compliquent le mélange de composants provenant de divers fabricants. La définition d'un bus normalisé simplifie partiellement le problème en le reportant au niveau de la liaison de chaque circuit sur le bus [80].

5.1.10 Sélection d'un registre

Le transfert d'information entre un commandant et un registre (§ 1.2.8) est l'opération de base rencontrée dans toutes les interfaces de périphériques. La figure 5.15 en donne le schéma logique et les contraintes temporelles: l'impulsion effectuant le transfert parallèle dans le registre de sortie doit avoir son flanc actif lorsque l'information est stable (temps s et h à respecter).

En entrée (lecture), le registre doit comporter des sorties à trois états, dont le temps d'accès a est un paramètre important. Ces sorties à trois états sont activées pendant toute

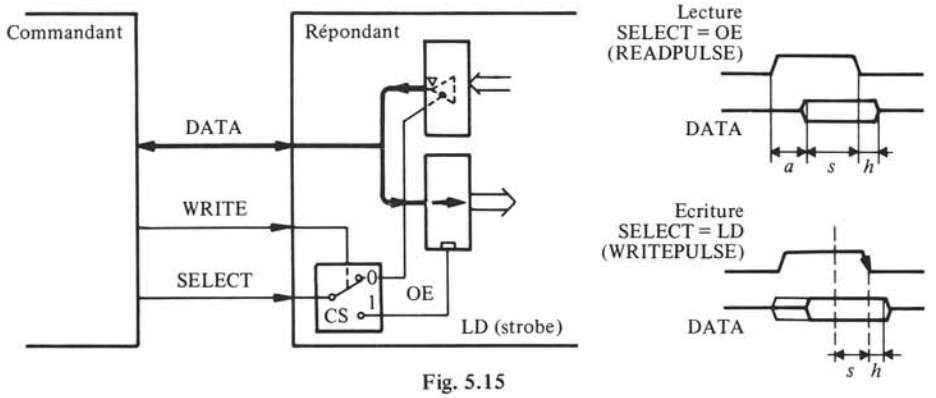


Fig. 5.15

la durée de sélection en lecture. La logique de sélection de l'impulsion de chargement (LD pour LOAD) du registre et de la ligne de commande des sorties à 3 états (OE pour output enable) est simple. Dans le cas d'un commandant générant les signaux READPULSE et WRITEPULSE, elle est inexistante, ces signaux pouvant être reliés directement aux registres d'entrée, respectivement de sortie.

5.1.11 Transferts par bus

Dans les cas réels, un commandant peut communiquer avec plusieurs esclaves chacun caractérisé par une adresse. Seul un esclave est sélectionné et répond à un instant donné. La figure 5.16 montre le schéma logique d'un système permettant la sélection de 4 registres d'entrée et 4 registres de sortie. Deux lignes d'adresse permettent de sélectionner le registre désiré, à l'aide d'un décodeur ou démultiplexeur à 8 positions (§ V.1.5.3). Quatre positions sont utilisées en lecture et quatre en écriture. Un seul décodeur pourrait être utilisé en lecture et quatre en écriture. Un seul décodeur pourrait être utilisé pour la sélection de tous les registres, mais des contraintes pratiques encouragent une décompostion en modules comportant un nombre restreint de registres et répondant chacun à une adresse différente. Pour une modularité parfaite, le décodeur est répété dans chaque unité esclave. Un comparateur d'adresse est parfois utilisé à la place du décodeur.

Les lignes d'information, d'adresse et de commande, relient chaque unité. Elles forment un bus dont les performances dépendent essentiellement du nombre de lignes d'information et d'adresse, et de la vitesse maximum des transferts.

La variante de transfert synchrone utilisant des signaux READPULSE et WRITEPULSE conduit à des différences mineures dans le décodage des signaux. La figure 5.17 donne le logigramme correspondant, qui peut être comparé avec celui de la figure 5.16.

5.1.12 Sélection des mémoires

Les mémoires, telles qu'elles ont été définies au paragraphe 1.2.7 sont fonctionnellement équivalentes à des ensembles de registres et se sélectionnent de façon analogue. La figure 5.18 montre comment connecter des modules mémoire de 2^n mots de k bits, en supposant que le bus d'adresse comporte $n + 2$ lignes. Quatre modules mémoire peuvent donc être sélectionnés et deux modules ont été associés dans l'exemple de la figure 5.18 pour former une carte mémoire connectée sur le bus.

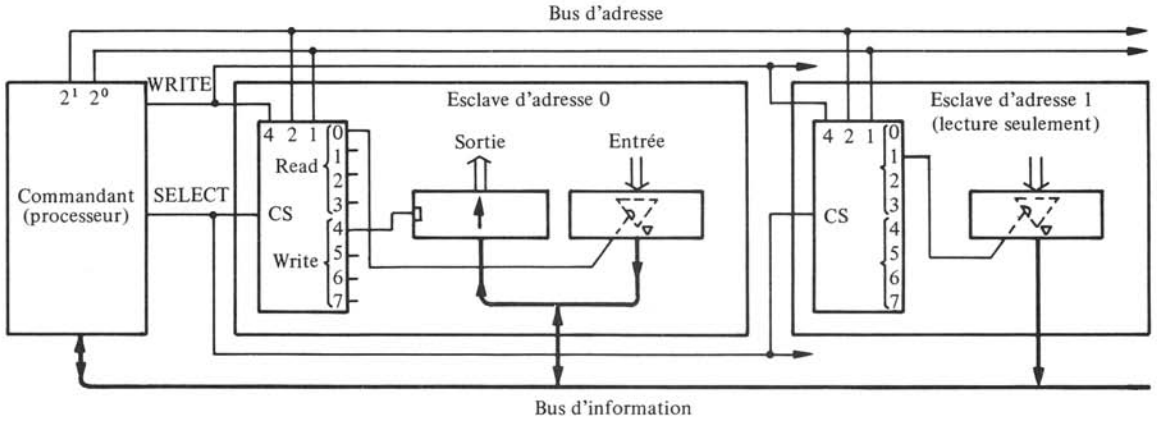


Fig. 5.16

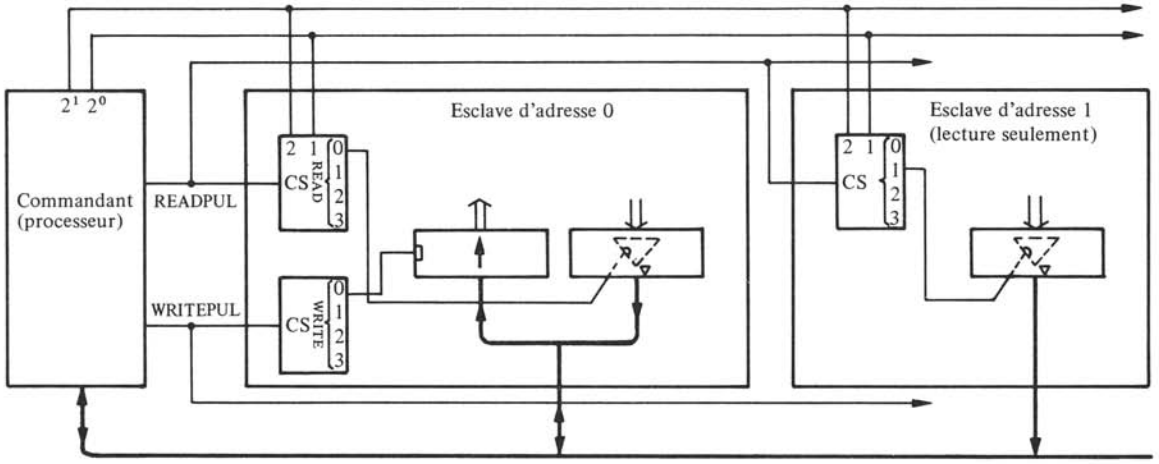


Fig. 5.17

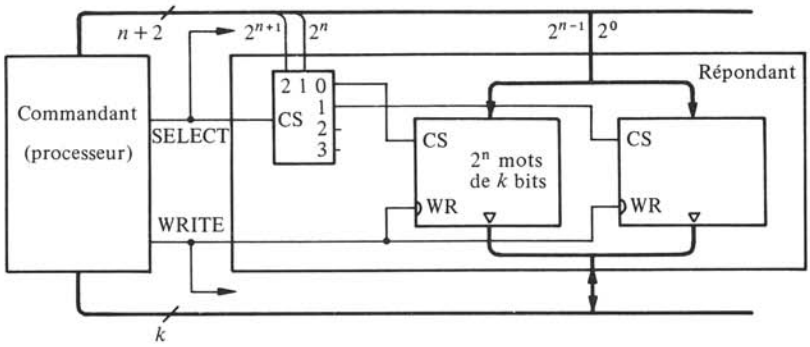


Fig. 5.18

L'utilisation de modules mémoire disposant d'une entrée de sélection CS et d'une entrée de lecture/écriture WR facilite la réalisation.

Dans le cas d'un commandant délivrant des signaux WRITEPULSE et READPULSE l'utilisation de modules mémoire ayant en plus une entrée directe de sélection en sortie est préférable. La figure 5.19 donne le schéma logique correspondant.

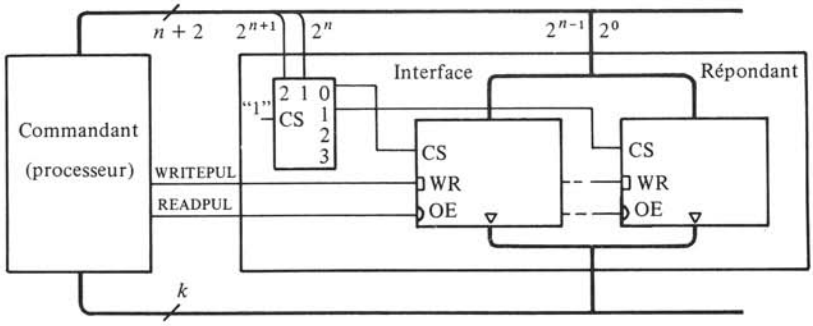


Fig. 5.19

5.1.13 Exercice

Etablir le schéma logique d'une carte mémoire utilisant les modules mémoire de la figure 5.18.

5.1.14 Remarque

Les schémas des précédents paragraphes, comme d'ailleurs presque tous ceux de ce livre, sont des schémas fonctionnels. La réalisation dans une technologie particulière implique usuellement l'adjonction de *muscleurs* (*buffers*) et d'inverseurs destinés à respecter les spécifications détaillées des circuits utilisés et à limiter la charge sur les bus. Pour plus de détails, on se référera aux notices des fabricants et aux références [23, 80]. Ces références décrivent aussi certains types de mémoire, par exemple les mémoires dites dynamiques dont les conditions d'utilisation sont plus complexes que ce qui a été vu précédemment.

5.1.15 Exemple d'application

Un système microprocesseur simple peut être réalisé avec 2048. mots de mémoire morte pour le programme (sous forme de deux circuits de 1 k octets), 4096. mots de mémoire vive (module avec entrées CS et WR) et deux registres de 8 bits en sortie. Ces deux registres sont sélectionnés comme des positions mémoire, mais un maximum de 16 registres prévus autorise un décodage incomplet. La figure 5.20 représente le schéma bloc du système et donne les adresses (en hexadécimal) choisies pour les zones mémoires et registres. Les adresses au-dessus de 4000 ne sont pas utilisées, et les adresses de H'300, à H'3FF sont toutes affectées aux registres d'entrée-sortie.

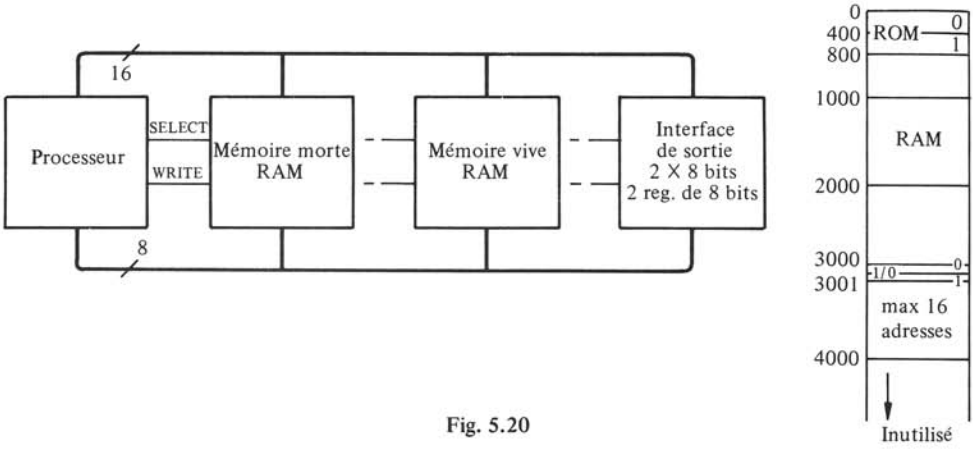


Fig. 5.20

Cette affectation d'adresse peut se résumer dans un tableau avec une colonne par bit d'adresse et une ligne par module. Dans chaque case peut se trouver l'un des quatre signes:

- 0 le bit considéré doit valoir 0 lorsque le module est sélectionné;
- 1 le bit considéré doit valoir 1 lorsque le module est sélectionné;
- – le bit considéré n'a pas d'importance pour la sélection;
- x le bit considéré est décodé à l'intérieur du module.

Le tableau correspondant à notre exemple est donné dans la figure 5.21.

	Lignes d'adresse																Adresses
	2^{15}			2^{11}				2^{7}				2^{3}				2^{0}	
ROM 0	–	–	0	0	0	0	x	x	x	x	x	x	x	x	x	x	0-3FF, 4000-43FF, etc.
ROM 1	–	–	0	0	0	1	x	x	x	x	x	x	x	x	x	x	400-7FF, 4400-47FF
RAM	–	–	0	1	x	x	x	x	x	x	x	x	x	x	x	x	1000-1FFF, 5000-5FFF
I/O 0	–	–	1	1	–	–	–	–	–	–	–	–	0	0	0	0	3000, 3010, 3020, etc.
I/O 1	–	–	1	1	–	–	–	–	–	–	–	–	0	0	0	1	3001, 3011, 3021, etc.

Fig. 5.21

Les deux lignes d'adresse de poids fort n'étant pas utilisées, les mémoires répondront chacune à quatre adresses émises par le processeur.

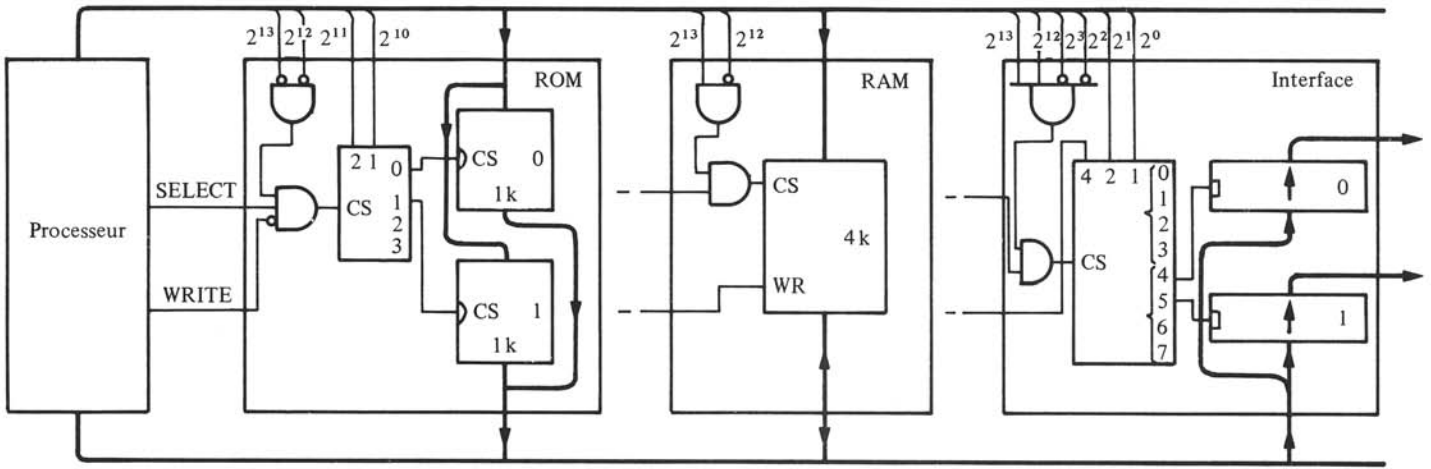

Processeur
SELECT
WRITE
2^{13} 2^{12} 2^{11} 2^{10}
2 1 0
CS
1
2
3
ROM
CS 0
1k
CS 1
1k
2^{13} 2^{12}
RAM
CS
4k
WR
2^{13} 2^{12} 2^3 2^2 2^1 2^0
Interface
4 2 1
CS
0
1
2
3
4
5
6
7
0
1

Fig. 5.22

La multiplicité des répétitions est encore plus grande avec le décodage des registres. La simplification apportée au niveau du schéma est importante, ainsi que le montre la figure 5.22, qui correspond à une approche modulaire avec un décodage indépendant pour chacun des trois modules.

Remarquons encore que de nombreuses variantes sont possibles, surtout au niveau du choix des circuits pour la réalisation détaillée. La porte ET qui dans chaque module décode les groupes de bits d'adresse et génère la condition de sélection locale est en fait un comparateur d'égalité. Deux portes ET sont dessinées chaque fois pour mettre en évidence la fonction de reconnaissance d'adresse par comparaison et la fonction de sélection. Ces deux fonctions peuvent naturellement se combiner.

5.1.16 Adresses mémoire et périphérique

L'exemple précédent identifie adresse mémoire et adresse périphérique. C'est le cas dans plusieurs processeurs, mais plusieurs autres ont des instructions spéciales de transfert avec les périphériques, et considèrent en général des adresses plus courtes (8 bits) pour la sélection des périphériques. Le décodage en est simplifié, mais une ligne supplémentaire doit permettre de distinguer entre adresse mémoire et adresse périphérique. Trois solutions de base sont possibles.

- Adjonction d'un signal PER/$\overline{\text{MEM}}$ valant 1 lorsqu'un périphérique est sélectionné et 0 lorsque la mémoire est sélectionnée. PER, WRITE et SELECT sont les trois signaux de commande correspondant à cette solution.
- Adjonction d'une impulsion de sélection des périphériques ADPER. Les signaux ADPER, ADMEM et WRITE forment alors un ensemble complet utilisé fréquemment dans la suite de cet ouvrage.
- Adjonction de deux impulsions de sélection des périphériques en lecture (INPULSE) et en écriture (OUTPULSE). Les signaux INPULSE, OUTPULSE, READPULSE et WRITEPULSE permettent toutes les sélections voulues.

D'autres combinaisons sont possibles, mais pas nécessairement cohérentes. Une très grande diversité existe dans la pratique, tant au niveau des lignes de sortie des processeurs [90] qu'au niveau des signaux transmis sur le bus du système [80].

5.1.17 Lignes de commande supplémentaires

Dans le cas d'un système processeur type avec bus d'adresse et d'information, on est amené à distinguer, en grande partie à cause de composants spéciaux tels que les mémoires dynamiques, l'instant où les adresses sont stables et permettent de sélectionner le répondant voulu et l'instant où le transfert d'information peut s'effectuer.

Ceci conduit à appeler ADVAL (*ad*dress *val*id) ou AS (*a*ddress *s*trobe), le signal appelé précédemment SELECT et DAVAL (*da*ta *val*id) ou DS (*d*ata *s*trobe) le signal indiquant que le transfert de données peut s'effectuer (fig. 5.23).

Comme pour le signal READY expliqué au paragraphe 5.1.7, il peut y avoir avantage à considérer plutôt un signal NODA (nodata valid) inverse du précédent aux instants de sélection. Lorsque NODA est actif, ou DS inactif, le bus d'information peut être utilisé dans d'autres buts.

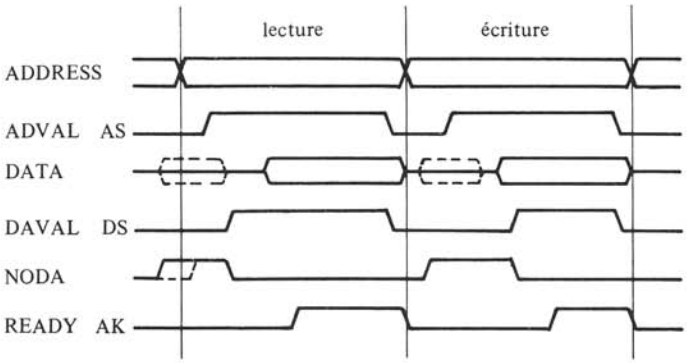


Fig. 5.23

Le signal d'acceptation (AK, READY) est généré par le répondant pour signaler au commandant qu'il peut terminer le cycle. On pourrait imaginer une acceptation séparée des adresses et des informations avec deux lignes ADK et DAK, mais aucun processeur n'est actuellement construit avec ces deux lignes.

5.1.18 Bus multiplexés

Un bus complet comporte un nombre important de lignes, en grande partie utilisées pour les adresses et les informations. Une économie sensible peut être réalisée parfois en multiplexant les adresses et les informations sur les mêmes lignes. Un multiplexage se fait sur la carte ou le circuit processeur. Un multiplexage doit être réalisé au niveau de chaque carte connectée sur le bus, sous forme d'un registre mémorisant les adresses et les conservant jusqu'à la fin du cycle (fig. 5.24).

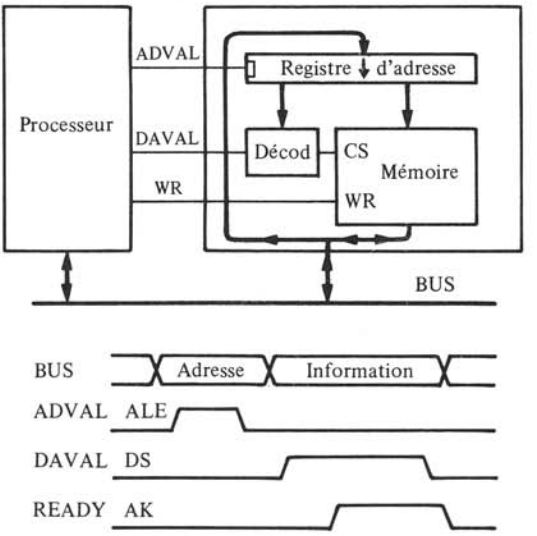


Fig. 5.24

Le signal ADVAL, souvent appelé ALE (*a*ddress *l*atch *e*nable), charge le registre d'adresse au début du cycle.

Le coût décroissant des circuits électroniques par rapport au coût des connexions encourage le multiplexage des adresses et données, voire encore d'autres signaux. Un ralentissement relativement faible des opérations en résulte, par le fait que les adresses doivent être décodées.

5.1.19 Cycles spéciaux

Dans le transfert simultané ou multiplexé d'une adresse et de la donnée correspondante, l'information d'adresse correspond souvent à des valeurs consécutives. C'est le cas en particulier lorsque l'on balaye une table pour chercher une valeur. Si le bus est multiplexé, on peut dans ce cas simplement remplacer le registre d'adresse par un compteur (fig. 5.24) et incrémenter ce compteur à chaque transfert. On parle alors de *transfert de blocs,* qui permettent d'atteindre des vitesses de transfert d'information maximales, même sur un bus non multiplexé, étant donné l'économie sur le temps de décodage.

Un autre type de cycle intéressant combine une lecture et une écriture à la même adresse. Ces cycles de *lecture-écriture* (*RMW: Read Modify Write*) sont indispensables dans un système multiprocesseur, pour éviter la modification de la même portion mémoire par un autre commandant, dans l'instant entre la lecture et l'écriture.

Dans les systèmes ordinateurs complexes, des *cycles partagés* (*split cycles*) évitent l'attente passive sur la quittance de synchronisation (SSYNC, READY ou NOTRDY) qui pourrait bloquer le processeur pour quelques millisecondes si l'information doit venir d'un disque. Chaque transfert de lecture est dédoublé: un cycle d'écriture du commandant dans le répondant précise l'information désirée, et le répondant transmet l'information lorsqu'elle est prête, après avoir interrompu le processeur qui s'est occupé d'autres tâches en attendant [80].

5.2 TRANSFERTS PROGRAMMÉS

5.2.1 Interface d'entrée

La lecture par le processeur d'une information arrivant par un câble unidirectionnel (§ 5.1.11) est simple. Le schéma en a été donné dans la figure 5.16, sans tenir compte du problème de la synchronisation des échanges (§ 5.1.4). Ce problème se résout à la fois par des circuits et par une programmation appropriée, afin de permettre au processeur de commander complètement les échanges.

Un registre, ou plus simplement un ensemble de portes à trois états, est utilisé pour transmettre l'information du périphérique (DATAIN) sur le bus du processeur, lorsque l'instruction de lecture s'exécute. Le plus souvent, le transfert se fait directement avec le registre accumulateur A du processeur, et l'instruction de lecture s'écrit LOAD A,$ADATA. ADATA est ici un symbole représentant l'adresse du registre et le signe $ indique qu'il s'agit d'une adresse de périphérique et non pas une adresse mémoire. Lorsque l'instruction LOAD A, $ADATA est exécutée, le processeur génère une impulsion ADPER avec WRITE = 0 et la valeur ADATA sur le bus d'adresse.

Les circuits de décodage de l'interface (fig. 5.25) reconnaissent cet ensemble de conditions et génèrent une impulsion INDA de sélection des sorties du passeur de données.

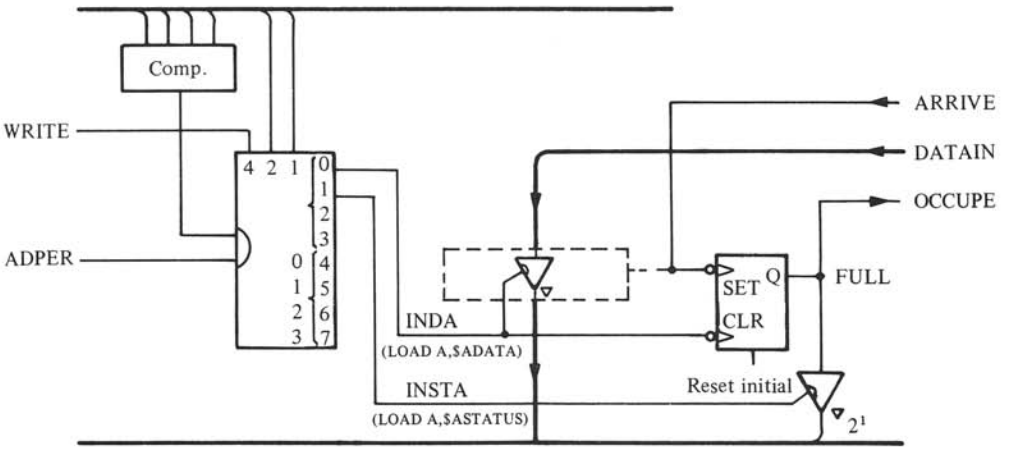


Fig. 5.25

Cette lecture de l'information de périphérique par le processeur ne doit pas se faire sans cesse, mais une fois seulement lorsqu'une nouvelle information a été transmise. Dans ce but, une bascule FULL est activée par le signal ARRIVE, et constitue en quelque sorte un second périphérique indiquant l'état de la transmission. Ce second périphérique est appelé *registre d'état* (*status register*). Le processeur doit lire fréquemment ce périphérique d'état à l'adresse ADSTATUS et surveiller la valeur du bit sur lequel le signal FULL a été relié. Tant que ce bit est égal à zéro, aucune action ne doit être entreprise. Dès que ce bit est actif (FULL = 1), alors le registre d'information peut et doit être lu: une nouvelle information a été transmise. L'exécution de l'instruction de lecture de l'information remet la bascule FULL à zéro. Toute lecture d'information étant précédée du test de l'état, chaque nouvelle information ne peut donc être lue qu'une seule fois.

On dit que la bascule FULL est un *sémaphore* entre l'interface et le périphérique. Cette méthode de synchronisation des échanges est la plus fréquemment utilisée.

Le schéma logique est donné dans la figure 5.25. La bascule FULL est une double bascule dynamique sensible aux fronts descendants, facile à réaliser avec deux bascules de types JK ou D [82]. Cette bascule est de plus connectée à la remise à zéro initiale du système. Ceci évite qu'un premier caractère non significatif ne soit lu.

Le diagramme des temps de la figure 5.26 montre une séquence type transfert en entrée. Le processeur teste régulièrement le registre d'état, et au niveau de l'interface le signal INSTA est chaque fois généré. Dès que FULL = 1 lors d'une impulsion INSTA, le processeur prend la décision de lire l'information. Une impulsion INDA est générée. Après avoir traité cette information, le processeur reprend ses tests de l'état.

Les instructions d'attente d'une information s'écrivent usuellement

```
ATINF :   LOAD      A, $STATUS
          AND       A, #MFULL          (5.1)
          JUMP, EQ  ATINF
```

Le symbole MFULL est un masque permettant d'isoler le bit FULL du registre d'état. La figure 5.25 montre que ce bit est câblé sur le bit 2^1. Le mot binaire B'00000010 = H'2

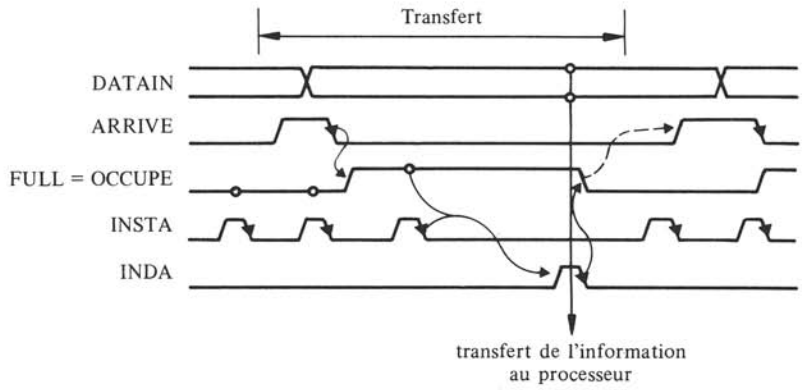


Fig. 5.26

permet d'isoler ce bit et d'initialiser le bit de nullité du processeur à un ou à zéro selon que le bit correspondant (FULL) vaut un ou zéro.

Si le processeur a une instruction permettant de tester directement un bit, on peut aussi écrire :

```
ATINF :    TEST       $ASTATUS : # BFULL
           JUMP, EQ   ATINF                    (5.2)
```

Dans ce cas, BFULL est le numéro d'ordre du bit FULL, donc BFULL = 1. L'instruction TEST, contrairement à l'instruction AND du programme (5.1), ne modifie pas l'opérande testé.

5.2.2 Exemple

Ecrivons le programme permettant de transférer en mémoire une information transmise par un périphérique, par exemple un lecteur de ruban papier. L'adresse de destination du bloc d'information commence à l'adresse 4000, et le transfert se termine dès que le 128 ème octet a été transféré.

Les programmes ci-après (fig. 5.27), écrits pour deux processeurs différents en CALM, déclarent les symboles utilisés, groupent dans une routine GETBYTE l'attente d'un octet transmis à l'interface, et se terminent par une instruction TRAP qui redonne le contrôle au moniteur de mise au point (§ 6.4.3).

5.2.3 Exercice

Modifier le programme précédent pour transférer en mémoire des caractères venant d'un clavier, avec arrêt du transfert dès que le code CR = H'D (retour de chariot) est rencontré, correction de la position précédente si DELETE = H'7F est transmis, et réinitialisation de tout le transfert si BS = H'8 est transmis. L'adresse du clavier et la position du bit FULL sont les mêmes que dans le programme de la figure 5.27.

5.2.4 Interface de sortie

Le schéma d'une interface de sortie avec synchronisation des échanges est donné dans la figure 5.28. Le signal OUTDA est créé lorsque le processeur transfère une nouvelle

```
.TITLE EX_5_27  ;Exemple de transfert programme
.PROC   Z80

DATABLOC= H'4000        ;Emplacement de l'information lue
LONGBLOC= 128.

LEC     = 6             ;Adresse de l'interface lecteur
SLEC    = LEC+1         ;Adresse du registre d'état
 BFULL  = 1             ;Position du bit FULL
 MFULL  = 2**BFULL      ;Masque pour le bit FULL

;+++ Programme

.LOC    H'1000

GETBLOC: LOAD   HL,#DATABLOC
        LOAD    B,#LONGBLOC
2$:     CALL    GETBYTE
        LOAD    {HL},A
        INC     HL
        DECJ,NE B,2$

        TRAP            ;Retour au moniteur

;--- Routine de lecture d'un caractère

;out    A caractère lu
;mod    A, F

GETBYTE: LOAD   A,$SLEC
        AND     A,#MFULL
        JUMP,NE GETBYTE   ;Attente FULL=1
        LOAD    A,$LEC
        RET

.PROC   M68000

DATABLOC= H'4000        ;Emplacement de l'information lue
LONGBLOC= 128.

LEC     = 346           ;Adresse de l'interface lecteur
SLEC    = LEC+1         ;Adresse du registre d'état
 BFULL  = 1             ;Position du bit FULL

;+++ Programme

.LOC    H'1200

GETBLOC:MOVE.32 #DATABLOC,A0
        MOVE.16 #LONGBLOC-1,D0
2$:     CALL    GETBYTE
        MOVE.8  D0,{A0+}
        DECJ.16,NMO D0,2$

        TRAP    #0

;--- Routine de lecture d'un caractère

GETBYTE:
2$:     TEST.8  (ADPER+SLEC):#BFULL
        JUMP,NE 2$
        MOVE.8  ADPER+LEC,D0
        RET

.END
```

Fig. 5.27

valeur dans le registre de sortie. Ce signal de fonctionnement est identique au signal ARRIVE, mais un retard est nécessaire pour que l'information soit stable au début de l'impulsion. Une impulsion STROBE pourrait être facilement générée avec un monostable. Le signal ARRIVE déclenche l'action de lecture et traitement de l'information au niveau du périphérique. Simultanément, une bascule de l'interface, appelée BUSY, est activée pour montrer que le périphérique a été activé, et qu'il ne faut pas lui transmettre de nouvelle information. Le signal OCCUPE a la signification vue (§ 5.1.3). La fin du signal OCCUPE remet BUSY à zéro pour utiliser le prochain transfert.

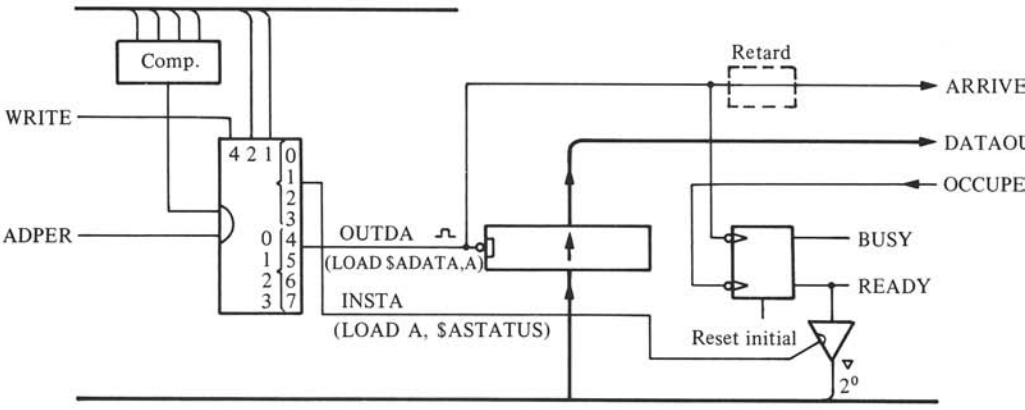


Fig. 5.28

Dans la pratique, le signal complémentaire de BUSY, appelé READY est de préférence considéré et dans la figure 5.28, c'est le signal READY qui est lu par le processeur, comme l'un des bits du registre d'état. Le programmeur n'a le droit de charger le registre de sortie que si READY = 1. Après enclenchement une ligne de remise à zéro garantit cet état initial : le périphérique est considéré comme un répondant, prêt à accepter de l'information après la phase d'initialisation.

Le diagramme des temps de la figure 5.29 montre une séquence type de transfert en sortie.

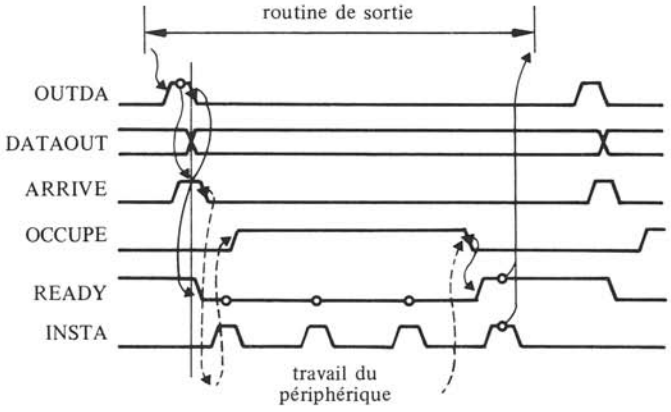


Fig. 5.29

La routine de sortie correspondant à la figure 5.29, s'écrit

```
WRBYTE :  LOAD      $ADATA,A        ; transmission du caractère
2$:       LOAD      A,$ASTATUS      ; attente tant que l'interface est
                                      occupée
          AND       A, # MREADY                              (5.3)
          JUMP, EQ  2$
          RET
```

On remarque que cette routine suppose qu'on a le droit d'écrire (il n'y a pas de test initial du READY), et que l'on attend que l'interface soit à nouveau libre avant de sortir de la routine.

Ceci n'est pas efficace, et il vaut mieux sortir de la routine tout de suite, pour pouvoir préparer l'information suivante pendant que le périphérique travaille et tester READY au début de la routine, lorsqu'il n'y a plus rien d'autre à faire que transmettre l'information.

```
WRBYTE :  LOAD       B, A              ; sauvetage temporaire
2$:       LOAD       A,$ASTATUS        ; attente éventuelle si
                                          c'est encore occupé
          AND        A, # MREADY                              (5.4)
          JUMP, EQ   2$
          LOAD       A,B
          LOAD       $ADATA,A          ; transmission
          RET
```

5.2.5 Remarque

Plutôt que d'attendre sur les indicateurs FULL et READY, on peut bloquer le processeur avec la ligne READY du bus (§ 5.1.7) en cas de sélection et tant que l'information n'a pas été acceptée par le périphérique.

Dans le cas des exemples simples de transferts programmés vus ci-dessus le résultat est le même puisque le processeur ne fait rien d'autre dans la boucle d'attente.

En pratique, le blocage du processeur n'est envisageable que pour des temps très courts (inférieurs à 10 microsecondes), et s'avère alors plus efficace qu'une boucle d'attente.

5.2.6 Double registre

Un programme de sortie d'information peut avoir tout à coup une grande quantité d'opérations à effectuer pour préparer le mot suivant. Une perte de temps peut en résulter au niveau du périphérique. Pour l'éviter, un système de *double ou multiple registre* (*multiple buffering*) est souvent incorporé. C'est l'équivalent d'une mémoire silo de petite dimension permettant d'avoir en attente dans l'interface une réserve de deux caractères à transmettre, parfois plus. Pour le programmeur, la gestion est identique, mais certaines contraintes de temps d'exécution de parties de programmes sont allégées.

5.2.7 Interface complet d'entrée et sortie

La comparaison des figures 5.25 et 5.28 montre que le registre d'état peut être commun à une interface d'entrée et de sortie. En pratique, une interface d'entrée-sortie a une structure très équilibrée si elle est formée de quatre registres:

- un registre d'entrée de l'information;
- un registre de sortie de l'information;
- un registre d'état;
- un registre de mode.

Le *registre de mode* (*mode register*) n'a pas été mentionné précédemment. Il est nécessaire avec plusieurs types de périphériques pour modifier un mode de fonctionnement (couleur du ruban d'une machine à écrire), agir sur une lampe de contrôle, sur le moteur principal ou modifier le protocole de synchronisation des échanges. Le registre d'état peut, en plus des signaux FULL et READY, contenir des indications d'erreur (épuisement du papier dans une imprimante, moteur bloqué, etc.).

Ces quatre registres peuvent être représentés comme dans la figure 5.30, qui est un modèle simplifié de l'interface.

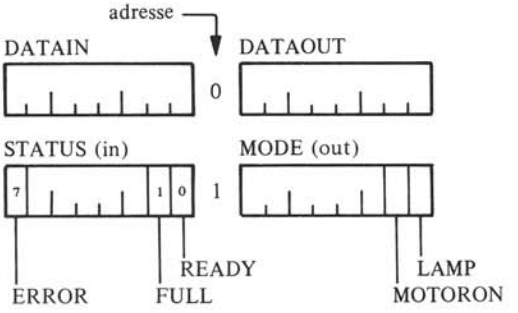


Fig. 5.30

Ce modèle est très proche du modèle utilisé par le programmeur dans les déclarations de début de programme :

```
DATAIN      = 0
DATAOUT     = DATAIN
STATUS      = DATAIN + 1
   BREADY   = 0
   BFULL    = 1                                    (5.5)
   BERROR   = 7
MODE        = STATUS
  BLAMP     = 0
  BMOTORON  = 1
```

5.2.8 Interface programmable

L'interface complète du précédent paragraphe, si elle était disponible sous forme d'un module, serait assez souvent mal utilisée.

Une application peut nécessiter 16 bits en entrée, et seulement quelques lignes de sortie, ou l'inverse. De plus, on a vu au paragraphe 5.1.3 que les signaux de synchronisation ne sont pas toujours les mêmes.

Une solution utilisée abondamment par les fabricants est de réserver des bits du registre de mode pour modifier la logique interne du module. Par exemple, un bit peut décider si un registre agit en entrée ou en sortie. C'est le cas de l'interface de la figure 5.31.

Avec cette solution, une adresse différente est utilisée pour chaque canal d'entrée ou de sortie. Mais on remarque qu'en sortie, il est possible de relire le mot transféré à l'interface. De plus, des transferts bidirectionnels sont possibles.

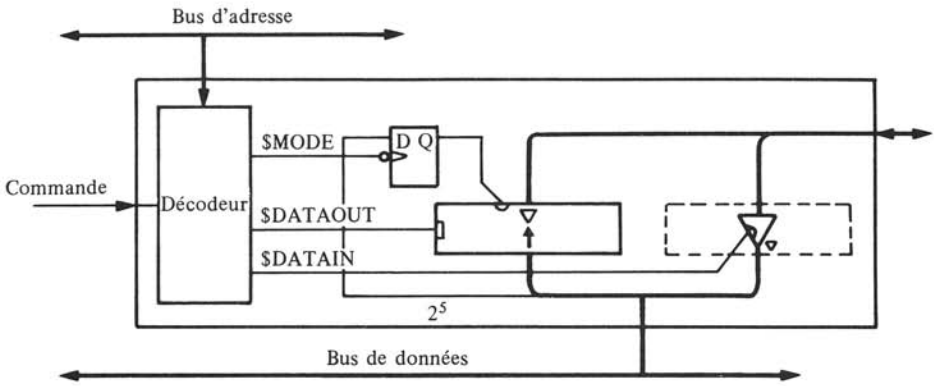


Fig. 5.31

De très nombreux types d'interfaces programmables existent, et leur complexité peut dépasser celle des processeurs avec lesquels ils travaillent [50, 81].

Par exemple, une interface programmable peut commander un écran et générer avec des compteurs les impulsions de synchronisation horizontale et verticale nécessaires pour un moniteur vidéo. Les chaînes de division sont alors programmables pour pouvoir s'adapter aux différentes normes et un tel circuit contient une dizaine de registres de mode.

Le nombre parfois important de registres des interfaces programmables et le nombre toujours très limité de broches autour d'un circuit intégré ont conduit les fabricants à utiliser différentes techniques d'adressage de registres internes.

5.2.9 Adressage direct

Un décodeur distingue chaque registre par une adresse distincte, dans certains cas, les registres d'entrée et de sortie sont distincts, et les bits correspondant des registres d'entrée et de sortie à la même adresse ont des significations différentes (fig. 5.32). Dans d'autres cas, les 2 registres sont confondus et le contenu du registre de sortie peut être

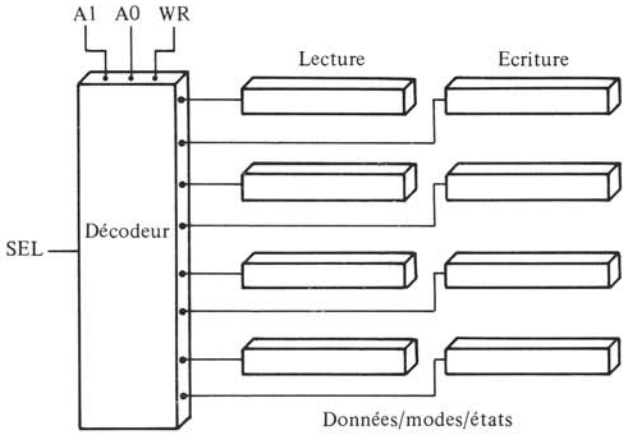


Fig. 5.32

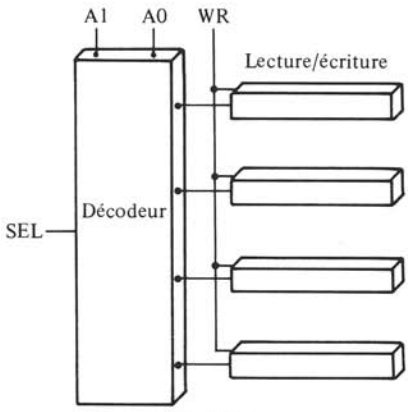


Fig. 5.33

relu, ce qui est intéressant du point de vue programmation, mais oblige à doubler le nombre d'adresses, si on a par exemple autant de fonctions d'entrée que de sortie (fig. 5.33). L'implémentation de la figure 5.31 correspond à ce 2e cas.

L'initialisation d'une série de registres consécutifs est effectuée à partir d'une table des valeurs définies dans le programme par copie avec des instructions de déplacement de blocs (§ 3.7.2).

On a par exemple avec des processeurs courants les blocs d'instructions de la figure 5.34.

Le programme pour 8085 lit les valeurs d'initialisation dans la table et les copie dans chaque adresse de périphérique successivement, par adressage direct (seul disponible pour les périphériques avec le 8085).

Le programme Z80 utilise l'adressage indexé pour accéder aux registres de l'interface programmable, qui doivent être à des adresses consécutives.

L'instruction OUTI correspond à l'exécution simultanée des deux instructions

LOAD $(C), (HL+)$
DEC B

et met à jour le fanion Z.

Le programme 68000 ne fait pas de distinction entre un espace mémoire et un espace d'entrée-sortie. Les adresses de périphériques vont de 2 en 2 étant donné que le processeur est 16 bits et l'instruction DEC.W, NMO, appelée DB par le fabricant, décompte et saute si le résultat est différent de moins un (Not Minus One).

```
8085                        Z80                          68000
      LOAD HL,#TABLE              LOAD HL,#TABLE               MOVE.32     #TABLE,A0
      LOAD A,(HL)                 LOAD C,#REG1                 MOVE.32     #REG1,A1
      INC HL                      LOAD B,#LONG                 MOVE.16     #LONG-1,D0
      LOAD $REG1,A          1$:   INC C                  1$:   MOVE.16     (A0+),(A1)
      LOAD A,(HL)                 OUTI                         ADD.16    A1,#2 ;adresses paires
      INC HL                      JUMP,NE 1$                   DECJ.16,NMO D0,1$
      LOAD $REG2,A
      ... (3 instr. par registre)
```

Fig. 5.34

Parmi les interfaces programmables très connues qui utilisent ce mode d'adressage, citons le 8255, le 6840 et le 6532 [81, 90].

5.2.10 Sous adresse dans un registre

L'adresse d'un groupe de registres peut être préalablement préparé dans un registre de mode (fig. 5.35). La sélection doit alors distinguer seulement entre le registre de commande et le registre de "données". Dans ce cas aussi, les registres peuvent être dédoublés avec signification différente des bits en lecture et en écriture.

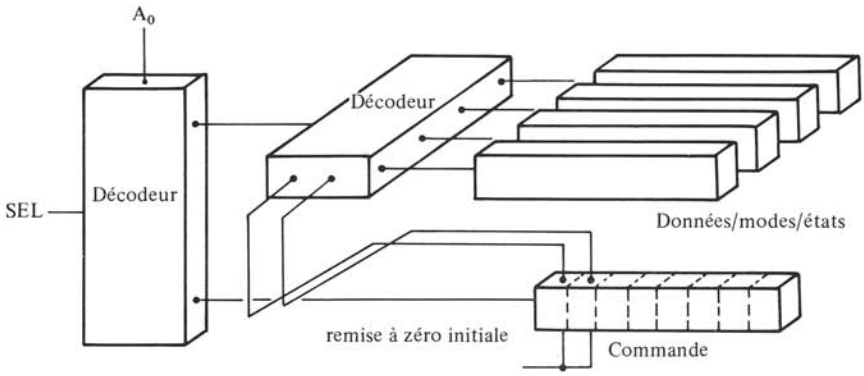


Fig. 5.35

L'initialisation du groupe de registres de données doit se faire en écrivant alternativement dans le registre de commande et de données. Si ces registres sont à des sous-adresses consécutives on peut écrire les instructions de la figure 5.36.

```
8085
        LOAD HL,#TABLE
        LOAD B,#LONG
        LOAD C,#MODIN
1$:     LOAD A,C
        LOAD $MODE,A
        ADD A,#ADREG1
        LOAD C,A
        LOAD A,(HL)
        INC HL
        LOAD $DATA,A
        DEC B
        JUMP,NE 1$
```

```
Z80
        LOAD HL,#TABLE
        LOAD B,#LONG
        LOAD C,#DATA
        LOAD A,#MODIN
1$:     INC C
        LOAD $(C),A
        DEC C
        ADD A,#ADREG1
        OUTI
        JUMP,NE 1$

(MODE = DATA+1)
```

```
68000
        MOVE.32     #TABLE,A0
        MOVE.32     #DATA,A1
        MOVE.16     #LONG-1,D0
        MOVE.16     #MODIN,D1
1$:     MOVE.8      D1,(A1)+2
        MOVE.8      (A0+),(A1)
        ADD.16      D1,#ADREG1
        DECJ.16,NMO D0,1$

(MODE = DATA+2)
```

Fig. 5.36

Parmi les interfaces programmables très connues qui utilisent ce mode d'adressage, citons le 8253, 6821, 6845 et 9519 [90].

5.2.11 Sous-adresses dans un compteur

Pour simplifier la programmation, lorsque les registres doivent être tous initialisés dans l'ordre, il y a avantage à mettre la sous-adresse dans un compteur, automatiquement post-incrémenté à chaque transfert. Un bit du registre de mode peut permettre la remise à zéro du compteur, pour réinitialisation, et on pourrait imaginer que d'autres bits du registre de mode permettraient de présélectionner une adresse quelconque comme dans le cas précédent (fig. 5.37).

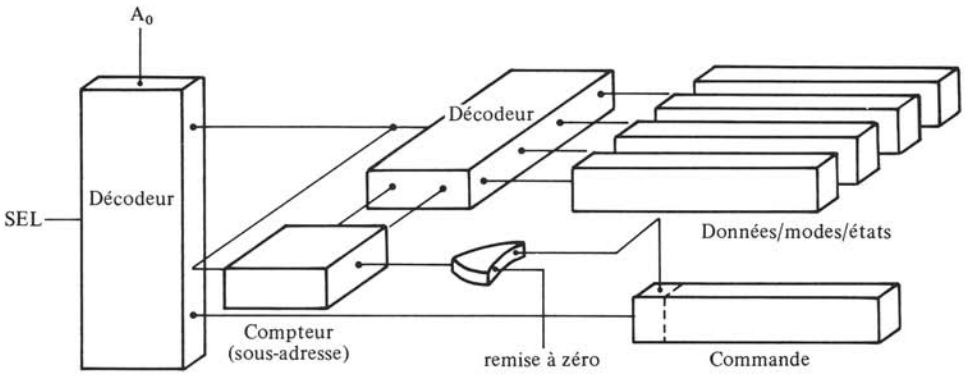


Fig. 5.37

Les instructions d'initialisation correspondantes sont données dans la figure 5.38. L'instruction OUTIR du Z80 est équivalente à la répétition de l'instruction OUTI tant que B est différent de zéro.

```
8085                        Z80                         68000
      LOAD HL,#TABLE              LOAD HL,#TABLE              MOVE.32     #TABLE,A0
      LOAD B,#LONG                LOAD B,#LONG                MOVE.32     #DATA,A1
      LOAD A,#MODIN               LOAD C,#DATA                MOVE.16     #MODIN,(A1)+2
      LOAD $MODE,A                LOAD A,#MODIN         1$:   MOVE.16     (A0+),(A1)
1$:   LOAD A,(HL)                 LOAD $MODE,A                DECJ.16,NMO D0,1$
      INC HL                      OUTIR
      LOAD $DATA,A                                      (MODE = DATA+2)
      DEC B
      JUMP,NE 1$
```

Fig. 5.38

Des interfaces programmables utilisant ce type de sous-adresse sont le 8251, 9513 [90].

5.2.12 Sous-adresse dans le registre adressé

Le registre adressé peut avoir un encodage du type rencontré dans les répertoires d'instructions de processeurs avec des champs codant des fonctions ou transferts d'autres parties du mot dans des registres plus courts. Dans un cas simple (fig. 5.39) on peut imaginer trois bits du mot sélectionnant un décodeur ayant pour effet de mémoriser les 5 bits

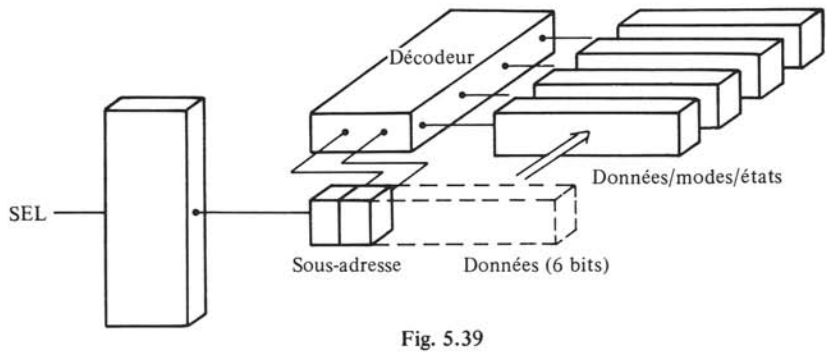


Fig. 5.39

restants dans l'un des 8 registres de 5 bits. Ce type d'encodage peut stocker 2 X 7 bits, 4 X 6 bits, 8 X 5 bits, 16 X 4 bits, 32 X 3 bits, 64 X 2 bits ou 128 bits (c'est dans ce dernier cas un "verrou adressable").

L'initialisation est très simple, puisqu'il n'y a qu'une seule adresse et que la sous-adresse peut être mémorisée dans la table avec les données d'initialisation (fig. 5.40)

```
8085                        Z80                         68000
      LOAD HL,#TABLE              LOAD HL,#TABLE              MOVE.32     #TABLE,A0
      LOAD B,#LONG                LOAD B,#LONG                MOVE.32     #DATA,A1
1$:   LOAD A,(HL)                 LOAD C,#DATA                MOVE.16     #LONG-1,D0
      INC HL                      OUTIR                 1$:   MOVE.8      (A0+),(A1)
      LOAD $DATA,A                                            DECJ.16,NMO D0,1$
      DEC B
      JUMP,NE 1$
```

Fig. 5.40

Une variante de ce mode d'adressage définit l'adresse d'un bit de l'un des registres et permet de mettre à 1 ou 0 ce bit unique. Cette variante est fréquemment utilisée, par exemple dans les interfaces 8255 et 9519 [90].

5.3 INTERRUPTION

5.3.1 Principe de l'interruption

Le principe du transfert programmé décrit précédemment oblige le processeur c'est-à-dire le programme, à lire sans cesse les registres d'état des périphériques avec lesquels les transferts sont en cours. Une technique plus efficace revient à interrompre le processeur, c'est-à-dire forcer l'exécution d'un programme toutes les fois qu'une interface est prête pour le transfert suivant. Une ligne spéciale appelée INTREQ (interrupt request) doit exister pour interrompre le programme effectué par le processeur. Cette ligne est fonctionnellement reliée au signal FULL ou READY de l'interface (fig. 5.41).

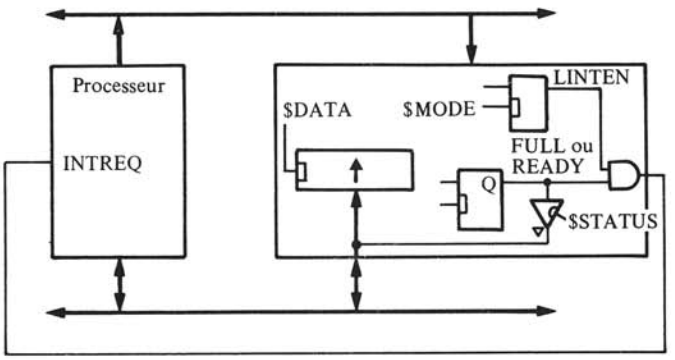


Fig. 5.41

La demande d'interruption peut être désactivée dans l'interface, sous commande d'une bascule de mode appelée ici LINTEN (Local INTerrupt ENable).

Au niveau du processeur, le signal d'interruption permet de forcer l'appel d'un sous-programme spécial, dit routine d'interruption. La logique correspondante est représentée dans la figure 5.42.

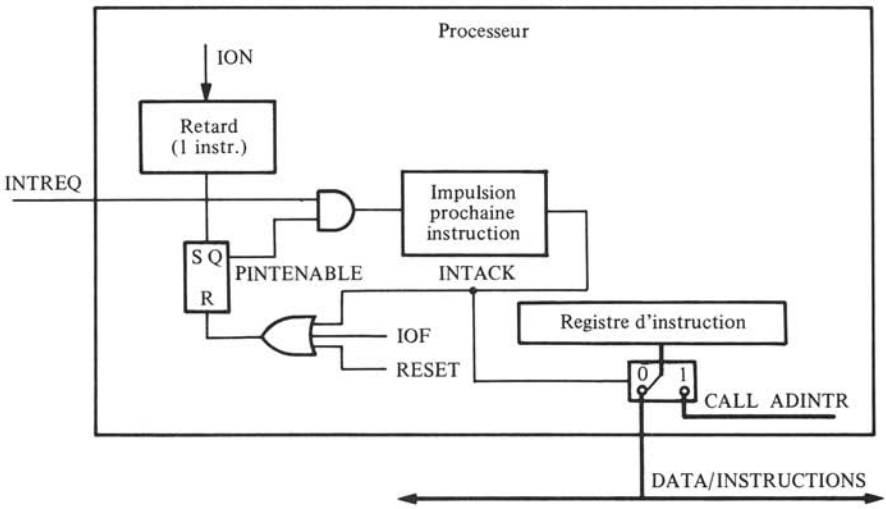


Fig. 5.42

Remarquons tout d'abord que les interruptions ne sont pas toujours acceptées par le processeur. Une bascule d'autorisation dans le processeur (PINTENABLE) décide si la demande d'interruption doit être servie ou non. Cette bascule est généralement mise à zéro à l'enclenchement du système, et les instructions ION et IOF permettent de modifier son état au cours du déroulement d'un programme.

Si l'interruption est autorisée (PINTENABLE = 1), une demande d'interruption (INTREQ = 1) a pour effet de créer, au moment où l'instruction suivante devrait être lue, une impulsion substituant à cette instruction une instruction d'appel de sous-programme à une adresse connue. Simultanément, la bascule PINTENABLE est remise à zéro pour éviter que, la demande subsistant, des appels imbriqués de la même routine d'interruption se répètent. L'appel de la routine d'interruption sauve l'adresse courante sur la pile, comme tout appel de sous-programme. C'est tout ce que fait la logique d'interruption du microprocesseur en général; le reste est entre les mains du programmeur et représente un exemple type de changement de contexte (§ 4.8.9).

En effet, le programme principal est interrompu à un instant quelconque. Le processeur a besoin de ressources (registres, positions mémoire) pour exécuter la routine d'interruption et doit commencer par sauver les états de toutes les ressources communes au programme et à la routine d'interruption, en particulier tous les registres employés dans la routine d'interruption. Si la routine d'interruption est rentrante (§ 4.8.13), l'autorisation d'interruption peut être redonnée dès que la source de l'interruption a été reconnue, avant la fin de la routine d'interruption.

5.3.2 Exemple

Gérons par interruption le transfert donné au paragraphe 5.2.2. La routine GETBYTE, qui attend un caractère n'est plus nécessaire: une interruption sera créé chaque fois que FULL passera à un. Pendant le transfert, un programme quelconque peut

s'exécuter. Pour terminer le transfert, il faut désactiver les interrupteurs au niveau de l'interface.

Avec un processeur disposant de modes d'adressage plus puissants que le M68000, le programme qui transfère par interruption le bloc de caractères venant du lecteur s'écrit:

```
            LOAD      SAVLONG, #LONGBLOC
            LOAD      SAVBLOC, #DATABLOC
            SET       $MODE: # BITLINTEN      ; activation locale
            ION                               ; activation du proces-
                                              ; seur
            ...
ADINTER:    PUSH      F
            LOAD      (SAVEBLOC +), $DATA
            DEC       SAVLONG                                  (5.6)
            JUMP, NE  2$
            CLR       $MODE: #BITLINTEN       ; désactivation locale
2$:         POP       F
            ION                               ; réactivation du proces-
                                              ; seur
            RET
```

Le programme réel de la figure 5.27 utilise les registres HL et B pour pointer le bloc et décompter les caractères. Ceci n'est plus possible dans une routine d'interruption, car ces deux registres ne seraient pas utilisables ailleurs. Tous les registres doivent rester dans le programme principal, et les positions mémoire SAVBLOC et SAVLONG remplacent ces deux registres. Cette routine d'interruption n'est pas rentrante puisque les paramètres sont en position mémoire fixe. Le contexte, qui doit être sauvé pendant l'exécution de la routine, se limite aux indicateurs, modifiés par l'instruction qui teste la fin du transfert.

Lorsque le transfert est terminé, l'interruption n'est pas réactivée et le lecteur va se bloquer avec le prochain caractère en attente d'être lu par le processeur.

Le programme principal teste la valeur SAVLONG pour savoir si le bloc est complètement transféré; on retrouve donc la notion simple de transfert programmé, mais pour des blocs d'information, la gestion du transfert de chaque caractère se faisant par interruption.

Si certaines zones du programme principal ne peuvent pas tolérer d'interruptions, par exemple à cause d'une durée d'exécution critique, il suffit de les encadrer par la paire d'instructions IOF/ION.

La figure 5.43 donne un listing plus complet de ce programme écrit pour le processeur Z80. La routine d'interruption est à l'adresse H'38, fixée par la structure interne.

Les premières instructions sauvent l'état des registres utilisés par la routine sur la pile, puis chargent les registres du processeur par les paramètres de la routine. Ce changement de contexte est suivi par la lecture du périphérique et le transfert en mémoire, puis par le rétablissement du contexte.

On remarque que l'instruction ION précède immédiatement l'instruction de retour. Le processeur retarde l'effet de l'instruction ION pour éviter qu'une nouvelle interruption ne soit servie avant que la pile ne soit vide de l'adresse de retour de l'interruption qui vient d'être servie. Plusieurs processeurs ont dans ce but une instruction combinée RETION.

```
.TITLE  EX5_43  ;Exemple de transfert par interruption
.PROC   Z80

DATABLOC= H'4000        ; Tampon pour l'information lue
LONGBLOC= 128.

LEC     = 6     ;Interface lecteur papier
SLEC    = LEC+1 ;Registre d'état
 MFULL  = 2     ;Masque pour le bit FULL
MLEC    = LEC+1
 MILECEN = 2**5 ;Activation de l'interruption locale
 MILECDIS= 0    ;Désactivation

;+++ Programme principal

.LOC    1000

        LOAD    HL,#DATABLOC
        LOAD    SAVBLOC,HL
        LOAD    A,#LONGBLOC
        LOAD    SAVLONG,A
        LOAD    A,#MILECEN
        LOAD    $MLEC,A
        ION

        . . .           ; suite du programme

;--- Routine d'interruption en mode 1 (non rentrante)

.LOC    H'38

ADINTER: PUSH   AF              ; sauvetage sur la pile
        PUSH    BC
        PUSH    HL

        LOAD    HL,SAVBLOC
        LOAD    A,SAVLONG
        LOAD    B,A             ; LOAD B,SAVLONG n'existe pas

        LOAD    A,$LEC          ; Lit et supprime la demande d'interruption
        LOAD    (HL),A          ; Transfère en mémoire
        INC     HL

        LOAD    SAVBLOC,HL      ; Changement de contexte
        DEC     B
        LOAD    A,B
        LOAD    SAVLONG,A
        JUMP,NE 2$
        LOAD    A,MILECDIS      ; Désactivation
        LOAD    $MLEC,A

2$:     POP     HL
        POP     BC
        POP     AF
        ION
        RET

.END
```

Fig. 5.43

5.3.3 Exercice

Ecrire le programme de la figure 5.34 pour le processeur M68000.

5.3.4 Exercice

Modifier le programme silo de la section 4.5 en sorte que le silo se remplisse à chaque interruption d'un périphérique lecteur de ruban. La sortie d'un caractère du silo reste tributaire de la touche CR d'un clavier.

5.3.5 Interruption multiple

La solution la plus simple pour gérer plusieurs périphériques différents en interruption est de disposer de plusieurs entrées d'interruption sur le processeur (fig. 5.44). Chaque ligne appelle une routine d'interruption différente et la gestion de l'ensemble est simple.

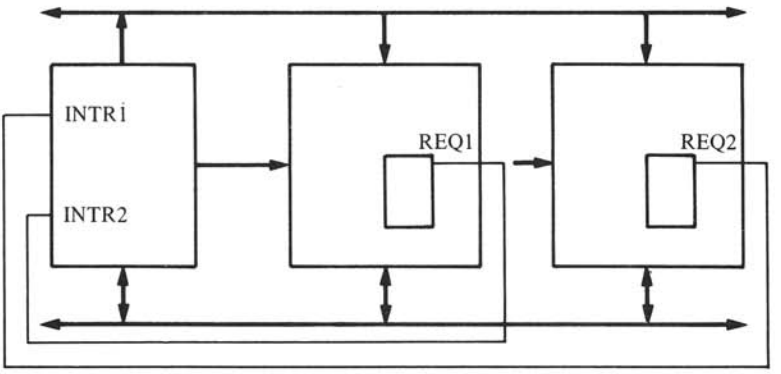


Fig. 5.44

Sur les microprocesseurs, le nombre de lignes est limité, mais la présence de 3 ou 4 lignes de demande d'interruption directe facilite beaucoup les petites applications.

Dans le cas où une seule ligne d'interruption peut être activée par plusieurs périphériques, il faut effectuer le OU des différentes demandes. La technique du OU câblé positif ou négatif [80] permet de garder les avantages d'un bus et ne pas limiter le nombre d'interfaces susceptibles d'être reliées (fig. 5.45).

Cette technique oblige naturellement à lire l'état de chaque périphérique pour déterminer la provenance de l'interruption. Cette *scrutation* (*polling*) des périphériques se fait au début de la routine d'interruption en lisant les registres d'état de chacune des interfaces susceptibles d'avoir créé une interruption.

```
INTER:      sauvetage des registres      F, A, ...
            LOAD        A, $STAT1
            AND         A, #REQ1
            JUMP, NE    SERVICE1
            LOAD        A, $STAT2
            AND         A, #REQ2
            JUMP, NE    SERVICE2
            ...                                              (5.7)
            CALL        ERROR       ; signaler une erreur si
            JUMP        RETOUR      ; la source de l'interruption
                                    ; n'a pas été trouvée

SERVICE 1 : service du périphérique 1
            JUMP        RETOUR

SERVICE 2 : service du périphérique 2
            JUMP        RETOUR

RETOUR :    rétablissement des registres F, A, ...
            ION
            RET
```

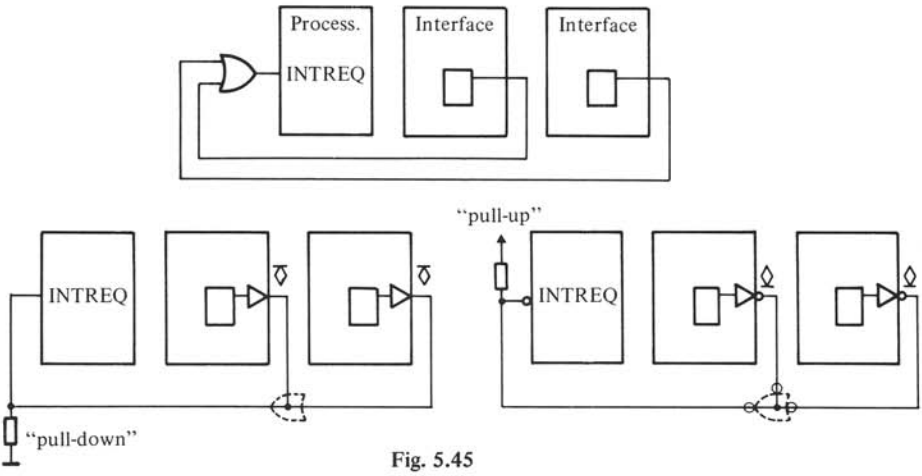


Fig. 5.45

S'il y a plusieurs interruptions simultanées, la routine sera appelée plusieurs fois consécutivement. L'ordre dans lequel les périphériques sont interrogés définit une certaine priorité d'interruption.

5.3.6 Exercice

Si les périphériques sont nombreux, le programme 5.7 comporte de nombreuses instructions prenant beaucoup de place en mémoire seulement pour déterminer l'adresse de la routine à servir. Pour chaque périphérique, le triplet STATi, REQi, SERVICEi comme défini au paragraphe précédent caractérise les opération à effectuer.

Ecrire un programme qui gère une table de triplets, dont la première valeur est le nombre de périphériques. L'allure de la routine d'interruption est la suivante:

```
INTER:      ...          sauvetage des registres
            ...          instructions faisant l'objet de cet exercice
            ION          rétablissement des registres
            RET                                                    (5.8)
TABLINTER:  .8           NBPERIPH ;nombre de périphériques
            .8.8.16      STATI, REQ1, SERVICE1
            .8.8.16      STAT2, REQ2, SERVICE2
            ...
```

Comparer la longueur des deux solutions en fonction du nombre de périphériques et déterminer le nombre de périphériques à partir duquel il y a avantage, du point de vue de la place mémoire, à utiliser une table contenant les caractéristiques des périphériques.

5.3.7 Masques

La technique de balayage du paragraphe 5.3.5 implique les possibilités de masquage local de l'interruption, au niveau de chaque périphérique. Le schéma logique correspondant est donné dans la figure 5.46, qui utilise des portes ET à collecteur ouvert positif. La technologie actuelle en fait des portes NAND en collecteur ouvert négatif.

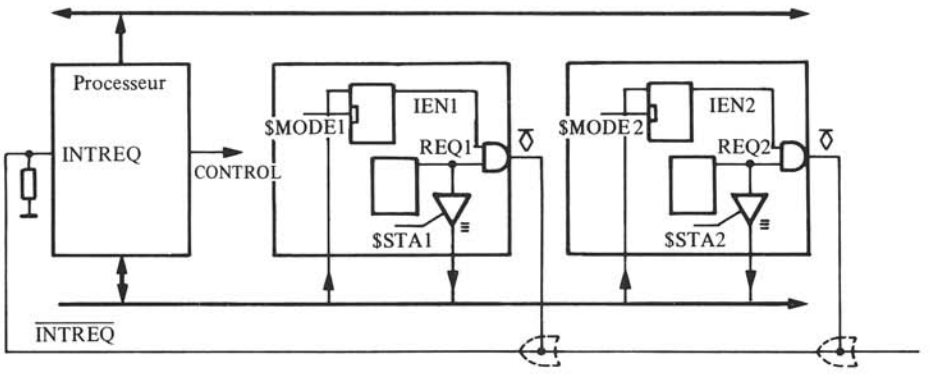


Fig. 5.46

Avant d'autoriser l'interruption au niveau du processeur, il faut définir par une suite d'instructions quels sont les périphériques susceptibles de créer des interruptions. Au moment du service d'une interruption, il est à nouveau possible de redéfinir la configuration d'interruption, et de décider d'accepter certaines interruptions de niveau très prioritaire, et d'en masquer d'autres.

5.3.8 Interruptions vectorisées

La nécessité de la scrutation peut être supprimée si le processeur a un moyen de connaître la provenance de l'interruption. Une instruction et une ligne spéciale peut permettre d'interroger tous les périphériques simultanément et de déterminer l'adresse du périphérique devant être servi en premier. Une technique plus flexible et plus fréquente consiste, au moment où l'interruption est reconnue et où le processeur force l'instruction d'appel de routine d'interruption dans le registre d'instruction (fig. 5.47) à lire cette adresse dans le périphérique lui-même. Une partie de l'adresse suffit pour donner au programmeur suffisamment de flexibilité, et cette partie est appelée *vecteur d'interruption*. Le bus d'information est naturellement utilisé pour ce transfert. Une porte à trois états dont l'entrée est câblée avec l'état du vecteur d'interruption (ou liée à un registre de mode contenant cette adresse) place le vecteur d'interruption sur le bus. L'instant est défini par

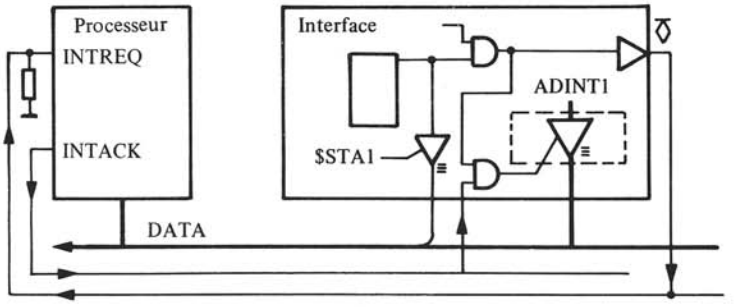


Fig. 5.47

le processeur à l'aide d'une ligne spéciale de reconnaissance d'interruption INTACK (interrupt acknowledge), équivalente à la ligne READPUL vue au paragraphe 5.1.8.

Le vecteur d'interruption a souvent 8 bits et est utilisé par le processeur pour générer une adresse de routine d'interruption selon un mode d'adressage en général indirect.

Par exemple, le Z80 prend les 8 bits de poids fort de l'adresse dans son registre I et concatène le vecteur V sur le bus pour exécuter un appel de routine d'interruption équivalent à CALL {{IV}}.

Le processeur 68000 multiplie le vecteur lu par 4 avant de chercher en mémoire l'adresse 32 bits de la routine d'interruption. L'instruction exécutée est donc CALL{{V*4}}

5.3.9 Chaîne de priorité d'interruptions

En cas d'interruptions simultanées, le schéma de la figure 5.47 n'est pas acceptable, car deux vecteurs peuvent être sélectionnés simultanément, et conduire à un mélange d'informations électriquement et logiquement peu favorable. La technique simple qui est usuellement adoptée est de chaîner le signal INTACK en le faisant passer par une porte d'inhibition (fig. 5.48). INTACK se propage à travers chaque interface, en formant ce qu'on appelle une *chaîne de priorité d'interruption* (*daisy chain*).

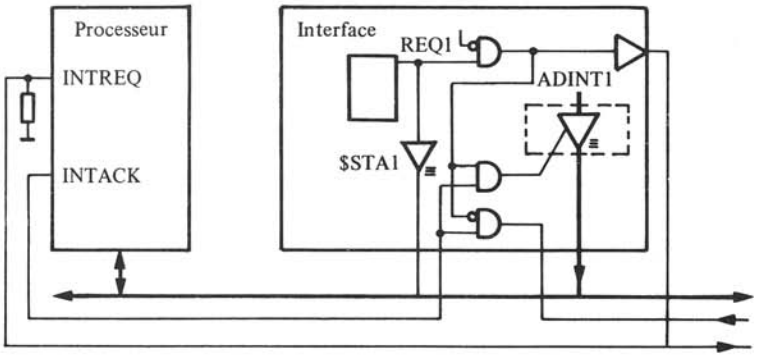


Fig. 5.48

En pratique, les chaînes d'interruption se font souvent avec deux fils pour économiser les délais.

L'interface la plus proche du processeur a la plus grande priorité: en cas de demande simultanée, c'est elle qui placera son vecteur, l'interface la plus éloignée sur le bas a la priorité la plus faible. Le seul problème est le temps de propagation au travers de la chaîne de priorité. L'utilisation d'une ligne supplémentaire, ainsi qu'une synchronisation des demandes sont nécessaires pour un fonctionnement correct [80, 81].

5.3.10 Contrôleur d'interruption

La distribution de la logique d'interruption dans chaque unité est très intéressante par son aspect modulaire, mais elle conduit à une quantité relativement importante de matériel dans chaque interface, et convient mal lorsque les interfaces sont simples et nombreuses. Une variante consiste à concentrer toute la logique d'interruption dans une

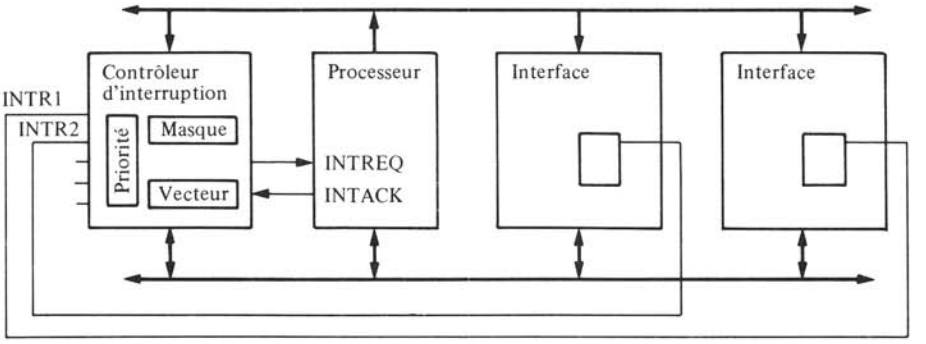


Fig. 5.49

interface unique, dotée d'une ligne de demande d'interruption pour chacune des autres interfaces (fig. 5.49).

Ce contrôleur d'interruption est en fait une interface programmable particulière, comportant les registres nécessaires au masquage des interruptions et à la génération du vecteur d'interruption. Une logique fixe a la priorité des demandes, selon une règle qui peut être programmable.

Du point de vue programmation, le contrôleur d'interruption est plus facile à gérer qu'un ensemble de bits de commande d'interruption répartis dans des périphériques, mais les liaisons en étoile depuis le processeur sont contraires à la notion de bus formé de lignes parallèles et de connecteurs tous équivalents.

Des contrôleurs d'interruption programmables existants sont les 9519, 8259 et 6828 [90].

5.3.11 Interruption non masquable

Il existe des causes d'interruption absolument prioritaires dont le contrôle doit échapper au programmeur. C'est le cas par exemple des pannes de courant, dont l'effet peut être retardé de quelques millisecondes pour permettre un sauvetage dans une mémoire non volatile de l'ensemble de contexte du programme en cours d'exécution.

Une entrée d'interruption *non masquable* (*non masquable interrupt, NMI*) est généralement sensible au flanc montant, mais une astuce évite qu'une impulsion parasite n'ait de l'effet. Autrement l'interruption n'étant pas masquable, des interruptions en cascade seraient crées. A part ce fait, l'interruption non masquable est identique à une interruption simple avec une adresse généralement unique de la routine d'interruption.

Si l'interruption non masquable est utilisable en cas de coupure de courant, le rétablissement du courant doit être suivi d'un programme reconstituant le contexte détruit par la panne avant de continuer l'exécution. Une grande attention, tant du point de vue matériel et logiciel, doit être portée à ce problème pour une solution satisfaisante.

5.3.12 Trappes

Le processeur lui-même peut être considéré comme une unité périphérique effectuant des calculs, des références mémoire, etc. et pouvant avoir des informations urgentes à communiquer au programme en exécution. Par exemple, si ce programme déborde les

limites assignées à une pile, si un dépassement de capacité apparaît dans l'unité arithmétique, ou si simplement une instruction illégale est rencontrée, l'exécution d'une routine adéquate peut être forcée. On parle de *trappes* qui sont en tout point semblables à des interruptions et peuvent, sous contrôle du programme être autorisées ou inhibées.

5.4 ACCÈS DIRECT EN MÉMOIRE

5.4.1 Principe

Pour transférer un bloc d'information d'un périphérique dans une zone mémoire, on a vu au paragraphe 5.2.2 la solution d'un transfert programmé et au paragraphe 5.3.2 la variante par interruption.

Une autre solution est de demander à l'interface de placer elle-même l'information en mémoire, sans solliciter l'attention continuelle ou fréquente du processeur. C'est plus compliqué et coûteux, puisque l'interface doit alors être un maître pouvant jouer un rôle de commandant et doit comporter au moins un registre remplaçant le registre d'index du processeur et un compteur déterminant la longueur du bloc transféré (fig. 5.50).

Le registre d'adresse et le compteur sont initialisés par le programme comme les registres d'une interface. Par la suite, à chaque mot reçu par l'interface, celui-ci prend le contrôle du bus d'adresse et de donnée et place directement l'information en mémoire à l'adresse pointée par le registre. On parle donc d'*accès direct en mémoire* ou *DMA* (*direct memory access*).

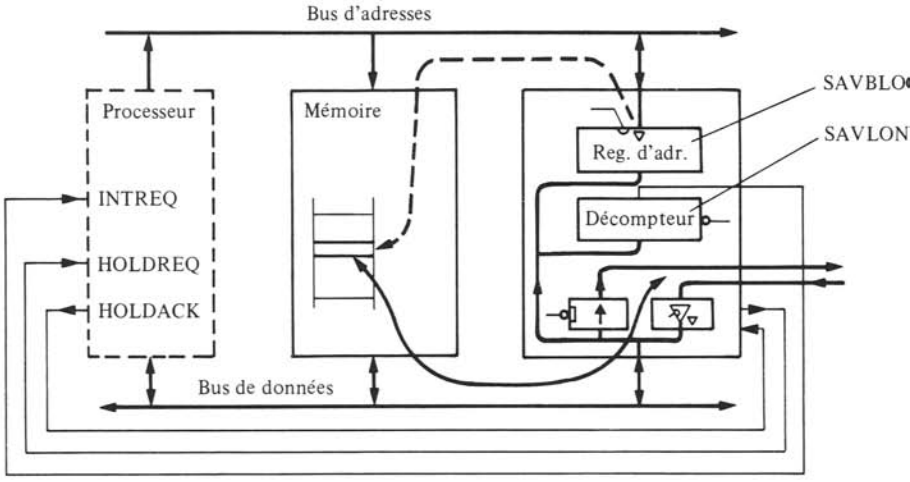


Fig. 5.50

Le processeur et l'interface DMA ne peuvent naturellement pas accéder simultanément à la mémoire. L'unité DMA doit demander au processeur de suspendre son activité, par une ligne spéciale appelée HOLDREQ. Le processeur termine le cycle de transfert en cours et signale lorsque le bus est libre en activant une ligne appelée HOLDACK. Dès que HOLDREQ est désactivé, le processeur reprend son activité, sans avoir été perturbé par la suspension.

Lorsque le bon nombre de caractères a été transféré, l'interface DMA crée une interruption pour signaler au processeur que le transfert est terminé. La routine d'interruption peut alors traiter le bloc et initialiser un nouveau transfert.

5.4.2 Priorité

Plusieurs unités DMA peuvent demander simultanément le contrôle du bus. Le problème de l'arbitration des priorités de DMA est identique à celui des priorités d'interruptions, les signaux HOLDREQ/HOLDACK étant les correspondants des signaux INTREQ/INTACK. On a donc le choix entre une chaîne de priorité d'interruption, donnant la plus haute priorité à l'unité la plus proche du processeur, et un contrôleur de priorité.

5.4.3 Modes de transfert

Le transfert d'un bloc par DMA peut se faire selon différents modes.

Dans le *mode par bloc* ou *mode continu* l'activité du processeur est suspendue pendans tout le transfert du bloc. Ceci n'est applicable naturellement que pour des transferts rapides, et permet des transferts plus rapides que lorsque le processeur reprend le contrôle entre chaque DMA.

Dans le *mode par rafale* (*burst mode*), l'interface garde le contrôle du DMA tant que le transfert d'information est suffisamment rapide pour saturer le bus. Dès qu'un caractère n'a pas été reçu à l'instant où il faudrait commencer à le transférer en mémoire, le contrôle est rendu au processeur.

Dans le *mode par caractère* ou par mots, le processeur est suspendu chaque fois pour la durée du transfert d'un seul caractère.

Une variante de ce mode isole le processeur du bus pendant de brefs instants, sans utiliser la ligne HOLDREQ, mais avec la ligne NOTREADY pour ralentir le processeur s'il a justement besoin du bus pour un transfert. Ce mode est appelé *vol de cycle* (*cycle stealing*) et implique une logique un peu plus complexe.

Un dernier mode, dit *mode transparent* permet le transfert des informations lorsque le processeur n'utilise pas le bus, par exemple lorsque des opérations arithmétiques internes sont effectuées. Le programme n'est dans ce cas absolument pas ralenti, mais une logique compliquée et des contraintes sévères de durée de transferts DMA doivent être respectées.

De plus, la fréquence maximum des transferts peut dépendre du programme exécuté, ou plus exactement des instructions utilisées dans le programme.

5.4.4 Remarque

Les transferts DMA ne se font pas nécessairement entre mémoire et périphérique. Une logique adéquate peut permettre le transfert direct entre deux périphériques, ou entre deux zones mémoire. Dans ce cas, l'interface DMA doit comporter deux pointeurs (registres d'adresse) et permet des déplacements de blocs effectués par DMA plutôt que par l'instruction vue au pragraphe 3.7.2.

5.4.5 Interface DMA programmable

Une interface comportant toutes les facilités DMA, vues précédemment peut être réalisée sous forme d'un circuit monolithique. La complexité approche celle d'un processeur et l'utilisation en est tout aussi difficile. De nombreux registres permettent de définir les modes, les adresses et les compteurs nécessaires aux transferts [62, 81, 90].

Dans presque tous les cas, le contrôle de DMA est séparé de l'interface programmable s'occupant des transferts d'information en parallèle ou en série. Plusieurs canaux peuvent être incorporés dans le même circuit DMA, pour servir plusieurs interfaces de transfert. Chaque canal est formé d'un compteur d'adresse, d'un décompteur de longueur et de registres de mode et d'état associés. Le mode définit le type d'accès (par rafale, par caractère, etc.), les priorités et les interruptions.

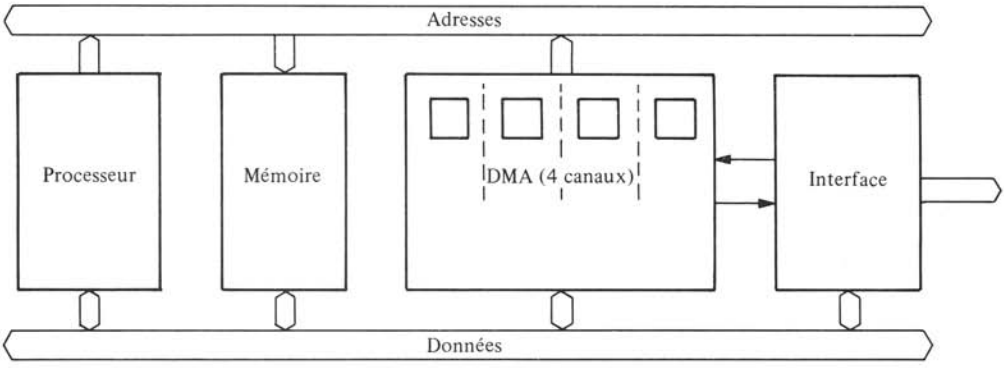


Fig. 5.51

La structure correspondant à la remarque du paragraphe 5.4.4 est donnée dans la figure 5.52. Les transferts DMA se font en deux temps, avec stockage intermédiaire dans le registre A. Un certain traitement peut être effectué simultanément (test sur des bits, masquage) et ce type d'unité DMA est une version primitive des processeurs d'entrée/sortie et coprocesseurs [90] dont l'étude sort du cadre de ce livre. Le circuit DMA Z80-DAM [90] a la structure représentée dans la figure 5.52.

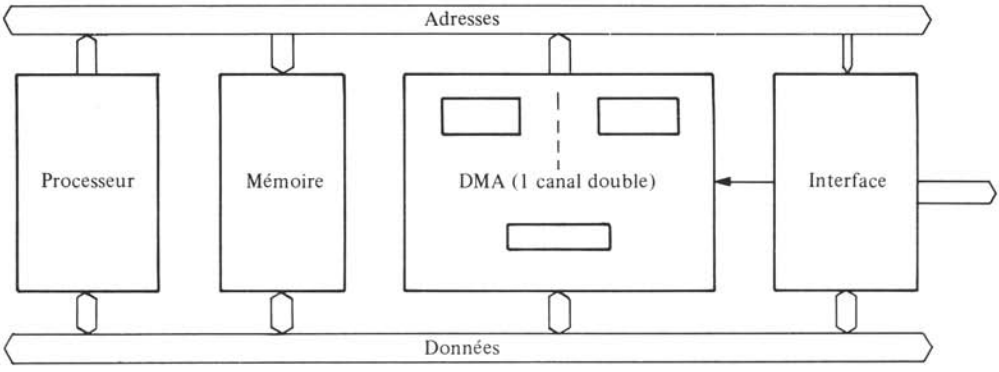


Fig. 5.52

5.4.6 Types de maîtres

Dans tout ce qui précède , le processeur commande le bus. Dans le cas du DMA, il prête son pouvoir sur l'ensemble des lignes du bus, mais garde le contrôle de la ligne HOLDACK tout en surveillant HOLDREQ.

Une approche un peu différente est utilisée dans certains systèmes; elle revient à considérer le bus comme une ressource à disposition des différents maîtres, processeurs, et unités DMA. Chaque fois qu'un processeur veut exécuter une instruction, il demande le bus, démarre son cycle dès qu'il l'obtient, et le libère au plus vite. Les interfaces DMA agissent de même.

Deux maîtres peuvent communiquer entre eux, mais alors l'un est le commandant, et a pris l'initiative du transfert, et l'autre est le répondant, et prend ou fournit l'information décidée par le commandant.

Lorsque le bus est libre, le premier maître qui prend le bus a la priorité, mais les cas de simultanéité de demande doivent être soigneusement analysés.

Il faut également éviter que le bus ne soit accaparé par deux ou trois maîtres de haute priorité et ne bloque l'accès aux unités de basse priorité. Ce travail est l'objet d'un contrôleur de bus qui peut être soit centralisé, soit distribué dans chacune des unités maîtres.

Cette approche permet un nombre quelconque de processeurs et unités DMA, l'efficacité diminuant toutefois avec le nombre d'unités [80, 83].

5.5 TRANSFERTS SÉRIE

5.5.1 Principe général

Dans un *transfert série*, les bits d'information sont transmis l'un après l'autre, séquentiellement (fig. 5.53). Une information de *synchronisation* est nécessaire pour reconnaître les bits, et une information de *verrouillage* permet de connaître le début des mots. En transmission série, les mots sont le plus souvent de 8 bits (octets).

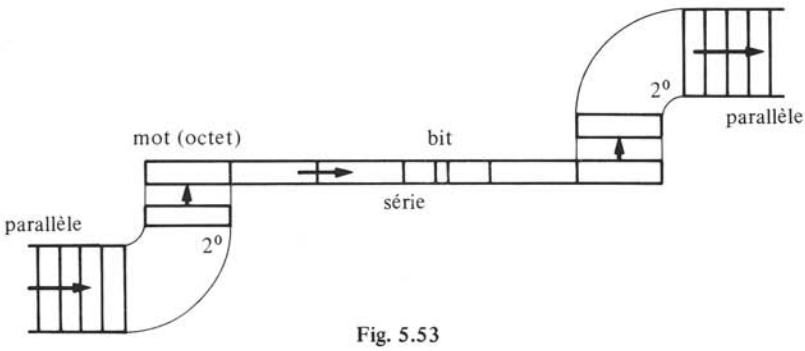


Fig. 5.53

Les transferts purement série n'utilisent qu'une seule ligne, par exemple un fil (plus le fil de référence de potentiel), une fibre optique, etc. Les informations de verrouillage et de synchronisation doivent alors être temporellement multiplexées ou encodées avec les signaux d'information et être décodables à la réception. De plus, une mise en forme des signaux, dépendant des caractéristiques physiques du canal est nécessaire (fig. 5.54).

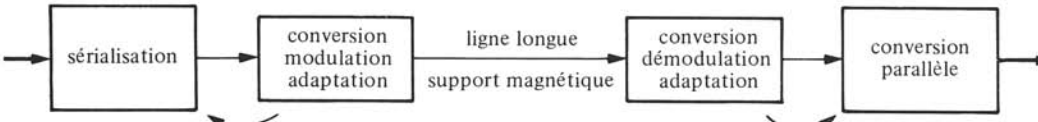


Fig. 5.54

Le problème des transmissions série est très vaste puisqu'il recouvre l'ensemble des problèmes suivants:

- structuration d'un bloc d'information;
- synchronisation des mots (verrouillage);
- synchronisation des bits;
- adaptation à la voie de transmission.

Notons que le transfert en série d'une information est semblable à son enregistrement en série sur un support magnétique par exemple. Les solutions choisies ne sont pas toujours les mêmes à cause de la fluctuation en vitesse du support magnétique au cours de la lecture.

5.5.2 Série complet

Un transfert série est *complet* lorsque les signaux de synchronisation sont transmis directement en plus de l'information. C'est le cas du registre à décalage [23], pour lequel l'impulsion d'horloge synchronise les transferts et permet l'enregistrement correct des bits. Pour transmettre des mots, il est nécessaire de disposer en plus de deux signaux équivalents aux signaux ARRIVE et OCCUPE vus au paragraphe 5.1.3.

La figure 5.55 montre un tel transfert série complet, dans lequel l'initiative du transfert est prise par la source. La source génère également les impulsions de décalage pour le registre destination. On remarque dans la figure 5.44 une inversion du signal d'horloge, permettant la mémorisation dans le registre source à un instant où l'information est bien stable.

Pour diminuer le nombre de lignes, le signal ARRIVE peut être généré avec un monostable comme l'enveloppe des impulsions d'horloge. Une fréquence minimale de celui-ci

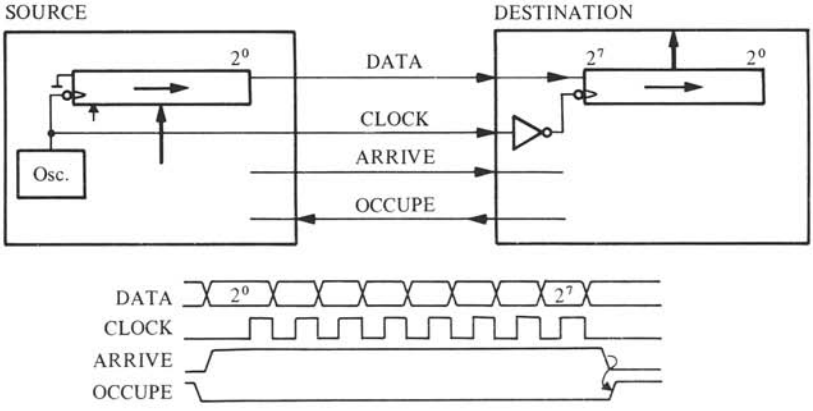


Fig. 5.55

est alors imposée et un espace équivalent à plusieurs impulsions d'horloge doit séparer la transmission de chaque mot.

Le signal OCCUPE peut également être supprimé, si la source transmet l'information à une vitesse inférieure à celle que peut accepter la destination.

Cette hypothèse est généralement faite avec les transmissions série, étant donné le fait que ce type de transmission est intrinsèquement lent. La synchronisation des échanges n'est pas possible mot par mot, et un contrôle est rappelé au niveau du message complet (§ 5.5.8).

5.5.3 Série modulé

La suppression de la ligne d'horloge dans la figure 5.55 nécessite un encodage temporel pour lequel de nombreuses techniques ont été proposées (vol. XVIII). Le code biphasé utilisé dans la figure 5.56 a une transition positive ou négative, selon que le bit à transmettre est 1 ou 0. Le décodage est possible si la déformation temporelle des signaux est inférieure à une valeur de l'ordre de 20 % d'un bit suivant.

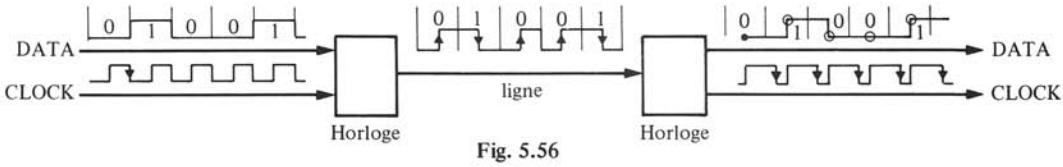


Fig. 5.56

Ce type d'encodage est abondamment utilisé dans l'enregistrement de l'information sur support magnétique. A cause des fluctuations importantes de vitesse en lecture, la synchronisation doit être intimement liée à l'information. Souvent, deux pistes magnétiques sont utilisées. Même dans ce cas, un encodage adéquat permet d'augmenter la bande passante, et tenir compte des caractéristiques du support magnétique et des amplificateurs [78].

La synchronisation par mot peut s'obtenir comme au paragraphe précédent par des silences entre chaque mot, ou avec l'une des techniques vues dans les paragraphes suivants.

5.5.4 Série asynchrone

La technique de *transfert arythmique* (souvent appelé *transfert asynchrone*) permet de générer une information de synchronisation de mots (verrouillage) grâce à un motif simple placé entre chaque mot et le suivant. Après cette transition, le signal reste à zéro pendant une période entière, et les bits du mot suivent avec la même période exprimée en bits par seconde. La fréquence des bits (bits par seconde) doit être très stable.

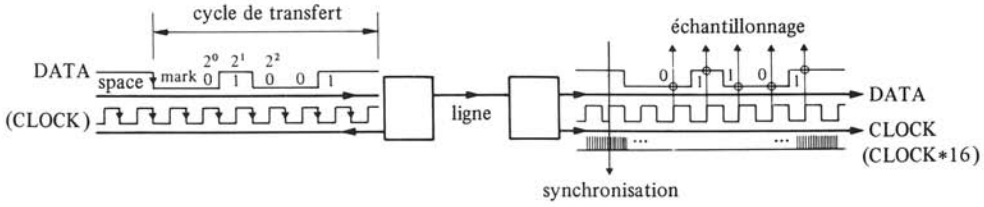


Fig. 5.57

La figure 5.57 montre la forme type des signaux dans le cas de mots de 5 bits. Un cycle de transfert commence par une transition de l'état 1 appelé *repos* (*space*) vers l'état 0 appelé *travail* (*mark*). Une période complète à l'état 0 est suivie ici par 5 périodes comportant les bits d'information, poids faibles en tête. Un 6e bit de repos d'une période (parfois d'une période et demi) doit être prévu avant le début du cycle suivant.

L'intérêt du transfert série est de permettre une vitesse quelconque de transfert de mots en dessous d'une fréquence maximale fixée par le débit du canal.

La vitesse de transfert est le bit par seconde, souvent abusivement appelé Baud. De façon générale, le *Baud* est une unité de débit de signaux élémentaires, ou rapidité de modulation. Des techniques d'encodage permettent de faire correspondre plusieurs bits par Baud (vol. XVIII), mais dans les cas simples de transmission binaire, il y a correspondance entre Baud et bit par seconde.

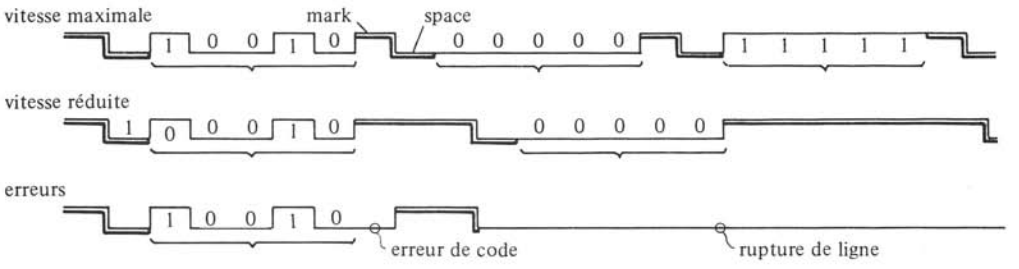


Fig. 5.58

La figure 5.58 montre quelques exemples types de signaux sur la ligne. Les erreurs qui peuvent être reconnues sont appelées *erreur de verrouillage* (*framing error*), et peuvent être dues à une mauvaise synchronisation ou à un parasite et à la *rupture de ligne* (*break*). La détection des ruptures de lignes justifie le choix d'un *espace* actif (état 1) et d'une *marque* inactive (état 0).

Le décodage d'une transmission asynchrone (figure 5.47), nécessite la reconstruction (par l'unité destination) d'un signal d'horloge en phase avec les bits reçus. La transition initiale permet cette synchronisation. Une horloge de fréquence plus élevée (16 fois le nombre de bits/sec) est généralement utilisée pour réduire l'erreur de synchronisation, et permettre un décodage purement numérique. La tolérance de fréquence entre les oscillateurs de l'unité source et de l'unité destination est fixée par l'écart maximum d'une demi période lors de l'échantillonnage du dernier bit; elle est de quelques pourcents. Naturellement, le débit binaire doit être le même de part et d'autre. Les valeurs standard sont de 110, 300, 600, 1200, 2400, 4800, 9600 et 19200, correspondant à des vitesses de transfert de 10 à 1900 octets par seconde.

Les caractéristiques électriques des signaux transmis sont également standardisées selon deux normes. L'une spécifie un courant de 20 mA pour l'état 1 (repos). C'est la norme du télex qui est en voie d'abandon [78].

L'autre norme spécifie deux tensions positives et négatives pour l'état 1 et l'état 0 avec des variantes mineures d'une norme à l'autre (EIA RS232, 423, CCITT V24 [80].

Un schéma bloc type est donné dans la figure 5.59. Les circuits émetteur et récepteur asynchrone existent sous forme de circuits intégrés programmables pour différents standards de conversion série-parallèle [21, 82].

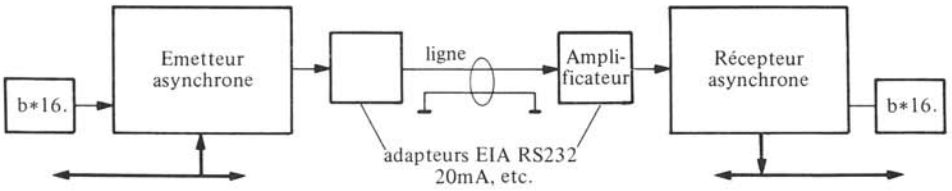


Fig. 5.59

5.5.5 Variante

Le système asynchrone a trois points faibles qu'il est facile de pallier par l'adjonction d'une ligne supplémentaire.

Ces trois défauts sont :

- exigence d'une bonne précision de l'horloge de référence du temps située dans chaque unité;
- multiplicité des valeurs de débits et nécessité d'avoir un débit identique de part et d'autre pour un bon fonctionnement;
- pas de synchronisation des échanges; la durée du transfert le plus lent fixe le dé-

Une ligne supplémentaire transmettant l'horloge de synchronisation du récepteur vers l'émetteur permet l'utilisation d'un oscillateur peu précis. Si cet oscillateur est bloqué tant que le récepteur n'a pas traité son information (fig. 5.60), une synchronisation des échanges parfaite en résulte, sans aucune perte dans le temps de transmission. Ce système appelé SIMSER (SIMplified SERial) est très bien adapté pour des transmissions à relativement courte distance entre systèmes à microprocesseurs et périphériques [80].

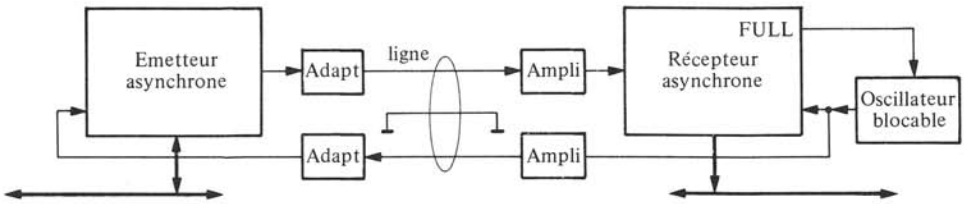


Fig. 5.60

5.5.6 Série synchrone

Dans un transfert *série synchrone*, la synchronisation des mots est obtenue par un motif répété un certain nombre de fois. Ce motif doit avoir une période de répétition égale à la longueur des mots transmis. Un comparateur dans l'unité destination permet, lorsqu'il y a égalité, d'initialiser un compteur de bits par mot, et de lire ensuite l'information correctement (fig. 5.61).

Ce qui vient d'être dit suppose que la synchronisation par bit existe. Si ce n'est pas le cas, une première recherche de cette synchronisation par bit doit se faire sur un motif initial (fig. 5.62). Une resynchronisation doit se faire ensuite sur chaque transition, et les messages comportant de longues suites de 1 ou de 0 ne sont pas possibles, à cause des exigences trop sévères que cela poserait au niveau de la stabilité et de la précision des oscillations de référence.

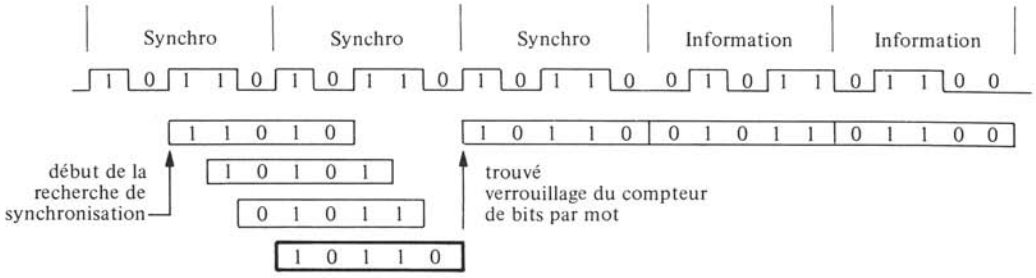


Fig. 5.61

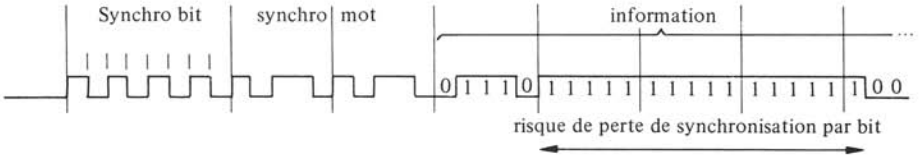


Fig. 5.62

Un autre type de liaison synchrone comporte un mot de synchronisation appelé fanion comportant six 1 consécutifs. Cette combinaison de bits ne peut jamais se produire dans le message, car l'émetteur insère automatiquement un 0 après une suite de cinq 1 consécutifs.

Ce *bourrage de bits* (*bit stuffing*) ne doit pas être confondu avec l'insertion de mots neutres ou mots de synchronisation, appelé *remplissage de temps entre trame* et destiné à maintenir un flux d'information sur la ligne lorsqu'il n'a pas d'information significative à transmettre.

Les protocoles HDLC (high level data link protocol) et X-25 sont de ce type, avec un format type donné dans la figure 5.63 [83, 85].

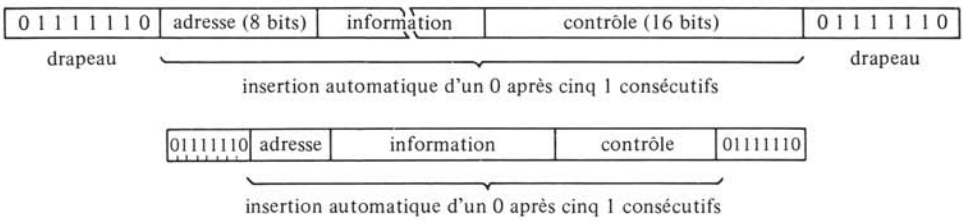


Fig. 5.63

Le fanion est suivi de l'adresse du destinataire, du message dont la longueur n'a pas besoin d'être multiple de 8 bits et d'un mot de contrôle (§ 5.5.8). La longueur transmise de cette partie est plus grande, à cause des zéros supplémentaires insérés automatiquement.

5.5.7 Transmission téléphonique

Dans une voie téléphonique, le signal doit être modulé. Un système électronique complexe, le *modem* (modulateur-démodulateur) assure l'interface avec la voix et s'occupe

de la synchronisation par bits (vol. XVIII). Le dialogue de mise en route d'une communication tient compte des signaux de modes et d'état suivant:

- Indication de sonnerie (ring indicator)
- Interface connectée (data terminal ready DTR)
- Ligne connectée (data set ready DSR)
- Demande de transmission (request to send RTS)
- Ligne prête à la transmission (clear to send CTS)
- Synchronisation faite/porteuse (carrier detected).

La figure 5.64 donne le schéma bloc d'une transmission par modem. Le modem est appelé *équipement de communication* (*DCE*: *data communication equipment*). Il est lié à un *équipement de transmission* (*DTE*: *data transmission equipment*) par un câble comportant trois fils au moins. La norme RS232/V24 [83] définit un connecteur à 25 poles dont la polarité et le brochage dépendent de la conversion DCE/DTE de l'équipement. Un mauvais respect de cette norme, et la nécessité de sélectionner le même débit dans chaque équipement lié en série (ordinateur-modem-terminal) nécessitent un certain soin lors de la mise en oeuvre d'une liaison série [80].

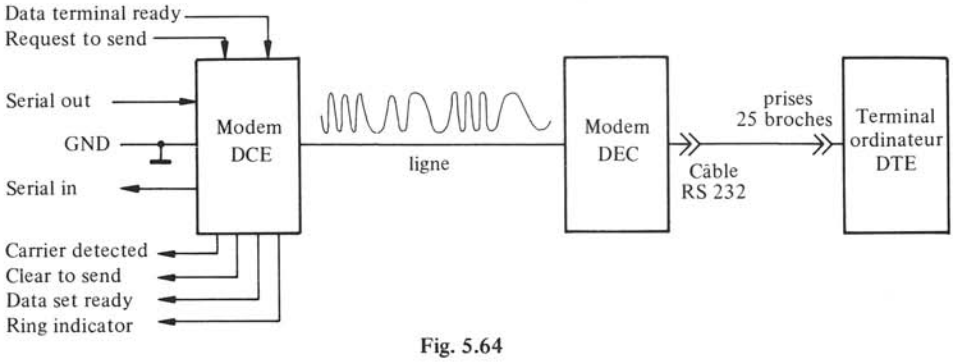


Fig. 5.64

Dans les transmissions série, on distingue les transmissions *simplex*, qui n'ont lieu que dans un sens, les transmissions en *duplex intégral* (*full duplex*) qui ont lieu dans les deux sens à la fois et nécessitent généralement deux voies et les transmissions *semi-duplex* pour lesquelles la transmission peut se faire à tour de rôle dans chaque sens [83].

5.5.8 Redondance

La fiabilité plus faible des transmissions série, due à la longueur des lignes et aux points de connexion que l'absence d'une synchronisation des échanges mot par mot (handshake), oblige à des contrôles directs et indirects de l'information.

La technique la plus fréquente est l'adjonction d'un *bit de parité* (ou d'imparité) à chaque mot transmis. Chaque mot de n bits est suivi d'un n + 1ième bit, dont la valeur est telle que le nombre total de bits à 1 dans le mot complété par son bit de parité, est un nombre pair (ou impair).

Dans les transmissions série, l'objectif est souvent de transmettre du texte alphanumérique sous forme de caractères de 7 bits (généralement en code ASCII). Un 8e bit de parité peut facilement être ajouté et testé par l'interface (fig,. 5.65).

Fig. 5.65

La notion de bits de parité peut se généraliser en celle de mot de contrôle, permettant non seulement de détecter les erreurs, mais de les corriger dans une certaine mesure. Par exemple, le code de Hamming de type 7.4 est formé de 4 bits d'information et 3 bits de contrôle (fig. 5.65). Il permet de corriger un bit incorrectement reçu et de détecter si deux bits sont incorrects.

Le contrôle peut également s'effectuer au niveau d'un bloc d'information. Tous les 20 à 200 caractères, un mot de contrôle peut être intercalé (fig. 5.66). Le récepteur applique la même règle de calcul du mot de contrôle et compare son résultat avec le mot transmis. Suivant la règle de calcul, on parle de *somme de contrôle* (*checksum*), de *contrôle longitudinal* (*longitudinal redundancy check, LRC*) ou de *contrôle de redondance cyclique* (*cycle redundancy checksum, CRC*) [83, 85].

Les contrôles transverseaux (par mot) et longitudinaux (par bloc) peuvent être effectués simultanément. Par exemple l'adjonction d'un bit de parité longitudinal et d'un bit de parité transversal permet la correction d'une erreur simple: son adresse est à l'intersection des lignes et colonnes erronées (fig. 5.67).

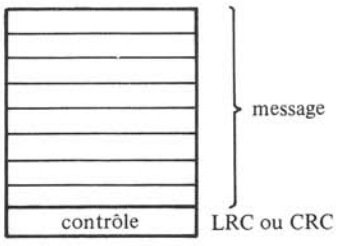


Fig. 5.66

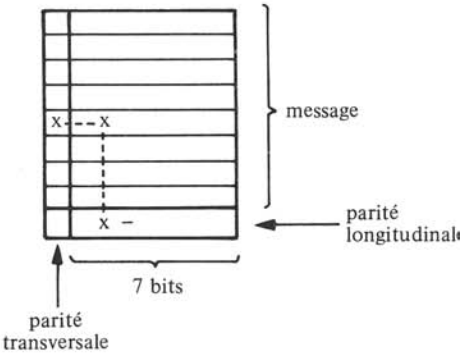


Fig. 5.67

Le problème avec les techniques de détection d'erreurs n'est pas tellement de détecter les erreurs, mais de savoir que faire en cas d'erreur. Une demande de répétition du message faux est généralement faite, il faut savoir depuis où, combien de fois maximum, et veiller aux possibilités de fautes dans la transmission de demande de répétition du message.

Ceci fait partie du *protocole* de transmission, qui fixe la structure du message en blocs d'information précédés et suivis d'identificateurs, indications de type, contrôles.

Des interfaces programmables facilitent l'implémentation des protocoles de transmissions synchrones et asynchrones. La détection automatique de la synchronisation et diverses procédures de contrôles sont généralement implémentées [21, 82].

5.5.9 Réseaux

Les liaisons série sont utilisées pour les transmissions entre ordinateurs autant que pour les liaisons entre terminaux. Les terminaux devenant stations de travail (§ 1.1.7), la distinction devient de toute façon difficile.

Les liaisons ne se font plus seulement point à point, mais chaque machine, dite *hôte* est noeud d'un réseau et des machines spécialisées jouant le rôle de *passerelle* (*gateway*), de *concentrateur* ou de *station intermédiaire de transfert des messages* (*interface message processor*) permettent l'interconnexion des réseaux et le transport de l'information à l'intérieur de chaque réseau (fig. 5.68).

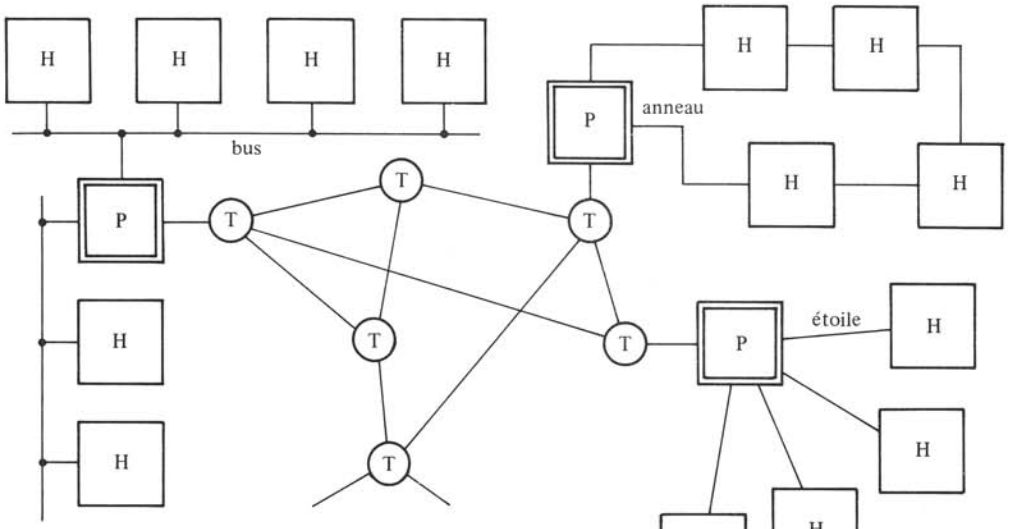


Fig. 5.68

On peut distinguer quatre types de réseaux, dont l'étendue varie entre quelques dizaines de mètres et toute la Terre, avec des vitesses de transfert moyennes de quelques mégabits par seconde à quelques kilobits par seconde, en fonction de la distance et du nombre de stations:

- les réseaux en étoile, par liaisons directes sur un concentrateur ou ordinateur centralisé; ils sont simples à mettre en œuvre (§ 5.5.4);
- les réseaux locaux à bus série; ils utilisent le plus fréquemment un câble coaxial sur lequel les paquets d'information sont envoyés [87];
- les anneaux locaux; ils sont caractérisés par des trains d'information qui tournent en rond dans le réseau en transportant les messages d'une station à l'autre [88];
- les réseaux à longue distance; ils passent par des lignes téléphoniques louées. Un maillage de stations intermédiaires permet le transport des paquets d'information selon des chemins qui peuvent varier d'un paquet à l'autre [85].

5.5.10 Modèle de référence ISO/OSI

Une meilleure compréhension des communications entre systèmes informatiques résulte d'une décomposition en couches [89]. Le modèle de la figure 5.69 résultant d'un groupe de travail de l'ISO (International Standard Organisation) et appelé OSI (Open System Interconnection), met en évidence 7 couches:

- La couche supérieure est dite d'application et est proche de l'utilisateur final. Un agent de voyages qui réserve une place ne voit que cette couche, et ne se préoccupe pas de toutes les transformations que doit subir l'information pour qu'il obtienne sa réponse. Pour lui, tout se passe comme s'il avait une communication directe avec l'ordinateur de réservation comprenant son langage.
- La couche de présentation définit la présentation des données et les codes des caractères. L'agent de voyages peut être dans ce niveau au début de son accès, quand il définit la langue qu'il préfère utiliser.
- La couche de session garantit la synchronisation de processus distants et le maintien du dialogue. A ce niveau, l'agent de voyages a connecté son terminal à un système de réservation particulier.
- La couche de transport garantit le transport correct de l'information. Cette information est formée de paquets plus ou moins grands acheminés individuellement. L'agent de voyages a dû composer le numéro de téléphone d'un réseau informatique, qui est responsable du transport correct des questions et réponses.
- La couche de réseau est responsable de l'acheminement des paquets, eux-mêmes formés de blocs d'information.
- La couche de liaison des données est responsable du transport sans erreur des blocs d'information.
- La couche physique définit le support physique utilisé pour transporter l'information et spécifie les caractéristiques associées.

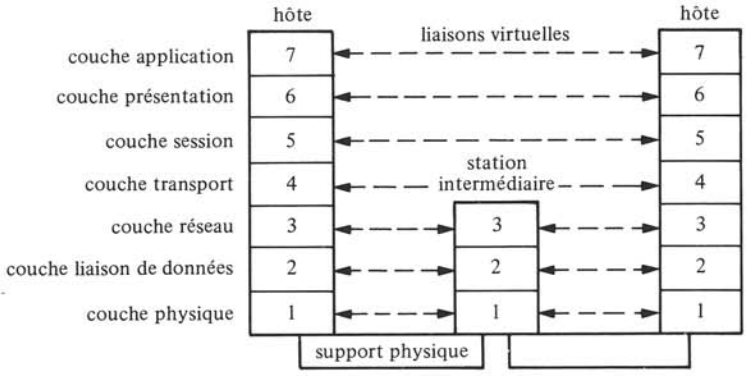


Fig. 5.69

Les transferts par bus étudiés au début de ce chapitre se placent au niveau de la couche 2, avec quelques commentaires concernant la couche 1 (temps de maintien, muscleurs, etc.).

Dans un réseau de transmission à distance comportant des stations intermédiaires aidant au routage de l'information, l'action de ces stations intermédiaires se limite au niveau 3, comme le montre la figure 5.69.

CHAPITRE 6

MICROPROCESSEURS ET SYSTÈMES

6.1 PÉRIPHÉRIQUES

6.1.1 Introduction

Le processeur, les mémoires et les interfaces ne forment qu'une relativement petite partie d'un système informatique, surtout du point de vue prix et volume, étant donné les progrès technologiques de l'intégration à large échelle. Les périphériques et le logiciel forment une part très importante de la connaissance des calculatrices, que ce dernier chapitre ne fait qu'évoquer.

6.1.2 Organes d'entrée

Le *clavier* est le principal périphérique de dialogue, car il est simple et relativement efficace. Une soixantaine de touches et des touches de fonction permettent de générer les codes des lettres, chiffres et signes spéciaux.

Les touches sont disposées physiquement et électriquement selon une matrice et des circuits spéciaux (encodeurs de clavier) [81] fournissent le code des touches pressées. Lorsque plusieurs touches sont activées simultanément tous les codes correspondants doivent être générés. La répétition automatique des touches pressées est faite parfois automatiquement dès que la touche est maintenue pressée. Les contrôleurs de clavier récents incorporent un microprocesseur et permettent d'innombrables possibilités, y compris la répétition automatique de phrases ou de séquences d'ordres.

En complément du clavier, on utilise un *crayon lumineux* ou *photostyle* (*light pen*), un *manche à balai* (*joystick*), une *boule roulante* (*track ball*), une *souris* (*mouse*) ou une *tablette graphique*. Ces périphériques facilitent et accélèrent l'interaction et ont chacun des avantages spécifiques pour dessiner, pointer sur l'écran ou encoder un dessin existant.

La reconnaissance directe de la voix présente un très grand intérêt. Le vocabulaire est limité et une phase d'apprentissage doit précéder l'utilisation [78].

La lecture directe de documents reste très délicate pour tous les documents qui n'ont pas été préparés spécialement pour être lus. Le codage de numéros de contrôle sous forme de code-barre s'est rapidement généralisé, car il permet la lecture avec un simple crayon doté du système optique et électronique adéquat [78].

6.1.3 Organes de sortie

L'information à communiquer en retour à l'utilisateur est généralement visuelle, sous une forme alphanumérique ou graphique. D'innombrables systèmes d'affichage et d'impression existent.

Pour le dialogue avec l'ordinateur, les *écrans de visualisation* (*visual display*) utilisant un écran cathodique sont le plus souvent utilisés [49]. Leur capacité usuelle (2000 caractères) est suffisante pour traiter du texte. L'affichage et le traitement graphique est plus coûteux de par le nombre important de points que l'on doit pouvoir afficher (plus de 500 000).

L'information affichée peut être mémorisée au niveau de l'écran lui-même, au niveau du contrôleur ou dans la mémoire principale du système. Dans ce cas, un *rafraîchissement* périodique de l'écran est nécessaire, 50 ou 60 fois par seconde, et s'effectue par accès direct dans la mémoire.

Une évolution rapide a lieu actuellement pour tous les systèmes d'affichage. Elle tend vers des écrans plats, en couleur, de grandes dimensions, à haute résolution et à plus faible consommation de puissance.

Les *imprimantes* (*printer*) utilisent des techniques très variées [78] et demandent souvent du papier spécial. Un affichage matriciel des caractères (§ 4.8.3) tend à se généraliser, la finesse des points offerte par la technologie actuelle permettant une qualité suffisante pour la plupart des applications. Les imprimantes matricielles ont par ailleurs l'avantage d'être utilisables pour des graphismes divers.

Les *traceurs* (*plotter*) ont sur les imprimantes matricielles l'avantage de permettre de plus grandes dimensions du papier, et un grand choix de couleurs par changement de plume automatique. Ils conviennent moins bien pour colorier des grandes surfaces.

6.1.4 Capteurs et actionneurs

Les *capteurs* (*sensors*) et *actionneurs* (*actuators*) sont des organes respectivement d'entrée et de sortie qui permettent de tenir compte d'un environnement physique et d'agir sur celui-ci. Ils jouent un rôle très important en robotique et l'imperfection des solutions actuelles encourage une évolution rapide tant du point de vue technologie que sur le plan du traitement de l'information. La reconnaissance visuelle des formes, l'asservissement des moteurs ou électrovalves commandant des mouvements dans l'espace prédéfinis exigent des systèmes microinformatiques performants dotés d'une programmation efficace.

Comme capteurs, des contacts mécaniques, sous forme de touches séparées ou de groupes de balais repérant une position, sont souvent utilisés. Une caractéristique importante est la présence de rebonds de contact, pendant quelques millisecondes, chaque fois que l'état du contact change. Une logique appropriée, ou un filtrage analogique ou programmé, permettent de s'affranchir de ce problème.

Les signaux analogiques à lire sont convertis sous forme numérique avec des circuits convertisseurs toujours plus précis et faciles à mettre en oeuvre.

En ce qui concerne les actionneurs, la commande d'un relais, d'une lampe ou d'un moteur en tout ou rien à partir de l'état d'un bit dans un registre est triviale. Un amplificateur fournissant la tension et le courant voulus suffit [81]. La commande d'un moteur pas à pas est un peu plus délicate. Chacun des bobinages du moteur est commandé en tout ou rien, mais la durée et le déphasage des impulsions de commande sont importants pour un bon fonctionnement. Si des vitesses d'avance relativement élevées sont nécessaires, les signaux ne peuvent pas être générés directement par un microprocesseur, qui nécessite souvent quelques dizaines de microsecondes pour une action simple. Des compteurs, diviseurs et trains d'impulsions, etc. sont nécessaires. Des interfaces programmables sont disponibles pour ce genre d'application.

La commande d'un moteur de façon proportionnelle nécessite une conversion du mot digital représentant la consigne de tension ou courant en un signal analogique. Celui-ci est ensuite amplifié. Les amplificateurs analogiques de puissance étant coûteux, on évite autant que possible ce type d'actionneur. Le plus souvent, à cause des inerties mécaniques, l'action proportionnelle peut être remplacée par une action impulsionnelle en tout ou rien, le rapport des durées d'action et de repos fixant l'intensité de l'action.

6.1.5 Perturbations électriques

Les sytèmes informatiques souffrent souvent de problèmes dus aux *perturbations* électriques, électrostatiques et électromagnétiques. Le concepteur d'équipements doit tout faire pour minimiser ces problèmes, qui réduisent la fiabilité et la sécurité de fonctionnement des systèmes.

Dans les systèmes informatiques ayant des périphériques répartis dans un local ou un bâtiment, les courants sur la ligne de masse perturbent les transmissions. Les décharges électrostatiques créées par les utilisateurs ont souvent un effet catastrophique sur les équipements. La proximité d'un écran et d'une unité de disque magnétique perturbe l'un et l'autre.

Les capteurs, dont le niveau d'énergie est très faible, sont influencés par les variations de courant résultant de l'activité du processeur. Le processeur lui-même peut être perturbé par les pointes de courant d'excitation d'un électro-aimant, ou par les étincelles d'un moteur.

Un découplage galvanique soigné, fait au moyen de diodes lumineuses et phototransistors (*photocoupleurs*) évite tous ces problèmes. Les alimentations, de part et d'autre du système optique, doivent être elles-mêmes soigneusement isolées. Les photocoupleurs usuels ont un temps de réaction d'une dizaine de microsecondes; ceci contribue à l'immunité aux parasites.

6.1.6 Bus d'instrumentation

La liaison de plusieurs instruments de mesure avec un processeur se fait souvent avec une interface série ou parallèle, avec une interface pour chaque instrument. La notion de bus vue au chapitre 5 peut s'appliquer pour permettre de lire, à tour de rôle, les valeurs calculées par divers instruments connectés sur un même chemin de données. Contrairement au bus du processeur, un bus d'instrumentation peut être un peu plus long (20 mètres).

Le bus standardisé IEEE 488 [80] possède un bus 8 bits pour les ordres, adresses et informations. Huit lignes de contrôle génèrent la synchronisation des échanges, la signification de l'information placée sur le bus, et permettent l'équivalent des interruptions vectorisées.

Pour recueillir l'information sur une distance plus grande, à l'intérieur d'une usine par exemple, des bus et anneaux série sont utilisés [80]. La transmission de l'information par fibre optique évite les problèmes de perturbation électrique vus au paragraphe précédent.

6.1.7 Mémoires auxiliaires

La mémoire propre du système doit presque toujours être prolongée par une possibilité de mémoriser l'information sur un support bon marché et peu destructible.

Contrairement à la mémoire centrale, qui a un *accès aléatoire*, c'est-à-dire que chaque position peut être accédée dans un temps constant, les mémoires auxiliaires ont un *accès séquentiel*, l'information étant déposée en série sur le support, et devant être relue dans un certain ordre avec un temps d'attente essentiellement variable.

L'information est toujours structurée sous forme de *fichiers* (*file*), comportant en plus de l'information proprement dite les éléments caractérisant cette information; type de fichier, date de création, taille, etc. Cette information associée est créée automatiquement par le système.

Une évolution très rapide a lieu dans les domaines des mémoires magnétiques et optiques. Les capacités augmentent et les prix diminuent sensiblement. On distingue les *disques souples* (*floppy disks*) d'une capacité de 100 koctets à 2 Moctets, les *disques durs* (*hard disk, winchester*) d'une capacité de 5 Moctets à 500 Moctets, les *disques optiques* d'une capacité de 2 à 10 Gigaoctets. Les disques optiques sont caractérisés par le fait que l'on ne peut pas effacer l'information écrite; ils conviennent spécialement pour l'archivage de l'information.

L'information stockée sur une mémoire auxiliaire doit être sauvée régulièrement sur une autre mémoire auxiliaire, pour éviter des pertes définitives d'informations précieuses en cas de panne grave de la mémoire. Des *bandes magnétiques*, en cassettes ou en bobines, sont souvent utilisées dans ce but.

Les mémoires à *bulles magnétiques*, utilisant le principe de charges magnétiques se déplaçant sur la surface d'un support convenablement préparé ont un avenir incertain. Elles sont par contre très intéressantes comme mémoires non volatiles dans un système informatique portatif.

6.1.8 Fichiers disques

Avec tous les disques, une tête de lecture est déplacée radialement et une centaine de *pistes* concentriques portent l'information. Le temps d'accès à l'information dépend de la vitesse de déplacement radiale et de la vitesse de déplacement angulaire. Des impulsions de synchronisation à chaque révolution permettent de découper le disque en quelques dizaines de *secteurs*. Un *bloc* d'information (128 à 512 octets) existe à l'intersection d'une piste et d'un secteur.

Un fichier est mémorisé sur un disque sous forme d'un ensemble de blocs. A chaque bloc correspond une adresse de piste et une adresse de secteur. Les caractéristiques des différents fichiers mémorisés sur le disque sont réunies dans un fichier séparé, appelé *répertoire*. Trois types d'organisations de fichiers se rencontrent dans les systèmes actuels.

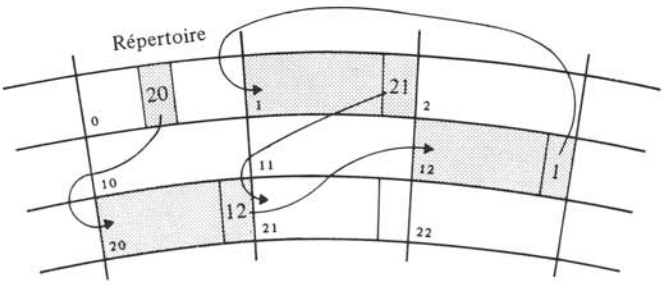


Fig. 6.1

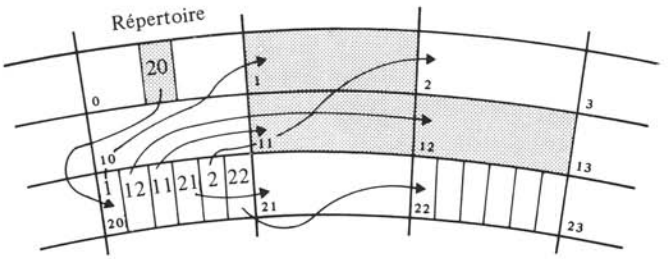


Fig. 6.2

Les *fichiers séquentiels* sont formés d'une liste de blocs chaînés (fig. 6.1). Le passage d'un bloc au suivant implique en général un changement de piste et l'attente d'une révolution.

Les *fichiers à accès aléatoire* sont formés d'une part d'un fichier séquentiel formé de pointeurs et d'autre part, de l'ensemble des blocs pointés. Le passage d'un bloc au suivant implique en principe la lecture de deux blocs, mais le bloc des pointeurs est conservé en mémoire après son premier transfert, afin d'accélérer les transferts suivants.

Les *fichiers contigus* sont simplement formés par des blocs consécutifs (fig. 6.3). Leur lecture est très rapide, puisque les déplacements de tête et attentes angulaires n'existent pas. Par contre, l'insertion d'une nouvelle information et la récupération des blocs inutilisés impliquent des transferts de quantités importantes d'informations.

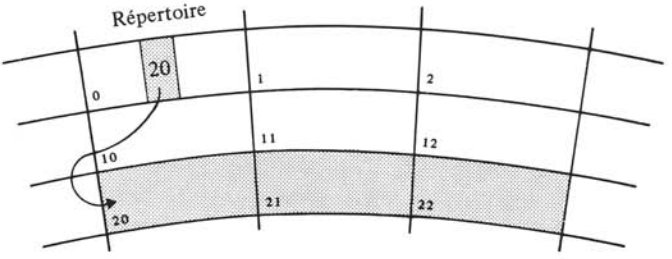


Fig. 6.3

Sur disque souple et bande magnétique, les fichiers sont toujours contigus. L'utilisation de fichiers séquentiels ou aléatoires imposerait des temps excessifs. La réorganisation de l'information lorsque le disque contient, après une certaine période d'utilisation, de nombreux trous de petite taille, implique une recopie de toute l'information.

6.2 SYSTÈME MICROPROCESSEUR

6.2.1 Evolution

Le premier microprocesseur date de 1971. Depuis lors, le nombre de processeurs a augmenté très rapidement.

Cette multiplicité dans le nombre de microprocesseurs est due partiellement à la variété des applications et de leurs contraintes, et principalement à l'évolution des archi-

tectures et de la technologie. Tous les trois ans environ, un nouveau dessin des circuits permet de façon substantielle :

- d'augmenter les performances (fonctions, vitesse, fiabilité);
- de simplifier la liaison du processeur à son environnement;
- d'abaisser les prix.

L'évolution de la technologie (nombre total de transistors) permet également, avec à peu près la même périodicité, de s'élever dans le niveau de complexité des architectures et des applications correspondantes:

- processeurs 4 bits (applications industrielles simples);
- processeurs 8 bits (applications générales, ordinateurs personnels);
- processeurs 16 bits (domaine d'application des miniordinateurs);
- processeurs 32 bits (domaine d'application des ordinateurs moyens).

Les performances des interfaces et périphériques évoluent de la même façon et conduisent à des réductions impressionnantes du prix et de l'encombrement des installations.

De nombreuses applications nouvelles sont rendues possibles. En particulier, la logique câblée est remplacée par la logique programmée et les éléments mécaniques permettant de mémoriser, de traiter ou de transporter de l'information (cames, leviers, etc.) sont appelés à disparaître lorsqu'aucune fonction énergétique ne doit leur être associée.

La mise en oeuvre d'un système dans lequel un ou plusieurs microprocesseurs sont incorporés implique des connaissances générales qui font l'objet de ce livre et des connaissances particulières sur les circuits et les techniques de mise au point associées.

6.2.2 Processeur idéal

Il est facile d'imaginer un processeur idéal. Du point de vue de son architecture interne, il doit disposer:

- de très nombreux registres de calcul et pointeurs;
- d'une unité arithmétique permettant des opérations binaires et décimales, en virgule fixe et en virgule flottante, avec de nombreux indicateurs de test et d'erreur;
- d'un répertoire d'instructions complet permettant tous les transferts et toutes les opérations arithmétiques, ainsi que tous les sauts, appels de routines et structures de commande;
- de tous les modes d'adressage, et de possibilités d'exécution de calculs complexes au moment de l'adressage de processus multiples;
- de facilités de changement de contexte et de synchronisation;
- de primitives permettant l'écriture de compilateurs efficaces.

Du point de vue de son architecture externe, il doit disposer:

- d'un bus d'adresses, d'informations et de contrôles permettant d'adresser au travers d'une unité de gestion mémoire toutes les mémoires et périphériques voulus et autorisant l'interruption, l'accès direct en mémoire et le dialogue avec d'autres processeurs;
- d'une mémoire interne et d'interfaces permettant de résoudre toutes les applications simples, avec comme seuls éléments extérieurs des amplificateurs de puissance liés à des éléments électromécaniques.

Du point de vue de ses caractéristiques électriques, il doit avoir:

- une consommation de puissance réduite, permettant le fonctionnement temporaire ou permanent sur batterie;
- une tension d'alimentation unique, égale à 5 V de préférence;
- des signaux d'entrée et de sortie compatibles avec la technologie TTL;
- une vitesse élevée restant compatible avec la technologie des mémoires TTL et MOS et permettant l'utilisation de la technologie habituelle des circuits imprimés;
- une immunité au bruit très grande.

Du point de vue de ses caractéristiques physiques, il doit être:

- de petite taille tout en permettant une manipulation et un remplacement faciles;
- insensible aux chocs, à l'environnement et au vieillissement.

Du point de vue commercial enfin, le microprocesseur doit être:

- bon marché;
- facile à comprendre et à utiliser, donc à documenter;
- d'une durée de vie élevée (obsolescence faible);
- complété par tous les équipements de développement et les langages de programmation nécessaires.

Aucune de ces exigences n'est contradictoire. Les efforts des physiciens, des électroniciens et des informaticiens tendent vers cet idéal.

6.2.3 Types de microprocesseurs

Deux contraintes principales empêchent les microprocesseurs actuels d'avoir des caractéristiques proches de l'idéal du paragraphe précédent. La première de ces contraintes est due au nombre maximum de transistors qu'une pastille de silicium peut contenir. Ce nombre, qui double tous les deux ans, a dépassé depuis quelques années le nombre minimum permettant de réaliser un processeur simple 16 bits, mais ne permet pas l'implémentation de toutes les caractéristiques souhaitées. Le grand nombre de processeurs actuellement disponibles s'explique par les nombreux compromis possibles.

L'autre contrainte est due au nombre de lignes d'entrée/sortie que l'on peut placer économiquement autour du circuit processeur. Ce nombre est de 40 à 120, mais dans des applications de grande série il y a avantage à le réduire. Ce nombre de connexions ne permet pas d'avoir à la fois un bus complet et des lignes d'interface pour des périphériques. Une nouvelle diversité des types de processeurs résulte de l'affectation des lignes. Cinq grandes catégories peuvent être considérées:

- les processeurs *orientés application* sont des ordinateurs complets et comportent sur le même circuit le processeur, la mémoire et les interfaces. On les appelle *microordinateurs monolithiques* et leur architecture est souvent proche de celle de Harvard [51];
- les processeurs *universels* simples doivent être complétés par de la mémoire et des interfaces. Les mots sont de 8 ou 16 bits et la mémoire est limitée à 32 ou 64 koctets [50, 51, 53];
- les processeurs universels *évolués* permettent la gestion d'une mémoire de grande dimension, la simultanéité de plusieurs tâches et une vitesse d'exécution élevée [14, 62];

- les processeurs *en tranche* permettent par justaposition, interconnexion et microprogrammation de définir une architecture propre adaptée à l'application et présentent de très grandes performances en vitesse [52];
- les *microcontrôleurs* effectuent quelques tâches simples faisant plus appel à des décisions qu'à des opérations arithmétiques [52] et sont utilisés dans des interfaces ou systèmes spécialisés.

6.2.4 Microordinateur monolithique

Les microordinateurs monolithiques sont utilisés pour résoudre d'innombrables applications comportant quelques touches de clavier et fins de course en entrée, un affichage, un haut-parleur ou quelques relais et moteurs en sortie. Ces systèmes peuvent être autonomes comme une machine à laver ou une calculatrice, ou être satellites d'un processeur plus puissant comme dans le cas d'un contrôleur de moteur pas à pas intelligent.

Une fois le programme de gestion de l'application au point, ce programme est figé dans une mémoire morte et copié en de très nombreux exemplaires. Le temps de développement peut être important, et la programmation de ces microordinateurs se fait en assembleur, en utilisant toutes les ressources du processeur de façon souvent peu structurée, afin d'obtenir le plus de fonctions possible avec la mémoire disponible.

6.2.5 Système microordinateur évolué

A l'opposé du système du paragraphe précédent, un système évolué comporte plusieurs cartes et peut s'adapter à des applications très complexes par augmentation de la taille mémoire, du nombre des interfaces d'entrée/sortie et de l'intelligence de celles-ci.

Ce système peut comporter plusieurs processeurs spécialisés ou universels, et être relié à des systèmes similaires au moyen de liaisons à haute vitesse, par exemple un bus parallèle de même nature que le bus du système [79].

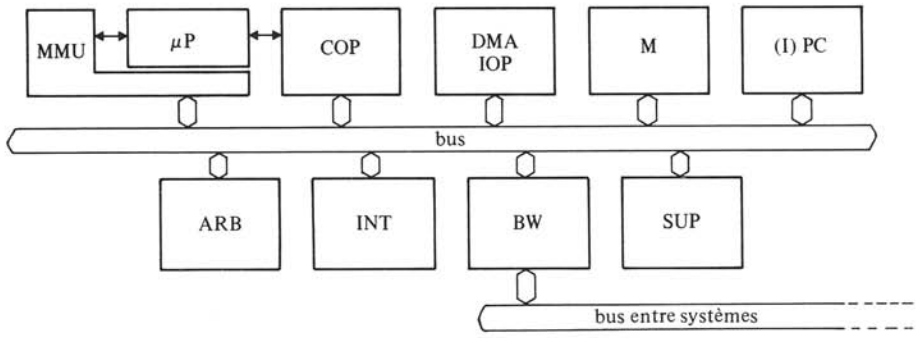


Fig. 6.4

Un système prend l'allure de la figure 6.4 et comporte les éléments suivants:

μP	microprocesseur. Traite l'information et sert les interruptions;
MMU	circuit de gestion de mémoire (memory management unit). Transforme les adresses logiques en adresses physiques en vérifiant les droits d'accès aux ressources du système;

COP	processeur esclave (coprocesseur). Remplace le processeur pour certaines opérations spécialisées (virgule flottante en particulier);
DMA, IOP	unité d'accès direct en mémoire et processeur d'entrée/sortie (direct memory access, input/output processor);
M	mémoire;
(I) PC	interface d'entrée/sortie ((intelligent) peripheral controller). Le terme intelligent signifie que cette interface comporte un processeur et une mémoire propre, programmable par l'utilisateur, mais non modifiable par le processeur principal du système;
ARB	arbitre. Attribue la ressource commune, le bus, au processeur ou à l'unité DMA qui en fait la demande, selon un schéma de priorité prédéfini;
INT	circuit d'interruption. Gère les demandes d'interruption des différents interfaces et processeurs d'entrée/sortie et présente au processeur une demande à la fois;
BW	fenêtre de communication entre bus (bus window). Interface mettant en communication le bus du système considéré et un bus de même type reliant d'autres systèmes microprocesseurs;
SUP	superviseur. Espionne le bus et enregistre (trace) son activité ou intervient pour modifier l'état ou l'évolution du système.

Sauf pour les deux derniers modules, des circuits intégrés existent qui réalisent la fonction mentionnée avec un nombre plus ou moins important de composants extérieurs. L'utilisation de circuits intégrés complexes caractérise les microordinateurs, par rapport aux miniordinateurs.

Le superviseur doit être réalisé dans une technologie très rapide, si on ne veut pas ralentir le système. L'interface entre deux bus n'est pas très différente d'une interface simple, si le bus demandeur est bloqué jusqu'à ce que l'autre soit libre et que la fenêtre entre les deux bus puisse s'ouvrir.

Remarquons encore qu'il faut bien distinguer une fonction qui doit être réalisée par le système, et l'emplacement où cette fonction se réalise. Les contraintes technologiques (nombre maximum de transistors et broches par circuit une année donnée, meilleur compromis prix/flexibilité), conduisent à distribuer différemment les fonctions dans des boîtiers, selon la complexité du système.

Par exemple, la complexité d'une unité arithmétique en virgule flottante force à l'utilisation d'un processeur esclave. Des processeurs futurs contiendront directement cette unité, mais le concept de processeur auxiliaire subsistera pour les fonctions plus spécialisées comme par exemple la transformée de Fourier rapide, réalisée actuellement avec des microprocesseurs en tranche et une logique encombrante.

6.2.6 Systèmes de développement

Le développement d'un programme pour microprocesseur peut se faire sur le système lui-même, si celui-ci dispose des périphériques de dialogue, de la mémoire et des programmes nécessaires. C'est le cas du système évolué de la figure 6.4, mais n'est jamais le cas des systèmes dédiés à une application simple traités au paragraphe 6.2.4.

Un système de développement doit alors être utilisé et peut être basé sur un processeur quelconque. Il doit au moins comporter un assembleur croisé pour le microprocesseur

choisi, et des facilités pour transférer les programmes objet assemblés du système de développement dans le système utilisateur.

Parmi les très nombreuses solutions proposées pour aider à la mise au point des programmes, ne retenons que trois approches.

La solution la plus économique procède par essais successifs, en observant l'effet du programme. Le programme est préparé sur le système de développement et transféré dans la mémoire du processeur de l'application par programmation directe d'une mémoire morte, ou par transfert dans une mémoire vive remplaçant la mémoire morte du système définitif. Le programme est ensuite corrigé jusqu'à ce qu'il ait le comportement correct. Un analyseur logique, qui enregistre la suite des instructions effectuées, est un outil indispensable pour permettre une certaine efficacité avec cette méthode tout-à-fait valable pour des petites applications [53].

La solution la plus fréquente et la plus efficace est l'utilisation d'un système de développement comportant un *émulateur en ligne* (*in circuit emulator*) qui substitue au processeur de l'application un matériel et logiciel complexe permettant d'effectuer toutes les opérations de l'application en pas à pas ou à vitesse normale, en disposant de toutes les facilités d'interaction et d'observation fournies par un clavier et écran. Des éléments de programme peuvent être exécutés séparément, afin de permettre une localisation rapide des défauts matériels et des erreurs de programmation [9]. Le recours à un système de développement permet de développer et tester le logiciel avant même que le matériel ne soit opérationnel. Pendant cette phase, les périphériques réels sont simulés par les périphériques du système de développement. L'intégration du logiciel et du matériel peut se faire progressivement, grâce au concept de l'émulateur en ligne: en premier, les périphériques simulés sont remplacés par les réels; ensuite la mémoire est transférée et ce n'est qu'en dernier lieu que le microprocesseur définitif est mis en oeuvre.

Une dernière solution est la simulation logicielle complète sur le système de développement. Ceci suppose que tous les paramètres de l'application ont été parfaitement définis et qu'un programme de simulation correct existe. Indispensables lorsqu'il s'agit de la conception d'un nouveau processeur intégré, les techniques de simulation sont appelées à se développer pour permettre une mise au point complète d'applications complexes, pour lesquelles on se contente trop souvent d'avoir observé un comportement correct dans les conditions d'utilisation usuelles.

6.2.7 Choix

Le choix d'un processeur ou d'un système de développement pour une application donnée est rendu difficile par la diversité des solutions possibles et la rapidité de l'évolution. Les principaux critères qui interviennent sont les suivants:

- nature de l'application;
- quantités prévues;
- durée de vie du produit souhaitée;
- expérience de l'équipe de développement;
- équipement à disposition pour la mise au point;
- langage de développement préféré;
- service après vente.

Le microprocesseur a rapidement gagné contre la logique câblée étant donné la plus grande facilité de mise en oeuvre due à un nombre restreint de composants, et étant donné surtout la possibilité de modifier et d'améliorer les spécifications jusqu'à un instant avancé dans la mise en fabrication. Il ouvre maintenant des applications nouvelles exigeant des capacités de traitement d'informations très grandes.

Une meilleure position de concurrence résulte du développement de produits ayant un comportement plus riche et mieux adapté à l'application. Malheureusement, l'évolution trop rapide de la technologie accélère l'obsolescence des produits et oblige tous les trois ans à reconcevoir complètement les systèmes réalisés en utilisant la dernière génération de processeurs.

La formation aux nouvelles techniques joue un rôle important; chaque technicien et ingénieur doit devenir informaticien, sans pour autant perdre ses compétences propres.

6.3 SYSTÈMES D'EXPLOITATION

6.3.1 Définition

Un *système d'exploitation* est formé par l'ensemble des programmes simplifiant la préparation et l'exécution des programmes de l'utilisateur. Dans un système simplifié à l'extrême, un système d'évaluation ne comportant qu'un clavier et affichage hexadécimal, le système d'exploitation se réduit à un moniteur entièrement résident en mémoire morte (taille de 2 à 4 octets). Ce moniteur permet d'examiner le contenu des positions mémoire, de modifier et d'exécuter les programmes ainsi créés. Une liaison série permet de charger des programmes depuis une cassette ou un autre système.

A l'autre bout du spectre, le système se décompose dans une pyramide de programmes dont seul le premier, le programme *amorce* (*bootstrap*) est en mémoire morte, et a pour but de permettre le chargement du noyau du système d'exploitation à partir d'un disque. Ce noyau comporte un ensemble de routines permettant la commande des périphériques, souvent abrégé *BIOS* (*basic input/output system*) et un programme exécutif permettant un premier dialogue avec l'utilisateur. Ce programme est souvent abrégé *CLI* (*Command Line Interpreter*) puisqu'il analyse les ordres tapés au clavier, sous forme de mots mnémotechniques et d'opérandes terminés par un retour de chariot.

Le système d'exploitation comporte un nombre important de programmes, transférés du disque en mémoire seulement lorsqu'ils sont nécessaires. Le système effectue lui-même les *permutations* (*swapping*). Il importe en effet de laisser suffisamment de mémoire pour l'utilisateur. Notons que par programme utilisateur, on comprend aussi l'éditeur, l'assembleur, etc. c'est-à-dire tous les programmes dont l'utilisateur peut demander lui-même le chargement.

Les programmes utilisateurs peuvent être plus grands que la place libre en mémoire. Ils doivent être dans ce cas segmentés. Une *partie commune* (*root*) doit subsister en mémoire et demande le chargement en mémoire de *morceaux superposables* (*overlay*) dont elle a besoin pour exécuter chaque partie de programme. Le programme peut ainsi être coupé en parties exécutées successivement en prenant garde qu'une partie commune à chaque *segment* ou *page* subsiste en mémoire pour assurer la communication avec le nouveau segment transféré du disque et les suivants. Cette technique s'appelle souvent *chaînage* du programme.

Une occupation mémoire typique est donnée dans la figure 6.5.

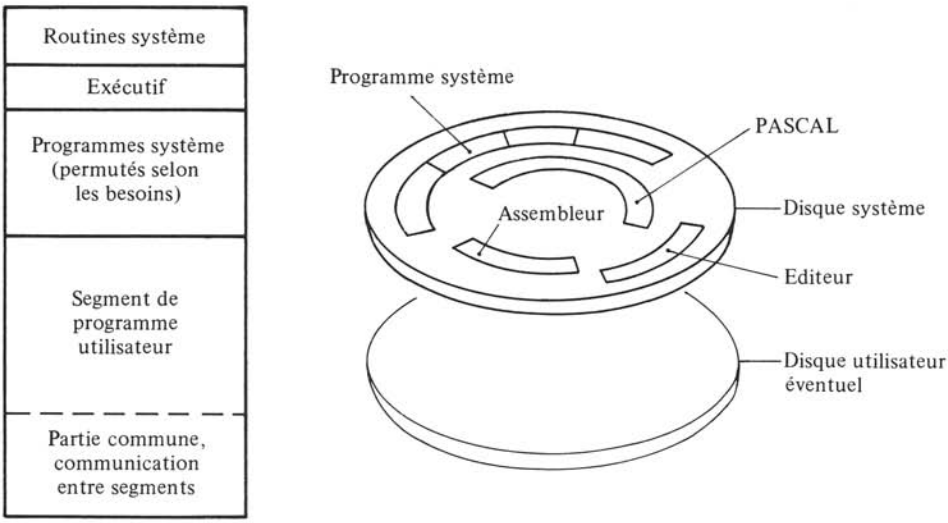


Fig. 6.5

Pour les petites applications et sur les systèmes récents disposant de très grande mémoire, il n'est pas nécessaire de segmenter les programmes de l'utilisateur et le système peut être entièrement résident.

La taille d'un système d'exploitation à disque est de l'ordre de 50 k octets, dans une variante relativement simple pour microprocesseurs ne comprenant pas des compilateurs de langages évolués; sur les plus gros systèmes, il atteint plusieurs millions d'octets.

6.3.2 Programme exécutif

Pour appeler les programmes existant sur disque, pour nommer, copier, vérifier ses fichiers, l'utilisateur tape des ordres interprétés par le *programme exécutif* (*filer*). La syntaxe comporte généralement une structure simple permettant de définir un ordre et le ou les noms des fichiers concernés. Des caractères *modificateurs* (*options*) permettent de préciser des conditions particulières. On trouve par exemple l'ordre

XFER FILE1 FILE2 ou FILE1 > FILE2 (6.1)

pour transférer le contenu d'un fichier FILE1 dans un fichier FILE2.

Au lieu de taper les noms, on peut faire apparaître un ensemble de noms sur l'écran, et choisir avec une souris par exemple les parties de l'ordre.

Le transfert dans un périphérique se fait avec le même ordre, étant donné l'identité entre la notion de fichier disque et le canal périphérique.

Par exemple, pour lire une bande papier on doit taper

XFER $PTR NEWFILE ou NEWFILE < PR: (6.2)

Un signal spécial peut distinguer les fichiers périphériques des fichiers disque.

Pour vérifier l'existence d'un fichier, l'ordre LIST avec des modificateurs éventuels

est utilisé. Par exemple, en tapant:

LIST FILE1	on obtient la dimension du fichier FILE1	
LIST T–	on obtient la liste et la dimension de tous les fichiers commençant par T;	
LIST/D T–	idem, avec en plus la date de création et de dernière modification;	(6.3)
LIST/A	on obtient la liste de tous les fichiers par ordre alphabétique;	

L'édition d'un programme peut se faire en tapant

EDIT FILE1 (6.4)

L'assemblage de ce programme, avec sauvetage de la table de symboles et un listage complet se note

AS FILE1/X/L (6.5)

Cette notation compacte est possible grâce aux extensions des fichiers et peut se taper de façon complète

AS FILE1.SR FILE1.ST/X FILE1.LS/L (6.6)

ce qui exprime que l'assembleur doit lire le fichier source FILE1. SR et créer le fichier table de symboles FILE1. ST (pour utilisation dans un . REF FILE1) et le fichier de listage FILE1. LS.

Ces quelques exemples ne correspondent pas tous à un système existant. Ils illustrent les différentes façons d'exprimer les paramètres associés aux ordres et sont complétés dans chaque système par des facilités supplémentaires permettant d'exécuter des suites d'ordres dépendant de certaines conditions [14].

6.3.3 Partage des ressources

Plusieurs utilisateurs se partagent fréquemment le même système, soit en se succédant sur le même clavier, soit en travaillant simultanément sur des ressources communes comme le disque et les périphériques.

Au niveau du disque, les noms des fichiers d'un utilisateur sont regroupés dans un fichier *répertoire* (*directory*), parfois subdivisé en sous-répertoires. Un mot de passe peut protéger l'accès à un répertoire, donc aux fichiers d'un utilisateur.

Au niveau des périphériques, par exemple l'imprimante, les fichiers à imprimer sont accumulés par le système d'exploitation dans une queue d'attente sur disque, et imprimés au fur et à mesure avec une en-tête caractérisant la tâche ou l'utilisateur.

L'utilisation de ressources communes impose la réservation préalable et la libération de la ressource. Lors de l'appel système réservant l'imprimante par exemple, un paramètre est rendu par le système à l'utilisateur pour lui indiquer si l'imprimante est libre, utilisée, non connectée, etc.

Si le processeur doit se partager entre plusieurs utilisateurs et/ou tâches (par exemple un seul utilisateur qui veut continuer à éditer pendant qu'un gros programme s'assemble), un programme appelé *moniteur de tâches* ou *superviseur* (*scheduler*) commute d'un processus à l'autre en fonction des interruptions et de points de commutation prévus dans chaque programme, de façon à donner l'illusion de la simultanéité aux utilisateurs du système.

Chaque tâche peut se trouver dans trois états différents [74]:

- en exécution. La tâche dispose du processeur;
- prêt ou dormant. La tâche attend que le processeur lui soit attribué;
- suspendu. La tâche attend qu'une ressource demandée (périphérique) soit libre.

Le moniteur de tâche gère ces états et des appels système permettent de créer et éliminer des tâches, selon les besoins des programmes lancés par l'utilisateur et par les tâches exécutées.

Plusieurs systèmes ne permettent que deux tâches simultanées: une tâche importante dite de *premier plan* (*foreground*), comme par exemple un programme d'application avec des contraintes de temps de réponse, et une tâche secondaire, dite de *second plan* (*background*), comme par exemple l'édition de texte, la traduction de programmes ou la réorganisation des fichiers disque.

Les tâches de second plan, dont les durées d'exécution ne sont pas critiques sont souvent utilisées sous forme de *lots de tâches* (*batch*) exécutées lorsque le processeur en a le temps.

6.3.4 Système en temps réel

Un contrôle de l'exécution de nombreuses tâches dont l'exécution dépend du temps nécessite un système dit *en temps réel* (*RTOS: real time operating system*). Un tel système permet de planifier les tâches en fonction du temps et des demandes, d'assigner des priorités aux interruptions, de transférer des données et paramètres d'une tâche à une autre, et de réserver des zones mémoire pour chacun des processus simultanés. Il doit être conçu de façon qu'aucun programme ne puisse interférer avec l'exécution des autres tâches. De nombreux circuits de protection, en particulier un système de *gestion des accès mémoire* (*memory management*) sont nécessaires. On a vu que l'évolution de la technologie permet, même au niveau d'un système microprocesseur, de disposer de toutes ces facilités (sect. 3.8).

6.3.5 Protection

Dans la mesure du possible, le système doit être protégé des erreurs des utilisateurs. Il doit vérifier les paramètres qui lui sont transmis et signaler de façon explicite les erreurs découvertes. La mise au point des programmes de l'utilisateur peut en être grandement facilitée.

Des circuits supplémentaires permettent, dans les miniordinateurs évolués et dans quelques microprocesseurs récents, de protéger certains périphériques et zones mémoire en interdisant leur accès à l'utilisateur. Le processeur a deux modes de fonctionnement, le mode système, sans protection et le mode utilisateur avec inhibition de certaines instructions, automatiquement rétablies lorsque l'interruption système est servie. Certains processeurs ont jusqu'à quatre modes de fonctionnement [55].

6.3.6 Editeur

L'éditeur est un programme très important qui permet la préparation des programmes et textes.

Le programme *éditeur* permet la création d'un fichier texte à partir des lettres frappées au clavier et par l'insertion d'éléments de texte préexistants. La configuration qui est nécessaire dans ce but est un terminal de dialogue, un processeur quelconque et une mémoire auxiliaire de taille suffisante.

Trois types d'éditeurs existent. Leur philosophie différente doit être bien présente à l'esprit lors de leur utilisation, car elle justifie la syntaxe des ordres et les limitations par rapport aux techniques naturelles crayon–papier–classeur. La nécessité de taper sur le même clavier le texte à mémoriser et les ordres à effectuer (transferts, recherches, corrections) impose des règles particulières d'utilisation.

6.3.7 Editeur pour terminal imprimant

Les éditeurs anciens sont conçus pour travailler avec une imprimante ou un écran qui simule une imprimante.

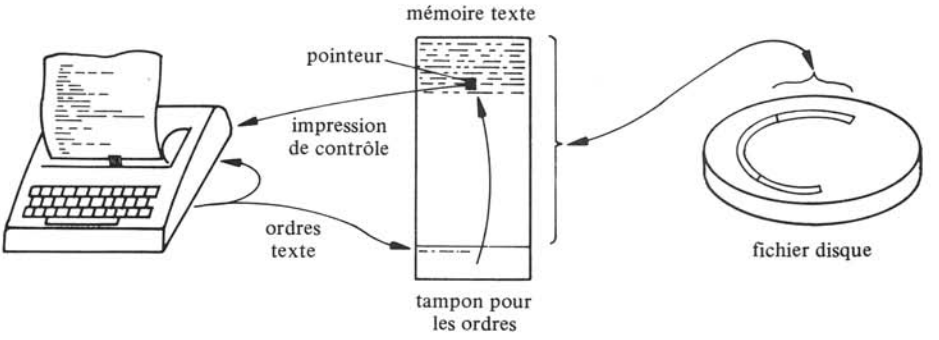


Fig. 6.6

Avec ces éditeurs, chaque caractère tapé est imprimé et transféré dans une zone temporaire en mémoire. En effet pour pouvoir distinguer les ordres et les textes à insérer et permettre une correction plus facile de certaines erreurs de frappe, chaque demande d'action est mémorisée temporairement avant d'être exécutée. Chaque ordre, y compris l'ordre d'insertion, est formé d'une chaîne de caractères comprenant une lettre caractéristique et est terminé par exemple par une double action sur la touche ESCAPE.

Le diagramme de syntaxe de la figure 6.7 correspond aux ordres principaux de l'éditeur du système RT11.

Pour insérer un mot ou une phrase, il faut taper Itexte$$ (la touche ESCAPE donne en écho le signe $). Un pointeur indique l'emplacement du prochain caractère qui sera inséré. Pour vérifier l'emplacement de ce pointeur invisible, et pour le déplacer, un certain nombre d'ordres sont disponibles. Le texte aussi est invisible. Son impression complète étant en général trop longue, des ordres permettent une impression partielle.

Un mot donné peut se trouver en tapant Gmot-cherché $$; le pointeur se trouve après exécution à la fin de la première occurence du mot cherché. Pour le modifier, il faut tuer un nombre de lettres égal aux lettres du mot et insérer le nouveau mot. D'autres éditeurs permettent des ordres de changement plus commodes.

La double touche ESCAPE à la fin de chaque ordre se justifie par la possibilité de générer des ordres comportant plusieurs commandes, séparées par un seul ESCAPE.

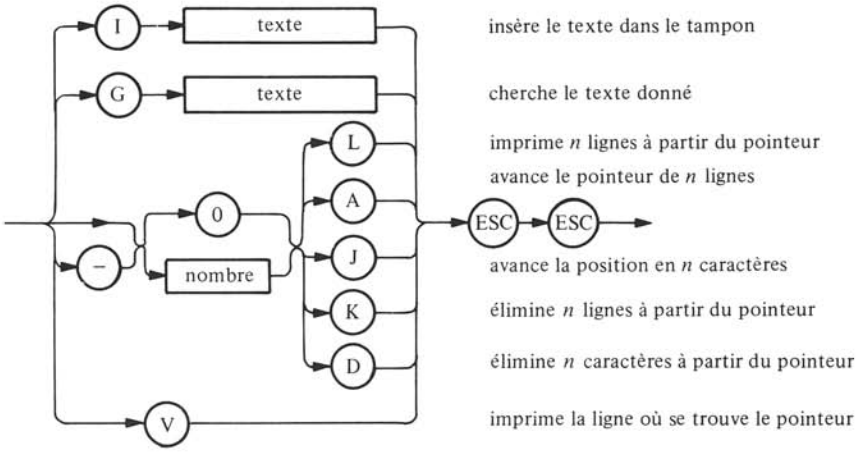


Fig. 6.7

Par exemple, pour remplacer le prochain mot TRUC par le mot CHOSE, il est possible de taper GTRUC$–4D$ICHOSE$V$$. Le dernier V a pour effet d'imprimer la ligne modifiée pour vérification.

Des ordres permettent le transfert de la mémoire texte sur disques. Lorsque le texte est long, il ne peut pas résider entièrement dans la mémoire principale. Il doit alors être décomposé en pages de quelques milliers de caractères transférées successivement.

6.3.8 Editeur de ligne

Si chaque ligne est numérotée et que la correction d'un caractère dans une ligne implique la frappe de la ligne complète, l'éditeur devient très simple. La plupart des interpréteurs interactifs, comme le BASIC, incorporent un tel éditeur de ligne.

La recherche d'un mot n'est généralement pas possible dans un éditeur de ligne. Si l'on dispose d'une copie imprimée du fichier, avec ses lignes numérotées, la recherche d'une ligne donnée et sa subsitution est très facile à décrire. La notion de pointeur n'est pas nécessaire, car l'insertion est facile si les lignes sont numérotées de 10 en 10 par exemple: il suffit de taper la nouvelle ligne précédée d'un numéro précisant sa position. Sur demande, les lignes sont renumérotées pour permettre de nouvelles modifications.

6.3.9 Editeur pour écran alphanumérique

L'utilisation d'un écran alphanumérique permet d'éviter des demandes continuelles d'impression des lignes entourant le pointeur. Elles sont en permanence affichées sur l'écran et le pointeur peut être visualisé par un signe spécial clignotant (fig. 6.8).

L'insertion et les déplacements de pointeur peuvent être indirects ou directs. Des touches supplémentaires, non utilisées en insertion sont alors nécessaires pour coder les différents ordres. Elles peuvent agir soit par effet de simultanéité, soit par effet de séquences.

Avec les touches de *simultanéité*, la touche pressée modifie la signification de toutes les autres touches. C'est le cas de la touche majuscule (SHIFT) des machines à écrire. Les

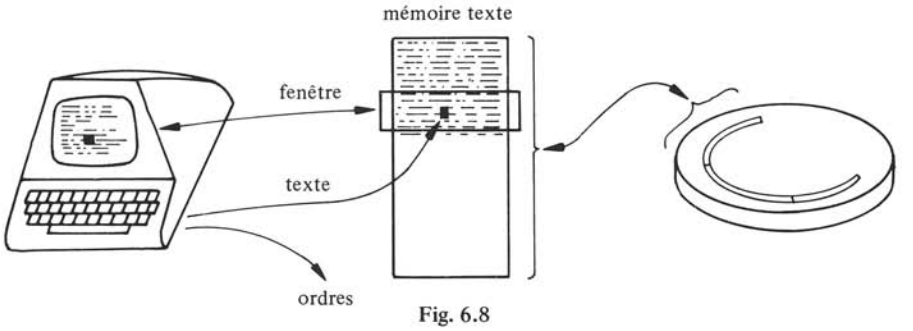


Fig. 6.8

claviers d'ordinateurs ont en plus une touche permettant de générer des codes supplémentaires (CONTROL). Rien n'empêche d'ajouter des touches supplémentaires, mais il est évident qu'un code de plus de 8 bits doit alors être généré pour représenter les nouvelles combinaisons créées.

Les *touches de séquence* doivent être mémorisées par le programme et modifient l'effet des touches du clavier jusqu'à la touche suivante, ou jusqu'à une touche donnée.

La figure 6.9 donne une représentation inspirée des diagrammes de syntaxe illustrant un éditeur ayant 3 touches de simultanéité marquées KILL, TEXT et SEARCH et une touche de séquence marquée DEF. Les touches C, R, F, et D sont repérées par des flèches. Pour déplacer le pointeur à droite, il faut maintenir TEXT pressé et agir sur la touche F. Pour tuer les caractères rencontrés dans ce déplacement, la touche KILL doit être également pressée en plus.

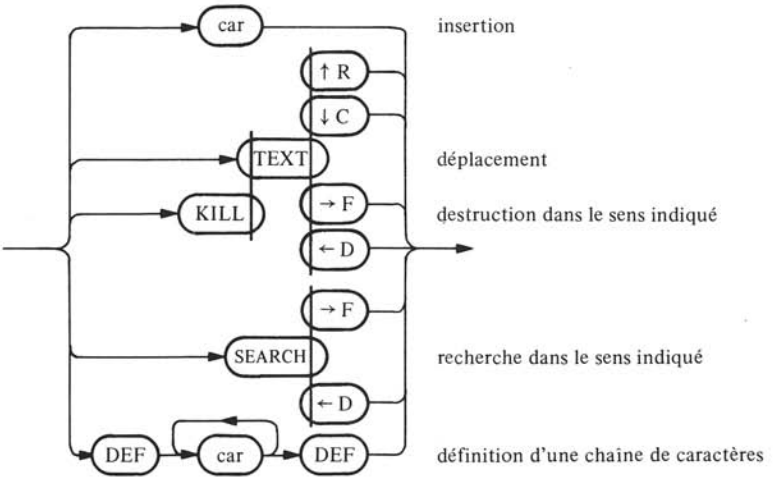


Fig. 6.9

Ceci permet au programme d'édition d'être en permanence dans le mode d'insertion, comme une machine à écrire. Le problème se complique toutefois pour la recherche d'un mot. Le mot tapé ne doit pas apparaître dans le texte. La touche DEF permet de sortir du mode d'insertion usuel, de placer les caractères tapés dans une zone spéciale de l'écran, et de retourner au mode normal en tapant à nouveau sur la touche DEF. La recherche du

mot tapé peut alors se faire facilement dans la direction désirée avec la touche de simultanéité SEARCH.

L'utilisation d'une souris et de menus sur l'écran facilitent et accélèrent la manipulation.

6.3.10 Autres programmes utilitaires

En plus des programmes décrits dans cette section et dans la prochaine, un système d'exploitation comporte un grand nombre de programmes de test, de transformation de l'information et de *commandes des périphériques* (*drivers*). La multiplicité des formats, protocoles, jeux de caractères, habitudes des utilisateurs, force à l'écriture de programmes multiples facilitant la tâche des utilisateurs, ces programmes étant tantôt incorporés dans le système ou appelés comme programmes indépendants.

6.4 LANGAGES ÉVOLUÉS

6.4.1 Introduction

Ce livre insiste sur la programmation en assembleur, qui permet de commander directement et efficacement toutes les ressources de la machine, et sa lecture doit être complétée en particulier par l'étude des langages évolués.

Deux raisons motivent l'utilisation de langages évolués. D'une part, la programmation d'un système en assembleur nécessite une connaissance détaillée de l'architecture du système, et de l'action de chaque bit des interfaces. D'autre part, les programmes écrits pour un processeur doivent être récrits pour la génération suivante de processeurs.

Dans le premier cas, les routines système en assembleur évitent à chaque utilisateur la compréhension détaillée des interfaces, mais pour rendre les programmes portables d'une machine à une autre, il faut cacher à l'utilisateur tout ce qui dépend d'un processeur donné. En fonction de l'application, il faut de plus fournir à l'utilisateur les types de données qu'il a appris à utiliser, comme les nombres réels (en virgule flottante) et ne pas le forcer à lire le chapitre 2 de ce livre pour comprendre les problèmes d'extension de signe et de comparaison de nombres arithmétiques et logiques.

Une tendance très forte favorise actuellement les langages dits bien structurés comme le Pascal [67] et ses dérivés [68, 69, 71]. Ces langages facilitent, et partiellement obligent, une programmation structurée. Toutefois, la structuration générale d'un programme est peu liée au langage finalement utilisé. Il a été montré abondamment dans le chapitre précédent que même un programme en assembleur peut être bien structuré. Inversément, le Pascal disposant de l'instruction GOTO, il est possible de mal structurer un programme!

L'utilisation d'un langage évolué permet d'élever le niveau d'abstraction des instructions. Les notions de registres, positions mémoire, adressage, disparaissent et le programmeur peut travailler directement avec des entités plus évoluées telles que des nombres en virgule flottante, des vecteurs, des piles. L'existence de telles *structures de données* accèlère considérablement la programmation et minimise les risques d'erreur.

En programmation structurée, les notions d'étiquettes (adresse d'instruction) disparaissent. Le programmeur ne dit pas où le programme doit continuer, mais ce qu'il doit faire. Des structures de commande adéquates (sect. 4.7) permettent un séquencement facile des actions.

Une plus grande lisibilité des programmes en résulte d'où une plus grande facilité de maintenance. Enfin, le programme est indépendant de la machine et peut être transporté sur toute machine disposant d'un compilateur ou d'un interpréteur pour ce langage.

Les langages évolués sont souvent appelés langages orientés problèmes et la multiplicité des problèmes à résoudre a généré, et génère encore, une multiplicité de langages. Quelques langages ont acquis une bonne popularité, comme le FORTAN [65] auprès des ingénieurs non informaticiens, le COBOL [70] dans les applications commerciales, le PASCAL [67] et le C [68] auprès des informaticiens et le BASIC [66] auprès d'une large couche d'utilisateurs de tout âge, scientifiques ou non.

Des langages mieux adaptés pour des environnements industriels, comme PORTAL, CHILL ou ADA [69], se propagent progressivement, mais la relative complexité de ces langages ralentit leur mise en oeuvre dans les nombreux types de systèmes qui apparaissent sans cesse, donc leur accessibilité et la portabilité réelle de ces programmes.

Les langages évolués présentent les programmes comme des suites d'instructions utilisant les structures de commande vues à la section 4.7 et résumées au paragraphe 7.3.3.

Chaque bloc ou procédure a un nom. Les variables sont déclarées par des noms symboliques, parfois de 1 à 2 lettres seulement, et un type de variable (caractère, réel, tableau) doit être déclaré, pour que la place correspondante puisse être réservée en mémoire.

Les langages évolués sont soit *interprétés* c'est-à-dire exécutés au fur et à mesure de la lecture des lignes du programme source, soit *compilés*, c'est-à-dire traduits en instructions en langage machine, avant d'être exécutés. Une solution mixte compile dans un langage intermédiaire codé correspondant à une machine idéale manipulant exactement les types de données et structures de commande prévues par le langage. Un interpréteur traite ensuite le langage codé (appelé par exemple P-code dans le cas du Pascal) sur la machine hôte.

Pour chaque langage, plusieurs interpréteurs et compilateurs existent, avec des performances très variables concernant la vitesse d'exécution des programmes générés et des incompatibilités mineures ou majeures. Sur une machine donnée, le choix est beaucoup plus restreint, et l'installation d'un compilateur ou d'un interpréteur est souvent délicate, à cause de petites différences de configuration et de la relativement mauvaise documentation.

6.4.2 Interpréteurs

Un interpréteur peut être un programme relativement simple analysant une suite d'ordres mémorisée dans une table et exécutant les actions correspondantes. Tout se passe alors comme si les ordres étaient directement exécutés par une machine virtuelle formée du processeur et de son programme interpréteur. Pour cette raison, les interpréteurs sont parfois appelés *simulateurs* et sont utilisés dans ce cas pour effectivement simuler une machine sur une autre machine. Dans le cas bien connu des langages interprétés comme le BASIC, l'interpréteur BASIC proprement dit est complété par un programme interactif permettant d'introduire et de modifier des ordres dans la table et de mettre au point le programme. Ce programme analyse la syntaxe et signale les erreurs au fur et à mesure de l'introduction des ordres dans la table. Parfois, chaque ordre peut être exécuté individuellement pour contrôle immédiat. Ce programme interactif permet de démarrer l'interprétation de la table ainsi préparée, c'est-à-dire l'exécution du programme BASIC.

Une instruction du programme BASIC (STOP) redonne la commande au programme interactif.

Les interpréteurs présentent un très grand intérêt dans le contexte actuel. Ce sont des programmes de dimension raisonnable permettant de définir des langages orientés problèmes, très localement "évolués" et parfaitement adaptés à l'application visée. Les codes du programme interprété peuvent être très compacts et une certaine indépendance vis-à-vis du processeur peut être atteinte. En cas de changement de machine, seul l'interpréteur doit être modifié; les programmes en code interprété peuvent être conservés.

L'inconvénient des interpréteurs est naturellement leur relative lenteur et la place fixe utilisée en mémoire par l'interpréteur. Ce second inconvénient est très réduit si le programme interprété est long ou si l'interpréteur se présente sous forme d'un circuit mémoire morte de grande capacité. Ce circuit, associé au processeur, forme un processeur virtuel et on retrouve à la limite une parenté très grande avec les unités microprogrammées.

6.4.3 Exemple

Pour montrer les possibilités et la structure des interpréteurs, analysons l'exemple très simple d'un langage permettant de définir des mouvements rotatifs ou linéaires créés par un moteur pas à pas.

Imaginons que le problème de l'utilisateur peut se ramener à quelques opérations élémentaires:

- avancer d'un certain nombre de pas;
- reculer d'un certain nombre de pas;
- attendre un certain temps;
- exécuter l'un parmi dix ordres;
- arrêter les opérations.

Fixons à un maximum de 999 le nombre de pas et la durée des attentes (en centièmes de secondes) et choisissons une syntaxe pour les ordres (fig. 6.10).

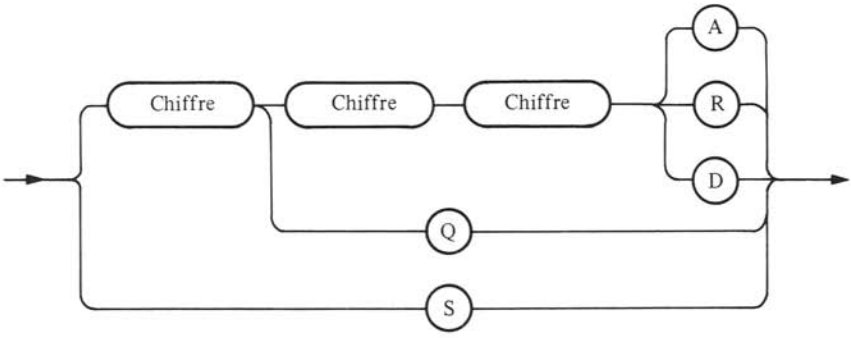


Fig. 6.10

En d'autres termes, les ordres suivants sont prévus:

- nnnA avance de nnn pas;
- nnnR recule de nnn pas;
- nnnD attente (delay) de nnn centisecondes;

- nQ sortie de l'ordre n;
- S arrêt.

Chaque ordre se terminant par une seule lettre, un terminateur spécial n'est pas nécessaire et un déplacement tel que celui représenté par la figure 6.11 peut se coder par le programme:

200A020D100R4Q200A030D2Q040D400R7Q010D100AS.

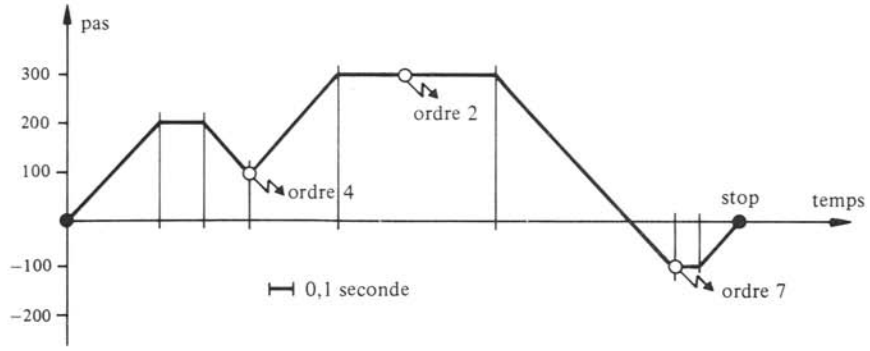


Fig. 6.11

La représentation de ce programme en mémoire peut se faire de différentes façons. Chaque chiffre ou lettre peut être représenté par son code ASCII. C'est la solution la plus simple, permettant l'adjonction de nouveaux ordres, tels que par exemple la répétition de certaines séquences. Si le nombre d'ordres est limité à 6, une compactification des chiffres et ordres sur 4 bits est possible. L'affectation de deux mots de 8 bits pour chaque ordre, y compris les ordres Q et S peut simplifier le programme de décodage (l'interpréteur) au détriment d'une place plus grande prise par le programme.

Un premier module de l'interpréteur doit permettre la lecture d'un ordre à la fois. Il peut s'appeler FETCH. Un compteur d'adresse virtuel pointe dans la table des ordres et transfère dans un registre d'instructions virtuel l'argument et le code de l'ordre. Celui-ci est alors interprété. Quatre modules principaux sont appelés lors de cette interprétation. On peut les appeler AVANCE, RECUL, DELAY et ORDRE. Ils reçoivent comme paramètre d'entrée l'argument de l'instruction, éventuellement préalablement converti de décimal en binaire. Les modules AVANCE et RECUL appelent eux-mêmes un module STEP, tenant compte des caractéristiques du moteur pas à pas. La programmation du coeur de cet interpréteur peut ainsi facilement se faire dans un langage quelconque. Le langage d'assemblage est toutefois recommandé, étant donné qu'un interpréteur est par essence relativement lent; l'écriture de l'interpréteur en langage évolué le ralentirait encore.

La préparation des suites d'ordres de déplacement peut se faire sur une installation quelconque disposant d'un éditeur de texte éventuellement spécialisé pour vérifier la syntaxe. Le transfert dans la mémoire s'effectue alors par un programme appelé *chargeur* qui effectue la conversion de code de compactification nécessaire. Ce programme peut résider dans le système et permettre une préparation interactive des programmes de déplacement. Complété par une possibilité d'exécution en pas à pas des déplacements et des ordres, un tel interpréteur est un complément essentiel du moteur pas à pas, faisant partie de l'installation au même titre que les amplificateurs du moteur.

6.4.4 Compilateur

Le travail d'un compilateur est sans commune mesure avec le travail d'un interpréteur simple. La compilation doit s'occuper de l'affectation des registres, de la conversion des structures de commande et de données, de l'analyse d'une syntaxe généralement riche et de l'optimisation du code produit.

Contrairement à l'assembleur, le compilateur génère pour chaque instruction en langage évolué un assez grand nombre d'instructions machine.

Il travaille selon 5 phases principales:

- lecture et conversion de la ligne source;
- analyse syntaxique;
- analyse sémantique;
- préparation du code;
- génération du code.

L'étape la plus délicate est la préparation du code, qui alloue la mémoire et remplace les instructions par des routines en s'efforçant d'optimiser le code localement et globalement. La dernière étape choisit les modes d'adressage et génère très souvent du langage d'assemblage plutôt que du binaire, afin de permettre certaines vérifications et faciliter la compatibilité [15].

6.4.5 Mélange de langages

Le compilateur crée à partir du source un code binaire translatable (§ 4.3.7) que le chargeur place en mémoire. A partir de plusieurs modules de programmes écrits dans différents langages, on peut donc charger le programme final comme on le fait en assembleur. La seule difficulté est le passage des paramètres qui ne peut se faire aussi facilement qu'en assembleur par des positions mémoire et registres, ces éléments étant inconnus des langages évolués.

Une solution peu élégante consiste à réserver néanmoins des positions fixes en mémoire et de permettre au langage évolué d'atteindre ces positions (ordres PEEK, POKE).

Une autre solution consiste à passer tous les paramètres sur la pile, avec l'avantage de la rentrance et de la récursivité. Une compatibilité doit naturellement exister dans le format des paramètres en mémoire et une connaissance assez profonde des mécanismes de compilation et d'édition du lien est nécessaire pour permettre le mélange de plusieurs langages. Chaque langage documente en général assez bien la liaison avec des parties écrites en assembleur, ce qui permet une bonne efficacité globale si toutes les parties critiques en temps d'exécution sont écrites en assembleur.

6.4.6 Conclusion

La science des calculatrices a pris une extension considérable en moins de 40 ans et forme à la fois une science en soi et l'un des domaines utiles à chaque science.

Nées grâce à l'electricité et ayant permis les progrès les plus spectaculaires de ce domaine, les calculatrices ont déjà vécu plusieurs réincarnations. La dernière, sous forme de système microinformatique doté de logiciel puissant, permet aussi bien au concepteur qu'à l'utilisateur de progresser plus rapidement grâce à des modèles toujours plus généraux.

Une bonne compréhension des notions de base reste toutefois nécessaire à celui qui veut pouvoir contribuer aux domaines de conception et d'application des calculatrices.

CHAPITRE 7

ANNEXES

7.1 NOMBRES

7.1.1 Puissance de 2

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 676 950 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125

Fairchild Camera and Instrument Corp., 464 Ellis St., Mountain View, Ca. 94042).

Fig. 7.1

7.1.2 Table des puissances de 10. en hexadécimal

		1	0	1,0000	0000	0000	0000	
		A	1	0,1999	9999	9999	999A	
		64	2	0,28F5	C28F	5C28	F5C3	$*16^{-1}$
		3E8	3	0,4189	474B	C6A7	EF9E	$*16^{-2}$
		2710	4	0,68DB	8BAC	710C	B296	$*16^{-3}$
	1	86A0	5	0,A7C5	AC47	1B47	8423	$*16^{-4}$
	F	4240	6	0,10C6	F7A0	B5ED	8D37	$*16^{-4}$
	98	9680	7	0,1AD7	F29A	BCAF	4858	$*16^{-5}$
	5F5	E100	8	0,2AF3	IDC4	611B	73BF	$*16^{-6}$
	3B9A	CA00	9	0,44BB	2FA0	9B5A	52CC	$*16^{-7}$
2	540B	E400	10	0,6DF3	7F67	5EF6	EADF	$*16^{-8}$
17	4876	E800	11	0,AFEB	FF0B	CB24	AAFF	$*16^{-9}$
E8	D4A5	1000	12	0,1197	9981	2DEA	1119	$*16^{-9}$
916	4E72	A000	13	0,1C25	C268	4976	81C2	$*16^{-10}$
5AF3	107A	4000	14	0,2D09	370D	4257	3604	$*16^{-11}$
3 8D7E	A4C6	8000	15	0,480E	BE7B	9D58	566D	$*16^{-12}$
23 8652	6FC1	0000	16	0,734A	CA5F	6226	F0AE	$*16^{-13}$
163 4578	5D8A	0000	17	0,B877	AA32	36A4	B449	$*16^{-14}$
DE0 B6B3	A764	0000	18	0,1272	5DD1	D243	ABA1	$*16^{-14}$
8AC7 2304	89E8	0000	19	0,1D83	C94F	B6D2	AC35	$*16^{-15}$

Fig. 7.2

7.1.3 Constantes en octal et en hexadécimal

π	=	3,110 375 524 21	=	3,243 F6A 888
π^{-1}	=	0,242 763 015 56	=	0,517 CC1 B70
e	=	2,557 605 213 05	=	2,B7E 151 628
e^{-1}	=	0,274 265 306 61	=	0,5E2 D58 D88
$\log_{10} e$	=	0,336 267 542 51	=	0,6F2 DEC 548
$\log_2 e$	=	1,342 521 662 45	=	1,715 476 528
$\log_2 10$	=	3,244 647 411 36	=	3,526 9E1 2F0
ln 2	=	0,542 710 277 60	=	0,B17 217 F80
ln 10	=	2,232 730 673 55	=	2,4D7 637 768
$\sqrt{2}$	=	1,324 047 463 20	=	1,6A0 9E6 680

Fig. 7.3

7.2 CODES

7.2.1 Code ASCII

ISO				0/	1/	2/	3/	4/	5/	6/	7/
	Hexa			0	10	20	30	40	50	60	70
		Octal		0	20	40	60	80	100	120	140
			Binaire	0000	0001	0010	0011	0100	0101	0110	0111
0	0	0	0000	NUL	DLE	space	0	@	P	`	p
1	1	1	0001	SOH	DC1	!	1	A	Q	a	q
2	2	2	0010	STX	DC2	"	2	B	R	b	r
3	3	3	0011	ETX	DC3	#	3	C	S	c	s
4	4	4	0100	EOT	DC3	$	4	D	T	d	t
5	5	5	0101	ENQ	NAK	%	5	E	U	e	u
6	6	6	0110	ACK	SYN	&	6	F	V	f	v
7	7	7	0111	BEL	ETB	'	7	G	W	g	w
8	8	10	1000	BS	CAN	(	8	H	X	h	x
9	9	11	1001	HT	EM	)	9	I	Y	i	y
10	A	12	1010	LF	SUB	*	:	J	Z	j	z
11	B	13	1011	VT	ESC	+	;	K	[	k	{
12	C	14	1100	FF	FS	,	<	L	\	l	|
13	D	15	1101	CR	GS	-	=	M	]	m	}
14	E	16	1110	SO	RS	.	>	N	^	n	~
15	F	17	1111	SI	US	/	?	O	_	o	DEL

NUL	Null	Nul
SOH	Start of Heading	Début d'en-tête
STX	Start of Text	Début de texte
ETX	End of Text	Fin de texte
EOT	End of Transmission	Fin de communication
ENQ	Enquiry	Demande
ACK	Acknowledge	Accusé de réception
BEL	Bell	Sonnerie
BS	Back Space	Retour Arrière
HT	Horizontal Tab	Tabulation horizontale
LF	Line Feed	Interligne
VT	Vertical Tab	Tabulation verticale
FF	Form Feed	Page suivante
CR	Carriage Return	Retour de chariot
SO	Shift out	Hors code
SI	Shift in	En code
DLE	Data Link Escape	Echappement transmission
DC1 XON	Device Control 1	Commande d'appareil aux
DC2	Device Control 2	
DC3 XOF	Device Control 3	
DC4	Device Control 4 (stop)	
NAK	Negative Acknowledge	Accusé de réception négatif
SYN	Synchronization	Synchronisation
ETB	End of Text Block	Fin de bloc de transmission
CAN	Cancel	Annulation
EM	End of Medium	Fin de support
SUB	Substitute	Substitution
ESC	Escape	Echappement
FS	File Separator	Séparateur de fichier
GS	Group Separator	Séparateur de groupe
RS	Record Separator	Séparateur d'article
US	Unit Separator	Séparateur de sous-article

Fig. 7.4

7.2.2 Code EBCDIC

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	NUL	DLE	DS		SP	&	-						{	}	\	0
0001	SOH	DC1	SOS				/		a	j	~		A	J		1
0010	STX	DC2	FS	SYN					b	k	s		B	K	S	2
0011	ETX	TM							c	l	t		C	L	T	3
0100	PF	RES	BYP	PN					d	m	u		D	M	U	4
0101	HT	NL	LF	RS					e	n	v		E	N	V	5
0110	LC	BS	ETB	UC					f	o	w		F	O	W	6
0111	DEL	IL	ESC	EOT					g	p	x		G	P	X	7
1000		CAN							h	q	y		H	Q	Y	8
1001		EM							i	r	z		L	R	Z	9
1010	SMM	CC	SM		cent	!	l	:								
1011	VT	CU1	CU2	CU3	.	$	,	#								
1100	FF	IFS		DC4	<	*	%	@								
1101	CR	IGS	ENQ	NAK	(	)	_	'								
1110	SO	IRS	ACK		+	;	>	=								
1111	SI	IUS	BEL	SUB	l	¬	?	"								

Caractères de commande

NUL	All zeros	RES	Restore	LF	Line Feed
SOH	Start Of Heading	NL	New Line	ETB	End Transm Block
SOT	Start Of Text	BS	Back Space	ESC	Escape
EOT	End Of Text	IL	Idle	SM	Set Mode
PF	Punch Off	CAN	Cancel	ENQ	Enquiry
HT	Horizontal Tab	EM	End of Medium	ACK	Acknowledge
LC	Lower Case	CC	Cursor Control	BEL	Bell
DEL	Delete	SP	Space	SYN	Snchronous/Idle
SMM	Start Manual Mess	IFS	Interchange File Sep	PN	Punch On
VT	Vertical Tabulation	IGS	Int. Group Separator	RS	Reader Stop
FF	Form Feed	IRS	Int. Record Separator	UC	Upper Case
CR	Carriage Return	IUS	Int. Unit Separator	EOT	End Of Transmission
SO	Shift Out	DS	Digit Select	NACK	Negative Acknowledge
SI	Shift In	SOS	Start Of Signifiance	SUB	Substitute
DLE	Data Link Esc	FS	Field Separator	DCi	Device Control 1,2,4
TM	Tape Mark	BYP	Bypass	CUi	Customer Control 1,2,3,

Fig. 7.5

7.2.3 Code Telex

Start	1	2	3	4	5	Stop	Lower case	Upper case
	o	o				o	A	-
	o			o	o	o	B	?
		o	o	o		o	C	:
	o			o		o	D	$
	o					o	E	3
	o		o	o		o	F	!
		o		o	o	o	G	&
			o		o	o	H	#
		o	o			o	I	8
	o	o		o		o	J	bell
	o	o	o	o		o	K	(
		o			o	o	L	)
			o	o	o	o	M	.
			o	o		o	N	,
				o	o	o	O	9
		o	o		o	o	P	0
	o	o	o		o	o	Q	1
		o		o		o	R	4
	o		o			o	S	'
					o	o	T	5
	o	o	o			o	U	7
		o	o	o	o	o	V	=
	o	o			o	o	W	2
	o		o	o	o	o	X	/
	o		o		o	o	Y	6
	o				o	o	Z	+
						o	blank	blank
	o	o	o	o	o	o	A...	A...
	o	o		o	o	o	1...	1...
			o			o	space	space
				o		o	CR	CR
		o				o	LF	LF

Start	1	2	3	4	5	Stop	Lower case	Upper case
						o	blank	blank
					o	o	T	5
				o		o	CR	CR
				o	o	o	O	9
			o			o	space	space
			o		o	o	H	#
			o	o		o	N	,
			o	o	o	o	M	.
		o				o	LF	LF
		o			o	o	L	)
		o		o		o	R	4
		o		o	o	o	G	&
		o	o			o	I	8
		o	o		o	o	P	0
		o	o	o		o	C	:
		o	o	o	o	o	V	;
	o					o	E	3
	o				o	o	Z	"
	o			o		o	D	$
	o			o	o	o	B	?
	o		o			o	S	bell
	o		o		o	o	Y	6
	o		o	o		o	F	!
	o		o	o	o	o	X	/
	o	o				o	A	-
	o	o			o	o	W	2
	o	o		o		o	J	'
	o	o		o	o	o	1...	1...
	o	o	o			o	U	7
	o	o	o		o	o	Q	1
	o	o	o	o		o	K	(
	o	o	o	o	o	o	A...	A...

Fig. 7.6

7.3 MICROPROCESSEURS

7.3.1 Répertoire d'instructions du Z80/8085

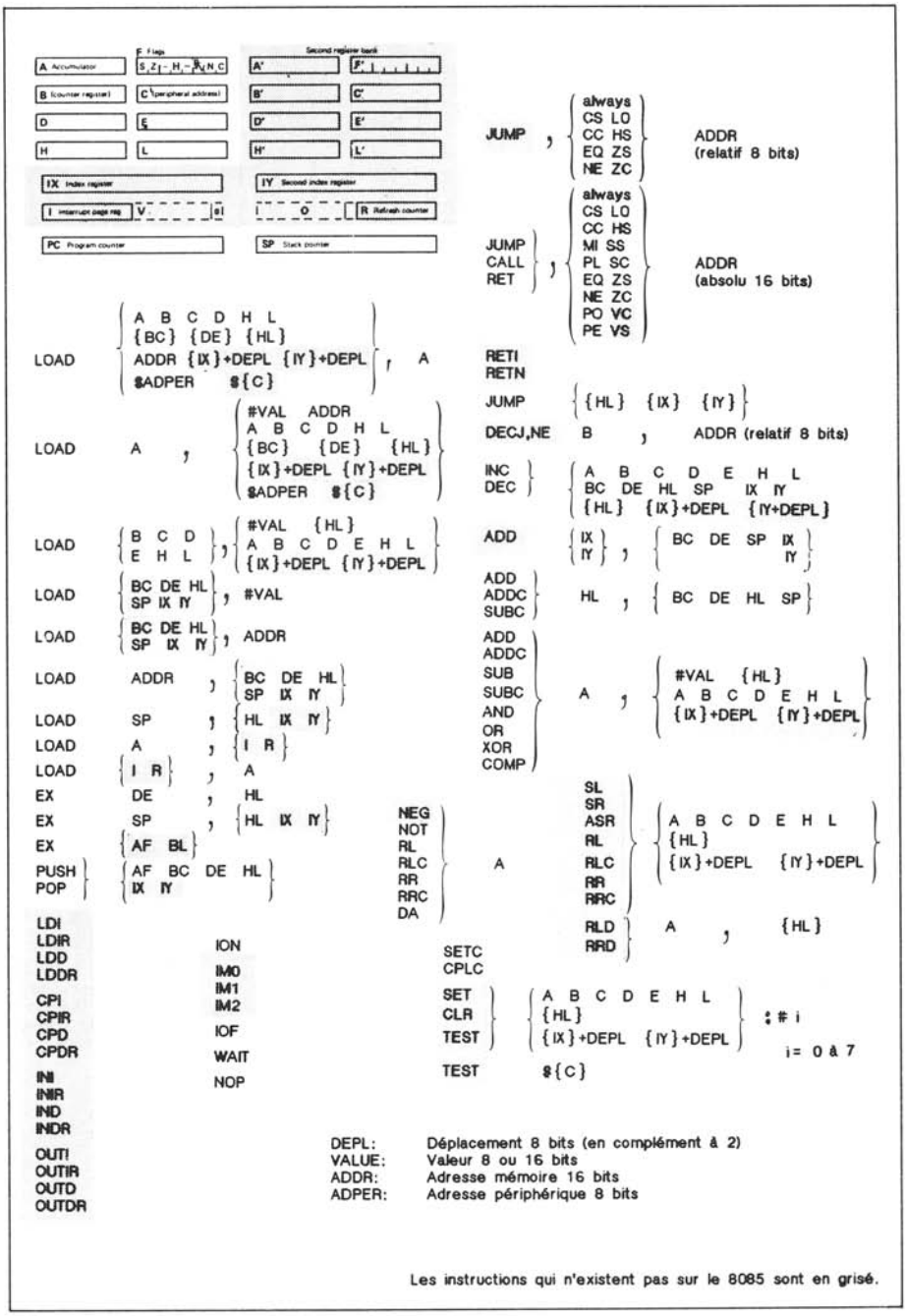
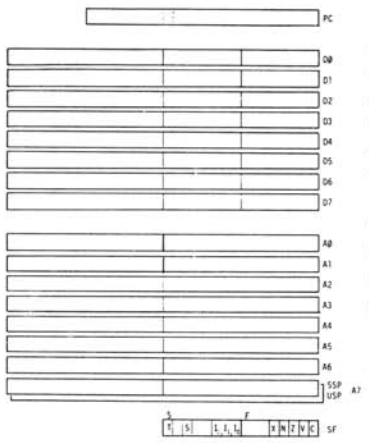


Fig. 7.7

7.3.2 Répertoire d'instructions du M68000



Di registre de donnée
Ai registre d'adresse

Modes d'adressage

{Ai} indexé simple
{Ai+} post-auto-incrémenté
{-Ai} pré-auto-décrémenté
{Ai}+DEPL indexé, déplacement 16 bit arith
{Ai}+{Rk}+DEPL indexé, déplacement dans Rk (A ou D) et déplacement suppl 8 bits arith
ADDR absolu (16 ou 32 bits)
RELAD relatif (16 bits arith)
RELAD+{Rk} indexé avec dépl 8 bits arith relatifs au PC
#VAL immédiat 3 à 32 bits (selon les instructions)

Pour chaque instruction, le nombre de modes est limité.

ad source: #VAL Ai Di {Ai} {Ai+} {-Ai} {Ai}+*DISP* *ADDR* *RELAD* *RELAD*+{Rk} {Ai}+{Rk}+*DISP*

ad destination : Di {Ai} {Ai+} {-Ai} {Ai}+*DISP* *ADDR* {Ai}+{Rk}+*DISP*

ad bit, ad amplitude : #VAL Di

ad saut : {Ai} {Ai}+*DISP* {Ai}+{Rk}+*DISP* *ADDR* *RELAD* *RELAD*+{Rk}

MOVE
ADD
SUB
COMP
AND
OR
XOR
→ .8 / .16 / .32 → ad source , ad destination

MUL
DIV
→ .16 / .A16 → ad source, Di

EX
SWAP
→ .16 / .32 → Ri,Rj / Di

ADDX
SUBX
→ .8 / .16 / .32 → ad source, ad destination

NEGX → .D8 → ad destination

PUSH
POP
SEX
INC
DEC
NEG
NOT
CLR
TEST
→ .8 / .16 / .32 → ad destination

TEST
TNOT
TSET
TCLR
→ .8 / .32 → ad destination : ad bit

ASR
ASL
SR
SL
RR
RL
RRX
RLX
→ .8 / .16 / .32 → ad destination (amplitude = 1) / ad amplitude , ad destination

X est fonctionnellement équivalent à C (carry)

DECJ.16,NMO Di, adresse
SKIP,cc DECJ.16,NMO Di,adresse

JUMP
CALL
→ ad saut

RET

cc code de condition

JUMP, → EQ NE SS SC VS VC → ad saut

JUMP, → LO LS HS HI → (nombre logique)

JUMP, → GT GE LE LT → (nombre arith en compl a 2)

Instructions spéciales (voir la documentation pour les types de données et les modes d'adressages permis)

MOVEM
PUSHM
POPM
MOVEP
CHECK.A16
LINK.32
UNLINK.32
SET,cc
TAS

RETF
RETSF
TRAP #valeur
TRAP,VS
TRAPA #valeur
TRAPF #valeur
RESET
MOVEWAIT #valeur,SF
NOP

Fig. 7.8

7.3.3 Structures de contrôle des langages évolués pour microprocesseurs

	. PASCAL	. MODULA II	. PORTAL	. C	PL/M	FORTRAN 77	. FORTH	. BASIC

m	BEGIN
	i1 ; i2;
	END ;
		i1 ; i2;	i1 ; i2;	{
				i1 ; i2;
				}
					i1 ;
					i2 ;
						i1
						i2
							i1
							i2
								i1
								i2

IF
THEN
ELSE
	IF c THEN m
	<ELSE IF c THEN m>
	[ELSE m]
		IF c THEN m
		<ELSIF c THEN m>
		[ELSE m] END ;
			IF c THEN m
			<ELSIF c THEN m>
			[ELSE m]
			END IF
				IF c m
				ELSE m
					IF c THEN m
					[ELSE ; ;]
						IF (c1) THEN
						m
						<ELSEIF c2 THEN
						m>
						[ELSE
						m]
						ENDIF
							c IF i ELSE j THEN	IF c THEN l
								
								IF c GOTO l

CASE	CASE v OF
	n1 : m
	n2 : m
	END ;
		CASE v OF
		n1 : i1
		n2 : i2
		[ELSE m]
		END ;
			CASE v
			OF n1 : m ;
			OF n2 : m ;
			ELSE m
			END CASE
				SWITCH v {
				CASE n1 : m [BREAK ;]
				CASE n2 : m [BREAK ;]
				[DEFAULT : m]
				}
						GOTO (l 1,l 2) v
						l 1 i1
						l 2 i2		

WHILE	WHILE c DO m	WHILE c DO m END ;	WHILE c DO m END WHILE	WHILE c DO m		DOWHILE c m ENDDO	BEGIN v IF i WHILE	
REPEAT	REPEAT m UNTIL c ;	REPEAT m UNTIL c ;		DO m WHILE c			BEGIN i v END	
LOOP		LOOP m EXIT END ;	LOOP m END LOOP ;					
FOR	FOR v := n1 TO n2 DO m FOR v := n1 DOWNTO n2 DO m	FOR v := n1 TO n2 [BY n3] DO m END ; FOR v := n1 DOWN n2 [BY n3] DO m END ;		FOR (v=n1 ; c ; i) m	DO v=n1 TO n2 [BY n3] ; m ENDDO	DOFOR v=n1, n2, n3 m ENDFOR	n1 n2 DO v LOOP	FOR v=n1 TO n2 [STEP n3] m NEXT v
LOOP complexe			LOOP [m] ON e [DO m] EXIT ... END LOOP					
JUMP	GOTO l				GOTO l	GOTO l	JMP l	GOTO l

c condition — i instruction isolée
l label — m suite d'instructions sur une ou plusieurs lignes
n nombre entier — [ ] optionnel (0 ou 1 fois)
v variable entière — < > répétition possible (0 à n)

Fig. 7.9

SOLUTIONS DES EXERCICES

CHAPITRE 2

2.1.8

```
H'57:     H'58  H'59  H'5A  H'5B  H'5C  H'5D  H'5E  H'5F
H'98:     H'99  H'9A  H'9B  H'9C  H'9D  H'9E  H'9F  H'A0
H'B9:     H'BA  H'BB  H'BC  H'BD  H'BE  H'BF  H'C0  H'C1
H'9F9:    H'9FA H'9FB H'9FC H'9FD H'9FE H'9FF H'A00 H'A01

O'57:     O'60   O'61   O'62   O'63
O'175:    O'176  O'177  O'200  O'201
O'657:    O'660  O'661  O'662  O'663
O'7774:   O'7775 O'7776 O'7777 O'10000
```

2.1.13 $6 * 10^{23}$ étant $\gg 1$, nous pouvons utiliser la formule 1.

$$k = \log_2 C = \log_2 10 \cdot \log_{10} C = \frac{\log_{10} C}{\log_{10} 2} = \frac{\log_{10} 6 \cdot 10^{23}}{\log_{10} 2} = 78{,}99 < 79$$

Il faudra donc un registre binaire d'au moins 79 cellules.

2.1.14 Le prix est donné par :

$p = k \cdot l \cdot p$ $\quad$ k = constante de proportionnalité
l = longueur du registre
p = base

On veut comparer les prix des registres de même capacité C. On a :

$l = \log_p C$ $\quad$ (formule 2.6)

$$p = k \cdot p \cdot \log_p C = k \cdot p \cdot \frac{\ln C}{\ln p}$$

Cherchons le minimum de cette fonction de p.

$$\frac{dP}{dp} = k \cdot \ln C \cdot \frac{d}{dp}\left(\frac{p}{\ln p}\right) =$$

$$= k \cdot \ln C \cdot \left(\frac{\ln p - p \cdot \frac{1}{p}}{\ln^2 p}\right) = 0$$

d'où

$$\ln p = 1$$

$$p = e$$

La base idéale est e = 2,7, mais il est évident que la base est un nombre entier.

Nous avons les rapports de prix suivants par rapport à cette base idéale:

Base 2: $\dfrac{k \cdot 2 \cdot \ln C}{\ln 2 \cdot k \cdot e \cdot \ln C} = \dfrac{2}{e \cdot \ln 2} = 1{,}0615$

Base 3: $\dfrac{k \cdot 3 \cdot \ln C}{\ln 3 \cdot k \cdot e \cdot \ln C} = \dfrac{3}{e \cdot \ln 3} = 1{,}0046$

Base 4: $\dfrac{4}{e \cdot \ln 4} = 1{,}0615$

Base 5: $\dfrac{5}{e \cdot \ln 5} = 1{,}143$

2.1.18 Si la base est $-p$ et les chiffres inférieurs à p, on peut écrire:

$$p = 1 \cdot (-p)^2 + (p-1) \cdot (-p)$$
$$p+1 = 1 \cdot (-p)^2 + (p-1) \cdot (-p) + 1 \quad \text{etc.}$$

et

$$-1 = 1 \cdot (-p) + (p-1)$$
$$-2 = 1 \cdot (-p) + (p-2) \quad \text{etc.}$$

Les nombres positifs peuvent s'exprimer avec un nombre impair de chiffres, et les nombres négatifs avec un nombre pair de chiffres.

En base -5, la correspondance entre les chiffres de -8 à $+20$ est:

base 10:	−8	−7	−6	−5	−4	−3	−2	−1	0
base −5:	22	23	24	10	11	12	13	14	0

base 10:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15...
base −5:	1	2	3	4	140	141	142	143	144	130	131	132	133	134	120...

L'avantage de ce système est la disparition du signe négatif. Un mécanisme d'addition peut être défini, mais il est un peu plus complexe qu'avec une base positive.

2.1.19 Si les chiffres sont:

$$-k, -k+1, \ldots, -1, 0, 1, \ldots l-1, l \quad \text{avec } k + l = p - 1$$

alors on peut écrire:

$$l+1 = 1 \cdot p + (-k)$$
$$-k-1 = -1 \cdot p + l$$

Par récurrence, on peut montrer que tous les entiers positifs et négatifs se laissent construire.

Dans la base proposée, on a pour équivalents des nombres de -10 à $+10$:

$-0-, -00, -0+, -+-, -+0, -++, --, -0, -+, -, 0,$

$+, +-, +0, ++, +--, +-0, +-+, +0-, +00, +0+$

Ce système pourrait être avantageux avec une électronique à trois états (+ 5 V, 0, − 5 V par exemple).

La complexité des opérateurs arithmétiques est plus grande et la fiabilité plus faible. Les essais faits dans ce sens ont été abandonnés.

2.1.24 Mantisse entière non normalisée:

$$\text{Max} = 999999 \cdot 10^{99} \cong 10^{105}$$
$$\text{Min} = 1 \cdot 10^{0} \cong 1$$

Mantisse entière normalisée:

$$\text{Max} = 999999 \cdot 10^{99} \cong 10^{105}$$
$$\text{Min} = 100000 \cdot 10^{0} \cong 10^{5}$$

Mantisse fractionnaire normalisée:

$$\text{Max} = 0{,}999999 \cdot 10^{99} \cong 10^{99}$$
$$\text{Min} = 0{,}100000 \cdot 10^{0} \cong 10^{-1}$$

2.2.6 On a comme codage lettre à lettre et résultat du calcul:

C	O	D	E		A	S	C	I	I	␣	␣
O′3	O′17	O′4	O′5	O′0	O′1	O′23	O′3	O′11	O′11	O′0	O′0
O′12434			O′17501			O′73501			O′34100		
H′151C			H′1F41			H′7741			H′3840		

2.3.7

$$A = a_{11} \cdot p^{11} + \ldots + a_0 = \alpha_2 (p^4)^2 + \alpha_1 (p^4) + \alpha_0$$
$$B = b_{11} \cdot p^{11} + \ldots + b_0 = \beta_1 (p^4)^2 + \beta_1 (p^4) + \beta_0$$

avec $\alpha_2 = a_{11} \cdot p^3 + a_{10} \cdot p^2 + a_9 \cdot p + a_8$, etc.

Il est facile de montrer que $\alpha_0 + \beta_0 = c_1 \cdot (p^4) + \gamma_0$ avec $\gamma_0 < p^4$ et $c_1 = 0$ ou 1.

Le rapport c_1 s'ajoute à la somme $\alpha_1 + \beta_1$ et on procède par récurrence comme dans le cas du paragraphe 2.3.5.

2.3.14

Si

$$A = a_n \cdot p^n + \ldots + a_0$$
$$A + 1 = a_n \cdot p^n + \ldots + (a_0 + 1)$$
$$= a_n \cdot p^n + \ldots + (c_1 \cdot p + r_0)$$

avec $c_1 = 0$ ou 1 et $r_0 < p$.

De plus, si $c_1 = 1$, on a nécessairement $r_0 = 0$ car $p + r_0 = a_0 + 1 < p + 1$ et donc $r_0 < 1$.

En continuant par récurrence sur les poids plus forts, on démontre la décomposition et que si $c_n = 1$, alors tous les r_i sont nuls.

2.4.5

00324	32547	
- 00017	- 63217	
00305	47330	
3F	03	
- 17	- 10	
28	F3	
00001101	01101101	00000011
- 00110110	- 01011011	- 00000111
11010111	00010010	11111100

2.4.19

O'1017 − O'3106 devient (O'1017) ADD (NEG O'3106)

(NEG O'3106) = O'4672

Résultat O'5711

−H'37 − H'712 devient (NEG H'037) ADD (NEG H'712)

(NEG H'037) = H'FC9, (NEG H'712) = H'8EE

Résultat H'8B7

H'37F − H'1F7 devient (H'37F) ADD (NEG H'1F7)

(NEG H'1F7) = H'E09

Résultat H'188

2.4.25 $B' = p^k - B = (p^k - 1) - (B - 1) = (B - 1)''$

2.4.27 Soient $A < 2^k$ et $B < 2^k$ représentés en longueur $k+1$, avec un chiffre de rang k égal à 0.

$-A$ est représenté par $A'' = (2^{k-1} - 1) - A$ et un chiffre de rang k égal à 1.

Les opérateurs d'addition et de soustraction ont une longueur $k+1$. Le report de sortie C_{out} a donc un poids 2^{k+1}.

Addition :

$A + B < 2^k + 2^k < 2^{k+1}$, donc $C_{out} = 0$

Si $A + B < 2^k$, le résultat est correct.

Si $A + B \geqslant 2^k$, il y a dépassement de capacité.

$(A > B)$ $A - B < p^k$. Il n'y a pas de dépassement de capacité. Le résultat doit être corrigé de 1 et il y a un report en sortie.

$(A < B)$ $A + (-B) = -(-A + B) = p^{k+1} - 1 - (p^{k+1} - 1 - A + B) = (A'' + B'')$. Le résultat ne doit pas être corrigé et il n'y a pas de report.

Soustraction:

$A - B$ identique à $A + (-B)$

$(-A) - B = -(A + B) = p^{k+1} - 1 - (A + B) + 1 - p^{k+1} = (A + B)'' + 1 - p^{k+1}$. Le résultat doit être corrigé et il n'y a pas de report en sortie.

On voit que dans tous les cas, la présence d'un report entraîne une correction. On relie donc le report de sortie de l'additionneur à son report d'entrée, et la logique est aussi simple que dans le cas du complément à 2.

Il y a une grande ressemblance avec les nombres arithmétiques, mais le type de donnée est en fait très différent (zéro positif et zéro négatif).

2.5.6 Soit $A = a_k \cdot A'$ A' de longueur k
$B = b_k \cdot B'$ B' de longueur k

Si $a_k = 0$ $b_k = 0$ alors $A < B$ si $A' < B'$
Si $a_k = 0$ $b_k = 1$ alors $A > B$ dans tous les cas
Si $a_k = 1$ $b_k = 0$ alors $A < B$ dans tous les cas
Si $a_k = 1$ $b_k = 1$ alors $A < B$ si $A' < B'$

Si C (carry) est égal à 1 dans le cas où $A' < B'$ et si Z est égal à 1 dans le cas où $A' = B'$, l'expression logique de la condition pour que $A < B$ est $a_k \cdot \overline{b}_k + \overline{b}_k \cdot c + a_k \cdot c$.

On voit donc qu'il est assez désagréable de devoir traiter séparément le signe et le reste du nombre sous forme complémentaire.

2.5.16

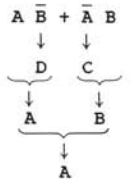
```
BIC  A,B    identique à   { NOT  B          ou   { NOT  A
                          { AND  A,B             { OR   A,B
                                                 { NOT  A
```

2.5.17

```
OR   A,B    identique à   { NOT  A
                          { NOT  B
                          { AND  A,B
                          { NOT  A
```

2.5.18

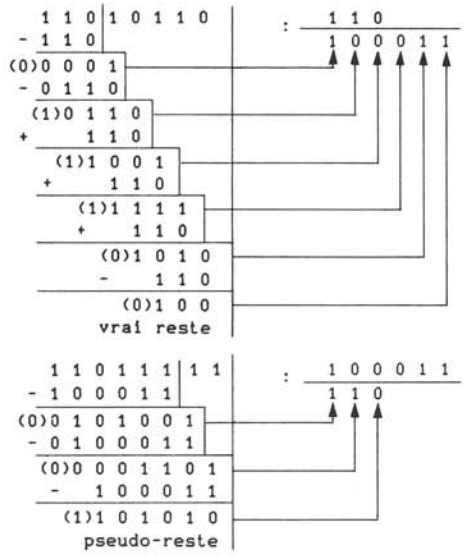
```
XOR  A,B    identique à   { LOAD  C,A
                          { LOAD  D,B
                          { NOT   C
                          { NOT   D
                          { AND   A,D
                          { AND   B,C
                          { OR    A,B
```



2.6.2 $113 \cdot 216 = ((0 + 113 \cdot 6) + 1130 \cdot 1) + 11300 \cdot 2 = 24'408$.

2.6.6 $113 \cdot 216 = 1000\,((113 \cdot 2 + 11{,}3 \cdot 1) + 1{,}13 \cdot 6)$.

2.6.14



2.7.15 Dans le format étendu, le nombre de bits de l'exposant n'est pas augmenté, et donc les nombres plus grands et plus petits (en valeur absolue) que l'on peut représenter sont ceux du paragraphe 2.7.14. La mantisse ayant 55 bits, la précision résultante est de $2^{-55} \cong 3 \cdot 10^{-17}$.

2.8.6 On ajoute 1 en binaire. Si le résultat est inférieur ou égal à 9, on le garde. S'il est égal à 10. = H'A, on le remplace par un zéro.

```
DINC  A      equivalent à {   INC      A
                          {   COMP     A,#10.
                          {   JUMP,LO  1$
                          {   CLR      A
                          { 1$:
```

Le schéma utilise les opérateurs INC, COMP, CLR et CASE.

2.8.10 Si A et B ont 4 bits,

```
DSUB  A,B    équivalent à {   ACOC     A,B
                          {   JUMP,CS  1$
                          {   ACOC     A,#6
                          { 1$:
```

2.8.13 Pour multiplier le contenu de A (a digits) par B (b digits) avec résultat dans C (a + b digits), on peut écrire :

```
CLR     C
LOAD    D,A             ; registre auxiliaire
répéter b fois          ; multiplication par les digits de B
| LOAD    A,D
| répéter 4 fois        ; mult. par les poids 1,2,4,8
| |   RRC     B         ; poids faible dans Carry
| |   JUMP,CC 1$
| |   DADD    C,A
| 1$:DADD    A,A       ; on double
| fin
| SL        #4,D        ; décalage d'un digit, mult. par 10
fin
```

2.9.4 $O'374 = (3 \cdot 8. + 7)8. + 4 = 252.$

$H'FC = 15. \cdot 16. + 12. = 252.$

2.9.6
$$O'0{,}52 = \left(2 \cdot \frac{1}{8.} + 5\right) \cdot \frac{1}{8.} = \frac{21.}{32.} \cong 0{,}66.$$

$$H'0{,}A8 = \left(8 \cdot \frac{1}{16.} + 10.\right) \cdot \frac{1}{16.} = \frac{21.}{32.} \cong 0.66.$$

2.9.8 $250.:8 = 31$ reste $2 = a_0$

$31:8 = 3$ reste $7 = a_1$

$3:8 = 0$ reste $3 = a_2$

Le résultat est $O'372$.

$250.:16. = 15.$ reste $10. = H'A$

$15.:16. = 0$ reste $15. = H'F$

Le résultat est $H'FA$.

2.9.10

$0{,}4 \cdot 8. = 3{,}2.$ partie entière $3 = a_{-1}$

$0{,}2 \cdot 8. = 1{,}6.$ partie entière $1 = a_{-2}$

$0{,}6 \cdot 8. = 4{,}8.$ partie entière $4 = a_{-3}$

$0{,}8 \cdot 8. = 6{,}4$ partie entière $6 = a_{-4}$

Donc $0{,}4. = O'0{,}3146\ 3146...$

$0{,}4. \cdot 16. = 6{,}4.$ partie entière $6 = a_{-1}$

$0{,}4. \cdot 16. = 6{,}4$ partie entière $6 = a_{-2}$

Donc $0{,}4. = H'0{,}6666.$

2.9.13 $O'372 = H'FA = 11111010$

$O'0{,}3146... = H'0{,}66... = 0{,}011001100110.$

2.9.15

$0{,}03125. = B'0{,}00001$

$1978. = B'11110111010$

$0{,}1. = B'0{,}000110011..$

$B'11010111 = 215.$

$B'0{,}1101 = 0{,}8125.$

$B'1101{,}011 = 13{,}375$

CHAPITRE 3

3.1.3 L'architecture est donnée dans la figure 3.54.

Si après le 1er cycle, le signe de E est négatif, donc le résultat est en complément à 2, un cycle supplémentaire est effectué avec la variable $S = 1$. Ce cycle conserve A et complément E (E reçoit la valeur $0 - E$).

La touche [CE] (Clear Entry) remet à zéro le registre E.

La touche [C] (Clear) met à zéro à la fois E et A.

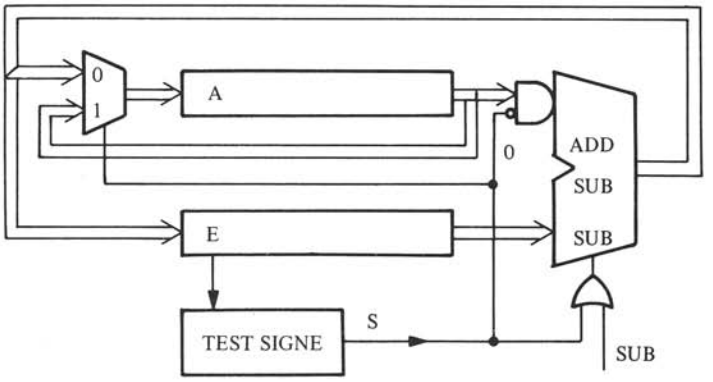


Fig. 3.54

3.2.2 Le programme sur la petite calculatrice TI53 s'écrit

[+] [2] [×] [RCL] [=] [STO] [R/S] [RST]

Avant d'exécuter le programme, il faut taper [0] [STO] pour initialiser la mémoire. Pendant l'exécution il faut, après avoir introduit un 1 ou un 0 correspondant à un chiffre binaire du nombre, taper sur [R/S].

Il n'y a aucun contrôle de validité des chiffres introduits.

3.4.8 a) solution qui modifie le registre C

```
PUSH    AF
LOAD    A,B
POP     BC
```

b) solution qui modifie le registre F

```
PUSH    BC
LOAD    B,A
POP     AF
```

c) solution compatible Z80, ne modifiant aucun registre autre que les registres A et B permutés

```
PUSH    DE
LOAD    D,C
PUSH    AF
LOAD    A,B
POP     BC
LOAD    C,D
POP     DE
```

CHAPITRE 4

4.2.5 Notation de BNW pour un nombre octal signé :

chiffre_octal = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"

nombre octal = ["+" | "−"] chiffre_octal {chiffre_octal}

Ceci exprime bien qu'il y a optionnalité entre le + et le − et qu'il doit y avoir ensuite au moins un chiffre octal.

Le diagramme de syntaxe est donné dans la figure 4.63.

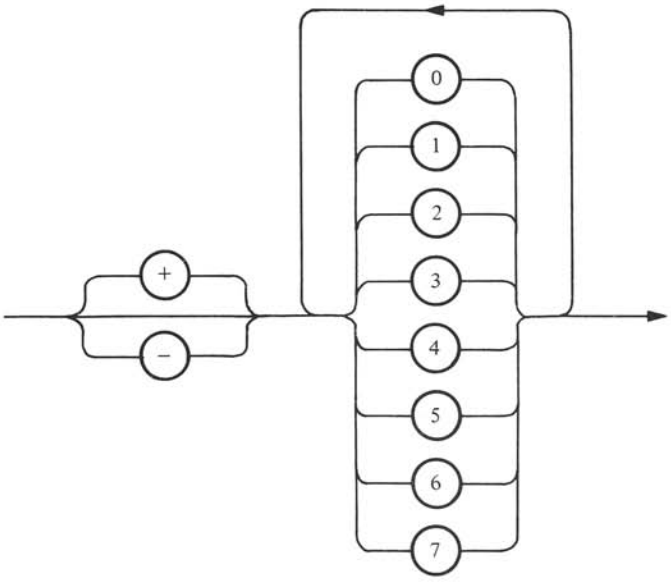


Fig. 4.63

4.2.7 La notation de BNW est :

"+"|"0"|"−"{"+"|"0"|"−"}

Le diagramme de syntaxe est trivial et n'est pas dessiné ici.

4.2.22 Il faut écrire :

.ASCIZ "A <'<> B si <CR> B,C et D>0"

4.2.25

A_ET_B	symbole
"TRUC"	chaîne
−36	expression
0A3F	nombre hexadécimal et expression
/APPELLE "DIEZE"/	chaîne
ADRESSEDEDEBUTDEROUTINE	symbole
"	nombre et expression
3 * (TRUC + 2) .OR .125	expression

4.2.26

```
TOTO*TRUC/H'100       = H'60C/H'100 = 6
TOTO*TRUC.AND.H'FF    = H'60C.AND.H'FF = H'C
(TRUC+CHOSE)*2        = H'A*2       = H'14
TOTO.AND.TRUC*CHOSE   = 2*4         = H'8
TOTO.OR.TRUC*CHOSE    = H'102.OR.H'18 = H'11A
```

4.5.5 La différence principale dans un programme Z80 est que l'on ne peut pas directement comparer avec une position mémoire. Il faut transférer dans un registre, après avoir sauvé sur la pile le registre qui doit être modifié.

Pour comparer les contenus de HL de DE, on a le choix entre la routine (modifie A et F):

```
COMPHLDE:
          LOAD          A,H
          COMP          A,D
          RET,NE
          LOAD          A,L
          COMP          A,E
          RET
```

ou la routine (modifie F seulement):

```
COMPHLDE:
          PUSH          HL
          OR            A,A         ; carry à zéro
          SUBC          HL,DE
          POP           HL
          RET
```

Le reste de la traduction se fait facilement.

4.6.3 Un programme de conversion simplifié peut s'écrire pour le M68000:

```
;in       D4.16 = DECI    nombre BCD de 4 digits
;out      D3.16 = BIN     résultat binaire 16 bits
;mod      D0.16, D1.16

NBDIGIT   =  4

CONVDB:   CLR.16          D3
          MOVE.16         #NBDIGIT-1,D0 ; compteur de cycles
2$:       SL.16           #4,D3
          MUL.16          #10,D3
          MOVE.16         D4,D1
          AND.16          #B'1111,D1 ; garde le chiffre de poids faible
          ADD.16          D1,D3
          SL.16           #4,D4
          DECJ.16,NMO     D0,2$
          TRAP            #0
```

4.8.5 Les instructions sont:

```
Z80       LOAD            HL,#ADADTABLE
          LOAD            B,#0    ;poids fort reg BC = i
          LOAD            D,#0    ;poids fort reg DE = j
          ADD             HL,BC    ;
          ADD             HL,BC    ;HL pointe ADTAi (adr de 16 bits)
          LOAD            C,{HL}    ; \
          INC             HL
          LOAD            L,{HL}    ;     LOAD  HL,{HL}
          LOAD            H,C    ; /
          ADD             HL,DE    ;HL pointe val ij
          LOAD            A,{HL}

M68000    MOVE.32         #ADADTABLE,A0
          SL.16           #2,D3    ;adr de 32 bits = 4 octets
          MOVE.32         {A0}+{D3},A0
          MOVE.8          {A0}+{D4},D0    ;D0 contient val ij
```

4.8.7 Cette routine travaille avec deux pointeurs et un capteur pour savoir quand s'arrêter.

DE est utilisé pour parcourir la table depuis le début, en surveillant le bit qui caractérise la fin d'un caractère.

L'adresse suivante est mémorisée dans une table pointée par HL.

```
GENERE:   LOAD      DE,#ADTABCAR
          LOAD      BC,#LONGTABCAR
          LOAD      HL,#ADTABADR

1$:       LOAD      {HL},E              ; sauve l'adresse du début
          INC       HL                  ; du caractère
          LOAD      {HL},D
          INC       HL

2$:       LOAD      A,B                 ; teste la fin de la table
          OR        A,C
          JUMP,EQ   FIN
          DEC       BC
          LOAD      A,{DE}              ; cherche le bit de fin de caractère
          OR        A,A                 ; test du signe
          JUMP,PL   2$
          JUMP      1$

FIN:      TRAP                          ; arrêt pour mise au point
```

Une variante plus rapide dans l'exécution de la boucle de recherche s'écrirait:

```
GENERE:   LOAD      HL,#ADTABCAR
          LOAD      BC,#LONGTABCAR      ; doit être > 0
          LOAD      DE,#ADTABADR

1$:       LOAD      A,L                 ; \
          LOAD      {DE},A              ;
          INC       DE                  ;
          LOAD      A,H                 ;     LOAD {DE+},HL
          LOAD      {DE},A              ;
          INC       DE                  ; /
          XOR       A,A                 ; CLR A

2$:       CPI                           ; COMP A,{HL+}, DEC BC (V=0 si BC=0)
          JUMP,VC   FIN
          JUMP,SC   2$
          JUMP      1$                  ; on a trouvé une fin de car

FIN:      TRAP
```

4.8.9 Il ne faut pas oublier de corriger l'adresse de retour après avoir lu le numéro de la routine.

La recherche de l'adresse de la routine est similaire à la recherche d'un caractère dans une table de caractères (§ 4.8.3).

Avec le Z80, la routine est assez astucieuse et fait appel à l'instruction qui échange le sommet de la pile (donc l'adresse de retour) avec le contenu de HL. Le registre A doit être sauvé en mémoire et grâce à la philosophie du processeur, aucun fanion n'est modifié par les premières instructions de transfert.

Les interruptions, qui peuvent modifier l'état de la pile, doivent être désactivées.

```
ROUTINE:  IOF
          LOAD      SAVEA,A             ; SAVEA est une position mémoire libre
          EX        {SP},HL             ; échange le sommet de la pile avec HL
          LOAD      A,{HL}
          INC       HL
          EX        {SP},HL             ; adresse de retour sur la pile

          PUSH      HL
          PUSH      AF                  ; sauve les fanions
          COMP      A,#MAX
          JUMP,HS   ERROR
```

```
        SL      A                   ; *2
        LOAD    HL,#TABLEROUT
        ADD     A,L                 ; \
        LOAD    L,A                 ;
        LOAD    A,H                 ;     ADD HL,0A  pour calculer
        ADDC    A,#0                ;                l'adresse de saut
        LOAD    H,A                 ; /
        POP     AF                  ; reprend les fanions
        LOAD    A,SAVEA
        EX      {SP},HL             ; adresse de saut sur la pile
        ION
        RET
```

Dans le cas du M68000, il n'y a pas besoin de désactiver les interruptions si on est dans le mode utilisateur, car elles interviennent sur la pile système en mode superviseur.

```
ROUTINE: PUSH.32    A0              ; place pour l'adresse de saut
         PUSH.16    F               ; sauve les fanions
         PUSHM.32   A0,D0
         MOVE.32    {A7}+14.,D0     ; adresse du mot contenant le numéro
         ADD.32     #2,{A7}+14.     ; correction pour le retour
         MOVE.32    #TABLEROUT,A0
         SL.16      #4,D0           ; multiplie par 4
         MOVE.32    {A0}+{D0},{A7}+10.
         POPM.32    A0,D0
         RETF                       ; reprend les fanions et saute
```

4.8.13 Le programme Z80 s'écrit:

```
...      ; récupération des fanions et des registres
         LOAD       SP,SAVSP
         LOAD       HL,SAVPC
         PUSH       HL              ; adresse de retour sur la pile
         LOAD       HL,SAVHL
         RET
```

Le programme I8085 est à peine plus complexe, puisque l'on doit remplacer la première instruction par

```
LOAD     HL,SAVSP
LOAD     SP,HL
```

4.8.17 La routine pour M68000 est:

```
;in      D4.16 = n
;        A4.32  pointeur à la pile des données
;out     D4.16 = SA(n)
;mod     D4.16
```

; la pile de données est pointée par A4

```
SA:      COMP.32    #TOPOPSTACK,A4
         JUMP,LO    ERROR
         MOVE.16    D4,{A4}
         DEC.16     D4
         JUMP,EQ    1$
         CALL       SA
1$:      ADD.16     {A4+},D4

         RET
```

CHAPITRE 5

5.1.13 Le schéma s'obtient en intercalant l'interface de la figure 5.13.

5.2.3 Seule la routine GETBLOC doit être modifiée. Avec le Z80, il faut écrire:

```
GETBLOC:  LOAD      HL,#DATABLOC
          LOAD      B,#LONGBLOC

2$:       CALL      GETBYTE
          COMP      A,#H'D              ; CR
          JUMP,EQ   4$
          COMP      A,#H'7F             ; DEL
          JUMP,NE   3$
          DEC       HL                  ; reculer le pointeur
          JUMP      2$

3$:       COMP      A,#H'8              ; BS
          JUMP,EQ   GETBLOC
          LOAD      {HL},A              ; sauve en mémoire
          INC       HL
          DECJ,NE   B,2$

4$:       TRAP                          ; fini
```

Avec le M68000, on a:

```
GETBLOC:  MOVE.32       #DATABLOC,A0
          MOVE.16       #LONGBLOC-1,D0

2$:       CALL          GETBYTE
          COMP.8        #H'D,D3         ; CR
          JUMP,EQ       4$
          COMP.8        #H'7F,D3        ; DEL
          JUMP,NE       3$
          DEC.32        A0              ; reculer le pointeur
          JUMP          2$

3$:       COMP.8        #H'8,D3         ; BS
          JUMP,EQ       GETBLOC
          MOVE          D3,{A0+}
          DECJ.16,NMO   D0,2$

4$:       TRAP          #0
```

5.3.3 L'interruption est supposée codée en autovecteur 1 [90].

```
          .TITLE    INT68000
          .PROC     M68000

DATABLOC  = H'4000              ; Adresse du bloc
LONGBLOC  = 128.                ; longueur

LEC       = 306                 ; adresse lecteur papier
SLEC      = LEC+2               ; registre d'état
BFULL     = 1                   ; rang du bit FULL
MLEC      = SLEC                ; registre de mode
MILECEN   = H'40                ; masque

;+++      Programme principal

.LOC      H'1000

DEB:      MOVE.32   #DATABLOC,SAVBLOC
          MOVE.16   #LONGBLOC-1,SAVLONG
          MOVE.32   #ADINTER,H'64       ; autovecteur 1
          MOVE.16   #H'700,SP           ; interruption processeur activées
                                        ; suite du programme en mode utilisateur
          MOVE.8    #MILECEN,MLEC       ; accepte les interruptions du lecteur

;+++      Routine d'interruption

ADINTER:  PUSH.32   A0
          MOVE.32   SAVBLOC,A0
          MOVE.8    LEC,{A0+}           ; transfert en mémoire
          MOVE.32   A0,SAVBLOC
          POP.32    A0
          DEC.16    SAVLONG
          JUMP,NE   1$
          MOVE.8    #MILECDIS,MLEC      ; déactive l'interruption lecteur
                                        ; en fin de bloc
1$:       RETSF
```

5.3.4 Le programme s'écrit :

```
.TITLE    EX534.TX
.PROC     M68000
.REF      SM8                    ; Références aux routines de du système
.REF      M8SYS
.START    TSILO
.LOC      1000

; **** Programme de test des routines silo, lecture par interruption

TSILO:    ;...                   ; Initialiser l'interruption comme dans lex
5.3.3
          MOVE.32   #SBUFF,SIN
          MOVE.32   #SBUFF,SOUT
LOOP:     .32       ?GETCAR   ?AFCAR  ; Lecture du clavier et echo
          COMP.8    #CR,D3
          JUMP,EQ   LOOP               ; Il n'y a rien d'autre à faire
GET:      CALL      STEMPTY            ; Avant de lire on vérifie
          JUMP,EQ   EREMPTY
          CALL      SREAD
          .32       ?AFSPACE ?AFCAR    ; Un espace précède les car relus
          JUMP      LOOP

EREMPTY:                             ; Procédures d'erreur simplifiées: un bruit
          .32       ?BUZZ
          JUMP      LOOP

; **** Variables en mémoire vive
SBUFF:    .BLK.8    64.              ; Silo
SBUFEND:
SIN:      .BLK.32   1                ; Pointeurs circulaires
SOUT:     .BLK.32   1
; --- Routine d'interruption (modifie A0.32)
INTERPUT:
          CALL      STFULL
          JUMP,EQ   ERFULL
          CALL      SWRITE
          RETSP
ERFULL:   .32       ?BUZZ
          RETSF
; Les routines SILO sont identiques
```

5.3.6 Le programme est :

```
INTER:    PUSH      AF
          PUSH      BC
          PUSH      HL
          LOAD      HL,#TABLINTER
          LOAD      B,{HL}           ; nombre de périphériques
          INC       HL
1$:       LOAD      A,{HL}           ; adresse du périphérique
          INC       HL
          LOAD      C,A
          LOAD      A,$(C)           ; contenu registre périphérique
          AND       A,{HL}           ; masquage et test du bit
          INC       HL
          JUMP,NE   2$               ; service si bit actif
          INC       HL               ; autrement on passe par dessus l'adresse
          INC       HL
          DECJ,NE   B,1$             ; et on continue la recherche selon la table
          CALL      ERROR
          JUMP      RETOUR
2$:       LOAD      A,{HL}           ; on lit l'adresse
          INC       HL
          LOAD      H,{HL}
          LOAD      L,A
          JUMP      {HL}             ; on saute à la routine
```

On voit que cette solution coûte 101. octets, plus 4 octets par périphérique. L'autre solution coûte un octet (PUSH AF), puis 7 octets par périphérique. Il faut donc au moins 10 périphériques pour que la première solution soit rentable (elle reste plus lente).

BIBLIOGRAPHIE

[1] *Data Processing Vocabulary*, Recueil de Normes ISO, ISO, Genève, 1982.

[2] *Terminologie du traitement de l'information*, IBM, Paris, 1980.

[3] C.J. SIPPL, *Microcomputer Dictionary*, Sams, Indianapolis, 1981.

[4] D. LINSE, *Dictionnaire de l'Informatique Français-Allemand*, Brandstetter, Wiesbaden, 1981.

[5] G. MICHEL *et al.*, *Les automates programmables industriels*, Dunod, Paris, 1979.

[6] P. VITRANT, *Calculatrices de poche et informatique*, Masson, Paris, 1980.

[7] H. SCHUMNY, *Taschenrechner und Mikrocomputer Jahrbuch 1981*, Vieweg, Braunschweig, 1980.

[8] M. WAITE, M. PARDEE, *Microcomputer Primer*, Sams, Indianapolis, 1980.

[9] A.B. FONTAINE, *Economie des projets à microprocesseurs*, Masson, Paris, 1980.

[10] R.P. UHLIG *et al.*, *The Office of the Future*, North-Holland, Amsterdam, 1979.

[11] J.D. FOLEY, J. VAN DAM, *Fundamentals of Interactive Computer Graphics*, Addison Wesley, Reading, 1982.

[12] J.P. HAYES, *Computer Architecture and Organisation*, McGraw-Hill, New York, 1978.

[13] R.W. HOCKNEY, *Parallel Processors*, Hilger, Bristol, 1981.

[14] R.E. ECKHOUSE, L.R. MORRIS, *Minicomputer Systems*, Prentice-Hall, London, 1979.

[15] G.A. KORN, *Microprocessors and Small Digital Computer Systems for Engineers and Scientists*, McGraw Hill, New York, 1980.

[16] N. BOURBAKI, *Eléments de mathématiques, 1ère partie*, Livre 1, chap I, Hermann, Paris, 1954.

[17] R. FAURE *et al.*, *Calcul booléen appliqué*, Albin Michel, Paris, 1963.

[18] J. KUNTZMANN, *Algèbre de Boole*, Dunod, Paris, 1969.

[19] T.R. BLAKESLEE, *Digital Design with Standard MSI & LSI*, Wiley, New York, 1979.

[20] E.R.HNATEK, *A User's Handbook of Semiconductor Memories*, Wiley, New York, 1977.

[21] D.R. McGLYNN, *Modern Microprocessor System Design: Sixteen-bit and Bit-slice Architecture*, Wiley, New York, 1980.

[22] J.B. PEATMAN, *Digital Hardware Design*, McGraw-Hill, Auckland, 1980.

[23] W.I. FLETCHER, *An Engineering Approach to Digital Design*, Prentice-Hall, Englewood Cliffs, 1980.

[24] J.L. BAER, *Computer Systems Architecture*, Pitman, London, 1980.

[25] S. ROSEN, Electronic Computers: A Historical Survey, *Computer Surveys ACM*, Vol 1, pp. 7-36, March 68.

[26] A.P. SPEISER, The Relay Calculator Z4, *Annals of the History of Computing*, Vol. 2 No 3, July 1980.
[27] E.E. SWARTZLANDER JR., *Computer Design Development*, Hayden, Rochelle Park, 1976.
[28] C.G. BELL, A. NEWELL, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.
[29] C. MEAD, L. CONWAY, *Introduction to VLSI Systems*, Addison-Wesley, Reading, 1980.
[30] T. FORESTER, *The Microelectronics Revolution*, Blackwell, Oxford, 1980.
[31] K.D. WISE *et al.*, *Microcomputers: a Technology Forecast and Assessment for the Year 2000*, Wiley, New York, 1980.
[32] A. WARUSFEL, *Les nombres et leurs mystères*, Seuil, Paris, 1961.
[33] B.L. VAN DER WAERDEN, *Science Awakening I*, (4e ed), Noordhoff, Leyden, 1975.
[34] D. KNUTH, *The Art of Computer Programming*, vol. 3, Pitman, London, 1969.
[35] J.H. WILKINSON, *Rundungsfehler*, Springer, Berlin, 1969.
[36] K. HWANG, *Computer Arithmetic, Principles, Architecture and Design*, Wiley, New York, 1979.
[37] J. CHINAL, *Circuits logiques de traitement numérique de l'information*, Cepadues, Toulouse, 1979.
[38] R.M.M. OBERMAN, *Digital Circuits for Binary Arithmetic*, Macmillan, London, 1979.
[39] J.P. VABRE, La retenue anticipée dans les circuits d'addition, *Onde électrique*, Vol. 61, No 1, pp. 57-64, 1981.
[40] S. WASER, High-Speed Monolithic Multipliers for Real-Time Digital Signal Processing, *Computer IEEE*, pp. 19-28, October 78.
[41] Multiplier/accumulators Expand Easily to Larger Arrays, *EDN*, pp. 88-94, July 80.
[42] *Arithmétique binaire en séparation flottante*, CEI, Genève, 1981.
[43] J.T. COONEN, An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic, *Computer Magazine IEEE*, pp. 68-79, January 80.
[44] R.L. KRUTZ, *Microprocessors and Logic Design*, Wiley, New York, 1980.
[45] R.K. RICHARDS, *Arithmetic Operations in Digital Computers*, van Nostand, Amsterdam, 1965.
[46] A.S. TANENBAUM, *Structured Computer Organization*, Prentice-Hall, Englewood Cliffs, 1976.
[47] J.D. NICOUD, Iterative Arrays for Radix Conversion, *IEEE Computer*, Vol C20, pp. 1479-1489, Dec. 71.
[48] J.D. NICOUD, Conversion binaire-décimale en virgule flottante, *Cahiers de la CSL*, No 2, Avril 1970.
[49] A. CAKIR *et al.*, *Visual Display Terminals*, Wiley, Chichester, 1980.
[50] G.A. GIBSON, YU-CHENG LIU, *Microcomputers for Engineers and Scientists*, Prentice-Hall, Englewood Cliffs, 1980.
[51] E.E. KLINGMAN, *Microprocessor Systems Design*, Vol. I, Prentice-Hall, Englewood Cliffs, 1977.
[52] E.E. KLINGMAN, *Microprocessor Systems Design*, Vol. II, Prentice-Hall, Englewood Cliffs, 1982.
[53] G. SCHNELL, K. HOYER, *Mikrocomputerfibel: vom 8-bit-chip zum Grundsystem*, Vieweg, Braunschweig, 1981.

[54] G. KANE *et al.*, *68000 Assembly Language Programming*, Osborne McGraw-Hill, New York, 1981.
[55] *VAX: Architecture Handbook*, Digital, Maynard, 1981.
[56] D.P. SIEWIOREK *et al.*, *Computer structures: Principles and Examples*, McGraw-Hill, New York, 1982.
[57] R.G. GARSIDE, *The Architecture of Digital Computers*, Clarendon, Oxford, 1980.
[58] Computer Architecture, Special Issue, *Communications of the ACM*, Vol. 21, No 1, January 78.
[59] G.J. LIPOWSKI, K.L. DOTY, Developments and Directions in Computer Architecture, *IEEE COMPUTER Magazine*, pp. 54-67, August 78.
[60] C. FOSTER, *Content Addresable Parallel Processor*, van Nostand, Amsterdam, 1976.
[61] J.F. WAKERLY, *Microcomputer Architecture and Programming*, Wiley, New York, 1981.
[62] R. DUBOIS, D. GIROD, *Les microprocesseurs 16 bits à la loupe*, Eyrolles, Paris, 1982.
[63] *Introduction to the IAPX432 Architecture*, Intel, 1981.
[64] R.P. POLIVKA, S. PAKIN, *APL, the Language and its Usage*, Prentice-Hall, Englewood Cliffs, 1975.
[65] A. STROHMEIER, *Fortran 77*, Eyrolles, Paris, 1982.
[66] B.S. GOTTFRIED, *Programming with Basic*, Schaum's, McGraw-Hill, New York, 1975.
[67] N. MAGNENAT-THALMANN *et al.*, *Le langage Pascal*, Gaëtan Morin, Chicoutimi, 1979.
[68] B.W. KERNIGHAN, P.J. PLAUGER, *Software Tools*, Addison-Wesley, Reading, 1976.
[69] P. WEGNER, *Programming with ADA*, Prentice-Hall, Englewood Cliffs, 1980.
[70] A. STROHMEIER, *Cobol 74*, Eyrolles, Paris, 1982.
[71] N. WIRTH, *Programming in Modula-2*, Springer, Berlin, 1982.
[72] D.R. MCGLYNN, *Fundamentals of Microcomputer Programming, including Pascal*, Wiley, New York, 1982.
[73] N. WIRTH, What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions, *Communications of the ACM*, Vol 20, No 11, Nov. 1977.
[74] H.P. BRINCH, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, 1977.
[75] G. BELL *et al.*, *Computer Engineering*, Digital, Bedford, 1978.
[76] E.C. POE, J.C. GOODWIN, *The S-100 and Other Micro Busses*, Sams, Indianapolis, 1979.
[77] A. ALLISON, Status Report on the P896 Backplane Bus, *IEEE Micro*, Vol. 1, No 1, Feb. 1981.
[78] L. HOHENSTEIN, *Computer Peripherals for Minicomputers, Microprocessors and Personal Computers*, McGraw-Hill, New York, 1980.
[79] B.A. BOWEN, R.J.A. BUHR, *The Logical Design of Multiple-Microprocessor Systems*, Prentice-Hall, Englewood Cliffs, 1980.
[80] D. DEL CORSO *et al.*, *Microprocessor Busses*, Academic Press, London, 1986.
[81] D.F. STOUT, *Microprocessor Application Handbook*, McGraw-Hill, New York, 1982.

[82] J. NIEVERGELT *et al.*, *Document Preparation Systems*, North-Holland, Amsterdam, 1982.

[83] J.E. McNAMARA, *Technical Aspects of Data Communication*, Digital, Maynard, 1977.

[84] B.W. HAMMING, Error Detecting and Error Correcting Codes, *The Bell System Technical Journal*, Vol. 26, No 2, April 1950.

[85] P.T. NGUYEN, *Transmission des données*, Infoprax, Pontoise, 1979.

[86] *Transfert de l'information*, Recueil de Normes 1, ISO, Genève, 1982.

[87] W. MYERS, Toward a Local Network Standard, *IEEE Micro*, pp. 28-45, August 1982.

[88] H. NUSSBAUMER, *Réseaux de Téléinformatique*, Cours EPFL, Lausanne, 1983.

[89] H.C. FOLTS, *McGraw-Hill Compilation of Data Standards*, McGraw-Hill, New York, 1982.

[90] *Documentation récente des fabricants.*

Pour la Suisse, il faut s'adresser aux représentants suivants:

M	Motorola:	Omni Ray Zurich ou Elbatex Wettingen
I	Intel:	Industrade Zurich
TMS	Texas:	Fabrimex Zurich
NS	National:	Fenner Sissach
R	Rockwell:	Aumann Zurich
Z	Zilog:	Stolz Baden
S	Signetics:	Philips Zurich
EF	EFCIS:	Modulator Berne
AM	AMD:	Hirt Zurich

INDEX ANALYTIQUE FRANÇAIS-ANGLAIS

Les références sont celles des pages

INDEX ANALYTIQUE ANGLAIS-FRANÇAIS

Les références sont celles des pages

Le Traité d'Electricité est l'œuvre collective
des membres du Département d'Electricité de l'EPFL,
assistés par quelques collaborateurs externes.
A ce volume ont contribué plus particulièrement:

Marianne Aiassa: coordination
Antonio Alabau: critique du manuscrit
Robert Arouette: critique du manuscrit
Théo Berney: critique du manuscrit
Marcel Berthoud: critique du manuscrit
Louis Bolliet: critique du manuscrit
Guy Boulaye: critique du manuscrit
Claude Capt: index
Edouard Cerny: critique du manuscrit
Paolo Conti: critique du manuscrit
Daniel-Jean David: critique du manuscrit
Claire-Lise Delacrausaz: Direction
des Presses polytechniques romandes
Michel Ducrey: critique du manuscrit
Roland Dumond: critique du manuscrit
Dominique Dutoit: critique du manuscrit
Marie-Claire Ebe: dactylographie du manuscrit
Pierre-Gérard Fontolliet: critique du manuscrit
Roland Forster: dessin des figures
Paul Gosset: critique du manuscrit
Michel Groslambert: dessin des figures
Paul L. Hazan: critique du manuscrit
Kurt Hofer: dessins et photographies
François Huguenin: critique du manuscrit
Raymond Juillerat: critique du manuscrit
Allen Kilner: mise en page et montage
Sylvie Kropf: composition des textes et équations
Wolfgang Mahr: critique du manuscrit
Fabio Manzini: correction des épreuves
Françoise Moinet: exercices, contrôle du manuscrit
Charles Musy: résolution des exercices
Jacques Neyrinck: Direction du Traité, critique du manuscrit
Cathi Nicoud: correction des épreuves, index
Jean-Daniel Nicoud: rédaction
Paul-Albert Nobs: dessins de figures
Jean-Marc Paratte: critique du manuscrit
Marie-José Pellaud: dactylographie
Renée Pittet: composition des textes et équations
Albert Prieto: critique du manuscrit
Daniel Roux: critique du manuscrit
Maria-Giovanna Sami: critique du manuscrit
Anne-Marie Schmidt: critique du manuscrit
André Spatz: critique du manuscrit
Jean-Stéphane Szijarto: contrôle des épreuves
Jacques Tiberghien: critique du manuscrit
Laurent Tourres: critique du manuscrit
Cong Thien Thang: dessin des figures
Philippe van Bastelaer: critique du manuscrit
Georges Vaucher: photographies
Jacques Virchaux: critique du manuscrit
Ida Wegmüller: montage du lettrage et corrections
Niklaus Wirth: critique du manuscrit
Bertrand Zufferey: critique du manuscrit

Imprimé en Suisse