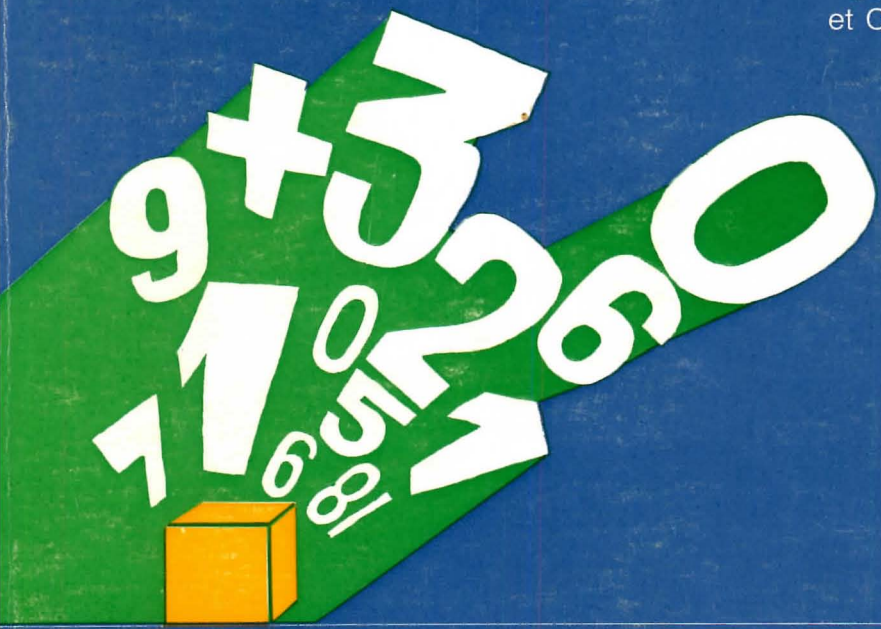
COLLECTION PRATIQUE DE L'INFORMATIQUE

INITIATION AUX MINICALCULATEURS ET MICROPROCESSEURS

Arpad BARNA
et Dan I. PORAT

Traduit par
Jean-Marc KERILIS
et Christian SAGUEZ



Eyrolles
EDITEUR-PARIS

INITIATION

AUX MINICALCULATEURS ET MICROPROCESSEURS

R 4

PRATIQUE DE L'INFORMATIQUE

INITIATION
AUX MINICALCULATEURS ET MICROPROCESSEURS

par

Arpad BARNA
Hewlett-Packard Laboratories
Palo Alto-Californie

et

Dan I. PORAT
Stanford Linear Accelerator Center
Stanford University-Stanford-Californie

TRADUIT DE L'ANGLAIS PAR

Jean-Marc KERILIS et Christian SAGUEZ
Ingénieurs des Arts et Manufactures

ÉDITIONS EYROLLES
61, boulevard Saint-Germain - 75005 PARIS
1977

Traduction autorisée de l'ouvrage anglais
Introduction to Microcomputers and Microprocessors

AVANT-PROPOS

Le nombre sans cesse croissant de types de minicalculateurs et de microprocesseurs a conduit à une grande diversité dans leurs applications. Ainsi la création de systèmes utilisant un microprocesseur demande la connaissance de plusieurs domaines tels que la logique, les systèmes digitaux, l'architecture d'un calculateur, la programmation et à un degré moindre la conception des circuits et la technologie des semiconducteurs. Ce livre d'initiation est écrit pour les personnes qui n'ont pas de connaissances précises dans ces domaines et qui désirent en acquérir suffisamment pour faire une bonne utilisation des minicalculateurs et des microprocesseurs.

Nous avons procédé en trois étapes. Les trois premiers chapitres donnent une vue générale du hardware et du logiciel élémentaires. Les cinq chapitres suivants détaillent le fonctionnement du minicalculateur et le dernier chapitre présente des compléments. Il y a un minimum de renvois entre les différents chapitres pour que le lecteur puisse sans problème lire seulement les parties qui l'intéressent.

Les 120 exemples et problèmes inclus dans le texte rendent le livre particulièrement apte à une étude personnelle, en fournissant une base solide pour la compréhension des caractéristiques de la plupart des minicalculateurs. Les réponses à certains problèmes sont données à la fin du livre.

TABLE DES MATIÈRES

LISTE DES ABRÉVIATIONS

ALU	unité arithmétique et logique (arithmetic-logic unit).
ASCII	American standard code for information interchange.
BCD	décimal codé binaire (binary-coded decimal).
CPU	unité centrale (central processor unit).
DMA	accès direct en mémoire (direct memory access).
FET	transistor à effet de champ (field-effect transistor).
I/O	entrée-sortie (input-output).
MAR	registre d'adresse mémoire (memory address register).
MDR	registre de données mémoire (memory data register).
MOS	métal-oxyde-silicium (metal-oxyde-silicon).
MPX	multiplexeurs d'entrée et de sortie (I/O multiplexer).
MUX	multiplexeurs d'entrée et de sortie (I/O multiplexer).
PC	compteur ordinal (program counter).
PLA	zone logique programmable (programmable logic array).
pROM	mémoires mortes programmables (programmable read-only memory).
RAM	mémoire à accès sélectif (random-access memory).
ROM	mémoires mortes (read-only memory).

1

INTRODUCTION

L'une des étapes les plus importantes dans l'évolution des calculateurs digitaux a été l'introduction des *calculateurs à programme mémorisé* (1). Contrairement à un boulier ou à un calculateur actionné manuellement, la suite des opérations dans un calculateur à programme mémorisé est commandée par un programme interne.

> ***Exemple 1.1.*** La circulation automobile à l'intersection d'une route principale et d'une route secondaire est régulée par un contrôleur de circulation qui a une période de 60 secondes. Les feux de la route principale sont verts pendant 30 secondes puis oranges pendant 5 secondes et rouges pendant 25 secondes. Aussi simple qu'il est, ce contrôleur de circulation peut être considéré comme un calculateur à programme mémorisé.

Cependant, suivant la conception habituelle, un calculateur à programme mémorisé a une possibilité supplémentaire : il est capable de choisir entre différentes parties de son programme. Un tel choix, ou *prise de décision*, peut être commandé par le résultat de calculs divers ; il peut être aussi commandé par l'information reçue d'une *unité d'entrée* du calculateur.

> ***Exemple 1.2.*** Le contrôleur de circulation de l'exemple 1.1 est développé pour recevoir deux capteurs de véhicule connectés comme unités d'entrée du calculateur. Les capteurs placés sur la route secondaire indiquent si un véhicule attend le changement de couleur du feu. A la fin des 30 secondes de feu vert de la route principale le contrôleur interroge les capteurs et ne change la couleur des feux que si un véhicule attend sur la route secondaire.

(1) Les termes nouveaux sont en italique.

Les calculateurs digitaux à programme mémorisé se sont largement développés au cours des deux dernières décennies. La première cause en était les progrès technologiques tels que l'introduction des transistors qui envahissent maintenant toutes les parties du calculateur, les améliorations des éléments de stockage utilisés dans la *mémoire*, la fiabilité accrue des *unités périphériques* électromécaniques et l'utilisation toujours croissante de *circuits intégrés*. Les calculateurs digitaux d'aujourd'hui se composent d'*ordinateurs spécialisés* conçus pour un seul usage et d'*ordinateurs à usages multiples* utilisés dans différents domaines tels que le contrôle, le traitement de données et les calculs scientifiques.

En parallèle avec la fiabilité accrue, les possibilités de calcul et la facilité d'utilisation des ordinateurs à usages multiples survinrent les miniordinateurs à usages multiples qui, bien que limités du point de vue possibilités de calcul, étaient plus petits et moins coûteux. En grande partie à cause de leur prix plus bas, les miniordinateurs s'imposèrent souvent dans ce qui était initialement le domaine exclusif des ordinateurs spécialisés. Le dernier fossé séparant les ordinateurs à usages multiples des contrôleurs et des ordinateurs spécialisés a été comblé par le dernier et le plus petit ordinateur à usages multiples, le *minicalculateur*.

Les premiers minicalculateurs étaient des calculatrices. A l'heure actuelle les minicalculateurs remplacent aussi et complètent beaucoup de petits ordinateurs et d'ordinateurs spécialisés, en particulier les *contrôleurs câblés* spécialisés.

> ***Exemple 1.3.*** Des aiguillages sont installés dans un système de transit rapide. Un aiguillage particulier est installé pour chaque « bloc » de la voie, cet aiguillage guidant les trains entrant et sortant du « bloc ». Dans l'étude initiale chaque aiguillage utilisait un contrôleur câblé spécialisé ; cependant, à cause des « cas particuliers » correspondant aux différentes parties de la voie, les contrôleurs ne pouvaient pas être identiques. Pour la réalisation pratique, les contrôleurs câblés ont donc été remplacés par des minicalculateurs et les cas particuliers sont traités par des programmes adéquats.

La simplicité et le coût réduit qui permettent une large utilisation des minicalculateurs ont aussi pour conséquence une complication des techniques de programmation par rapport aux ordinateurs. En outre, le côté *hardware* est souvent plus lié au côté programmation ou *logiciel* dans un minicalculateur que dans un ordinateur. Donc, en dépit du fait que le

travail est souvent divisé entre des spécialistes hardware et des spécialistes logiciel, la création d'un système qui utilise un miniordinateur réclame une connaissance élémentaire des deux domaines. Pour cette raison ces deux domaines sont imbriqués dans la plus grande partie de ce livre, de façon à favoriser une introduction équilibrée au hardware et au logiciel des minicalculateurs.

2

STRUCTURE DE BASE DES MINICALCULATEURS ET MICROPROCESSEURS

La figure 2.1 montre le schéma simplifié d'un minicalculateur. Il comprend trois blocs fonctionnels : le *bloc entrée-sortie* (I/O), l'*unité centrale* (CPU) et la *mémoire centrale* (1).

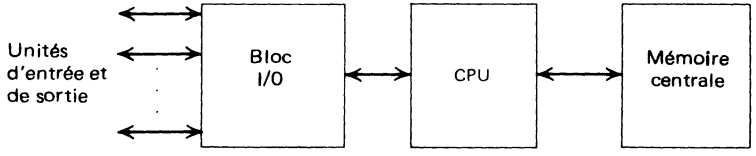


FIG. 2.1. — Schéma simplifié d'un minicalculateur (2).

2.1 LE BLOC ENTRÉE-SORTIE

Les flèches à gauche du bloc Entrée-Sortie dans la figure 2.1 relient le minicalculateur aux *unités d'entrée et sortie* (I/O), également appelées *unités périphériques*.

> ***Exemple 2.1.*** Un calculateur manuel a 10 clés numériques étiquetées de 0 à 9, 5 clés fonctions +, −, ×, + et =, et un affichage à 6 chiffres décimaux. Il contient un minicalculateur qui traite et stocke les données. Les clés sont les unités d'entrée et les chiffres affichés sont les unités de sortie.

(1) Un schéma plus détaillé est présenté au chapitre 8.
(2) Les flèches dans le schéma peuvent représenter plusieurs connexions.

La figure 2.2 montre le schéma simplifié d'un bloc Entrée-Sortie. Le choix des unités entrée-sortie (I/O) est réalisé par des *multiplexeurs d'entrée et de sortie* (souvent notés en abréviation MPX ou MUX), également appelés sélecteurs de données. L'information de sortie est stockée dans les *mémoires tampons*. Le *registre d'Entrée-Sortie* permet un stockage temporaire pendant la transmission des informations entre l'Unité Centrale (CPU) et le bloc Entrée-Sortie (I/O).

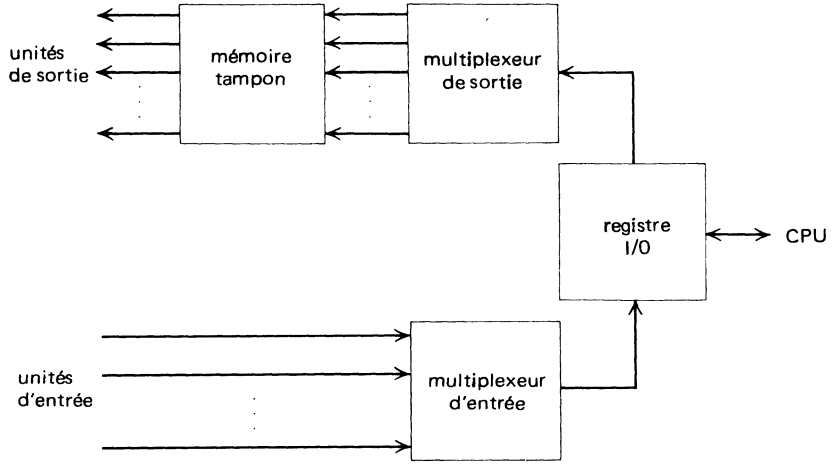


FIG. 2.2. — Schéma simplifié d'un bloc Entrée-Sortie.

Exemple 2.2. Un système de contrôle de la circulation à un carrefour utilise quatre capteurs indiquant la présence de véhicules et quatre feux tricolores. Le système de contrôle contient un minicalculateur auquel les capteurs sont connectés en tant qu'unités d'entrée et les feux en tant qu'unités de sortie. Les quatre feux doivent être en permanence vert, orange ou rouge.

Puisque la vitesse des véhicules est bornée, un capteur détecte un véhicule pendant au moins 0,1 seconde. Ainsi les quatre capteurs peuvent être appelés successivement par le minicalculateur à une vitesse uniforme d'au moins 10 lectures/seconde. Le bloc I/O du minicalculateur peut être représenté comme dans la figure 2.2.

Les caractéristiques et les connexions des unités d'entrée-sortie seront abordées au chapitre 4.

2.2 L'UNITÉ CENTRALE DE TRAITEMENT

La structure interne de l'unité centrale varie beaucoup selon les minicalculateurs. Ci-dessous nous décrivons une unité centrale simple. Elle comprend une *unité arithmétique logique* (ALU), plusieurs *registres* et une *unité de commande* (fig. 2.3). Le nombre de lignes reliant l'unité arithmétique et logique, l'*accumulateur* (registre A) et les registres B et M est déterminé par la *longueur du mot*, c'est-à-dire le nombre maximum de « *bits* » (chiffres binaires) que l'unité arithmétique et logique peut traiter en parallèle.

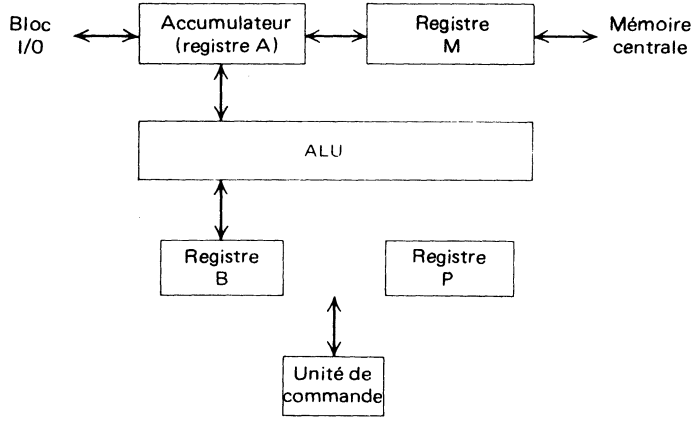


FIG. 2.3. — Schéma simplifié d'une CPU.
La figure ne montre pas les connexions de l'unité de commande et du registre P.

L'unité arithmétique et logique

L'ALU opère sur un ou deux nombres. Elle exécute des opérations arithmétiques comme l'addition et la soustraction et des *opérations* logiques comme le test d'égalité. La structure et le fonctionnement d'une unité arithmétique et logique seront abordés au chapitre 6.

Les registres

L'unité centrale comprend plusieurs registres tels que les *registres de données*, les *registres de travail* ou *mémoires à accès très rapide*. Le registre A

(accumulateur) et le registre B permettent le stockage des données utilisées par l'unité arithmétique et logique.

> ***Exemple 2.3.*** Dans l'unité centrale de la figure 2.3, l'addition est obtenue en ajoutant le contenu du registre B au contenu de l'accumulateur et en plaçant le résultat dans l'accumulateur.

La capacité en bits de l'accumulateur ainsi que celle du registre B sont déterminées par la longueur d'un mot. Pour certaines opérations arithmétiques, on peut utiliser ensemble les deux registres comme un simple registre de longueur double.

> ***Exemple 2.4.*** Dans une unité centrale avec une longueur de mot de 16 bits, la multiplication est obtenue en multipliant le contenu de l'accumulateur par le contenu du registre B. Le résultat est un nombre de 32 bits qui est stocké en plaçant les 16 bits les plus significatifs dans l'accumulateur et les 16 autres dans le registre B.

L'unité centrale de la figure 2.3 est reliée aux unités d'entrée-sortie au moyen du bloc entrée-sortie et à la mémoire centrale par le registre M.

> ***Exemple 2.5.*** On utilise un minicalculateur comprenant l'unité centrale de traitement de la figure 2.3 dans un contrôleur industriel de température. Le système mesure la température à l'aide de cinq capteurs, chacun d'eux est interrogé selon une loi uniforme de 12 lectures/minute. La température est commandée par un radiateur électrique à partir des 3 dernières lectures des 5 capteurs de température.
>
> Les données des capteurs sont transmises à la mémoire principale par le bloc entrée-sortie, l'accumulateur et le registre M. Les données de température des 3 dernières lectures sont fournies à l'unité arithmétique et logique ce qui rend nécessaire des transferts de données entre l'unité centrale et la mémoire centrale au moyen du registre M. L'information de commande résultant est envoyée au radiateur par l'intermédiaire du bloc entrée-sortie.

Le registre P est le *compteur ordinal.* Il détermine la suite des opérations du minicalculateur. Le compteur ordinal progresse d'un pas à la fois sauf ordre particulier. Chaque pas peut désigner une simple opération comme l'addition de deux nombres ou une suite d'opérations.

Exemple 2.6. Un système simple de chauffage comprend un thermostat, un radiateur allumé ou éteint et la CPU de la figure 2.3. La température désirée est chargée dans le registre M à partir de la mémoire. Le cycle de commande est mis en séquence par le compteur ordinal, c'est-à-dire par le registre P. Une suite simplifiée de commandes est représentée tableau 2.1. Elle est initialisée toutes les 10 secondes par la remise à zéro du registre P.

TABLEAU 2.1. — **Suite simplifiée de commandes dans le système de chauffage de l'exemple 2.6.**

Contenu du registre P	*Opération*
0	Départ.
1	Transfert de la température désirée du registre M au registre B.
2	Lecture du thermostat et transfert de l'information dans l'accumulateur par le bloc entrée-sortie.
3	Comparaison dans l'ALU des contenus de l'accumulateur et du registre B.
4	Allumage du radiateur par le bloc entrée-sortie si le contenu de l'accumulateur est inférieur à celui du registre B.
5	Extinction du radiateur par le bloc entrée-sortie si le contenu de l'accumulateur est supérieur ou égal à celui du registre B.
6	Fin.

L'unité de commande

Le rôle principal de l'unité de commande dans un minicalculateur est de fournir les ordres convenables pour le déroulement du calcul.

Exemple 2.7. L'addition de deux nombres dans l'unité centrale de la figure 2.3 est faite comme dans l'exemple 2.3. L'unité de commande établit d'abord *les chemins de données* pour faire passer successivement les sorties de l'accumulateur et du registre B à l'ALU et ordonner de faire l'addition. Après l'exécution de l'addition, l'unité de commande effectue le transfert des données de l'ALU à l'accumulateur.

Le fonctionnement de l'unité de commande sera étudié au chapitre 8.

2.3 LA MÉMOIRE CENTRALE

La CPU et le bloc I/O du minicalculateur contiennent plusieurs registres ou *mémoires-tampons* qui stockent des informations temporaires. Cependant, le stockage de données principales dans un minicalculateur se fait dans la *mémoire centrale.* Par rapport aux registres de la CPU et aux mémoires-tampons de sorties dans le bloc I/O, les éléments de stockage de la mémoire centrale sont plus nombreux et généralement plus lents.

La mémoire centrale peut contenir plusieurs types d'éléments de stockage. Les types les plus courants dans les minicalculateurs sont les *mémoires mortes* (ROM) et les *mémoires à accès sélectif* (RAM). En fait, toutes deux peuvent être à accès sélectif et les *temps d'accès* à tous les emplacements sont presque identiques. Dans une mémoire morte, on ne peut que lire les données de chaque emplacement ; dans une mémoire à accès sélectif on peut lire ou écrire des données dans chaque emplacement. Les données des mémoires à accès sélectif sont cependant généralement détruites par la coupure du circuit alors que dans une mémoire morte les données sont préservées lorsque le circuit est fermé. Ainsi on préfère les mémoires mortes pour le stockage d'informations permanentes telles que la suite des commandes du tableau 2.1.

En général un circuit intégré d'une mémoire morte ou d'une mémoire à accès sélectif peut stocker un grand nombre de bits : des microplaquettes de mémoires mortes de capacité de stockage de 16 384 bits (« $16-k$ bits ») et des microplaquettes de mémoires à accès sélectif de capacité de stockage de 4 096 bits (« $4-k$ bits ») se rencontrent couramment. Les bits stockés dans la mémoire centrale d'un minicalculateur sont groupés en *mots.* Chaque mot est identifié par son *emplacement mémoire* ou *adresse.*

> ***Exemple 2.8.*** Une mémoire à accès sélectif de 2 048 bits et une mémoire morte de 4 096 bits sont utilisées dans la mémoire centrale d'un minicalculateur qui a une longueur de mot de 8 bits. La mémoire à accès sélectif est organisée en 256 mots et la mémoire morte en 512. Chaque mot est identifié par l'adresse de son emplacement mémoire qui est un des 768 entiers de 0 à 767.

Les éléments et le fonctionnement de la mémoire centrale seront étudiés en détail au chapitre 7.

2.4 LES MICROPROCESSEURS

La technologie moderne permet souvent l'incorporation de tout le minicalculateur de la figure 2.1 dans une microplaquette de semiconduc-

teur et certains calculateurs ont en fait été construits de cette façon. Dans une autre approche, on met une grande partie ou tout le bloc entrée-sortie et la CPU sur une ou plusieurs microplaquettes. De telles microplaquettes sont appelées *microplaquettes de microprocesseurs*, *ensemble de microprocesseurs* ou *microprocesseurs*.

En grande partie, les caractéristiques et les limitations d'un minicalculateur sont déterminées par le microprocesseur qu'il contient. Donc une parfaite compréhension des propriétés du microprocesseur est essentielle pour l'élaboration et l'utilisation d'un minicalculateur.

PROBLÈMES

1. Combien faut-il de lignes pour connecter les clés et les affichages dans le calculateur de l'exemple 2.1 en supposant qu'aucun balayage des clés ou de l'affichage n'est employé ? On suppose que chaque affichage de chiffre requiert 4 lignes.
2. Établir une liste de la suite des opérations permettant l'addition de deux nombres de 1 chiffre dans le calculateur décrit dans l'exemple 2.1. Dans la liste, chaque entrée devra représenter une seule opération.
3. Estimer les besoins en mémoire du système de contrôle de circulation de l'exemple 2.2.
4. Déterminer les besoins en mémoire du contrôleur de température de l'exemple 2.5.
5. Établir une liste d'opérations équivalentes à la suite du tableau 2.1 ne comprenant que des opérations simples.

3

INTRODUCTION AUX TECHNIQUES DE PROGRAMMATION

La suite des opérations dans un minicalculateur est déterminée par un *programme* ou *code*. Un programme se compose d'une suite d'*instructions* et de *données*. Chaque instruction est stockée dans la mémoire centrale sous forme d'un ou plusieurs *mots instructions ;* les données sont stockées sous forme de *mots données*. Chaque instruction se compose d'un *opérateur* et d'un certain nombre d'*opérandes*. L'opérateur décrit l'*opération* à exécuter sur les données représentées par le ou les *opérandes*.

> ***Exemple 3.1.*** L'instruction « ADD 4 » est une abréviation pour « ajouter au contenu de l'accumulateur le contenu de l'emplacement mémoire 4 ». Le mot « ADD » est l'opérateur et le nombre « 4 » l'opérande.

Le programme commandant la suite des opérations est stocké dans la mémoire centrale du minicalculateur. Chaque instruction est d'abord *appelée* de la mémoire centrale puis *exécutée*.

> ***Exemple 3.2.*** L'instruction adressée par $P = 3$ dans le tableau 2.1 « Comparer dans l'ALU le contenu de l'accumulateur à celui du registre B » est une *instruction de comparaison* provenant de l'emplacement mémoire 3. Elle est exécutée par l'unité de commande en faisant passer les sorties de l'accumulateur et du registre B à l'entrée de l'ALU et en ordonnant de faire la comparaison.

3.1 INSTRUCTIONS EN LANGAGE MACHINE

Chaque instruction est stockée dans la mémoire centrale du minicalculateur sous la forme d'un code numérique approprié appelé *code*

d'instruction en langage machine qui comprend l'opérateur ainsi que le ou les opérandes.

> ***Exemple 3.3.*** Dans un minicalculateur ayant des mots de 8 bits, l'instruction abrégée « ADD 4 » (cf. exemple 3.1) est stockée sous un code d'instruction en langage machine constitué de deux mots de 8 bits. Le premier est 00111000 ($= 56_{10}$) qui dans ce minicalculateur représente l'opérateur « ADD ». Le second mot est 00000100 ($= 4_{10}$) qui est l'opérande « 4 » (1).

L'*ensemble des instructions* d'un minicalculateur peut comprendre des instructions logiques, des instructions de chargement, des instructions de stockage, des instructions arithmétiques, des instructions de saut et des instructions I/O. Quelques exemples de ces instructions sont donnés ci-dessous ; des instructions supplémentaires seront étudiées dans les chapitres suivants.

Instructions logiques

Un exemple d'instructions logiques est l'instruction de comparaison de l'exemple 3.2.

Instructions de chargement

Charger le contenu d'un emplacement mémoire spécifié dans l'accumulateur (« Charger l'accumulateur »). Si l'opérateur de cette instruction est suivi de l'opérande 00000110 ($= 6_{10}$), alors le contenu de l'emplacement mémoire 6 est chargé dans l'accumulateur. Le contenu de l'emplacement mémoire 6 reste inchangé.

Charger un nombre dans l'accumulateur (« Chargement direct »). Si l'opérateur de cette instruction est suivi de l'opérande 00001111 ($= 15_{10}$), le *nombre* 15 est chargé dans l'accumulateur.

Instructions de stockage

Stocker le contenu de l'accumulateur dans un emplacement mémoire spécifié. Si l'opérateur de cette instruction est suivi de l'opérande 0000110 ($= 13_{10}$), le contenu de l'accumulateur est stocké à l'emplacement mémoire 13. Le contenu de l'accumulateur reste inchangé.

---

(1) Pour des explications sur la notation, voir paragraphe 5.1.

Instructions arithmétiques

Ajouter au contenu de l'accumulateur le contenu de l'emplacement mémoire spécifié. Le résultat est stocké dans l'accumulateur ; le contenu de l'emplacement mémoire spécifié reste inchangé.

Instructions de saut

Sauter à un emplacement mémoire spécifié (« Saut inconditionnel »). Le contenu du compteur ordinal (registre P) devient l'adresse spécifiée ; donc l'instruction suivante est cherchée à partir de cette adresse.

Sauter à un emplacement mémoire spécifié si le contenu d'un indicateur spécifié est égal au nombre 1 (« Saut conditionnel »). L'indicateur qui est un registre à 1 bit peut être un indicateur de retenue ou de débordement, un indicateur de signe ou de nullité ou encore un indicateur quelconque.

Instructions d'entrée-sortie

Recevoir des données d'une unité d'entrée spécifiée. Les données de l'unité d'entrée spécifiée sont chargées dans le registre I/O par le multiplexeur d'entrée.

Envoyer des données à une unité de sortie spécifiée. Les données du registre I/O sont chargées dans la mémoire tampon de l'unité de sortie spécifiée par le multiplexeur de sortie.

3.2 INSTRUCTIONS EN LANGAGE D'ASSEMBLAGE

Le paragraphe précédent décrivait des instructions en langage machine. Un programme peut comprendre des centaines de ces instructions et doit être préparé avec un soin particulier. Une limitation intervenant dans l'utilisation des instructions en langage machine est l'incapacité des hommes à se servir de grands nombres binaires sans risque d'erreur. Par exemple, le code d'instruction en langage machine 0011100000000100 peut correspondre à « additionner au contenu de l'accumulateur le contenu de l'emplacement mémoire 4 » (cf. exemples 3.1 et 3.3). Le nombre d'erreurs en copiant un tel code d'instruction peut être considérablement réduit en utilisant une abréviation mnémonique telle que « ADD 4 » comme dans les exemples 3.1 et 3.3. Un langage programmé dans lequel le code d'ins-

truction en langage machine est remplacé par une abréviation mnémonique est dit *langage d'assemblage* ou *langage symbolique.*

Pour obtenir un programme en langage machine comme le demande le minicalculateur, un programme écrit en langage d'assemblage doit être traduit ; cette traduction est faite par un programme *assembleur* (1). L'assembleur peut être en permanence dans la mémoire centrale du minicalculateur ou peut être un *cross-assembleur* mis dans un ordinateur différent, en général plus grand.

> ***Exemple 3.4.*** Un programme pour un minicalculateur est écrit en langage d'assemblage en utilisant une perforatrice de ruban papier ayant un clavier ressemblant à celui d'une machine à écrire. Le ruban obtenu est lu dans un ordinateur en utilisant un lecteur de ruban comme unité d'entrée. Le programme « cross-assembleur » écrit en un langage approprié à l'ordinateur est également lu. L'ordinateur traduit le programme en langage d'assemblage en un programme en langage machine pour le minicalculateur d'après le programme cross-assembleur. Le programme en langage machine résultant est perforé par l'ordinateur sur un ruban qui est alors lu par le minicalculateur.

Macroinstructions

Une fois qu'on dispose d'un assembleur sachant traduire des instructions du langage d'assemblage en langage machine, on peut l'utiliser pour d'autres tâches. Par exemple, un grand groupe d'instructions répétées de nombreuses fois dans le programme peut être défini comme une macroinstruction et alors utilisé à la place de ce groupe. L'assembleur substitue au groupe d'instructions la macroinstruction.

> ***Exemple 3.5.*** Un groupe de 20 instructions se reproduit 10 fois dans un programme en langage d'assemblage. Pour économiser l'écriture, on définit une macroinstruction nommée MAC1 comme suit :
>
> DEFINE MACRO MAC1
>
> ...
> ... } (liste des 20 instructions définissant MAC1)
> ...
>
> END

(1) De plus amples détails sur les programmes assembleurs sont donnés au chapitre 9.

Nous écrivons donc MAC1 aux 10 endroits différents dans le programme. L'assembleur substitue les 20 instructions pour chacune des 10 macroinstructions MAC1.

Ainsi l'utilisation des macroinstructions ne change pas le programme résultant en langage machine mais évite de réécrire une longue suite d'instructions dans le programme en langage d'assemblage. Un effet malencontreux de l'utilisation des macroinstructions est que le programme en langage d'assemblage peut paraître extrêmement court. Il faut donc être prudent dans l'estimation de la taille du programme (emplacement mémoire) chaque fois que des macroinstructions sont utilisées.

Exemple 3.6. Un programme en langage d'assemblage comprend 30 instructions parmi lesquelles 12 sont des macroinstructions MAC1 définies dans l'exemple 3.5 (sans les instructions DEFINE et END). Le programme en langage machine assemblé par l'assembleur se compose de 258 instructions, dépassant les 256 emplacements mémoire disponibles dans le minicalculateur considéré.

3.3 LANGAGES ÉVOLUÉS DE PROGRAMMATION

L'utilisation d'un langage d'assemblage représente, pour le programmeur devant écrire ou comprendre un programme, un progrès sur le langage machine. Néanmoins, il reste encore une grande quantité d'écritures et mêmes des programmes simples, comme celui présenté dans l'exemple 2.6, peuvent demander des dizaines d'instructions.

Le travail du programmeur a encore été allégé par l'introduction de *langages évolués de programmation.* Ces langages de programmation combinent plusieurs instructions en une instruction générale et une suite d'instructions constitue un programme.

Exemple 3.7. En ALGOL, langage évolué de programmation, l'instruction $Y \leftarrow Z$ signifie : « la valeur de Z est assignée à Y ». L'instruction suivante est plus complexe :

$$\text{IF } V > W \text{ THEN } X \leftarrow Y - 1 \text{ ELSE } X \leftarrow Y + 1.$$

Un programme écrit en langage évolué est en général beaucoup plus court qu'un programme écrit en langage d'assemblage et peut même être plus court que la présentation initiale du problème.

Exemple 3.8. Le programme de l'exemple 2.6 peut être écrit en ALGOL comme suit :

```
BEGIN
A ← THERMOSTAT;
B ← M;
IF A < B THEN HEATER ← 1 ELSE HEATER ← 0;
END
```

Un langage évolué de programmation est traduit en langage machine par un *compilateur*. Le compilateur inspecte chaque instruction, assigne des emplacements mémoire pour le stockage des variables et des constantes et génère les instructions en langage machine.

Une importante question sur le compilateur est son efficacité. Puisqu'il fait le programme de manière routinière, le programme en langage machine qu'il génère comprend habituellement beaucoup plus d'instructions et peut utiliser davantage d'emplacements mémoire que ne le fait un programme en langage machine écrit par un programmeur spécialisé. Cette inefficacité peut être grave dans un petit minicalculateur et peut être significative dans un ordinateur important. Pour cette raison, le programmeur a souvent la possibilité de combiner dans un seul programme le langage d'assemblage et le langage évolué de programmation.

3.4 SOUS-PROGRAMMES

Nous avons déjà vu que l'utilisation des macroinstructions simplifie l'écriture d'un programme en langage d'assemblage mais que le programme en langage machine résultant n'est pas changé. Lorsqu'une suite d'instructions revient souvent dans un programme il peut être préférable de la mettre à part et de s'y référer ou de l'*appeler* si nécessaire. Une telle suite d'instructions est appelée *sous-programme*. Il peut être écrit en langage machine, en langage d'assemblage ou en langage évolué de programmation. Lorsqu'il est utilisé de nombreuses fois dans un programme, un sous-programme permet un gain important d'emplacement mémoire.

Exemple 3.9. La solution d'une équation du second degré $ROOT = (-B + \sqrt{B^2 - 4AC})/2A$ est utilisée 100 fois dans un programme, chaque fois avec des valeurs différentes de A, B et C. Un sous-programme calculant $ROOT$ peut être écrit en ALGOL comme suit :

```
PROCEDURE ROOT (A, B, C);
ROOT ← 0;
D ← B × B − 4 × A × C;
IF D ≧ 0 THEN ROOT ← (−B + SQRT(D))/(2 × A);
PRINT (A, B, C, D, ROOT);
END;
```

Notons que SQRT(D) mis pour $\sqrt{D}$ est calculé par un sous-programme séparé quand $D \geqslant 0$. Quand $D < 0$ on imprimera la valeur 0 pour *ROOT* et la valeur négative de D, indiquant ainsi que la racine (complexe) correcte ne peut être calculée par ce sous-programme simple.

Les gains d'emplacements mémoire obtenus par l'utilisation d'un sous-programme sont quelque peu réduits par le coût d'emplacements mémoire et le temps d'exécution utilisés par l'éditeur de lien pour l'appel du sous-programme et pour le retour de celui-ci. L'éditeur de lien est étudié au chapitre 9.

3.5 ORGANIGRAMMES

Un outil important pour la mise au point d'un programme est l'organigramme. A moins que le programmeur ne soit très expérimenté et que le problème ne soit très simple, il est souhaitable de construire d'abord un organigramme fondé sur la présentation initiale du problème, puis d'écrire le programme. La figure 3.1 montre divers éléments d'un organigramme. Un organigramme peut se composer de toutes les combinaisons de ces éléments.

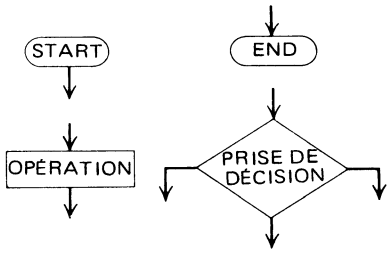


FIG. 3.1. — Divers éléments d'un organigramme.

Exemple 3.10. Un sous-programme pour trouver la racine d'une équation du second degré était décrit dans l'exemple 3.9. La figure 3.2 montre un organigramme pour ce sous-programme. Il comprend sept blocs : START, END, quatre blocs OPERATION et un bloc DECISION.

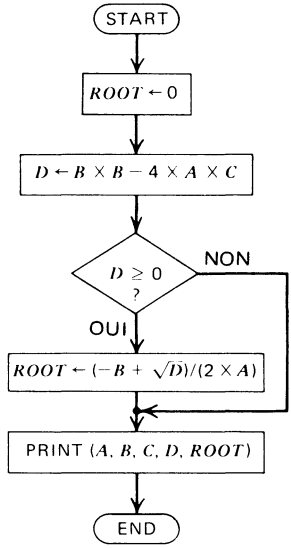


FIG. 3.2. — Organigramme pour l'obtention d'une racine d'une équation du second degré.

On peut passer sur chaque élément de l'organigramme un certain nombre de fois pendant l'exécution du programme sauf pour START et END. Une structure importante pour l'exécution multiple d'une partie d'un sous-programme est la *boucle* aussi appelée *boucle DO*.

Exemple 3.11. La figure 3.3 montre l'organigramme d'un programme imprimant le contenu d'un bloc de mémoire. Après avoir sélectionné le premier emplacement mémoire pour l'imprimer, il imprime le contenu de chaque emplacement mémoire en passant dans la boucle de l'organigramme jusqu'à ce que toutes les données soient imprimées.

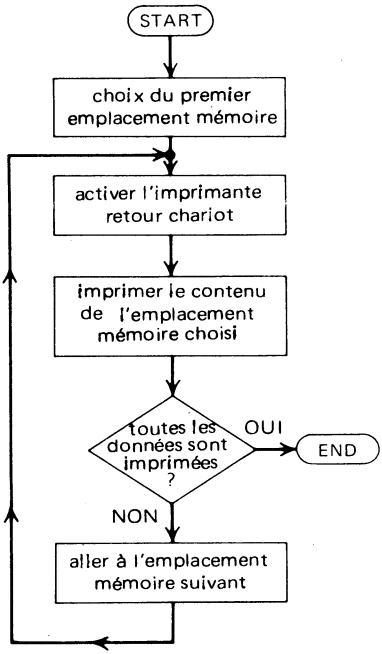


FIG. 3.3. — Organigramme pour l'impression du contenu d'un bloc mémoire.

PROBLÈMES

1. Le code instruction en langage machine pour l'opérateur « chargement de l'accumulateur » est 00001000 et pour « chargement direct » 00001001. Le contenu de l'emplacement mémoire 11_{10} est 5_{10}. Quel est le contenu de l'accumulateur après l'exécution des instructions suivantes 0000100000001011, 0000100100001011 et 0000100111111111 ?
2. Définir un ensemble d'abréviations mnémoniques et écrire un programme en langage d'assemblage pour le système de chauffage de l'exemple 2.6.
3. Résumer les avantages et les inconvénients des programmes écrits en langage machine, en langage d'assemblage et en langage évolué.
4. Comparer les avantages et les inconvénients d'un sous-programme et d'une macroinstruction.
5. Tracer un organigramme et écrire un sous-programme pour calculer la moyenne des quatre nombres A, B, C et D.
6. Préparer un organigramme pour le système de contrôle de température décrit dans l'exemple 2.5.
7. De combien d'emplacements mémoire a-t-on besoin dans le programme de l'exemple 3.6, si les macroinstructions sont remplacées par un sous-programme? Inclure une instruction d'appel pour chaque appel de sous-programme et dans le sous-programme, une instruction de retour.

4

ENTRÉES ET SORTIES

Les échanges entre le calculateur et l'extérieur se font au moyen des *unités d'entrée-sortie* (I/O), également appelées *unités périphériques*. Ce chapitre aborde la structure des instructions utilisées pour le fonctionnement des unités I/O, la structure et le fonctionnement du bloc I/O, l'interconnexion des unités I/O, les interruptions et l'accès direct en mémoire.

4.1 INSTRUCTIONS D'ENTRÉE-SORTIE

Les instructions I/O d'un minicalculateur ont aussi été décrites dans le paragraphe 3.1 : « les données de l'unité d'entrée spécifiée sont chargées dans le registre I/O par le multiplexeur d'entrée » et « les données du registre I/O sont chargées dans la mémoire tampon de sortie de l'unité de sortie spécifiée par le multiplexeur de sortie ». Ces instructions commandent le transfert de données de même que les paramètres de commande. Ces paramètres peuvent comprendre un signal « prêt » provenant du mécanisme d'impression d'une machine à écrire, un signal « fin de carte » d'un lecteur de cartes, un signal « prêt pour carte suivante » d'une perforatrice de cartes, une commande « rebobinage » pour une unité de bandes magnétiques, un signal « fin de rebobinage » ou « fin de bande » d'une unité de bandes magnétiques, etc. Divers minicalculateurs autorisent ou exigent à un degré variable la mise en œuvre d'un programme pour l'utilisation des paramètres qui doivent être inclus dans les instructions I/O.

4.2 LE BLOC ENTRÉE-SORTIE

Les composantes de base du bloc I/O ont été données figure 2.2 en tant que registre I/O, multiplexeur I/O et mémoires tampons de sortie. Ce paragraphe présente plus de détails sur le fonctionnement de ces circuits.

Registre d'entrée-sortie

Le registre I/O organise la transmission des informations entre le bloc I/O et la CPU du minicalculateur. Quand la capacité en bits du registre I/O est inadéquate pour la transmission simultanée des données, des paramètres et de l'*identification de l'unité*, la transmission s'effectue en groupes successifs.

> ***Exemple 4.1.*** Un minicalculateur avec une longueur de mot de 8 bits a 16 unités I/O, certaines demandant 4 bits pour les paramètres de commande. Ainsi, le bloc I/O, donc le registre I/O, doit transmettre 4 bits d'identification d'unités et 4 bits de paramètres de commande en plus des 8 bits de données. L'ensemble est transmis en deux groupes consécutifs : le premier de 8 bits comprend les 4 bits d'identification des unités et les 4 bits des paramètres de commande, le second comprend les 8 bits de données.

Une caractéristique du registre I/O est qu'il fournit un flot de données bidirectionnel comme le montre le circuit de la figure 4.1 pour un simple bit. L'élément de stockage dans le circuit est une bascule *D* dont l'état suivant est assigné par l'horloge *C* en fonction de l'état présent de l'entrée *D*. On utilise également des circuits logiques avec des *sorties à 3 états* qui présentent un circuit ouvert quand l'entrée *ENABLE* (désignée fréquem-

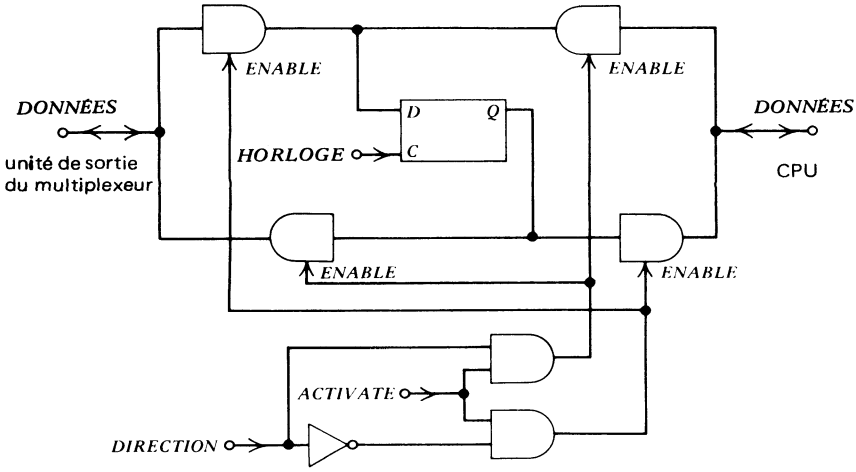


FIG. 4.1. — Flot bidirectionnel de données d'un bit dans un registre I/O.

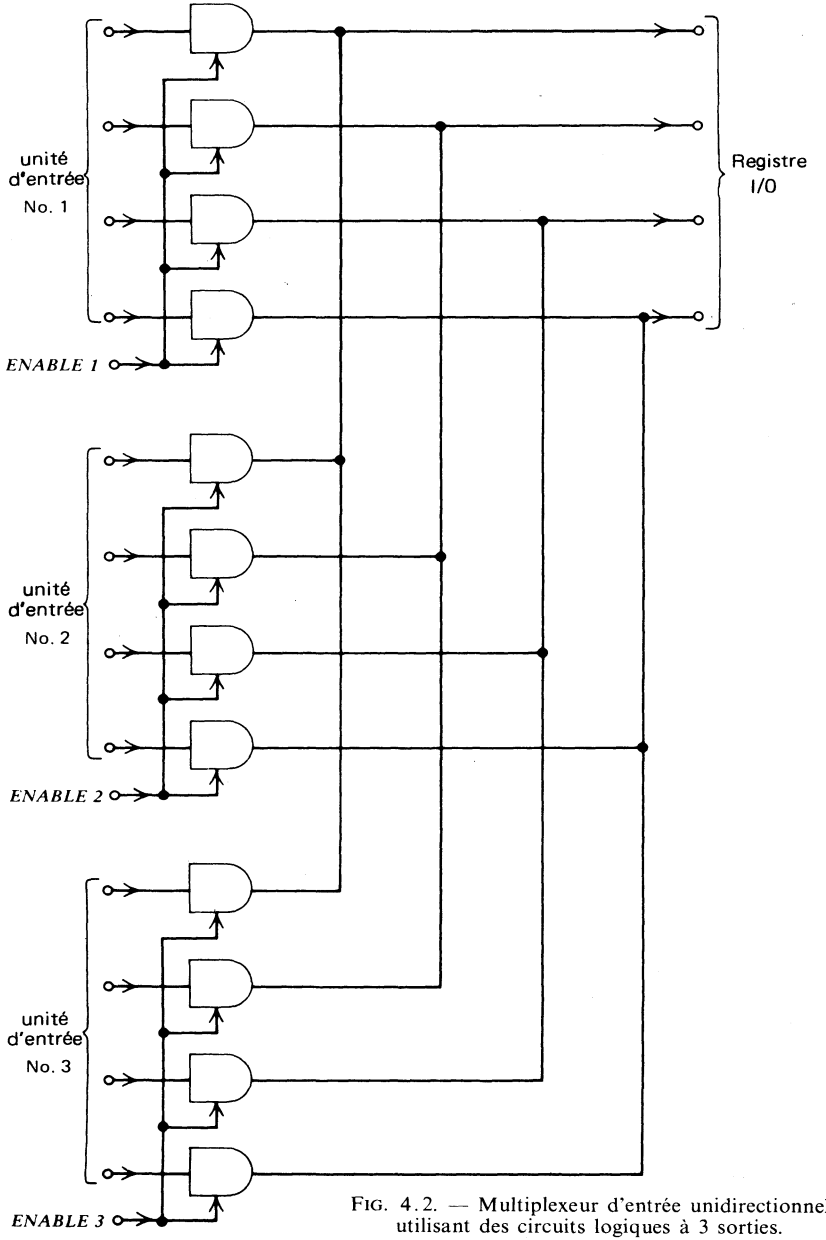


FIG. 4.2. — Multiplexeur d'entrée unidirectionnel utilisant des circuits logiques à 3 sorties.

ment comme *INHIBIT*) prend la valeur logique 0. L'entrée *DIRECTION* détermine si le flot de données va vers la droite ou vers la gauche, pourvu que l'entrée *ACTIVATE* autorise le transfert des données.

Multiplexeurs et mémoires tampons

Comme nous l'avons vu au paragraphe 2.1, un minicalculateur peut comprendre plusieurs *multiplexeurs*. La structure d'un multiplexeur diffère selon que le flot de données peut exister dans une seule ou deux directions. Dans le cas simple du bloc I/O de la figure 2.2, les multiplexeurs d'entrée-sortie sont tous les deux unidirectionnels, mais leurs structures sont différentes.

La figure 4.2 montre le schéma du circuit d'un multiplexeur d'entrée unidirectionnel. Il présente trois unités d'entrée avec des longueurs de mots de 4 bits. Un ensemble de données est réalisé par des circuits logiques avec des sorties à 3 états qui sont activés par une des trois entrées *ENABLE ;* une seule entrée *ENABLE* à la fois peut prendre la valeur binaire 1.

La figure 4.3 montre le schéma d'un multiplexeur de sortie unidirectionnel comprenant des mémoires tampons de sortie. Il comprend des *bascules D-E* dont l'état suivant est assigné par l'horloge *C* en fonction de l'état présent de l'entrée *D* si l'entrée *ENABLE* est à 1. L'état reste inchangé indépendamment de *D* si *ENABLE* est à 0. Les entrées *ENABLE (E)* appartenant à la même unité d'entrée sont reliées en parallèle.

> ***Exemple 4.2.*** Dans le bloc I/O du contrôleur de circulation de l'exemple 2.2, les unités d'entrée sont séparées des unités de sortie. Ainsi des multiplexeurs semblables à ceux des figures 4.2 et 4.3 peuvent être utilisés.

Quand les lignes reliant les unités d'entrée et de sortie ne sont pas séparées, on doit utiliser un multiplexeur bidirectionnel. Le circuit de base du multiplexeur bidirectionnel est identique à celui de la figure 4.1 et son utilisation est illustrée par la figure 4.4. Toutes les entrées *DIRECTION (DIR)* pour une unité donnée sont reliées en parallèle ainsi que toutes les entrées *ACTIVATE (A)*. Une unité est choisie en mettant l'*ENABLE* correspondant à la valeur 1 ; une seule des lignes *ENABLE* à la fois peut prendre la valeur binaire 1.

La structure générale d'un bloc entrée-sortie est montrée figure 4.5. Elle comprend un *multiplexeur auxiliaire* qui transfère l'identification d'unité I/O au *registre ID d'unité I/O* et fournit également un transfert

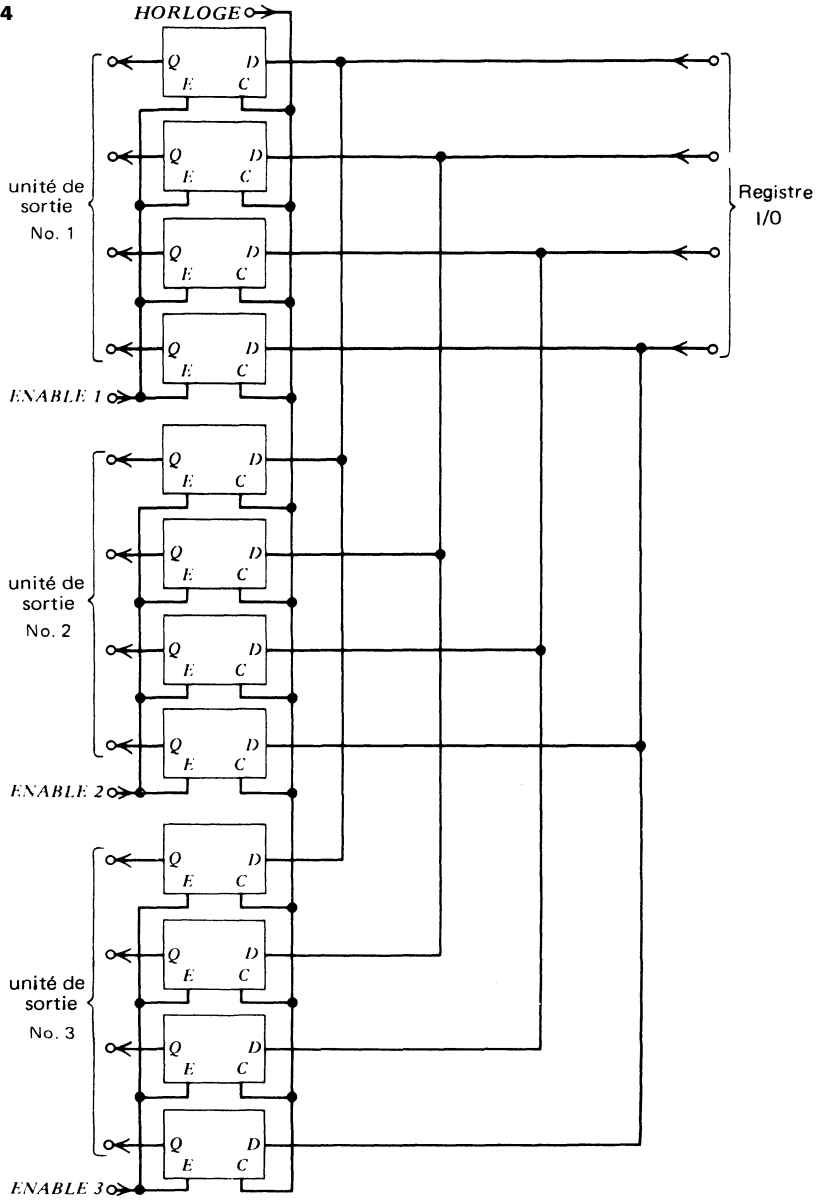


FIG. 4.3. — Multiplexeur de sortie unidirectionnel et mémoires tampons utilisant des bascules *D-E*.

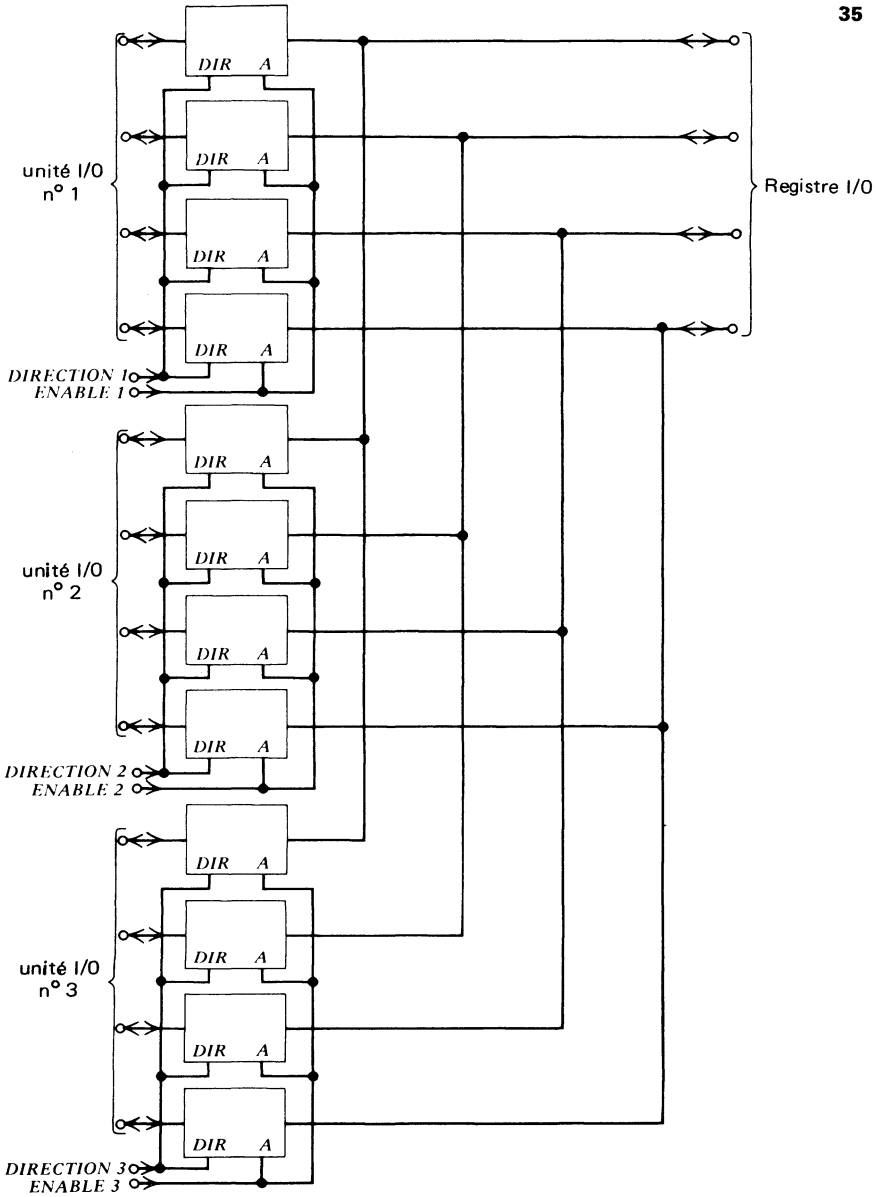


FIG. 4.4. — Multiplexeur I/O bidirectionnel, chacun des 12 blocs représente le circuit de la figure 4.1. *DIRECTION* est notée par *DIR*, *ACTIVATE* par *A*, les lignes de données ne sont pas marquées.

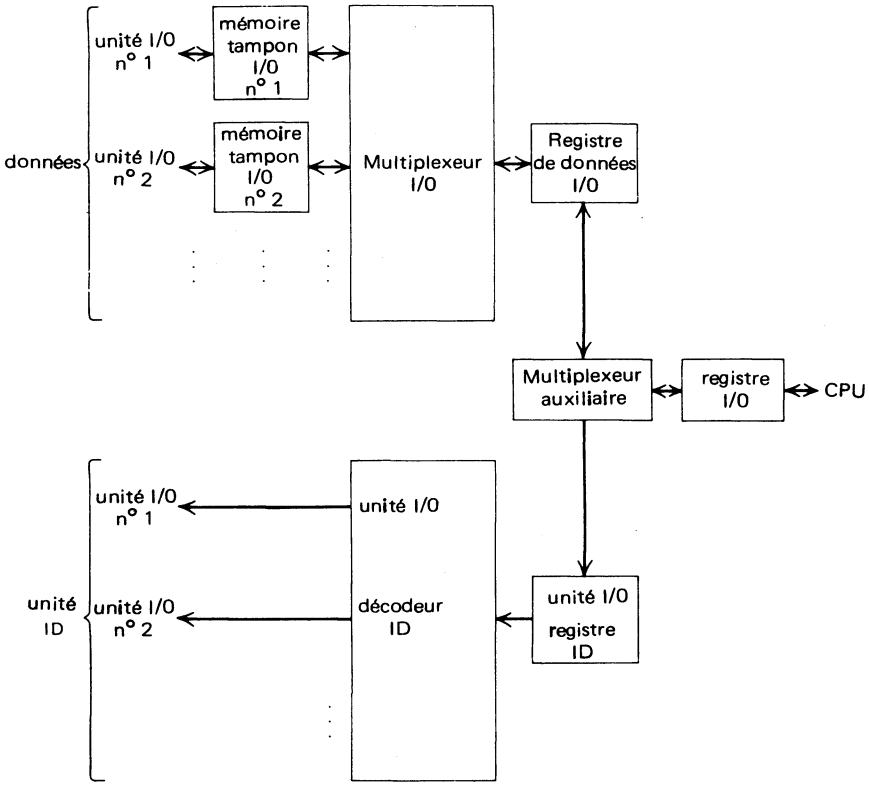


FIG. 4.5. — Schéma hors tout simplifié d'un bloc I/O.

bidirectionnel des paramètres de commande et des données entre le *registre de données I/O* et le registre I/O. Le multiplexeur auxiliaire n'est pas nécessaire dans des minicalculateurs ayant des *canaux* de données et d'adresses séparés. Le contenu du registre ID d'unité I/O est décodé par le *décodeur ID*. Dans certaines installations, surtout quand de nombreuses unités I/O sont utilisées, il peut être plus économique de placer ces décodeurs ainsi que les *mémoires tampons I/O* dans les unités I/O plutôt que dans le minicalculateur. La décision doit être fondée sur plusieurs facteurs tels que le nombre des unités I/O, la longueur d'un mot, l'emplacement des unités I/O par rapport au minicalculateur et les circuits intégrés disponibles.

4.3 INTERRUPTIONS

Jusqu'à présent les calculs et les opérations entrée-sortie ont été coordonnés en faisant fonctionner les unités I/O au moyen du programme de gestion *(appel sélectif)*. Dans un autre mode de fonctionnement, une unité I/O peut demander la main par l'envoi d'un signal de *demande d'interruption* au minicalculateur.

> ***Exemple 4.3.*** Le fonctionnement du contrôleur de température de l'exemple 2.5 est commandé au moyen d'une machine à écrire en tapant la température désirée et en effectuant un retour chariot. Chaque fois qu'un chiffre, un nombre décimal, une lettre ou un retour chariot est entré au clavier, l'imprimante demande l'utilisation de l'information reçue en envoyant un signal d'interruption au minicalculateur.

Après avoir reçu un signal de demande d'interruption, le minicalculateur achève l'instruction en cours et appelle un *sous-programme d'interruption.* Dans certains minicalculateurs un des sous-programmes d'interruption peut être choisi par un circuit donné *(interruption dirigée)*. La commande est renvoyée du sous-programme d'interruption à la suite normale des instructions dès que l'unité d'entrée a fini son travail.

Les demandes d'interruption peuvent aussi provenir des unités de sortie. Certaines unités de sortie, telles qu'une perforatrice de ruban, peuvent attendre indéfiniment que le microprocesseur fournisse le caractère suivant et peuvent utiliser ou l'appel sélectif ou l'interruption. Pourtant, d'autres unités de sortie telles qu'une unité de bandes magnétiques peuvent exiger un certain délai d'utilisation et alors la procédure d'interruption doit être utilisée avec soin.

> ***Exemple 4.4.*** Un enregistreur à cassettes d'un modèle bon marché démarre avec une commande de départ du minicalculateur. Après avoir atteint sa vitesse de déroulement, il enregistre 1 000 caractères/seconde. Chaque fois qu'il est prêt à enregistrer un caractère, l'enregistreur envoie un signal de demande d'interruption au minicalculateur, demandant le caractère suivant. Pour un fonctionnement correct de l'unité d'enregistrement, le minicalculateur doit effectuer cette interruption en une période d'environ 1 millième de seconde au plus.

De nombreux calculateurs fournissent un système d'*interruptions multiples* auquel plusieurs unités d'entrée-sortie sont branchées. Les unités peuvent être assignées avec des *priorités* égales ou différentes qui correspondent à des niveaux d'interruption différents.

4.4 ACCÈS DIRECT EN MÉMOIRE

L'*accès direct en mémoire* (DMA) évite le bloc I/O et fournit des transferts directs de données à grande vitesse dans n'importe quelle direction entre la mémoire centrale et une unité périphérique. Habituellement, le fonctionnement du bloc I/O et de la CPU s'arrête pendant l'accès direct en mémoire et la commande de la mémoire est faite par un circuit logique DMA. Cependant, le fait d'avoir un circuit extérieur atténue quelque peu l'avantage d'un gain de rapidité. En conclusion, le DMA est utilisé surtout quand un transfert à grande vitesse de grands blocs est demandé.

PROBLÈMES

1. Déterminer la direction du flot de données dans la figure 4.1 quand l'entrée *DIRECTION* a la valeur 1.
2. Établir un diagramme temporel pour les signaux dans le multiplexeur bidirectionnel de la figure 4.4, le multiplexeur fonctionnant séquentiellement sur une unité d'entrée et deux unités de sortie.
3. Étendre le diagramme temporel du problème 2 en incluant les signaux *DIRECTION* et *ACTIVATE* du registre I/O.
4. Faire un circuit logique montrant les détails du bloc I/O de la figure 4.5, en prenant 3 unités périphériques et des mots de longueur de 4 bits.
5. Dresser un diagramme temporel pour les signaux du circuit du problème 4, le circuit fonctionnant séquentiellement sur une unité d'entrée et deux unités de sortie.
6. Les mémoires tampons I/O et le décodeur ID d'unités I/O peuvent être implantés dans le minicalculateur ou dans les unités I/O. Considérer les avantages et les inconvénients impliqués par chacun de ces usages.
7. Dresser une suite de priorités raisonnables entre une machine à écrire, un enregistreur de cassettes et une perforatrice de ruban. Établir un circuit qui commande les interruptions multiples et fournit aussi au minicalculateur l'identification des unités I/O demandant l'interruption.

5

OPÉRATIONS ARITHMÉTIQUES

Ce chapitre décrit les divers *systèmes de numération* utilisés dans un minicalculateur, par exemple la *représentation en virgule flottante*, et les méthodes de calcul associées. Il présente également plusieurs *techniques de codage* comprenant le *code ASCII* pour représenter les *caractères* de l'alphabet et d'autres symboles.

5.1 SYSTÈMES DE NUMÉRATION

Usuellement, le système de numération utilisé est le *système en base décimale*. Chaque chiffre dans un mot de plusieurs chiffres a une *pondération* qui lui est propre. Par exemple, le nombre décimal 4 956 peut être exprimé comme la somme des coefficients pondérés :

$$4\,956 = 4 \times 10^3 + 9 \times 10^2 + 5 \times 10^1 + 6 \times 10^0.$$

La base décimale n'est pas la seule dans laquelle des nombres peuvent être exprimés. Ainsi, à cause de la nature binaire des circuits du calculateur, le *système binaire* a été largement employé. Ce système utilise uniquement les chiffres 0 et 1. Ainsi $1 + 0 = 0 + 1 = 1$; $1 + 1 = 10$ (lire un, zéro) ; $10 + 1 = 11$, etc. Les règles des opérations arithmétiques sont semblables à celles du système décimal à part le fait que le système binaire utilise un ensemble de deux chiffres binaires $\{ 0, 1 \}$ au lieu d'un ensemble de dix chiffres décimaux $\{ 0, 1, \ldots, 9 \}$.

En général, tout nombre N_b qui comprend n chiffres et est donné en base b peut être exprimé comme la somme de ses coefficients pondérés :

$$N_b = a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + \cdots + a_1b^1 + a_0b^0 = \sum_{i=0}^{n-1} a_ib^i. \quad (5.1)$$

Sous la forme :

$$N_b = a_{n-1}a_{n-2} \cdots a_1a_0. \quad (5.2)$$

la pondération est seulement implicite.

Nombres binaires

Les nombres binaires utilisent la base deux et sont souvent notés avec un indice 2. Par exemple 11_2 exprime un nombre en notation binaire équivalent au nombre décimal 3_{10}. En raison de la grande utilisation du système décimal et du système binaire, les *règles de conversion* entre ces deux systèmes sont décrites en détail. Un nombre donné en une base peut être exprimé par une valeur numérique équivalente dans une autre base au moyen d'un *algorithme de conversion.*

Conversion binaire-décimale

Deux méthodes pour la conversion binaire-décimale sont présentées ici. L'une est fondée sur la formule 5.1 et est illustrée par les exemples 5.1 et 5.2.

Exemple 5.1. Convertir en base dix le nombre entier binaire $N_2 = 111001_2$.

D'après le formule 5.1, N_2 représente la somme :

$$N_2 = 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0.$$

d'où :

$$N_2 = (32 + 16 + 8 + 1)_{10} = 57_{10}.$$

Des nombres fractionnaires peuvent être convertis de la même manière en se rappelant que la valeur de chaque chiffre binaire ou *bit*, après la *virgule binaire* est pondérée par un facteur 1/2. La formule 5.1 peut être étendue aux cas des bits fractionnaires, ainsi a_{-1}, a_{-2}, ..., a_{-m} devient :

$$N_2 \text{ (fractionnaire)} = a_{-1}2^{-1} + a_{-2}2^{-2} + \cdots + a_{-m+1}2^{-m+1} + a_{-m}2^{-m}$$

$$= \sum_{i=-m}^{-1} a_i 2^i. \qquad (5.3)$$

Exemple 5.2. Convertir en base dix la fraction binaire $N_2 = 0{,}11001_2$:

$$N_2 \text{ (fractionnaire)} = 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} = (0.5 + 0.25 + 0.03125)_{10} = 0.78125_{10}.$$

Une autre manière de convertir un nombre binaire en nombre décimal, applicable uniquement aux nombres entiers, est la méthode suivante.

Elle est fondée sur la représentation parenthésée ci-dessous des nombres binaires :

$$N_2 = \{[(a_{n-1} \cdot 2 + a_{n-2}) \cdot 2 + a_{n-3}] \cdot 2 + \cdots + a_1\} \cdot 2 + a_0 \cdot \quad (5.4)$$

La formule 5.4 montre la méthode de conversion. Le bit le plus significatif (bps) est doublé puis ajouté au coefficient du bit suivant qui peut être 0 ou 1. Le résultat est à nouveau doublé et le processus est terminé lorsque le bit le moins significatif (bms) a été finalement ajouté.

Exemple 5.3. Pour convertir le nombre binaire 1001101 en nombre décimal au moyen de cette méthode, nous procédons comme dans la figure 5.1.

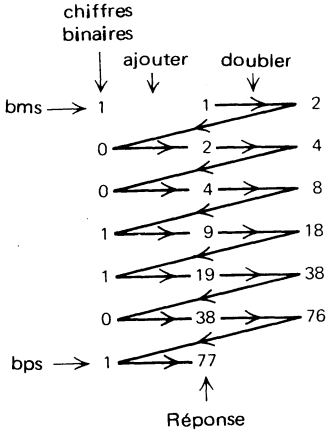


FIG. 5.1. — Conversion binaire-décimale.

Conversion décimale-binaire

Une méthode pour la conversion décimale-binaire à la main, commode pour les petits nombres entiers ou fractionnaires, est fondée sur la recherche des puissances de 2 contenues dans un nombre décimal. Elle démarre en soustrayant la plus grande puissance de 2 contenue dans le nombre décimal à convertir et en continuant jusqu'à ce que la conversion soit faite.

Exemple 5.4. Pour convertir 167_{10} en nombre binaire, on note que 2^7 est la plus grande puissance de 2 contenue dans le nombre à convertir. Ainsi $167 - 2^7 = 167 - 128 = 39$. Puis $39 - 2^5 = 7 = 111_2$. D'où la réponse: $167 = 2^7 + 2^5 + 2^2 + 2^1 + 2^0$ c'est-à-dire 10100111_2.

Cette méthode est peu pratique pour de grands nombres. Une autre méthode pour passer de décimal en binaire consiste en divisions successives par 2 du nombre entier décimal. Les restes représentent le nombre binaire, le premier étant le bit le moins significatif.

Exemple 5.5. Pour convertir l'entier 167_{10} en binaire, nous procédons comme suit :

	Reste	Poids binaire
$167 \div 2 = 83$	1	2^0 (bms)
$83 \div 2 = 41$	1	2^1
$41 \div 2 = 20$	1	2^2
$20 \div 2 = 10$	0	2^3
$10 \div 2 = 5$	0	2^4
$5 \div 2 = 2$	1	2^5
$2 \div 2 = 1$	0	2^6
$1 \div 2 = 0$	1	2^7 (bps)

La réponse est obtenue en lisant la colonne des restes de bas en haut : $167_{10} = 10100111_2$.

Quand le nombre est fractionnaire, il est multiplié successivement par 2 et le résultat est obtenu en lisant la colonne des *parties entières* de haut en bas. Les nombres fractionnaires décimaux ne peuvent pas tous être convertis en fraction binaire avec un nombre fini de bits et une erreur d'*arrondi* en résulte quand le processus est arrêté.

Exemple 5.6. Pour convertir $0{,}57_{10}$ en fraction binaire, nous procédons comme suit :

	Partie entière	Poids binaire
$0.57 \times 2 = 1.14$	1	2^{-1} (bps)
$0.14 \times 2 = 0.28$	0	2^{-2}
$0.28 \times 2 = 0.56$	0	2^{-3}
$0.56 \times 2 = 1.12$	1	2^{-4}
$0.12 \times 2 = 0.24$	0	2^{-5}
$0.24 \times 2 = 0.48$	0	2^{-6}
$0.48 \times 2 = 0.96$	0	2^{-7}
$0.96 \times 2 = 1.92$	1	2^{-8}

A ce point nous arrêtons arbitrairement le processus. Le résultat, en lisant la colonne partie entière de haut en bas, est $0{,}57_{10} = 0{,}10010001 + \varepsilon$, avec une erreur d'arrondi de $\varepsilon < 2^{-8} = 1/256$.

La conversion d'un nombre mixte consistant en une partie entière et une partie fractionnaire peut être obtenue comme suit. Les parties entière et fractionnaire sont d'abord séparées puis la conversion des parties entière et fractionnaire est effectuée au moyen des algorithmes de conversion respectifs et finalement les deux résultats sont associés.

Représentations de nombres négatifs

Jusqu'ici nous avons considéré les nombres binaires entiers et fractionnaires sans nous occuper de leur signe. Dans ce qui suit, nous décrivons trois représentations dans lesquelles on distingue les nombres positifs et négatifs. Chacune de ces représentations utilise un *bit supplémentaire de signe* qui devient le bit le plus à gauche du nombre binaire avec signe. Les trois représentations sont la *représentation signe-valeur absolue*, la *représentation complément à 1* et la *représentation complément à 2*.

Représentation signe-valeur absolue

La valeur absolue du nombre dans cette représentation est exprimée sous la forme binaire de la formule 5.1. Le bit de signe, placé à la gauche du bit le plus significatif, est 0 pour « + » et 1 pour « − ». Aussi le nombre positif $+ 5{,}25_{10}$ est représenté par 0101,01 et le nombre négatif $- 5{,}25_{10}$ par 1101,01. Un nombre binaire de n bits entiers et m bits fractionnaires exprimé dans la représentation signe-valeur absolue par $n + m + 1$ bits a la valeur :

$$N = (-1)^{b_n} \cdot \left(\sum_{i=0}^{n-1} b_i 2^i + \sum_{i=-m}^{-1} b_i 2^i \right)$$

$$= (-1)^{b_n} \cdot \sum_{i=-m}^{n-1} b_i 2^i, \qquad (5.5)$$

où les b_i sont les coefficients binaires 0 ou 1.

Cette représentation, quoique facilement lisible, ne se prête pas à une mise en œuvre facile dans des circuits arithmétiques digitaux pour plusieurs raisons. L'une d'elle est qu'il y a deux expressions différentes pour le nombre zéro : 000 . . . 0 et 100 . . . 0. Également, quand on additionne un nombre positif et un nombre négatif on doit d'abord comparer les

valeurs absolues des deux nombres, soustraire la plus petite valeur absolue de la plus grande et ensuite assigner au résultat le signe du nombre ayant la plus grande valeur absolue.

Représentation complément à 1

Dans cette représentation les nombres positifs sont exprimés de la même façon que dans la représentation précédente. La convention du bit de signe est également la même dans les deux représentations, 0 pour « + » et 1 pour « − ». Cependant pour exprimer la partie correspondant à la valeur absolue d'un nombre négatif dans la représentation complément à 1 on doit prendre son *complément* à 1. Le complément à 1 d'un nombre est obtenu par *inversion logique* de chaque bit du nombre, c'est-à-dire en changeant les 0 en 1 et les 1 en 0. Notons que le complément du complément est le nombre lui-même. Notons également que le bit de signe, qui est bit le plus significatif, est (contrairement à la représentation signe-valeur absolue) traité de la même façon dans l'addition et la soustraction que les bits représentant la valeur absolue.

> ***Exemple 5.7.*** Le nombre positif $+ 5{,}25_{10}$ est exprimé en représentation complément à 1 par $0101{,}01_2$ (de la même façon que dans la représentation signe-valeur absolue). Pour exprimer $- 5{,}25_{10}$, on exprime d'abord la valeur absolue, on prend le complément de $101{,}01_2$ qui est 010,10, puis on assigne un bit de signe négatif (qui est 1) à la gauche du bit le plus significatif. Ainsi la représentation complément à 1 de $- 5{,}25_{10}$ est $1\ 010{,}10_2$.

On peut montrer qu'un nombre binaire de n bits entiers et m bits fractionnaires exprimé en représentation complément à 1 par $n + m + 1$ bits a la valeur :

$$N = (1 - b_n) \cdot \sum_{i=-m}^{n-1} b_i 2^i - b_n \cdot \sum_{i=-m}^{n-1} (1 - b_i)\, 2^i, \qquad (5.6)$$

où les b_i sont les coefficients binaires 0 ou 1. Notons que $1 - b_i$ est le complément à 1 du $i$ème bit. Pour les nombres positifs la deuxième partie du terme de droite de la formule (5.6) se réduit à 0 et pour les nombres négatifs, la première partie devient zéro.

Notons également que (comme c'était le cas dans la représentation signe-valeur absolue) dans la représentation complément à 1, il y a deux expressions différentes du nombre 0 qui sont maintenant 00 . . . 0 et 11 . . . 1.

Représentation complément à 2

La représentation complément à 2 des nombres est largement utilisée dans les minicalculateurs. Les nombres positifs sont exprimés de la même façon dans les trois représentations, le bit de signe étant le même, 0 pour « + » et 1 pour « − ». Pour exprimer la valeur absolue d'un nombre négatif dans la représentation complément à 2, on doit prendre son *complément à 2.* Le complément à 2 d'un nombre peut être obtenu en calculant d'abord son complément à 1 et en ajoutant 1 au bit le moins significatif, qu'il soit entier ou fractionnaire. Notons que (comme dans la représentation précédente) le bit de signe dans l'addition et la soustraction est traité de la même façon que les bits représentant la valeur absolue.

Exemple 5.8. *a*) Le complément à 2 de $58_{10} = 111010_2$ est obtenu comme suit :

$$\left.\begin{array}{rl} 58_{10} = & 0\ 111010 \\ \text{complément à 1 de } 58_{10} = & 1\ 000101 \\ \text{ajouter 1 au bms :} & \qquad\quad 1 \end{array}\right\} \text{ajouter}$$

résultat : complément à 2 de $58_{10} = 1\ 000110$.

b) Le complément à 2 de $42{,}5_{10}$ est obtenu comme suit :

$$\left.\begin{array}{rl} 42{,}5_{10} = & 0\ 101010{,}1 \\ \text{complément à 1 de } 42{,}5_{10} = & 1\ 010101{,}0 \\ \text{ajouter 1 au bms :} & \qquad\quad\ \ 1 \end{array}\right\} \text{ajouter}$$

résultat : complément à 2 de $42{,}5_{10} = 1\ 010101{,}1$.

On peut montrer qu'un nombre binaire comprenant n bits entiers et m bits fractionnaires exprimé dans la représentation complément à 2 par $n + m + 1$ bits a la valeur :

$$N = (1 - b_n) \cdot \sum_{i=-m}^{n-1} b_i 2^i - b_n \cdot \sum_{i=-m}^{n-1} [(1 - b_i)2^i] + 2^{-m}. \quad (5.7)$$

Pour les nombres entiers, la formule 5.7 se réduit à :

$$N = (1 - b_n) \cdot \sum_{i=0}^{n-1} b_i 2^i - b_n \cdot \sum_{i=0}^{n-1} [(1 - b_i)2^i] + 1. \quad (5.8)$$

Notons que, contrairement aux deux autres représentations, celle-ci n'a qu'une expression pour le nombre 0, c'est-à-dire 000 . . . 0. Le tableau 5.1 montre plusieurs nombres exprimés dans les trois représentations ci-dessus.

TABLEAU 5.1. — **Nombres binaires dans les trois représentations.**

Nombre décimal	Représentation signe-valeur absolue	Représentation complément à 1	Représentation complément à 2
+7	0 111	0 111	0 111
+6	0 110	0 110	0 110
+5	0 101	0 101	0 101
+4	0 100	0 100	0 100
+3	0 011	0 011	0 011
+2	0 010	0 010	0 010
+1	0 001	0 001	0 001
0	0 000 1 000	0 000 1 111	0 000
−1	1 001	1 110	1 111
−2	1 010	1 101	1 110
−3	1 011	1 100	1 101
−4	1 100	1 011	1 100
−5	1 101	1 010	1 011
−6	1 110	1 001	1 010
−7	1 111	1 000	1 001

5.2 REPRÉSENTATIONS DES NOMBRES EN OCTAL ET EN HEXADÉCIMAL

Une longue suite de 1 et 0 représentant une certaine valeur numérique n'est pas facilement manipulable par l'opérateur. Deux solutions sont en général adoptées : le *codage* dont on parlera dans le paragraphe suivant et l'utilisation de nombres dont la base est une puissance entière de 2 comme le *système octal* (base 2^3) et le *système hexadécimal* (base 2^4).

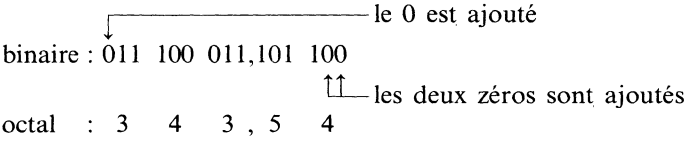
Le système de numération octal

Dans ce système la base est $b = 2^3 = 8_{10}$; ainsi 8 *symboles* de 0 à 7 sont nécessaires pour écrire un nombre. Un nombre octal N_8 comprenant n chiffres entiers et m chiffres fractionnaires peut être représenté par la somme :

$$N_8 = a_{n-1}8^{n-1} + a_{n-2}8^{n-2} + \cdots + a_1 8^1 + a_0 8^0 + a_{-1}8^{-1} + \cdots + a_{-m}8^{-m} = \sum_{i=-m}^{n-1} a_i 8^i. \qquad (5.9)$$

Les huit combinaisons pour les huit symboles utilisés dans le système de numération octal peuvent être représentées exactement par 3 bits binaires. Ainsi la conversion d'un nombre binaire en un nombre octal peut être réalisée d'après la règle suivante : mettre la partie entière du nombre binaire par groupes de 3, en partant du bit entier le plus à droite ; la partie fractionnaire est groupée de la même façon mais à partir du bit fractionnaire le plus à gauche.

Exemple 5.9. Pour convertir le nombre binaire 11100011,1011 en nombre octal on a :



Des représentations en système octal de divers nombres sont données dans le tableau 5.2.

TABLEAU 5.2. — **Représentation en décimal, binaire, octal et hexadécimal.**

Décimal	Binaire	Octal	Hexadécimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Le système de numération hexadécimal

Dans ce système la base est $b = 2^4$; ainsi 4 bits sont nécessaires pour exprimer un chiffre hexadécimal et il y a $2^4 = 16_{10}$ symboles. Les symboles sont 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F ; A représente dix, B onze, etc. jusqu'à F qui représente quinze. Le tableau 5.2 montre les représentations hexadécimales de divers nombres.

Les 16 combinaisons différentes (symboles) peuvent être représentées exactement par 4 bits. Ainsi la règle de conversion d'un nombre binaire en un nombre hexadécimal est la suivante : mettre la partie entière d'un nombre binaire en groupes de 4 en partant du bit entier le plus à droite ; de même pour la partie fractionnaire mais à partir du bit fractionnaire le plus à gauche.

Exemple 5.10. Pour convertir le nombre binaire 11100011,1011 ($= 227{,}6875_{10}$) en nombre hexadécimal on a :

binaire : 1110 0011 , 1011
hexadécimal : E 3 , B .

Comparaison des systèmes de numération

Un nombre N peut être exprimé en représentation binaire par b chiffres, en représentation octale par e chiffres, en représentation décimale par d chiffres et en représentation hexadécimale par h chiffres où N, b, e, d et h sont reliés par :

$$N = 2^{b}{}_{10} = 8^{e}{}_{10} = 10^{d}{}_{10} = 16^{h}{}_{10}. \tag{5.10}$$

D'après la formule 5.10 on obtient également :

$$\frac{e}{b} = 3, \quad \frac{h}{b} = 4, \quad \frac{h}{e} = \frac{4}{3}, \quad \frac{b}{d} = \frac{1}{\log_{10}2} = 3.32, \quad \frac{e}{d} = \frac{1}{\log_{10}8} = 1.05,$$

$$\frac{h}{d} = \frac{1}{\log_{10}16} = 0.895. \tag{5.11}$$

Dans la conversion binaire-décimale décrite dans l'exemple 5.5 on divise à plusieurs reprises le nombre décimal entier par 2 pour obtenir le nombre binaire équivalent. On peut généraliser cette méthode comme suit. Si un nombre en base b_1 doit être converti en base b_2, le calcul pour la conversion doit être effectué en base b_1. En outre, la conversion de nombres entiers nécessite des divisions successives tandis que la conversion d'un

nombre fractionnaire nécessite des multiplications successives. Ainsi la conversion d'octal (respectivement d'hexadécimal) en décimal utilise l'addition et la multiplication en octal (respectivement en hexadécimal).

De telles conversions et de telles opérations arithmétiques octales et hexadécimales sont facilitées par l'utilisation des tables d'addition et de multiplication données dans les annexes A et B.

5.3 CODAGE

Il y a quatre applications principales du codage dans un mini-calculateur : (1) la réduction du nombre de chiffres à un niveau qui puisse être manipulable par un opérateur ; (2) le codage par des 0 et des 1 pour obtenir des nombres en décimal-codé-binaire qui sont pratiques comme représentations I/O ; (3) le codage pour détecter les erreurs se produisant pendant la transmission des informations entre les diverses parties du calculateur ; (4) le codage pour mettre sous forme binaire les lettres de l'alphabet et d'autres symboles, en particulier ceux nécessaires pour les liaisons entre le calculateur et ses unités périphériques (par exemple, « retour chariot » d'une machine à écrire).

Codage pour réduire le nombre de chiffres

Les nombres en octal et hexadécimal dont on a parlé au paragraphe 5.2 sont de bons exemples des représentations de nombres qui demandent moins de chiffres que la représentation binaire.

> ***Exemple 5.11.*** Le nombre de chiffres nécessaires pour exprimer un nombre entier en octal est trois fois plus petit qu'en binaire et quatre fois plus petit en hexadécimal qu'en binaire.

Nombres en décimal-codé-binaire

La représentation en décimal-codé-binaire (BCD) utilise 4 bits pour exprimer les 10 combinaisons correspondant aux chiffres décimaux de 0 à 9. En réalité, 4 bits pourraient représenter au total 16 nombres. Ainsi existe une redondance permettant divers codes BCD. Trois de ces codes sont montrés dans le tableau 5.3. Le code BCD utilisé le plus couramment est le code 8-4-2-1 ou *code BCD à pondération naturelle* (voir la colonne 2 du tableau). La colonne 3 montre le *code BCD 2-4-2-1* tandis

que la dernière colonne montre le *code BCD « plus trois »* dérivé du code BCD 8-4-2-1 en ajoutant 3_{10} à chaque nombre codé. Les deux derniers codes ont des propriétés de symétrie (montrées par la ligne en pointillé) facilitant la *représentation « complément à 9 »* utile dans la soustraction en BCD.

Codage pour détecter des erreurs (tests de parité)

Des codes numériques ont été développés pour détecter et même corriger des erreurs qui se produisent dans les données pendant leurs diverses transmissions dans le calculateur et entre le calculateur et ses unités périphériques. De tels codes nécessitent des bits de test supplémentaires qui aident à détecter les erreurs et parfois aussi à localiser le ou les bits erronés. Nous ne parlerons ici que du code de détection d'erreur le plus simple et le plus couramment utilisé : le test de parité. Un test de parité améliore l'assurance de non-erreur dans la transmission des données en ajoutant un bit supplémentaire à l'information codée. Il existe deux types de test de parité : la *parité impaire* et la *parité paire*. Pour le test de parité impaire le bit de contrôle additionnel, p, a une valeur telle que :

$$p \oplus X_n \oplus X_{n-1} \oplus \cdots \oplus X_1 \oplus X_0 = 1, \qquad (5.12)$$

où le symbole $\oplus$ indique la *somme modulo 2* : $1 \oplus 0 = 0 \oplus 1 = 1$, $1 \oplus 1 = 0$, $1 \oplus 1 \oplus 1 = 1$, etc. Dans le test de parité paire le bit de contrôle p est choisi tel que :

$$p \oplus X_n \oplus X_{n-1} \oplus \cdots \oplus X_1 \oplus X_0 = 0. \qquad (5.13)$$

TABLEAU 5.3. — **Nombres BCD.**

Nombre décimal	BCD 8421	BCD 2421	BCD plus trois
0	0 0 0 0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 0 0 1	0 1 0 0
2	0 0 1 0	0 0 1 0	0 1 0 1
3	0 0 1 1	0 0 1 1	0 1 1 0
4	0 1 0 0	0 1 0 0	0 1 1 1
5	0 1 0 1	1 0 1 1	1 0 0 0
6	0 1 1 0	1 1 0 0	1 0 0 1
7	0 1 1 1	1 1 0 1	1 0 1 0
8	1 0 0 0	1 1 1 0	1 0 1 1
9	1 0 0 1	1 1 1 1	1 1 0 0

Exemple 5.12. Le tableau 5.4 montre les bits de test de parités paire et impaire pour 10 nombres BCD.

Notons que les deux tests de parités paire et impaire ne détectent qu'un nombre impair d'erreurs. Cependant la probabilité d'erreurs double, quadruple, etc., est négligeable quand le taux d'erreur simple est bas. Néanmoins des codes de détection d'erreurs plus sophistiqués ont été également mis au point pour détecter des blocs d'erreurs pendant la transmission. Nous ne présentons pas ces codes ici.

TABLEAU 5.4. — **Tests de parités paire et impaire.**

Nombre décimal	Code BCD 8 4 2 1	bit de test de parité	
		parité impaire	parité paire
0	0 0 0 0	1	0
1	0 0 0 1	0	1
2	0 0 1 0	0	1
3	0 0 1 1	1	0
4	0 1 0 0	0	1
5	0 1 0 1	1	0
6	0 1 1 0	1	0
7	0 1 1 1	0	1
8	1 0 0 0	0	1
9	1 0 0 1	1	0

Codage de lettres et d'autres symboles (codes ASCII)

Plusieurs codes ont été développés pour représenter en binaire les lettres de l'alphabet et d'autres symboles qui sont habituellement utilisés sur les claviers des unités périphériques. Le *code ASCII* a atteint une large diffusion dans les systèmes des minicalculateurs ; ASCII étant les initiales pour American Standard Code for Information Interchange.

Le code ASCII complet demande 8 bits ; donc 256 caractères et symboles, comprenant les lettres majuscules et minuscules, peuvent être représentés. Puisque de nombreux minicalculateurs ont des longueurs de mot multiples de 4, le code ASCII le plus courant sera le code à 8 bits. Pourtant des codes ASCII à 6 et 7 bits ont été également mis au point et peuvent être utilisés avec l'inconvénient d'une programmation plus compliquée.

5.4 REPRÉSENTATION EN VIRGULE FLOTTANTE ET RÈGLES DE CALCUL

Représentation en virgule flottante

La *représentation* de nombres en *virgule flottante*, semblable à la notation scientifique, est importante pour obtenir le maximum de précision quand on opère sur de grands nombres ou de petits nombres. Elle nécessite deux quantités : l'*exposant* E ou *caractéristique* et la *mantisse* M.

> ***Exemple 5.13.*** La vitesse de la lumière est de 300 000 000 m/s. En notation scientifique cela s'écrit $0{,}3 \times 10^9$ m/s et en représentation en virgule flottante avec un exposant 9 et une mantisse 0,3.

Le nombre de mots requis pour exprimer E et M est déterminé par la précision désirée et par la longueur de mot du minicalculateur. La mantisse M est choisie telle que :

$$1/2 \leqslant |M| < 1 \qquad (5.14)$$

L'exposant et la mantisse peuvent être exprimés en représentation « complément à 2 », la valeur de M doit être *normalisée* pour satisfaire à la formule 5.14.

> ***Exemple 5.14.*** Pour exprimer $0{,}0074_8 = 0{,}000000111100_2$ en représentation en virgule flottante, on note que M doit satisfaire à la formule 5.14, donc doit être normalisé. La normalisation peut être effectuée en rejetant le nombre binaire dans le domaine donné par la formule 5.14. Dans cet exemple, ceci est obtenu en décalant de 6 bits à gauche. Donc $0{,}0074_8$ en virgule flottante est exprimé par un exposant $E = 1\ 1010$ et par une mantisse $M = 0\ 1111$ où E est en représentation complément à 2 et où les zéros suivants de M ont été supprimés.

La figure 5.2 montre une représentation en virgule flottante utilisant quatre mots de 8 bits. L'exposant est exprimé par les 7 bits du premier mot dans une notation « plus 64 » dans laquelle 64_{10} est ajouté à l'exposant afin de manipuler les exposants positifs et négatifs sans utiliser de bit de signe. Le bit le plus significatif du premier mot est utilisé comme bit de signe de la mantisse et les 24 bits restants de la mantisse sont contenus dans

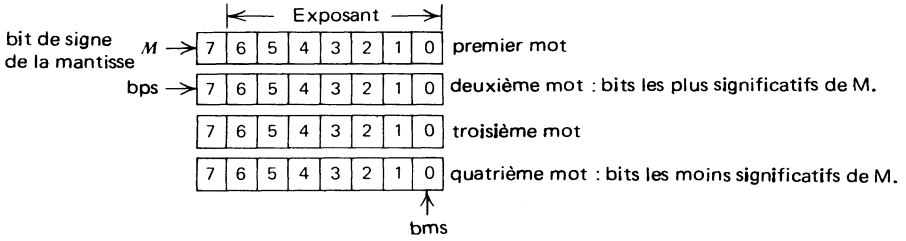


FIG. 5.2. — Représentation en virgule flottante avec 4 mots de 8 bits.

les second, troisième et quatrième mots. L'ensemble des nombres N qui peuvent être représentés par les 4 mots de la figure 5.4 peut s'écrire (voir également le problème 10) :

$$2^{-65} \leq N \leq (1 - 2^{-24}) \cdot 2^{63}. \tag{5.15}$$

Règles de calcul

Pour additionner ou soustraire deux nombres exprimés en virgule flottante, on écrit les deux nombres, de sorte que l'exposant du nombre de plus grande valeur absolue devienne l'exposant du résultat.

Pour la multiplication, les exposants sont additionnés et les mantisses multipliées. Ainsi :

$$X \cdot Y = (2^{E_X + E_Y}) \cdot (M_X \cdot M_Y). \tag{5.16}$$

La division des nombres est faite en soustrayant les exposants et en divisant les mantisses :

$$X \div Y = (2^{E_X - E_Y}) \cdot (M_X \div M_Y). \tag{5.17}$$

Le résultat d'une opération arithmétique utilisant les nombres en virgule flottante doit également être normalisé si la mantisse M du résultat ne vérifie pas la formule 5.14.

PROBLÈMES

1. Convertir en décimal les nombres binaires 101101 ; 0,101110 ; 110,101.
2. Trouver l'équivalent binaire des nombres décimaux : 123 ; 0,28125 ; 224,375.
3. Écrire la formule 5.3 en la généralisant pour qu'elle soit applicable dans toute base b.
4. Comparer les formules 5.1 et 5.4 et établir la validité de l'algorithme de l'exemple 5.3 pour la conversion binaire-décimale.

5. On donne les nombres binaires : 1 1010,01 ; 0 1010,01 ; 1 110010 ; 0 110010 ; 1 0100,11 ; 0 0100,11. Utiliser les formules 5.5 et 5.8 pour trouver les nombres décimaux équivalents si les nombres sont (*a*) en représentation signe-valeur absolue, (*b*) en représentation « complément à 1 », (*c*) en représentation « complément à 2 ».

6. Exprimer les nombres suivants dans la représentation « signe-valeur absolue », « complément à 1 » et « complément à 2 » : $-1001{,}5_{10}$; -853_{10} ; $-0{,}153_{10}$; $-52{,}0625_{10}$. L'erreur d'arrondi devra être inférieure au plus petit chiffre significatif donné.

7. Exprimer les nombres suivants dans la représentation signe-valeur absolue 8-4-2-1 BCD : $-93{,}2_{10}$; $-107{,}25_{10}$; $-0{,}769_{10}$.

8. Établir des algorithmes pour l'addition et la soustraction en utilisant le code BCD « plus 3 ». Considérer les corrections à appliquer quand (*a*) le résultat d'une addition est inférieur à 10_{10} dans le code considéré (*b*) une retenue au prochain chiffre BCD est générée dans une addition, (*c*) le résultat d'une soustraction est inférieur à 10_{10} et (*d*) une retenue pour le prochain chiffre est générée dans une soustraction.

9. Exprimer les nombres suivants dans la représentation en virgule flottante : $+364_8$; $-27{,}3_8$; $+0A{,}04_{16}$; $-0{,}025_{16}$.

10. Vérifier la formule 5.15. On remarquera que la plus petite valeur de $|M|$ est 2^{-1} et la plus grande valeur $(2^{24}-1).2^{-24}$.

11. Déterminer le décalage nécessaire pour effectuer l'addition de deux nombres en virgule flottante quand un nombre a un exposant positif et l'autre un exposant négatif.

12. Utiliser la représentation binaire en virgule flottante et calculer $763{,}2_{10} - 0{,}421_{10}$.

6

CIRCUITS ARITHMÉTIQUES ET LOGIQUES

Les opérations arithmétiques et logiques dans un minicalculateur sont effectuées par des *circuits arithmétiques et logiques.* Les circuits pour les opérations logiques, les additions et les soustractions forment habituellement une *unité arithmétique et logique* (ALU). Ce chapitre fournit des détails sur les *additionneurs et soustracteurs binaires*, les additionneurs BCD et les fonctions logiques de l'ALU ainsi qu'une brève description de l'accumulateur et des *circuits multiplicateurs et diviseurs.*

6.1 ADDITIONNEURS ET SOUSTRACTEURS

Additionneurs binaires

Dans un minicalculateur, deux nombres sont additionnés en prenant le premier opérande de l'addition dans l'accumulateur (registre A) et le deuxième opérande dans un autre registre ou dans la mémoire centrale et en les additionnant dans un circuit additionneur. Le résultat est placé dans l'accumulateur, en effaçant le contenu précédent. Des additions successives peuvent également être effectuées de cette façon, la valeur de l'accumulateur étant la nouvelle somme après chaque opération. Un additionneur contient également une *bascule de dépassement de capacité (ou indicateur) et/ou un indicateur de retenue* (appelé également *bascule de lien*) qui est un bit supplémentaire de l'accumulateur. Dans tous les cas, toutes les fois qu'il y a l'indication d'un dépassement de capacité ou d'une retenue, ceci est reconnue seulement par la CPU mais aucune action automatique n'est effectuée. Le programmeur a la responsabilité de tester l'état de l'indicateur, d'agir sur lui et d'effacer l'indicateur pour une utilisation future.

Le circuit de base dans un additionneur est l'*additionneur élémentaire* de la figure 6.1 (*a*) dans lequel A_i et B_i représentent les bits respectivement du premier opérande et du deuxième opérande de l'addition, S_i la somme, C_{in} le report d'entrée et C_{out} le report de sortie. Cet additionneur est capable de faire des additions *en série* bit par bit et est aussi le bloc de base pour les *additionneurs parallèles*.

L'additionneur parallèle le plus simple est l'*additionneur en cascade* qui est réalisé par une cascade de n additionneurs élémentaires pour l'addition de deux nombres de n bits. Le signal de report de sortie C_{out}, doit cependant se propager par n *paliers* ce qui peut causer un retard signifi-

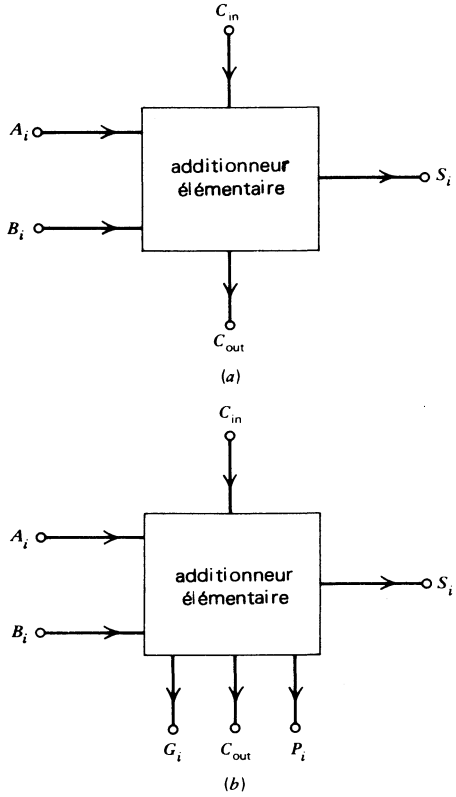


FIG. 6.1. — Additionneur élémentaire.
(*a*) Avec report d'entrée C_{in} et report de sortie C_{out}.
(*b*) Avec C_{in} et C_{out} et avec sorties G_i et P_i.

catif dans l'obtention du résultat final. Un *additionneur à report automatique* ou *additionneur à report simultané*, plus rapide peut être obtenu au prix d'un circuit plus complexe. Dans ce type d'additionneur (voir fig. 6.1) deux signaux auxiliaires, G_i (générateur) et P_i (propagateur) sont générés à chaque bit, avec $G_i = A_i B_i$ et $P_i = A_i \oplus B_i = A_i \bar{B}_i + \bar{A}_i B_i$. On peut montrer que les reports d'entrée et de sortie dans un additionneur de 4 bits sont donnés par les formules :

$$C_{in_0} = 0, \tag{6.1a}$$

$$C_{in_1} = G_0, \tag{6.1b}$$

$$C_{in_2} = G_1 + G_0 P_1, \tag{6.1c}$$

$$C_{in_3} = G_2 + G_1 P_2 + G_0 P_1 P_2, \tag{6.1d}$$

$$C_{out} = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3. \tag{6.1e}$$

Notons que le temps de propagation d'un report de sortie C_{out} d'un additionneur à report automatique est plus court que celui d'un additionneur en cascade, puisque C_{out} ne nécessite pas le passage par quatre additionneurs élémentaires mais seulement par deux niveaux d'un circuit logique. Un additionneur à report automatique de 4 bits est schématisé figure 6.2. Un total de n additionneurs de ce type peut être mis en cascade pour obtenir un *additionneur hybride* à 4 n bits.

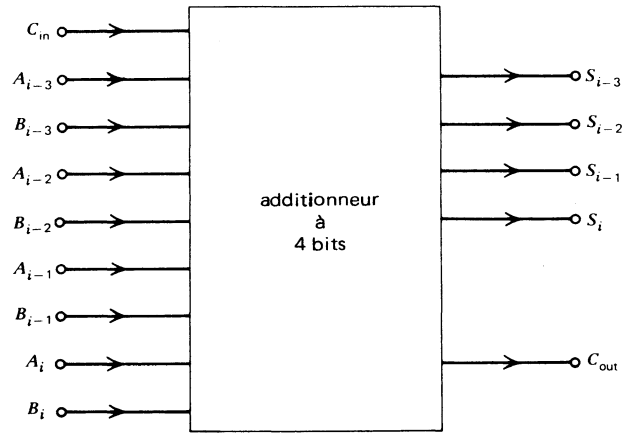


FIG. 6.2. — Additionneur à report automatique à 4 *bits* comprenant 4 circuits de la figure 6.1 *b* et des circuits logiques introduisant les formules 6.1.

Soustracteurs binaires

Puisque les nombres négatifs peuvent être exprimés dans trois représentations différentes, il existe au moins trois types de soustracteurs : les *soustracteurs signe-valeur absolue*, les *soustracteurs « complément à 1 »* et les *soustracteurs « complément à 2 »*.

Soustracteurs signe-valeur absolue

Ils utilisent un soustracteur élémentaire, semblable à l'additionneur élémentaire, comme bloc de base ; des *soustracteurs à report automatique* peuvent également être construits comme les additionneurs correspondants. En effectuant la soustraction $A - B$, on doit d'abord savoir lequel des deux nombres A et B a la plus grande valeur absolue. Le résultat aura le signe de ce nombre. La valeur absolue du résultat est obtenue par $A - B$ quand $A > B$ et par $B - A$ quand $A < B$.

Soustracteurs « complément à 1 »

Dans cette représentation la différence $A - B$ peut être obtenue par $-(B - A)$ en utilisant un additionneur et en ajoutant le complément à 1 du terme A à la quantité B à soustraire. Cependant, dans un tel circuit une correction est nécessaire quand $A - B$ est négatif ; la correction peut être obtenue en utilisant un *report bouclé* de C_{out} à C_{in} comme le montre la figure 6.3.

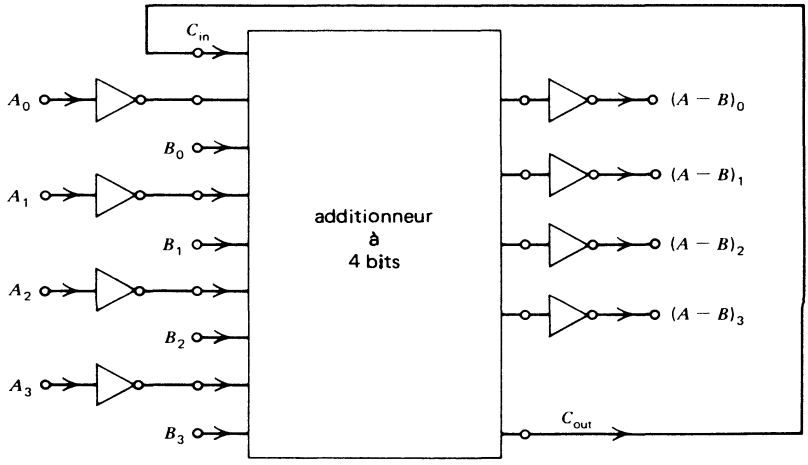


FIG. 6.3. — Soustracteur « complément à 1 » utilisant l'additionneur à 4 bits de la figure 6.2 et un report bouclé.

Exemple 6.1. La différence $A - B$ pour $A = 35_8$ et $B = 57_8$ peut être obtenue en utilisant le circuit de la figure 6.3 comme suit :

$A-B=-(B-A)$, donc on peut écrire :

$B= \quad 57_8=0\ 101111$
$-A=-35_8=1\ 100010 \leftarrow$ complément à 1 de A
$1\ \ 0\ 010001 \leftarrow$ somme incorrecte
$1 \leftarrow$ report bouclé
$B-A=0\ 010010 \leftarrow$ report bouclé ajouté à la somme incorrecte
$A-B=1\ 101101 \leftarrow$ représentation « complément à 1 » de $(A-B)$.

Soustracteurs « complément à 2 »

Dans cette représentation, le complément à 2 de la quantité à soustraire est d'abord généré et ensuite additionné au terme dont on soustrait. Donc les opérations suivantes sont effectuées : inverser tous les bits de la quantité à soustraire au moyen d'inverseurs, additionner 1 au résultat pour obtenir le complément à 2 de la quantité à soustraire, additionner le terme dont on soustrait au complément à 2 de la quantité à soustraire.

Additionneur décimal-codé-binaire

Dans certains minicalculateurs, les opérations arithmétiques sont effectuées en décimal-codé-binaire (BCD) le plus souvent en BCD 8-4-2-1. Tandis que les opérations sur les bits d'un chiffre décimal sont traitées en parallèle, les chiffres sont souvent traités en série.

L'addition de deux nombres BCD demande deux registres de 4 bits qu'on suppose être l'accumulateur (registre A) et le registre B. Les chiffres les moins significatifs des deux opérandes sont d'abord additionnés fournissant le chiffre de la somme le moins significatif et peut-être également une retenue. Le chiffre somme résultant est mis dans l'accumulateur à l'instant d'une impulsion d'horloge tandis que la retenue est retardée d'une période d'horloge et ensuite ajoutée au chiffre somme suivant. Les contenus de l'accumulateur et du registre B sont décalés à droite à l'addition de chaque paire de chiffres jusqu'à ce que tous les chiffres aient été traités. La figure 6.4 illustre les trois premières étapes dans l'addition des deux nombres BCD à 4 chiffres 64,327 et 2 189.

Un additionneur BCD doit considérer pour l'addition de deux nom-

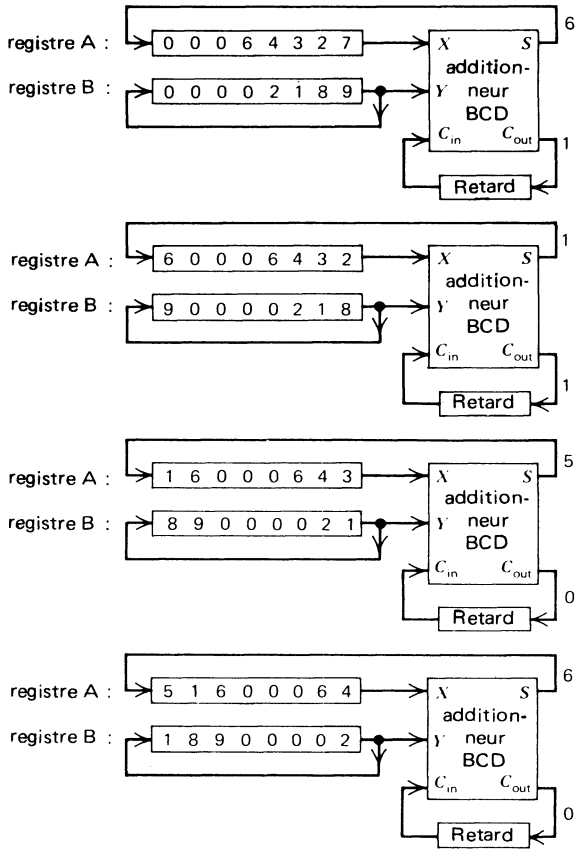


FIG. 6.4. — Trois étapes dans l'addition des deux nombres BCD $64{,}327_{10}$ et $2\,189_{10}$. Chaque bloc additionneur BCD contient un additionneur élémentaire binaire de 4 bits et un circuit correcteur (voir ci-dessus).

bres BCD les trois situations suivantes qui peuvent surgir dans le code BCD 8-4-2-1 : (1) la somme de deux nombres, S, est telle que $000 \leqslant S \leqslant 1001$, le résultat est correct ; (2) $1010 \leqslant S \leqslant 1111$, dans ce cas le résultat est une combinaison BCD non valable, de sorte que l'additionneur BCD doit faire une correction ; et (3) $10000 \leqslant S \leqslant 11001$ dans ce cas un report est généré et le résultat (lu dans le code BCD donné) est incorrect. Une correction est donc requise dans certains cas. Un circuit correcteur dans l'additionneur BCD examine le code BCD de la somme S et lui ajoute 6_{10} quand les cas (2) ou (3) se présentent.

Exemple 6.2. Cet exemple montre les résultats de trois additions BCD, les conditions nécessitant des corrections et les corrections faites.

(1) 5_{10} =	0101		
3_{10} =	0011		
somme =	1000	$=8_{10}$	(résultat correct)
(2) 6_{10} =	0110		
8_{10} =	1000		
(somme =)	1110		(nombre BCD non valable)
	0110		(ajouter 6_{10})
somme =	1 0100	$=14_{10}$	(résultat BCD correct)
(3) 9_{10} =	1001		
8_{10} =	1000		
(somme =)	1 0001		(un report est généré et le résultat est incorrect)
	0110		(ajouter 6_{10})
somme =	1 0111	$=17_{10}$	(résultat BCD correct)

6.2 MULTIPLICATEURS ET DIVISEURS

En général les circuits pour ces opérations arithmétiques ne sont pas incorporés dans la CPU du minicalculateur. Cependant les fabricants fournissent souvent des sous-programmes de bibliothèque pour la multiplication, fondés sur la méthode manuelle.

Bien que les sous-programmes soient efficaces dans de nombreuses applications, certaines applications de commande en temps réel nécessitent des techniques plus rapides. L'une d'elles utilise un multiplicateur élémentaire 4×2 (construit sur une simple microplaquette) qui génère des produits partiels. Plusieurs de ces unités sont connectées de manière adéquate pour des longueurs de mot plus importantes. Le temps total de la multiplication est égal au délai de propagation maximum des produits partiels : dans un circuit, la multiplication de deux nombres de 16 bits est effectuée en environ 600 nanosecondes.

Une technique un peu plus lente utilise les ROM comme *multiplicateurs à tableau de référence.*

Exemple 6.3. Un multiplicateur ROM pour deux mots de 4 bits a $(2^4)^2 = 256$ combinaisons de sortie de 8 bits maximum. Donc le nombre requis de bits ROM est $256 \times 8 = 2048$.

Cependant, multiplier deux mots de 8 bits, demanderait déjà $(2^8)^2 \times 16 = 1$ million de bits, ce qui est impossible pour le moment. La situation est allégée par l'emploi de techniques plus sophistiquées et l'utilisation de plusieurs additionneurs, donc en réduisant considérablement le nombre de bits ROM nécessaires.

Pour la division, comme pour la multiplication, un programme bibliothèque est également fourni par le fabricant. Des diviseurs combinés et des diviseurs ROM sont également possibles pour des mots de petite longueur.

6.3 ACCUMULATEUR ET UNITÉ ARITHMÉTIQUE ET LOGIQUE

L'accumulateur (Registre A) occupe une position importante dans la CPU puisque toutes les données à traiter doivent passer par lui. Il est souvent réalisé par un registre de décalage avec un nombre de bits égal à la longueur des mots du minicalculateur.

On peut effectuer sur le contenu de l'accumulateur soit une permutation circulaire soit un décalage à droite ou à gauche et ces opérations peuvent comprendre le bit de report si cela a été spécifié. Les diverses possibilités sont illustrées dans la figure 6.5.

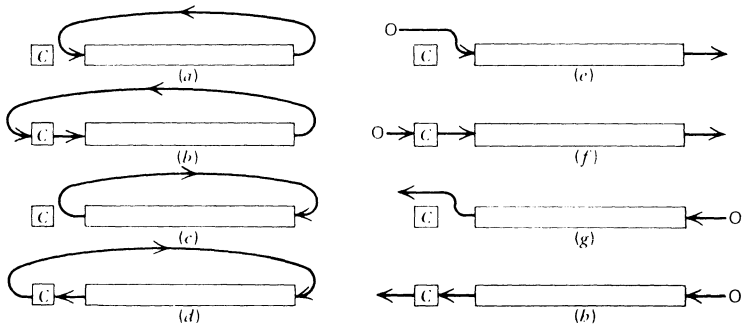


FIG. 6.5. — Opérations de permutation et de décalage dans l'accumulateur.
(*a*) Permutation à droite sans report, (*b*) permutation à droite avec report, (*c*) permutation à gauche sans report, (*d*) permutation à gauche avec report, (*e*) décalage à droite sans report, (*f*) décalage à droite avec report, (*g*) décalage à gauche sans report et (*h*) décalage à gauche avec report.

En plus des opérations de permutation et de décalage présentées dans la figure 6.5, un minicalculateur peut aussi effectuer des opérations *arithmétiques de permutation et de décalage*. Les opérations sont semblables à celles de la figure 6.5, à part qu'elles n'interviennent que sur les bits représentant la valeur absolue du nombre.

La structure et les fonctions de l'ALU varient selon les minicalculateurs. La figure 6.6 schématise une ALU de 4 bits. Elle est semblable à l'additionneur élémentaire de 4 bits avec report bouclé et incorpore un circuit qui lui permet d'effectuer de nombreuses opérations arithmétiques et logiques comme dans le tableau 6.1. Le niveau binaire *MODE* détermine si une opération logique ou une opération arithmétique doit être effectuée tandis que les entrées de *choix de fonction* S_2, S_1 et S_0 déterminent la fonction arithmétique ou la fonction logique.

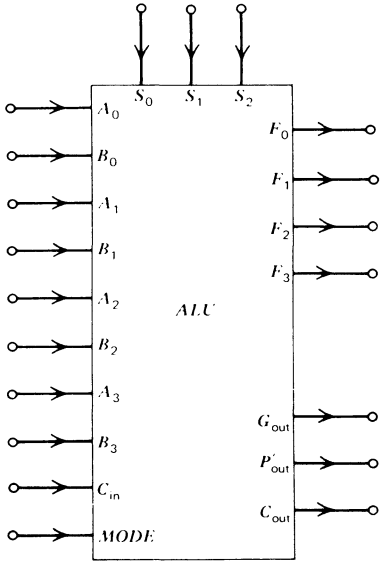


FIG. 6.6. — Schéma fonctionnel d'une *ALU* de 4 bits.

Plusieurs ALU de la figure 6.6 peuvent être mises en cascade pour des longueurs de mot supérieures à 4 bits. Dans de telles applications, la propagation du report est facilitée par les connexions C_{in}, C_{out}, G_{out} et P_{out}.

TABLEAU 6.1. — **Tableau des fonctions d'une ALU de 4 bits.**

Entrée de choix de fonction $S_2S_1S_0$	$MODE = 1$: opération logique F	$MODE = 0$: opération arithmétique F
0 0 0	$A + B$	A plus B
0 0 1	$A \cdot B$	Moins 1 (complément à 1)
0 1 0	$A \oplus B$	A moins 1
0 1 1	$A \odot B$	A plus $(A + B)$
1 0 0	0	A fois 2
1 0 1	1	A plus $(A \cdot B)$
1 1 0	B	A moins B moins 1
1 1 1	A	A

REMARQUE : les symboles « + », « . », « $\oplus$ », « . » font référence aux opérations OU, ET, OU EXCLUSIF et EGALITE, « plus », « moins » et « fois » font référence aux opérations arithmétiques.

PROBLÈMES

1. Utiliser la représentation « complément à 2 » et montrer des exemples d'addition avec des retenues arithmétiques.
2. Utiliser la représentation « complément à 2 » et montrer les exemples dans lesquelles l'indicateur de report est utilisé.
3. Utiliser la formule 6.1 et dresser le schéma logique d'un additionneur de 4 bits avec report automatique.
4. Utiliser l'additionneur de 4 bits de la figure 6.2 et tracer le schéma d'un additionneur hybride de 16 bits. Étudier le retard total de propagation.
5. Terminer l'addition BCD de la figure 6.4.
6. Établir la table de vérité et démontrer la nécessité d'une correction dans l'addition de deux nombres BCD ayant une somme $> 1001_2 = 9_{10}$.
7. Décrire les opérations arithmétiques de décalage à droite et à gauche nécessaires pour la normalisation de nombres en virgule flottante dans des opérations arithmétiques.

7

LA MÉMOIRE CENTRALE

Dans ce chapitre, nous décrivons les différents éléments, l'organisation et le fonctionnement de la *mémoire centrale* du minicalculateur. En principe la mémoire centrale peut inclure tout moyen de stockage pouvant être organisé pour remplir une fonction de mémorisation ; cependant nous ne nous intéressons ici qu'aux *mémoires à semiconducteurs* qui sont largement utilisées dans les minicalculateurs.

7.1 MÉMOIRES A SEMICONDUCTEURS

Les mémoires à semiconducteurs sont des zones cellulaires d'éléments semiconducteurs de stockage, ou *cellules*, qui contiennent des informations binaires. En général, une seule microplaquette contient 256 à 16 384 cellules identiques. Les mémoires à semiconducteurs peuvent être classées d'après leur utilisation, leur technique de fabrication et leur vitesse de fonctionnement.

Utilisation des mémoires

Les mémoires les plus fréquemment utilisées sont de deux sortes : les *mémoires à accès sélectif* (RAM) de type lecture-écriture et les *mémoires mortes* (ROM) employées pour stocker des informations fixes telles que des programmes, des sous-programmes et des tableaux de référence. Les RAM et les ROM sont toutes deux des zones de cellules-mémoire ; cependant le contenu d'une ROM ne peut être modifié au cours d'une opération puisque l'information déterminant la valeur binaire (0 ou 1) de chaque bit ne peut être introduite que pendant la fabrication de la mémoire.

Lors de la conception d'un programme, le travail de l'utilisateur serait facilité par la possibilité d'introduire des informations dans les ROM.

Cela a conduit à la création de *ROM programmables* (ou *pROM*). Il existe deux méthodes principales de programmation de zones. Dans l'une des méthodes, chaque cellule de la zone possède une liaison métallique qui peut être détruite au cours de la programmation par l'envoi d'une impulsion à forte intensité de durée précise. La liaison coupée dans une cellule définit une valeur binaire et la liaison intacte représente l'autre valeur binaire. Dans l'autre méthode une zone mémoire se compose de cellules qui sont programmées par la création sélective d'un court-circuit au moyen de coupures en cascade.

Dans une pROM la structure de la cellule est modifiée de façon irréversible pour l'une des deux valeurs 0 ou 1. Les *ROM reprogrammables* fournissent la possibilité supplémentaire de changer ces valeurs binaires. Dans une ROM reprogrammable, l'information est introduite grâce à un processus en cascade. L'information peut être effacée par projection d'une lumière ultra-violette à travers un quartz, ce qui permet une nouvelle programmation. Cette méthode peut être répétée de nombreuses fois sans altérer le bon fonctionnement de la mémoire.

Techniques de fabrication

Les cellules-mémoire peuvent être fabriquées à partir de transistors métal-oxyde-silicium à effet de champ (MOSFET) ou de transistors bi-polaires. Grâce à d'importants efforts, le nombre de transistors nécessaires par cellule a été réduit de 8 à 1. Nous décrivons ici en détail une configuration à trois transistors. La cellule est de *type dynamique :* elle n'a pas de rétro-action positive entre les éléments actifs pour conserver l'information dans un état stable mais au contraire l'information est stockée sur la capacité isolée d'une électrode du FET.

Le fonctionnement d'une cellule mémoire MOS à trois transistors et n canaux est étudié ci-dessous en se référant à la figure 7.1. La cellule se compose des transistors Q_1, Q_2 et Q_3 et de la capacité C_G. Le transistor Q_4 est commun à toutes les cellules-mémoire d'une *colonne* de la zone et sert à *précharger* la capacité C_D. Pour lire le contenu de la cellule-mémoire, C_D est préchargée initialement à une tension proche de V_{DD} et la ligne *READ SELECT* commune à une ligne d'une zone cellulaire est mise à la valeur 1, c'est-à-dire à $V_{DD} = +12$ volts. Si la tension à travers C_G est initialement au-dessus du seuil de tension de Q_2, C_D est déchargée à travers Q_2 et Q_3 ; à l'inverse C_D reste chargée à une tension proche de V_{DD} si la tension à travers C_G est initialement en dessous du seuil de tension de Q_2. A partir de là, le complément de l'information binaire de C_G est transféré à la capacité C_D par la ligne *READ DATA*, sans modifier l'état

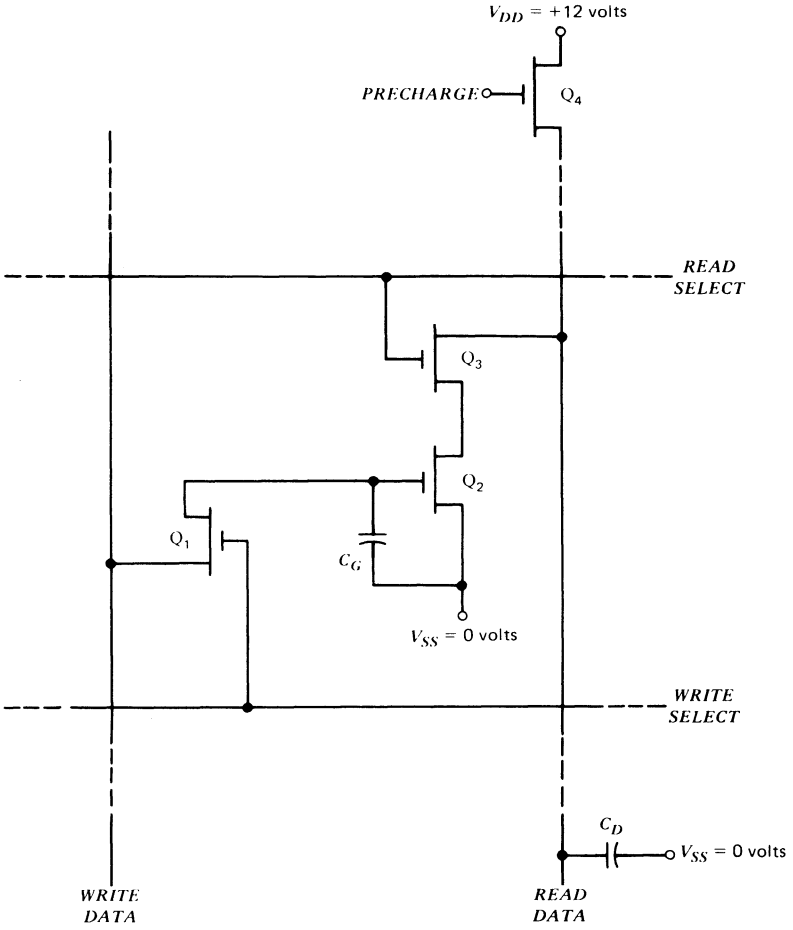


FIG. 7.1. — Cellule-mémoire dynamique MOS à trois transistors et *n* canaux.

de C_G. Pour écrire dans la cellule, il faut mettre à 1 la ligne *WRITE SELECT* qui est commune à une ligne de la zone cellulaire. Cela commute Q_1 et transfert à la capacité C_G l'état de la ligne *WRITE DATA* qui est commune à une colonne de la zone cellulaire.

Bien que la lecture ne soit pas destructive, la charge du condensateur C_G se modifie à cause de fuites, ce qui exige que l'information soit périodiquement réécrite dans chaque cellule de la mémoire, même si le contenu de

certaines cellules n'est pas modifié. Ceci est réalisé par une *restauration* périodique qui consiste à lire le contenu d'une cellule et à réécrire la donnée dans la même cellule. Les circuits réalisant cette tâche sont étudiés dans le paragraphe 7.5.

Les mémoires bipolaires sont habituellement de type statique, c'est-à-dire que deux inverseurs sont connectés dans une configuration à rétro-action positive de manière à atteindre deux états stables. Une cellule-mémoire bipolaire utilisant des transistors à émetteur triple pour la sélection des mots et l'aiguillage des données est représentée à la figure 7.2.

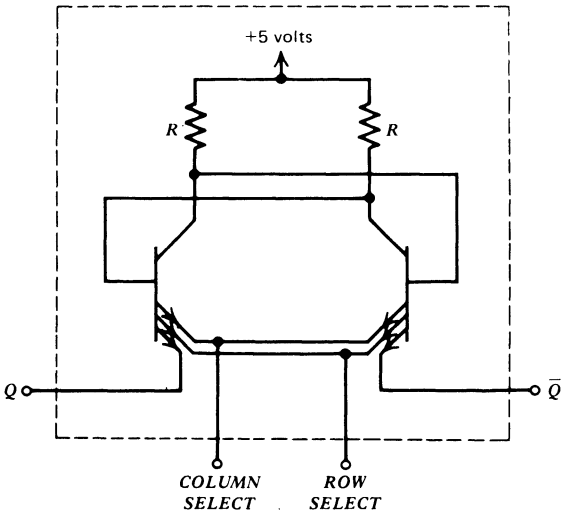


FIG. 7.2. — Cellule-mémoire bipolaire à émetteur triple avec adressage par coïncidence.

Pour les opérations d'écriture et de lecture, les deux lignes *COLUMN SELECT* et *ROW SELECT* doivent avoir simultanément la valeur binaire 1. Pendant une opération d'écriture, l'information est introduite dans la cellule en mettant la ligne Q ou $\bar{Q}$ à la valeur binaire 0. Pour la lecture, on mesure le courant circulant dans ces lignes.

L'utilisation de la cellule-mémoire à émetteur triple de la figure 7.2 est illustrée à la figure 7.3 pour une zone composée de 16 bits. Elle comprend des entrées adressées de A_0 à A_3 ainsi qu'une entrée *CHIP SELECT* (CS) qui commande le fonctionnement de la zone et facilite

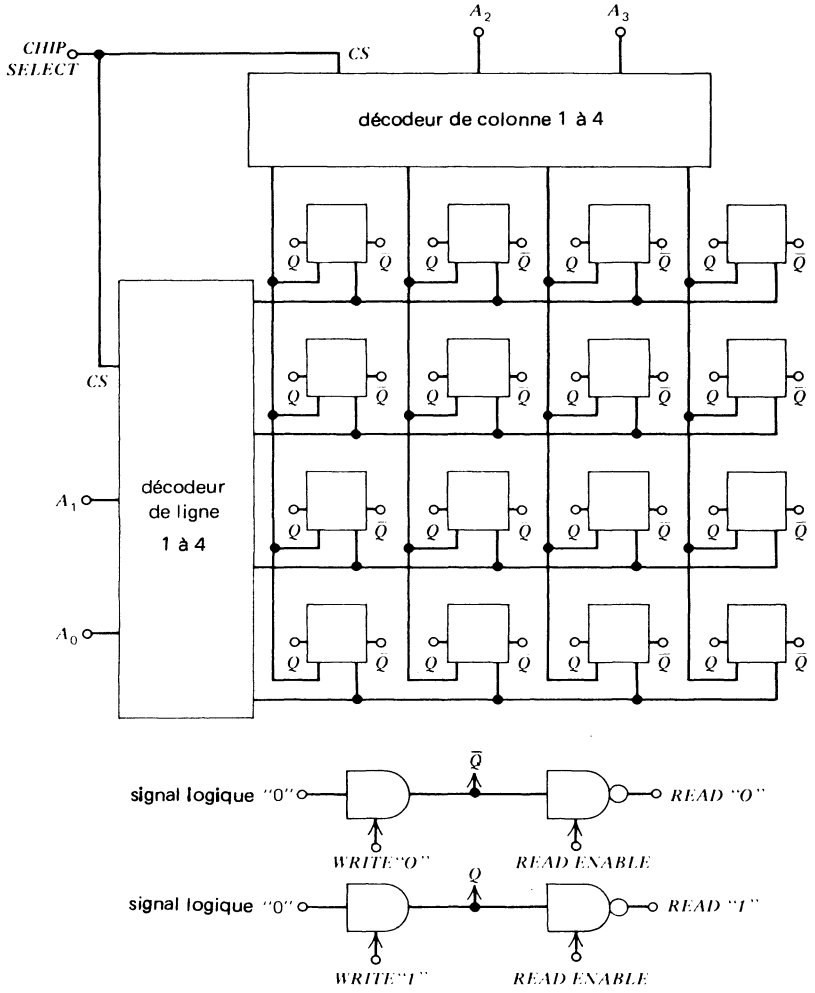


FIG. 7.3. — Schéma fonctionnel d'une zone RAM de 16 bits avec adressage par coïncidence. Chaque bit utilise une cellule de la figure 7.2.

l'utilisation de nombreuses zones dans une mémoire. Les sorties *READ* « 0 » et *READ* « 1 » du type à 3 états sont commandées par un signal commun *READ ENABLE* permettant des connexions en parallèle avec des sorties identiques d'autres zones.

Vitesse de fonctionnement

La vitesse de fonctionnement d'une mémoire peut être exprimée au moyen de plusieurs paramètres. L'un d'eux est le *temps d'accès* qui est le temps écoulé entre l'adressage d'une mémoire et la sortie de cette mémoire. Ce temps peut varier de plusieurs nanosecondes à deux microsecondes et dépend de la technologie et du nombre de mots dans la mémoire. Un autre paramètre est le *temps de cycle* qui est le temps minimum nécessaire entre le début des opérations successives de *lecture mémoire*, *écriture mémoire* ou *lecture-modification-écriture mémoire*.

En général les mémoires MOS sont plus lentes que les mémoires bipolaires. Les temps d'accès les plus courts possibles pour des mémoires bipolaires sont d'environ 5 nanosecondes.

7.2 STRUCTURE DE LA MÉMOIRE

La structure d'une RAM bipolaire utilisant une cellule mémoire à émetteur triple était montrée à la figure 7.3. Quand la vitesse de fonctionnement plus élevée des circuits bipolaires n'est pas nécessaire, des circuits MOS

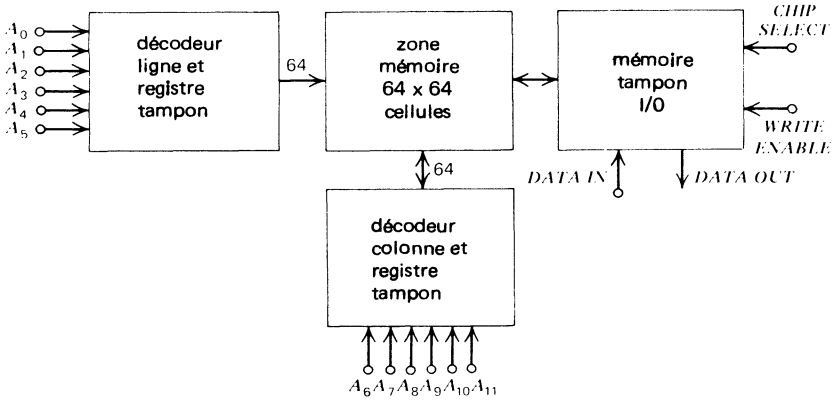


FIG. 7.4. — Schéma fonctionnel simplifié d'une RAM MOS de 4 096 mots × 1 bit.

à plus haute densité peuvent être utilisés. La figure 7.4 montre le schéma fonctionnel d'une RAM MOS de 4 096 bits. Chaque bit de la RAM est adressable individuellement (structure par bit). L'adresse à 12 bits $A_{11} - A_0$ est décodée par les *décodeurs ligne et colonne* à deux fois 64 lignes et un bit

est sélectionné par l'intersection entre une ligne de sortie du décodeur ligne et une ligne de sortie du décodeur colonne.

Alors que la *structure par bit* domine dans les grandes RAM, la *structure par mot* est utilisée de préférence dans des RAM plus petites et dans les ROM. Dans une mémoire à structure par mot un *décodeur d'adresse* sélectionne un mot d'une longueur de plusieurs bits au lieu d'un bit.

7.3 REGISTRE DE DÉCALAGE

Un minicalculateur peut aussi utiliser des *registres de décalage* en plus des différents types de mémoires étudiés jusqu'ici. Un registre de décalage se compose d'une zone linéaire d'éléments de stockage : l'information dans la zone est décalée grâce à l'utilisation d'une *horloge de décalage* (qui est habituellement une horloge à deux ou quatre phases) et le sens du décalage est ou fixe ou déterminé par la commande *sens de décalage*. L'information peut être entrée et retirée aux extrémités de la zone linéaire ; dans certains registres de décalage, il est aussi possible d'entrer ou de sortir des informations en d'autres points de la zone. Grâce à leur structure itérative combinée à une grande densité de composants, les registres de décalage fournissent un moyen bon marché de stockage de l'information et des capacités de 2 × 1 024 bits sur une seule microplaquette semiconductrice sont tout à fait réalisables. Un autre type de registre de décalage est le *dispositif à charge couplée* (CCD) qui autorise jusqu'à 16 384 bits sur une microplaquette semiconductrice.

7.4 REGISTRES AUXILIAIRES

En plus des éléments de mémoire décrits ci-dessus, le flot ordonné d'informations allant à la mémoire et provenant de la mémoire exige aussi plusieurs registres auxiliaires ainsi que des canaux de données et d'adressage. La figure 7.5 montre un schéma fonctionnel qui comprend plusieurs registres auxiliaires. Le *registre de données mémoire* (MDR) permet de transférer les données à partir de la mémoire centrale vers la mémoire centrale. Le *registre d'adresse mémoire* (MAR) a la possibilité d'adresser de façon sélective chaque mot dans la mémoire centrale. Ses informations viennent du *registre instruction* ou du *compteur ordinal* qui contient l'adresse de la prochaine instruction à exécuter et est en général incrémenté après que cette adresse ait été transmise au MAR.

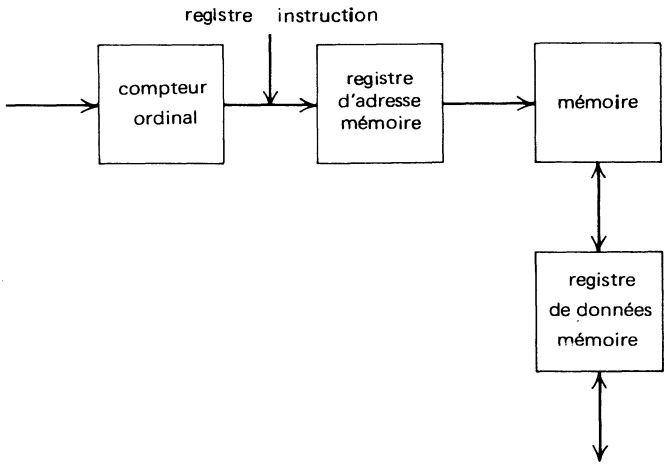


FIG. 7.5. — Schéma fonctionnel simplifié d'une mémoire montrant le compteur ordinal, le registre d'adresse mémoire et le registre de données mémoire.

7.5 CIRCUIT DE RESTAURATION POUR LES RAM MOS DYNAMIQUES

La figure 7.1 nous a montré que l'information contenue dans une cellule-mémoire MOS dynamique est conservée grâce à une petite capacité (qui est en général nettement inférieure à un picofarad). Un stockage aussi volatil peut amener la destruction de l'information, en particulier aux températures élevées, puisque les courants de fuite doublent chaque fois que la température augmente de 10° C (environ). Donc certains moyens durent être trouvés pour ramener le condensateur à son état de charge initial. Le fait de ne pas dépasser un certain intervalle de temps entre deux restaurations successives permet de conserver les données. En général cet intervalle est de quelques millisecondes.

La figure 7.6 donne une illustration d'un circuit de restauration pour une mémoire organisée à partir d'une microplaquette RAM MOS de 4 096 mots × 1 bit. Les 4 096 bits sont ordonnés en $2^6 = 64$ lignes et en 64 colonnes. L'opération de restauration est commandée en séquence par un *compteur de restauration* à 6 bits, puisqu'en général les signaux de restauration sont appliqués uniquement aux lignes d'une RAM. Une opération de restauration est initialisée par une *horloge de restauration* une fois toutes les 30 microsecondes ; de cette façon chacune des 64 lignes

est restaurée à intervalles réguliers d'environ 2 millisecondes. Un *cycle de restauration* arrête les opérations de la CPU *(gel de la CPU)*, neutralise (grâce à une haute impédance) les sorties du registre d'adresse mémoire, incrémente le compteur de restauration et active les sorties à 3 états du compteur de restauration.

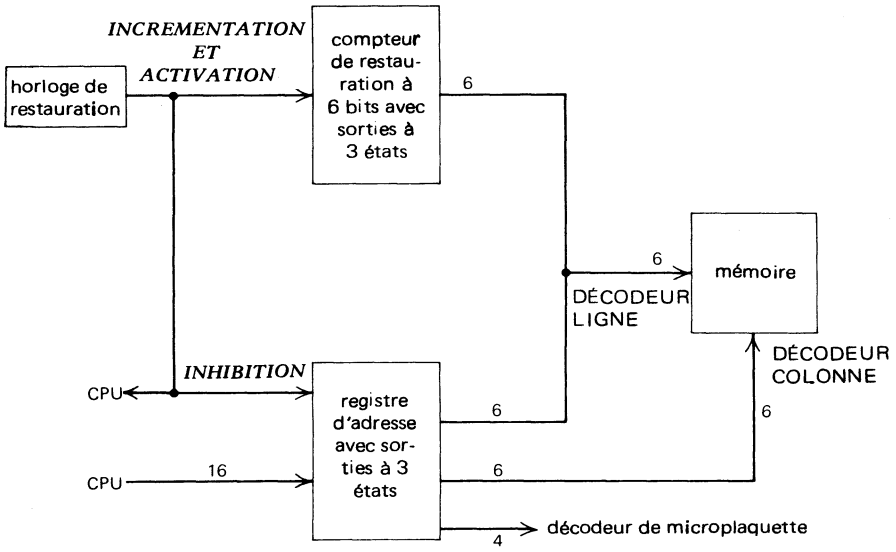


FIG. 7.6. — Circuit de restauration pour une RAM MOS dynamique utilisant des zones mémoire de 4 096 mots × 1 bit. Remarquer que l'adresse et les lignes de données de la CPU doivent être inhibées pendant la restauration.

Dans certaines structures de cellules-mémoire il est aussi possible de restaurer toute la mémoire par l'envoi périodique d'*une seule* impulsion de restauration.

7.6 MODE D'ADRESSAGE

La longueur d'une instruction dans un minicalculateur permet dans la plupart des cas d'inclure l'adresse complète de l'emplacement mémoire concerné.

Exemple 7.1. Un minicalculateur avec une longueur de mot de 8 bits possède $2^{16} = 65636$ emplacements mémoire. Une instruction « charger l'accumulateur » se compose d'un opéra-

teur de 8 bits qui désigne l'instruction ; cet opérateur est suivi par un opérande de 16 bits qui désigne l'adresse mémoire dont le contenu doit être chargé dans l'accumulateur.

Cependant certains minicalculateurs ont un nombre limité de bits dans les formats d'instructions faisant référence à la mémoire ce qui, en conséquence, ne permet pas l'adressage de toute la mémoire. Ceci a conduit à la création de différents *modes d'adressage*. Dans la plupart d'entre eux une *adresse effective* est calculée en combinant l'adresse contenue dans le mot de l'instruction avec un autre mot qui a été entré précédemment dans un registre ou dans un emplacement mémoire déterminé ; donc deux mots sont nécessaires pour préciser complètement un emplacement

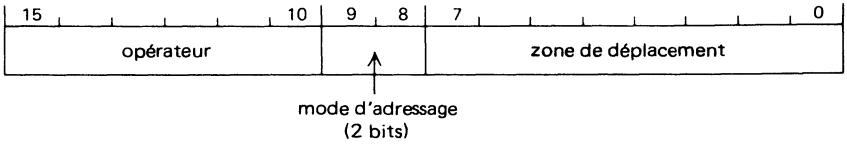


FIG. 7.7. — Mot instruction de 16 bits d'un minicalculateur possédant quatre modes d'adressage différents.

mémoire. La figure 7.7 illustre quatre de ces modes d'adressage avec un mot instruction de 16 bits. Les 6 bits les plus significatifs sont réservés à l'opérateur alors que les 8 bits les moins significatifs représentent la *zone d'adressage* appelée aussi *zone de déplacement* qui permet l'adressage de 256 mots mémoires. Deux bits supplémentaires, les bits 8 et 9, appelés *bits de mode d'adressage* fournissent les quatre modes d'adressage.

Adressage à la page de base

L'adressage à la page de base est le mode d'adressage le plus simple. Il est aussi connu sous le nom d'*adressage à la page 0*.

> ***Exemple 7.2.*** Supposons que les bits 8 et 9 de la figure 7.7 soient 00, désignant l'adressage à la page de base. La zone de mots mémoire qui peut être adressée grâce à ce procédé va de 0 à 255.

Adressage avec un registre de page

Le procédé d'adressage à la page de base décrit ci-dessus n'est suffisant que pour les plus petits systèmes de mémoire. Pour augmenter la possibilité

d'adressage dans le cas d'une mémoire plus grande nous mettons en œuvre un *registre de page*. Comme précédemment chaque page se compose seulement de 256 mots ; cependant le registre de page permet au programme de reconnaître quel jeu de 256 mots, c'est-à-dire quelle *page*, doit être utilisé dans chaque cas de référence à la mémoire. En effet, le registre de page ajoute à l'adresse de l'instruction des bits d'ordre plus élevé, ce qui permet l'adressage d'emplacements mémoire supplémentaires comme le montre la figure 7.8. Bien qu'une instruction supplémentaire soit maintenant nécessaire pour donner une valeur au registre de page, le procédé peut être intéressant si le programmeur prend soin d'effectuer une grande partie des références mémoire à l'intérieur de la page choisie.

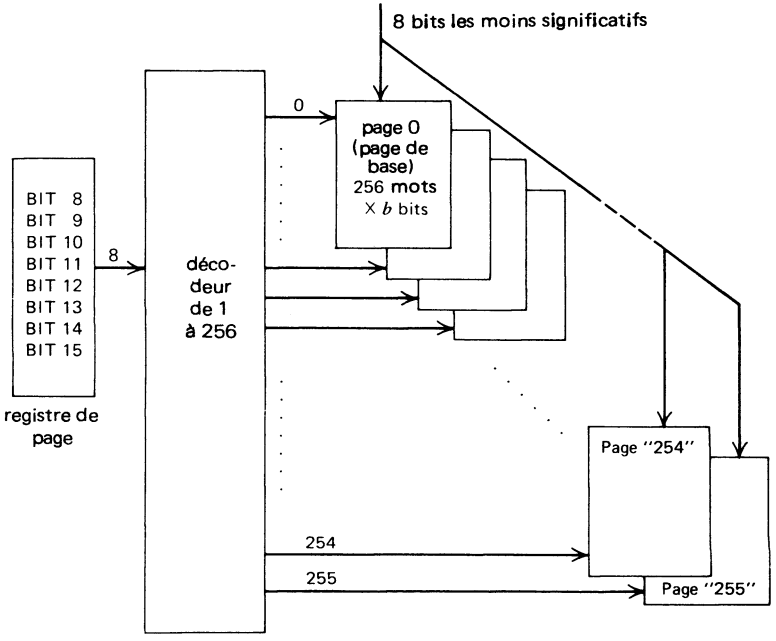


FIG. 7.8. — Adressage avec un registre de page. Chacune des 256 pages est sélectionnée au moyen du registre auxiliaire de page et chaque page a une capacité de 256 mots.

Exemple 7.3. Supposons que les bits 8 et 9 de la figure 7.7 soient 01, désignant l'adressage avec un registre de page. Le registre de page de la figure 7.8 est chargé à la valeur 10000001 (= 129) qui sont les bits les plus significatifs d'une

adresse effective de 16 bits ; donc la page 129_{10} est sélectionnée par le décodeur. Les adresses qui peuvent être atteintes en incluant les 8 bits les moins significatifs sont alors dans l'intervalle de 129×2^8 à $129 \times 2^8 + 255$, c'est-à-dire 33024_{10} à $33\ 279_{10}$.

Adressage relatif au compteur ordinal

Dans ce mode d'adressage les 8 bits les moins significatifs du mot instruction sont traités comme un nombre binaire signé dont le bit 7 représente le signe. L'adresse effective est obtenue en ajoutant le contenu du compteur ordinal *(PC)* au nombre signé ainsi formé. Cela permet l'adressage relatif dans l'intervalle de $(PC) - 128_{10}$ à $(PC) + 127_{10}$. Si le registre programme a déjà été incrémenté au moment où l'adresse est calculée et désigne alors l'instruction suivante, l'intervalle d'adressage va de $(PC) - 127_{10}$ à $(PC) + 128_{10}$.

L'adressage à la page de base et l'adressage relatif au compteur ordinal sont économiques, puisqu'ils ne demandent pas d'instruction complémentaire pour désigner la page. Bien que leurs intervalles d'adressage soient faibles, une programmation attentive peut permettre à l'intérieur de ces intervalles l'utilisation de nombreuses instructions avec référence mémoire. De plus l'adressage relatif au compteur ordinal simplifie le déplacement des programmes, ce qui facilite la combinaison de différents segments de programmes et de sous-programmes internes (bibliothèque).

Adressage relatif à un registre d'index

Dans ce mode d'adressage l'adresse effective est calculée en faisant la somme du contenu d'un registre d'index et du contenu de la zone de déplacement. L'avantage de l'utilisation des registres d'index repose sur leur possibilité d'être modifiés (c'est-à-dire effacés, chargés, incrémentés ou décrémentés) en une fraction d'un cycle mémoire. Donc la modification d'index est une façon commode d'adresser différents éléments de zones mémoire et d'autres structures de données.

7.7 ADRESSAGE INDIRECT

Dans l'adressage indirect, l'adresse fournie par l'un des modes d'adressage ci-dessus ne contient pas elle-même l'opérande cherché mais seulement son adresse. Donc l'instruction faisant référence à la mémoire contient

l'adresse d'un emplacement mémoire dont le contenu sert seulement d'*indicateur* de l'adresse recherchée. Habituellement l'instruction contient un bit qui indique si le mode d'adressage est direct ou indirect.

Exemple 7.4. La figure 7.9 illustre l'adressage indirect. Contrairement à la figure 7.7, le mot instruction se compose d'un opérateur de 7 bits, d'un bit D/I indiquant si le mode d'adressage est direct ou indirect et d'une zone de déplacement de 8 bits. La figure 7.9 montre aussi des parties de la mémoire centrale. Nous supposons que le bit D/I du registre instruction correspond au mode d'adressage indirect. Alors quand le MAR contient l'adresse 1012, le contenu de l'adresse 1012 est interprété comme l'*adresse* 10 128 c'est-à-dire l'emplacement où se trouve l'opérande.

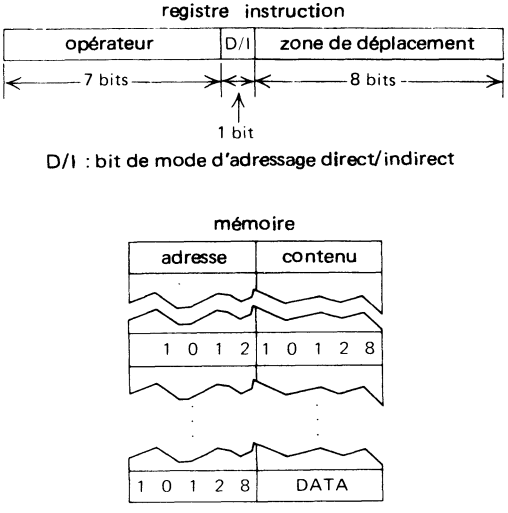


FIG. 7.9. — Illustration de l'adressage indirect.

PROBLÈMES

1. Combien de circuits RAM de 256 mots × 1 bit sont nécessaires pour un système mémoire de 1 024 mots × 8 bits ? Établir un schéma fonctionnel du système comprenant les décodeurs d'adresse pour la sélection des microplaquettes.
2. Soit un registre de décalage bouclé de 1 024 bits en série tel que celui décrit dans le paragraphe 7.3 ; concevoir une mémoire de 1 024 mots × 8 bits à registre

de décalage bouclé. Établir les circuits digitaux complémentaires nécessaires pour déterminer la position du bit où des informations peuvent être sorties ou de nouvelles données entrées.

3. Comparer les avantages des RAM et des registres de décalage et décrire quelques applications des deux types de mémoires.
4. Comparer les avantages de l'adressage avec registre de page et de l'adressage relatif au compteur ordinal. Donner un exemple dans chaque cas.
5. Dans l'adressage avec registre de page de la figure 7.8 le registre de page est mis à la valeur 10100100 et les bits les moins significatifs sont 01011011. Indiquer quelle sera la page choisie et quelle sera l'adresse de l'emplacement mémoire.

8

L'UNITÉ DE COMMANDE

L'unité de commande est le « centre nerveux » du minicalculateur. Elle permet la mise en séquence et la synchronisation lors de la réalisation des instructions et commande la circulation des données entre les différentes parties du minicalculateur.

8.1 MISE EN SÉQUENCE

Une instruction dans un minicalculateur est réalisée en un ou plusieurs *cycles processeur* aussi appelés *cycles machine*, *microcycles* ou *cycles d'instruction de base*. Un cycle processeur se compose de deux *phases* principales : la *phase d'adressage* (ou *de mise en place*) et la *phase d'exécution*. Chaque phase d'adressage est utilisée pour adresser un emplacement mémoire par le moyen du registre d'adresse mémoire (MAR) et pour déplacer des données entre le registre de données mémoire (MDR) et l'emplacement adressé. Dans le cas de quelques instructions simples, une instruction complète est mise en place au cours d'une seule phase d'adressage et est exécutée en une seule phase d'exécution.

> ***Exemple 8.1.*** Dans l'exemple 2.3, l'addition a été réalisée en ajoutant le contenu du registre B au contenu de l'accumulateur et en plaçant le résultat dans l'accumulateur. L'instruction correspondant à cette addition occupe seulement un emplacement mémoire et donc elle peut être mise en place en une seule phase d'adressage. En outre, puisque seule la CPU est utilisée et qu'aucune utilisation d'une mémoire supplémentaire n'est nécessaire, l'instruction peut être exécutée en une seule phase d'exécution. En résumé, l'instruction est achevée en une seule phase d'adressage et une seule phase d'exécution, donc en un seul cycle processeur.

Certaines instructions, en particulier pour les minicalculateurs qui ont des longueurs de mot inférieures à 16 bits, peuvent occuper plus d'un emplacement mémoire.

Exemple 8.2. Un microprocesseur a une longueur de mot de 8 bits et une mémoire centrale comprenant $2^{16} = 65\,536$ mots. La mémorisation d'une instruction de saut nécessite trois emplacements mémoire ; le premier contient le code de l'instruction de saut en langage machine ; les deuxième et troisième emplacements contiennent l'adresse de 16 bits indiquant l'emplacement mémoire où le programme doit se brancher.

Une instruction qui occupe plus d'un emplacement mémoire a besoin de plus d'une phase d'adressage, donc de plus d'un cycle processeur. Cependant, la phase d'exécution d'une instruction ne peut en général commencer que lorsque l'instruction entière a été transférée dans le *registre instruction.* Pour cette raison, les phases d'exécution sont inemployées dans les (ou sont omises des) cycles processeurs jusqu'à ce que toutes les phases d'adressage nécessaires aient été achevées. Dans beaucoup de cas, comme dans l'exemple 8.1, l'instruction de mise en place ne réclame pas d'autres références mémoire ; une telle instruction est exécutée pendant la phase d'exécution qui suit la phase d'adressage qui achève le transfert de l'instruction dans le registre instruction. Quand l'exécution de l'instruction de mise en place réclame d'autres références mémoire, cela conduit à un cycle processeur supplémentaire.

Exemple 8.3. Cet exemple illustre la mise en séquence de cycles processeur et l'utilisation de phases d'adressage et d'exécution pour une instruction d'addition. Le minicalculateur de l'exemple a une longueur de mot de 8 bits et une mémoire centrale comprenant $2^{16} = 65\,536$ mots.

Le tableau 8.1 montre certaines parties de la mémoire centrale. L'instruction « Ajouter le contenu de l'emplacement mémoire 1205 au contenu de l'accumulateur et placer le résultat dans l'accumulateur » est mémorisée dans les emplacements 101, 102 et 103. L'emplacement (l'adresse) 101 contient le code d'instruction de l'addition qui est 11000001 ($= C1_{16}$) dans le langage machine du minicalculateur. L'emplacement 102 contient les 8 bits les plus significatifs de l'équivalent binaire de 1205, l'emplacement 103, les 8 bits les moins significatifs. L'emplacement 1205 contient le terme à ajouter $00001111 = 15_{10} = F_{16}$.

TABLEAU 8.1. — **Déroulement de l'instruction d'addition de l'exemple 8.3.**

Adresse		Contenu	
Décimal	Binaire	Binaire	Hexadécimal
0	0000000000000000	????????	??
1	0000000000000001	????????	??
.	.	.	.
.	.	.	.
.	.	.	.
101	0000000001100101	11000001	C1
102	0000000001100110	00000100	04
103	0000000001100111	10110101	B5
.	.	.	.
.	.	.	.
.	.	.	.
1205	0000010010110101	00001111	0F
.	.	.	.
.	.	.	.
.	.	.	.
65534	1111111111111110	????????	??
65535	1111111111111111	????????	??

La figure 8.1 montre la mise en séquence des cycles processeur et des phases d'adressage et d'exécution. Le tableau 8.2 montre le contenu de l'accumulateur, le registre d'adresse mémoire, le registre de données mémoire et le registre instruction pendant la réalisation de l'instruction.

Pendant la phase d'adressage du premier cycle processeur (t_0 à t_1), l'emplacement mémoire 101 est adressé au moyen du MAR à 16 bits et le code instruction $11000001 = C1_{16}$ est déplacé de cet emplacement aux 8 premiers bits du registre instruction au moyen du MDR. Ces 8 bits montrent que l'instruction n'est pas complète, donc la phase d'exécution du premier cycle processeur (t_1 à t_2) est inemployée.

Pendant la phase d'adressage du second cycle processeur (t_2 à t_3), les 8 bits les plus significatifs de l'adresse 1205 sont

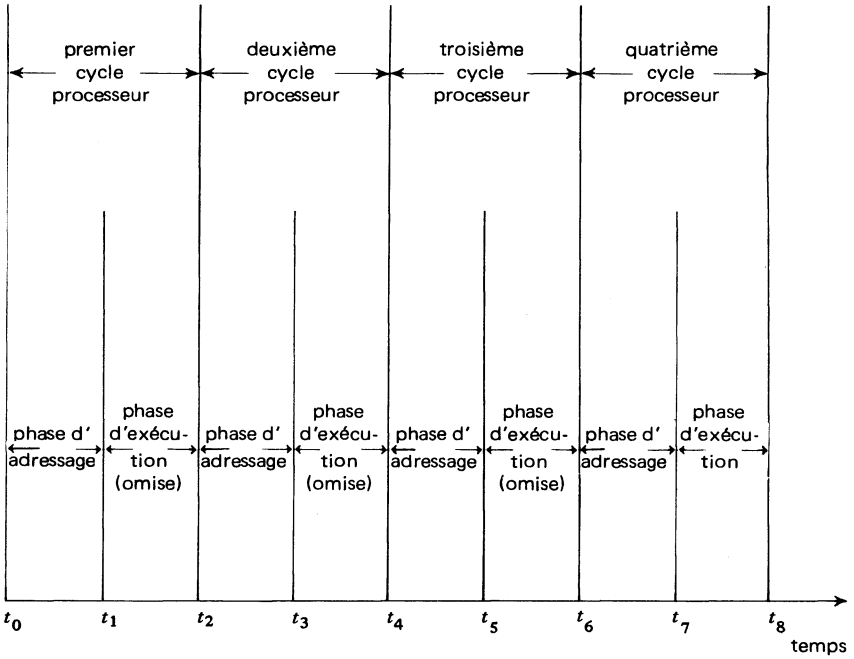


FIG. 8.1. — Déroulement de l'instruction d'addition de l'exemple 8.3 ; mise en séquence des cycles processeur et des phases d'adressage et d'exécution.

déplacés de l'emplacement mémoire 102 aux 8 bits suivants du registre instruction. L'instruction n'est pas encore complète, donc la phase d'exécution du second cycle processeur est aussi inemployée.

Pendant la phase d'adressage du troisième cycle processeur (t_4 à t_5), les 8 bits les moins significatifs de l'adresse 1205 sont déplacés de l'emplacement 103 aux 8 derniers bits du registre instruction. Alors l'instruction dans le registre instruction est complète et elle indique que le contenu de l'emplacement mémoire 1205 doit être mis en place. La phase d'exécution du troisième cycle processeur (t_5 à t_6) est donc inemployée.

Pendant la phase d'adressage du quatrième cycle processeur (t_6 à t_7), l'emplacement mémoire 1205 est adressé au moyen du MAR à 16 bits et son contenu est mis en place dans le MDR. L'addition est réalisée pendant la phase d'exécu-

TABLEAU 8.2. — **Déroulement de l'instruction d'addition de l'exemple 8.3;**
contenu de l'accumulateur, du MAR, du MDR et du registre instruction des instants t_0 à t_8.

Cycle processeur	Phase	Temps	Accumulateur		MAR		MDR		Registre instruction	
			Binaire	Décimal	Binaire	Décimal	Binaire	Hexadécimal	Binaire	Hexa-décimal
		t_0	00011000	24	0000000001100101	101	????????	??	??????????????????????????	??????
1	adressage									
	exécution	t_1	00011000	24	0000000001100101	101	11000001	C1	11000001????????????????	C1????
		t_2	00011000	24	0000000001100110	102	11000001	C1	11000001????????????????	C1????
2	adressage									
	exécution	t_3	00011000	24	0000000001100110	102	00000100	04	11000001000001000????????	C104??
		t_4	00011000	24	0000000001100111	103	00000100	04	11000001000001000????????	C104??
3	adressage									
	exécution	t_5	00011000	24	0000000001100111	103	10110101	B5	11000001000001001011010 1	C104B5
		t_6	00011000	34	0000010010110101	1205	10110101	B5	11000001000001001011010 1	C104B5
4	adressage									
	exécution	t_7	00011000	24	0000010010110101	1205	00001111	0F	11000001000001001011010 1	C104B5
		t_8	00100111	39	????????????????	????	00001111	0F	11000001000001001011010 1	C104B5

tion (t_7 à t_8), en ajoutant le contenu du MDR, qui est 15, au contenu de l'accumulateur qui est 24, et en plaçant le résultat (39) dans l'accumulateur.

Le fait de se référer à la mémoire peut entraîner le transfert d'une instruction, d'une partie d'instruction ou d'une donnée de la mémoire centrale au MDR, comme dans le cas de l'exemple 8.3 ; à l'inverse, il peut aussi entraîner le transfert de données du MDR à la mémoire centrale.

Exemple 8.4. Un minicalculateur a une longueur de mot de 8 bits et une mémoire centrale comprenant $2^{16} = 65\,536$ mots. L'instruction « Placer le contenu de l'accumulateur dans l'emplacement mémoire 1305 » est mémorisée aux adresses 201, 202 et 203. L'emplacement 201 contient le code instruction qui est 00001101 dans le langage machine du minicalculateur. L'emplacement 202 contient les 8 bits les plus significatifs de l'équivalent binaire de 1305, l'emplacement 203 les 8 bits les moins significatifs. Le contenu de l'accumulateur est $11000011 = 195_{10}$.

Le contenu de l'emplacement mémoire 1305 n'est pas connu initialement ; l'instruction transfert le nombre 195_{10} de l'accumulateur à l'emplacement mémoire 1305 au moyen du MDR, en détruisant le contenu précédent de l'emplacement 1305. Le contenu de l'accumulateur reste identique, c'est-à-dire 195_{10}.

Le transfert de données du MDR à la mémoire centrale est réalisé pendant une phase d'adressage, même quand, comme dans l'exemple 8.4, le transfert fait partie de l'exécution de l'instruction.

Exemple 8.5. La figure 8.2 montre la mise en séquence de l'instruction de l'exemple 8.4. L'instruction est mise en place durant les trois premières phases d'adressage, l'exécution de l'instruction, c'est-à-dire le stockage du nombre 195_{10} dans l'emplacement mémoire 1305, est réalisée pendant la quatrième phase d'adressage.

La phase d'adressage est souvent prolongée par l'addition d'une *phase d'attente* auxiliaire (aussi appelée *état d'attente*) quand le *temps d'accès* à l'emplacement mémoire adressé dépasse la durée de la phase d'adressage. La durée de la phase d'attente est choisie suffisamment longue pour assurer l'achèvement du transfert de données entre le MDR et l'emplacement mémoire adressé.

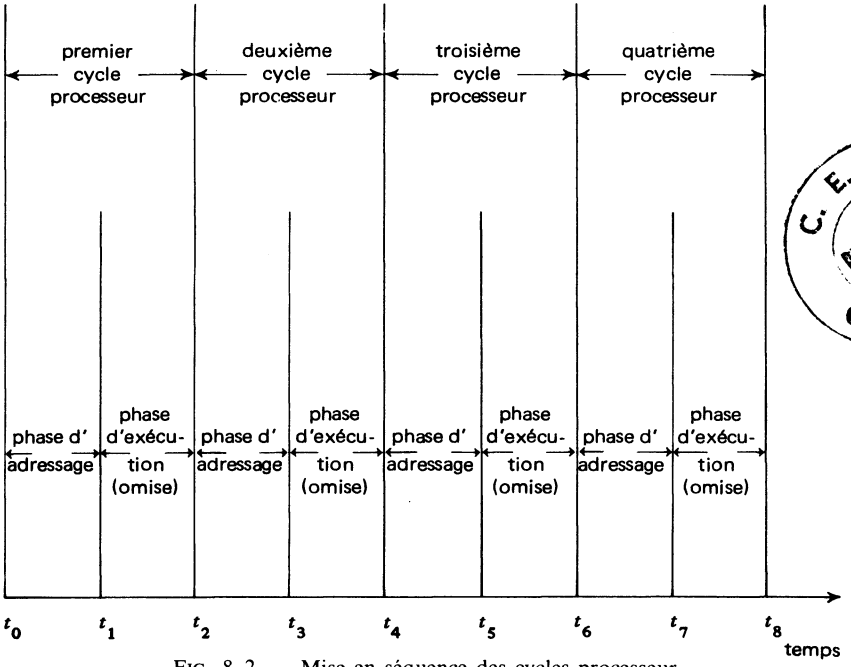


FIG. 8.2. — Mise en séquence des cycles processeur et des phases d'adressage et d'exécution de l'exemple 8.5.

Une autre phase auxiliaire est la *phase d'arrêt* (aussi appelée *état d'arrêt*) qui arrête les opérations de la CPU (cependant les mémoires dynamiques sont toujours restaurées si nécessaire). La phase d'arrêt qui est commandée par une instruction d'arrêt ou par un signal sur une ligne spéciale d'arrêt prend fin soit par une interruption soit par une intervention manuelle au niveau du tableau de commande.

8.2 SYNCHRONISATION

Un cycle processeur est divisé en *états temporels* de durées identiques. Cette durée est déterminée par l'horloge du système, donc est fixée pour un minicalculateur donné ; elle varie, en général, entre 0,1 et 3 microsecondes. Le nombre d'états temporels peut varier pour une phase d'exécution, une phase d'attente ou une phase d'arrêt. Cependant, le nombre d'états temporels pour une phase d'adressage est en général constant pour un minicalculateur donné.

Exemple 8.6. Un minicalculateur a une longueur de mot de 8 bits et une mémoire se composant de $2^{16} = 65\,536$ mots. L'adressage des emplacements mémoire utilise 16 bits qui sont envoyés de la CPU à la mémoire centrale au moyen d'un canal de données de 8 bits pendant deux états temporels consécutifs. Donc, la phase d'adressage se compose de deux états temporels. Cependant elle peut être suivie d'une phase d'attente composée d'un nombre quelconque d'états temporels.

8.3 CHEMINS DE DONNÉES ET STRUCTURE DES CANAUX

Les flots de données entre les différentes parties d'un minicalculateur se réalisent grâce à la connexion de chemins de données. Les flots de données, qui comprennent aussi bien les flots d'instructions, peuvent être unidirectionnels, comme dans la transmission d'une adresse du MAR à la mémoire centrale, ou bidirectionnels, comme entre le MDR et la mémoire centrale. Quand le chemin de données est unidirectionnel il peut être activé grâce à une porte AND pour chaque bit du chemin. Quand le chemin de données est bidirectionnel, un signal de commande de sens doit aussi être fourni à un circuit semblable à celui de la figure 4.1.

La *structure des canaux* d'un minicalculateur se compose de tous les chemins de données. L'exemple suivant illustre le fonctionnement d'un canal.

Exemple 8.7. La figure 8.3 montre les échanges de données entre trois registres d'un minicalculateur à longueur de mot de 4 bits. Les sorties des bascules $D-E$ sont reliées au canal de données au moyen de circuits logiques possédant des sorties à 3 états (voir p. 31) qui sont activées par les commandes *SEND*. Les entrées D des bascules $D-E$ sont aussi reliées au canal de données et sont activées par les commandes *RECEIVE*. A un instant donné, une seule commande *SEND* peut être activée alors qu'un nombre quelconque de commandes *RECEIVE* peuvent l'être. La figure 8.3 montre le fonctionnement du canal de données avec trois registres à quatre bits ; des registres supplémentaires peuvent être reliés en parallèle au canal de données.

Pendant qu'une instruction est effectuée, l'unité de commande active les chemins de données nécessaires pour déplacer l'instruction de la mémoire centrale au registre instruction. L'instruction est décodée par le *décodeur*

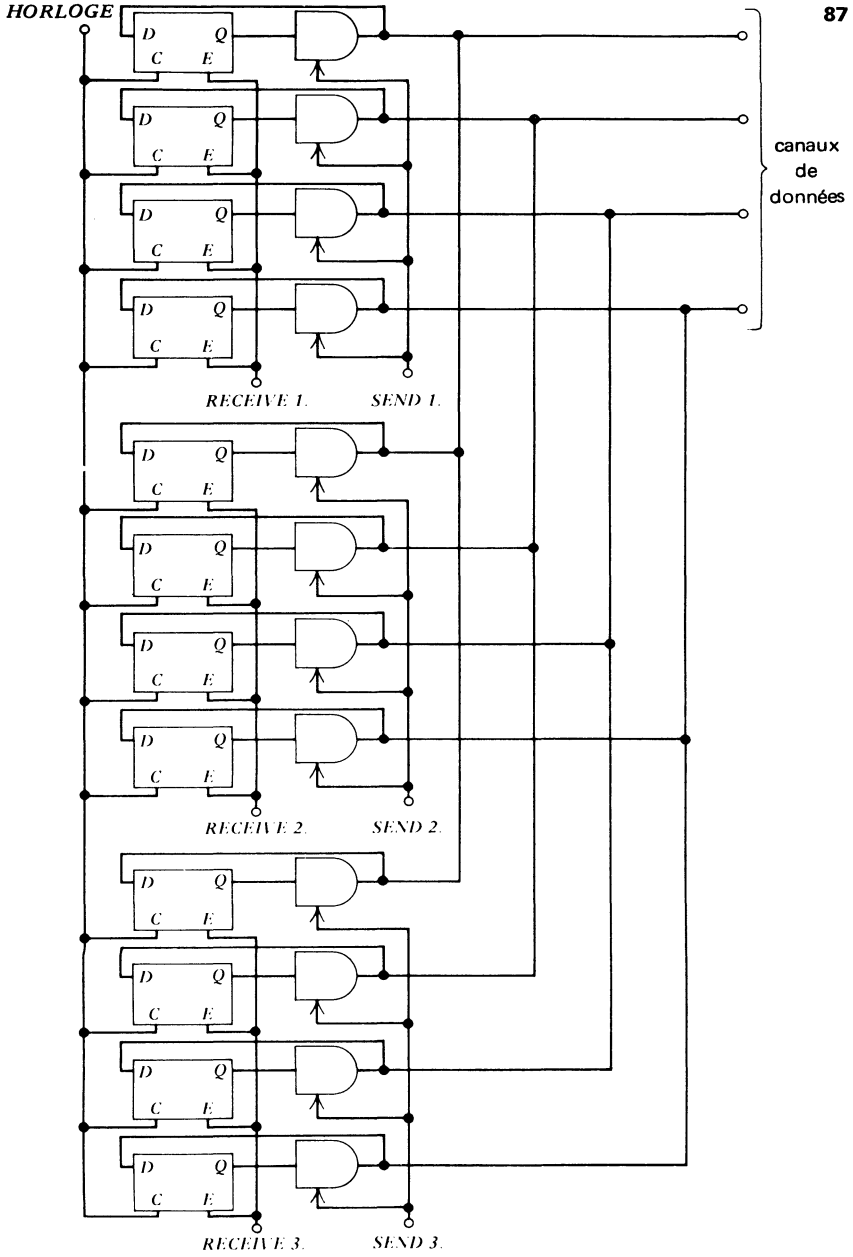


FIG. 8.3. — Canaux de données
activant les chemins de données au moyen de 3 registres à 4 bits.

d'instruction qui détermine quels chemins supplémentaires de données sont nécessaires.

Exemple 8.8. Dans l'exemple 8.3, les chemins de données nécessaires pour mettre en place les 8 premiers bits de l'instruction sont activés à l'instant t_0 (voir la fig. 8.1 et le tableau 8.2). Ces 8 bits sont mis en place et décodés par le décodeur d'instruction à l'instant t_1. Les 8 bits indiquent que le déroulement de l'instruction demande trois phases d'adressage supplémentaires et une phase d'exécution. Les chemins de données pour ces phases seront activés respectivement aux instants t_2, t_4, t_6 et t_7.

8.4 MICROPROGRAMMATION

Les chemins de données et la suite de leurs instants d'activation sont commandés par le jeu d'instructions du minicalculateur. Donc une fois le jeu d'instructions choisi, le décodeur d'instruction et les chemins de données nécessaires construits dans le hardware du minicalculateur, il n'est plus possible de modifier ou d'étendre le jeu d'instructions. Le fait de réaliser le décodeur d'instruction et les commandes des chemins de données en *firmware*, par exemple avec des mémoires mortes (ROM) donne une plus grande souplesse : on aboutit au minicalculateur *microprogrammé*. Il faut bien remarquer que le mot de microprogrammation décrit une *méthode d'activation* des chemins de données.

En microprogrammation, les chemins de données d'une instruction en langage machine sont activés par un *microprogramme*, les chemins de données d'un cycle processeur (microcycle) sont activés par une *microinstruction* et les chemins de données d'un état temporel sont activés par une *micro-opération*. Donc, un microprogramme se compose d'une ou de plusieurs microinstructions et une microinstruction se compose d'une ou de plusieurs micro-opérations.

En principe, chaque instruction dans un minicalculateur microprogrammé pourrait être représentée par un microprogramme stocké dans une ROM. Cependant, en réalité cette méthode demanderait une ROM beaucoup trop grande. Cette situation peut être évitée en remplaçant la ROM par une *zone logique programmable* plus élaborée (PLA). Dans une autre approche, les suites de micro-opérations qui se retrouvent dans beaucoup de microinstructions sont stockées une seule fois sous forme de sous-programmes microprogrammés ou micro sous-programmes.

Exemple 8.9. Tous les cycles processeur qui demandent le transfert d'un mot de la mémoire centrale au MDR utilisent des chemins de données identiques pour leurs phases d'adressage. Les informations de mise en séquence pour la commande des chemins de données correspondant à ces phases d'adressage sont stockées dans un micro sous-programme.

L'ensemble des circuits nécessaires pour la microprogrammation comprend une ROM qui est programmée pour un jeu d'instructions particulier. La ROM fournit la *commande des chemins de données* qui active les chemins de données nécessaires et la commande d'adresse suivante qui détermine l'emplacement de la prochaine micro-opération. Des circuits logiques externes à la ROM comprennent une *commande d'adresse ROM* et un *registre d'adresse ROM*.

8.5 SCHÉMA FONCTIONNEL D'UN MINICALCULATEUR

La figure 8.4 donne le schéma fonctionnel simplifié d'un minicalculateur imaginaire, schéma qui montre les détails de la CPU. Notons que la structure est assez arbitraire et peut beaucoup varier suivant les différents types de minicalculateurs.

La longueur de mot est de 8 bits donc l'ALU, l'accumulateur, le registre B et le MDR ont tous une capacité de stockage de 8 bits. Le MAR de 16 bits peut adresser $2^{16} = 65\,536$ emplacements mémoire ; le compteur ordinal et chacun des 7 niveaux de la pile d'adresses (voir chap. 9) ont aussi une capacité de 16 bits. Le registre instruction peut stocker 24 bits : 8 bits pour l'opérateur et 16 bits pour l'opérande. La commande des chemins de données est réalisée par une ROM microprogrammée se composant de $2^7 = 128$ mots de 30 bits chacun. L'ensemble des circuits auxiliaires comprend une commande d'adresse ROM et un registre d'adresse ROM de 7 bits.

La CPU du minicalculateur est reliée à l'extérieur au moyen d'une unité I/O décrite en détail dans le chapitre 4. Les instructions arithmétiques et logiques sont réalisées par l'ALU, l'accumulateur, le registre B et les 4 indicateurs.

Le chapitre 7 donne tous les détails sur la mémoire centrale ; celle-ci échange des données avec la CPU au moyen du MDR de façon bidirectionnelle et est adressée par le MAR qui peut être chargé soit à partir du compteur ordinal, soit à partir de la zone opérande du registre instruction. Les instructions sont transférées de la mémoire centrale au registre instruction au moyen du MDR en trois multiplets de 8 bits.

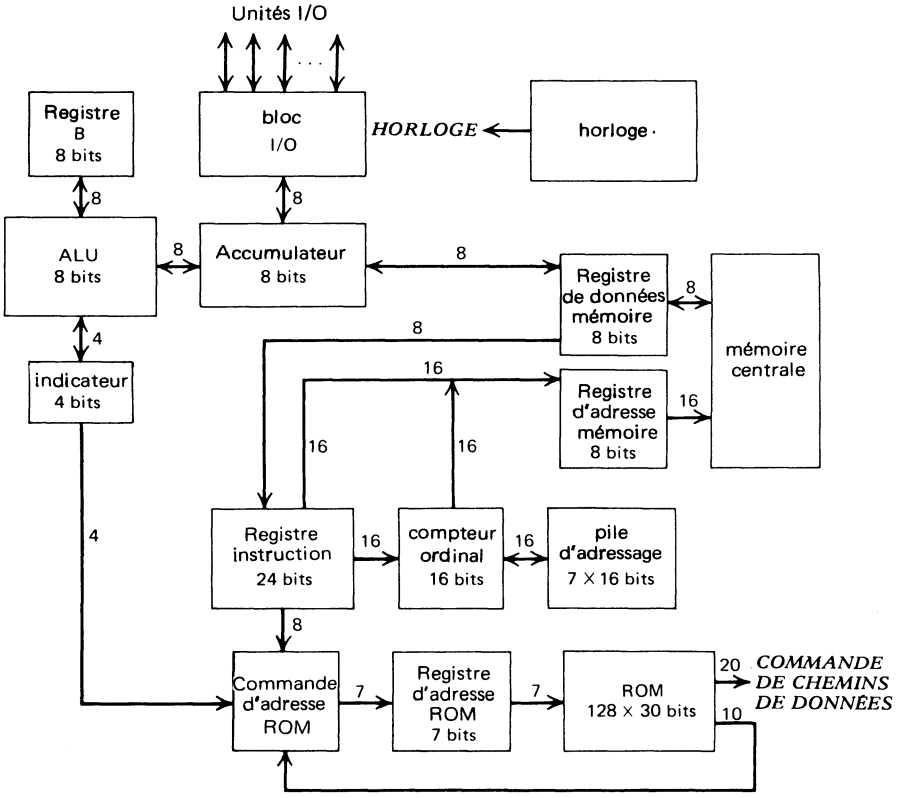


FIG. 8.4. — Schéma fonctionnel simplifié d'un minicalculateur.

Les instructions sont exécutées par la ROM, dirigée par la commande d'adresse ROM qui combine les informations des indicateurs de la zone opérateur du registre instruction et de la commande d'adresse suivante de la ROM. Les chemins de données ainsi que les opérations internes de l'ALU et de l'unité I/O sont régis par la commande de chemins de données de la ROM.

PROBLÈMES

1. Quelles sont les instructions du paragraphe 3.1 qui peuvent être réalisées en un seul cycle processeur ?

2. Remplacer les points d'interrogation des instants t_0 et t_8 dans le tableau 8.2

par les valeurs 0 ou 1 appropriées dans le cas où l'instruction est précédée et suivie d'instructions identiques.

3. Quelles modifications subirait le tableau 8.2 si l'emplacement mémoire 1205 contenait 21 au lieu de 15 ?

4. Établir un tableau qui montre les emplacements concernés de la mémoire centrale avant et après l'exécution de l'instruction décrite dans les exemples 8.4 et 8.5.

5. Établir un tableau qui montre le contenu de l'accumulateur, du MAR, du MDR et du registre instruction de l'instant t_0 à l'instant t_8 dans le cas de l'instruction des exemples 8.4 et 8.5.

6. Une phase d'attente est-elle nécessaire dans le cas du minicalculateur de l'exemple 8.6 si le temps d'accès mémoire a une durée de (*a*) 0,5 état temporel, (*b*) 2,5 états temporels ?

7. Trouver le temps nécessaire à la réalisation de l'instruction décrite dans l'exemple 8.3 si les durées d'une phase d'adressage et d'une phase d'exécution sont d'une microseconde et si le temps d'accès mémoire est inférieur à une microseconde. Même question dans le cas où les phases d'exécution inutiles sont omises.

8. Établir les valeurs binaires des signaux *RECEIVE* et *SEND* qui permettent dans le cas de la figure 8.3 de charger le contenu du plus haut registre à quatre bits dans les deux autres registres, en supposant que l'activation soit réalisée par la valeur binaire 1.

9. Indiquer quels sont les chemins de données qui sont activés pendant la réalisation de l'instruction d'addition de l'exemple 8.3.

10. Quelles sont les phases d'adressage et d'exécution des instructions du paragraphe 3 qui pourraient être réalisées par des micro sous-programmes dans un minicalculateur microprogrammé ?

9

COMPLÉMENTS

Ce chapitre présente des caractéristiques logicielles plus approfondies que celles étudiées dans le chapitre 3, en particulier sur les assembleurs, les chargeurs, les structures de données, l'édition de liens entre sous-programmes, les simulations, les systèmes d'exploitation et certaines caractéristiques hardware qui y sont associées.

9.1 ASSEMBLEURS

Un programme *assembleur* ou *assembleur symbolique* transforme un programme du langage d'assemblage en langage machine. Il affecte aussi des emplacements mémoire aux instructions, aux variables et aux constantes. Chaque instruction en langage d'assemblage est transformée en une instruction en langage machine à l'exception des macroinstructions (voir p. 24) et des *pseudo-instructions*. Les pseudo-instructions comprennent l'affectation de valeurs données (par exemple, EQUATE $W = 1{,}5$), la définition des macroinstructions (par exemple DEFINE MACRO MAC1), la caractérisation des I/O, différentes instructions de début et de fin, la caractérisation du stockage de blocs de données et la définition de formats.

Le fonctionnement de l'assembleur est illustré par un programme associé à l'instruction de l'exemple 3.7 : « IF $V > W$ THEN $X \leftarrow$ Y $-$ 1 ELSE $X \leftarrow Y + 1$ ». La figure 9.1 montre un organigramme pour le programme et le tableau 9.1 le programme en langage d'assemblage. A l'exception des pseudo-instructions EQUATE et END chaque instruction est numérotée par un numéro de ligne. Ces numéros de ligne ne sont pas utilisés par l'assembleur mais sont inscrits pour aider le programmeur. Chaque ligne numérotée représente une seule instruction ; donc le nombre de lignes dans le cas d'une macroinstruction est égal au nombre d'instructions en langage d'assemblage à l'intérieur de la macroinstruction.

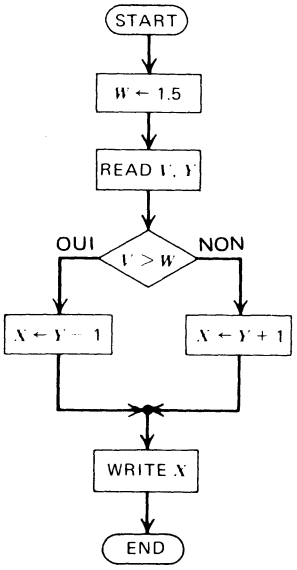


FIG. 9.1. — Organigramme du programme correspondant à « IF $V > W$ THEN $X \leftarrow Y - 1$ ELSE $X \leftarrow Y + 1$ ».

Exemple 9.1. Dans l'exemple 3.5 (p. 24) la macroinstruction MAC1 est définie comme suit :

```
DEFINE MACRO MAC1
...
...  (liste des vingt instructions définissant MAC1)
...
END
```

Cette instruction correspond à 20 lignes numérotées à chaque fois qu'elle est utilisée dans un programme en langage d'assemblage.

Les *étiquettes* du tableau 9.1 identifient les instructions JUMP et GO TO et sont placées en mémoire par l'assembleur. Les *opérateurs* sont des *codes mnémoniques d'instruction* connus de l'assembleur, les *opérandes* sont des variables et des constantes données par le programmeur et les commentaires (en général en langage non informatique) sont destinés au programmeur et sont ignorés par l'assembleur.

TABLEAU 9.1. — **Programme correspondant à « IF $V > W$ THEN $X \leftarrow Y - 1$ ELSE $X \leftarrow Y + 1$ ».** Programme en langage d'assemblage.

N° de ligne	*Étiquette*	*Instruction*		*Commentaires*
		Opérateur	Opérandes	
		EQUATE	*W*, 1.5	La valeur 1,5 est affectée à *W*.
1		READ	*V*	La valeur de *V* est lue sur une unité d'entrée.
2		READ	*Y*	La valeur de *Y* est lue sur une unité d'entrée.
3		LOAD	*W*	La valeur de *W* est chargée dans l'accumulateur.
4		SUBTRACT	*V*	La valeur de *V* est soustraite du contenu de l'accumulateur ; le signe résultant est donné par un indicateur de signe.
5		JUMP ON FLAG < 0	L1	Le programme saute à l'étiquette L1 si l'indicateur de signe montre que le contenu de l'accumulateur est négatif.
6		LOAD	*Y*	La valeur de *Y* est chargée dans l'accumulateur.
7		ADD IMMEDIATE	1	Le nombre 1 est ajouté au contenu de l'accumulateur.
8		STORE	*X*	Le contenu de l'accumulateur est stocké sous le nom de *X*.
9		GO TO	FINISH	Aller à l'étiquette FINISH.
10	L1	LOAD	*Y*	La valeur de *Y* est chargée dans l'accumulateur.
11		SUBTRACT IMMEDIATE	1	Le nombre 1 est soustrait du contenu de l'accumulateur.
12		STORE	*X*	Le contenu de l'accumulateur est stocké sous le nom de *X*.
13	FINISH	WRITE	*X*	Écrire la valeur de *X* sur une unité de sortie.
		END		

Les assembleurs les plus communément utilisés sont l'*assembleur à un passage* et l'*assembleur à deux passages*. Un assembleur à un passage transforme un programme en langage d'assemblage en un programme en langage machine en balayant le premier une seule fois. Cependant un tel assembleur entraîne des étapes supplémentaires de programmation quand

il est fait référence à une étiquette antérieure dans le programme en langage d'assemblage.

> ***Exemple 9.2.*** Le programme en langage d'assemblage donné par le tableau 9.1 est transformé en programme en langage machine par un assembleur à un passage. A la ligne 5, une référence antérieure à l'étiquette L1 est rencontrée. Cependant, l'emplacement mémoire correspondant à L1 (ligne numéro 10) n'est pas encore connu. Par conséquent, ou l'assembleur doit chercher et trouver L1 ou il doit retourner à la ligne 5 plus tard pour insérer l'emplacement mémoire correspondant à l'étiquette L1.

Le processus d'étiquetage est plus simple dans le cas de l'assembleur à deux passages. Un tel assembleur passe en revue le programme en langage d'assemblage deux fois. Au cours du premier passage il identifie les étiquettes et crée une *table des symboles* qui est composée des étiquettes et de leur emplacement mémoire. Les instructions en langage d'assemblage sont transformées en instruction en langage machine au cours du second passage. Les renseignements tirés du premier passage sont souvent transmis en vue de l'utilisation au second passage au moyen d'une mémoire externe telle qu'un ruban papier ou un disque magnétique.

9.2 CHARGEURS

Un programme en langage machine peut être lu dans la mémoire centrale du minicalculateur grâce à un programme de chargement (chargeur). Le chargeur lit les instructions de façon séquentielle et recherche les sous-programmes nécessaires au programme et déjà présents en mémoire *(programmes bibliothèque)*. Certains chargeurs sont aussi capables de reloger le programme ; cette caractéristique est particulièrement intéressante quand plusieurs programmes différents sont chargés dans la mémoire centrale.

Pour initialiser le fonctionnement du minicalculateur, un petit chargeur, souvent désigné sous le nom de *chargeur amorce* (ou *bootstrap*) est utilisé pour préparer le minicalculateur au chargement du premier programme (en général un chargeur plus important) en mémoire centrale. Le chargeur amorce peut être stocké dans une mémoire permanente telle qu'une logique câblée, une mémoire morte, un ruban papier ou un disque magnétique. Par ailleurs, il peut être lu de façon manuelle au moyen du tableau de commande.

9.3 STRUCTURES DES DONNÉES

Les structures de données concernent le stockage et le recouvrement de données dans un système de minicalculateur. Toute méthode qui permet le recouvrement de données peut être appelée structure de données ; cependant, dans la suite, nous nous limiterons aux piles et aux files qui sont toutes deux stockées dans des *emplacements mémoire consécutifs*. Une autre structure de données importante, la *zone*, est étudiée brièvement dans les problèmes 3 et 4 à la fin de ce chapitre.

Piles

L'une des plus simples structures de données, habituellement utilisée dans les liens de sous-programmes est la pile aussi appelée *pile refoulante* ou *pile LIFO* (Last in, first out). Dans une pile, toutes les manipulations sont faites à l'une des extrémités de la suite d'emplacements mémoire. La figure 9.2 montre 3 représentations d'une pile à 7 mots. Dans la colonne

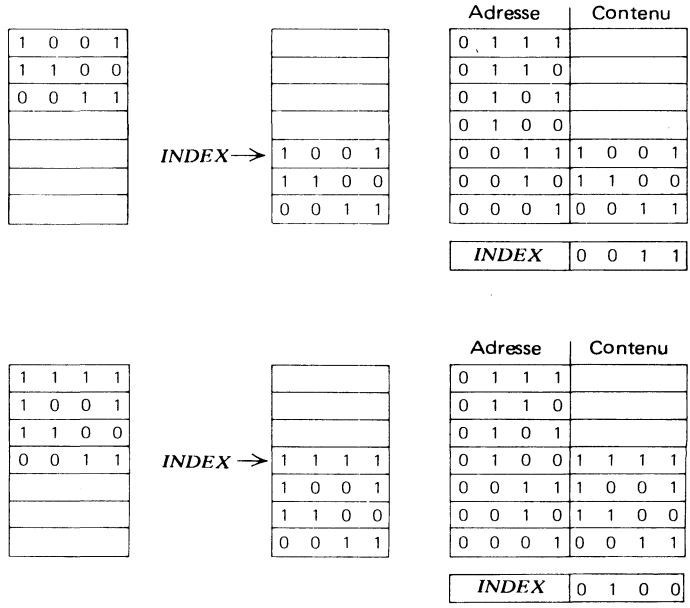


FIG. 9.2. — Trois représentations d'une pile d'une capacité de stockage de 7 mots de 4 bits. Partie supérieure : 3 mots dans la pile ; partie inférieure : 4 mots dans la pile.

de gauche, la pile est représentée sous forme refoulante. La partie supérieure de la figure 9.2 montre la pile avec 3 mots remplis et la partie inférieure la pile avec un mot supplémentaire 1111 ajouté au sommet.

Nous pouvons voir que dans la représentation de gauche, le contenu de la pile est refoulé et le nouveau mot est ajouté à l'entrée. Si un mot est alors retiré, ce sera le dernier à avoir été entré, c'est-à-dire le mot 1111 et la pile reviendra à son état précédent représenté dans la partie supérieure de la figure 9.2. Cette représentation de la pile peut être réalisée directement en utilisant des registres de décalage bidirectionnels.

La seconde colonne de la figure 9.2 montre une autre représentation d'une pile. On peut la comparer à un empilement de livres. Les mots sont ajoutés et retirés au sommet de la pile. La hauteur de la pile est indiquée par un index. Cette représentation peut être réalisée dans la mémoire centrale ou dans une mémoire intermédiaire comme le montre la colonne de droite de la figure 9.2. Remarquons qu'une adresse particulière a été réservée à l'index.

La figure 9.3 montre l'organigramme des opérations réalisées sur la pile pour la représentation de droite de la figure 9.2. La figure 9.3 *a*, montre l'entrée d'un mot dans la pile et la figure 9.3 *b* le retrait d'un mot

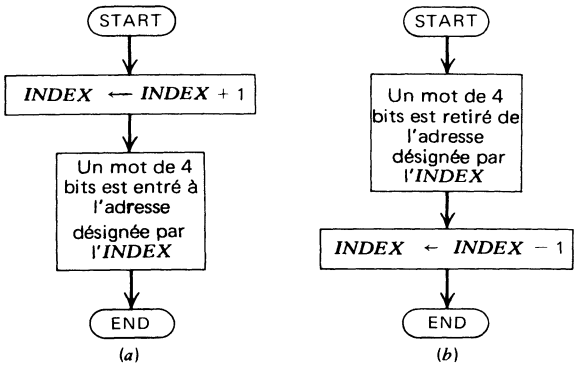


FIG. 9.3. — Programmes d'utilisation d'une pile.
(*a*) Entrée d'un mot dans la pile. (*b*) Retrait d'un mot de la pile.

de la pile. Remarquons que ces programmes ne permettent pas de détecter un *débordement de pile* qui se produit lorsque l'on veut ajouter un mot à la pile et que celle-ci est pleine. De même, rien ne permet de détecter si la pile est vide quand on veut retirer un mot. Dans des systèmes plus évolués ces protections sont habituellement présentes ; cependant, elles restent du ressort du programmeur dans les petits systèmes.

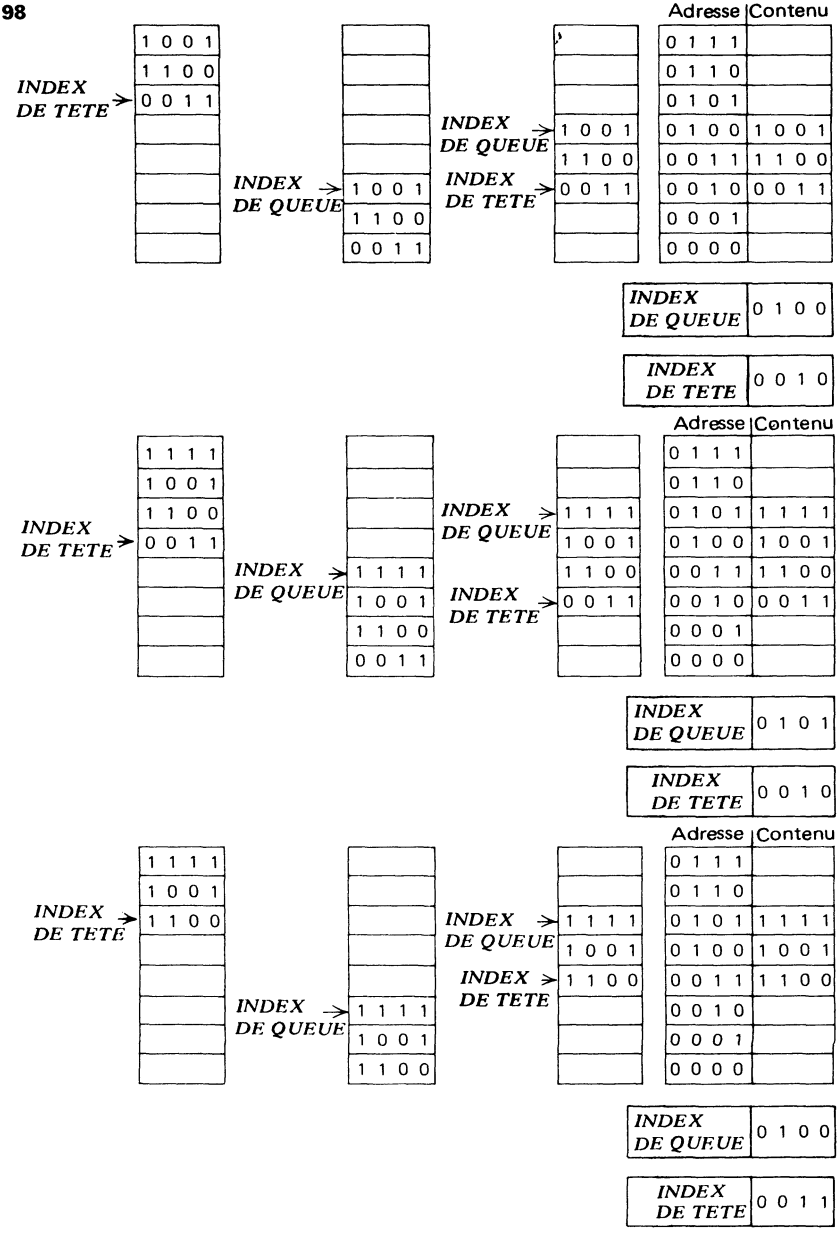


FIG. 9.4. — Quatre représentations d'une file d'une capacité de stockage de 8 mots de 4 bits. Ligne supérieure : 3 mots dans la file ; ligne intermédiaire : le mot 1111 est ajouté ; ligne inférieure : le mot 0011 est retiré.

Files

Une autre structure de données souvent utilisée pour l'attente de l'accès aux unités entrée-sortie est la *file d'attente*. Toute addition à la file est faite à sa fin et tout retrait est fait à son début. La figure 9.4 montre 4 représentations d'une file. La représentation de la colonne de gauche peut être

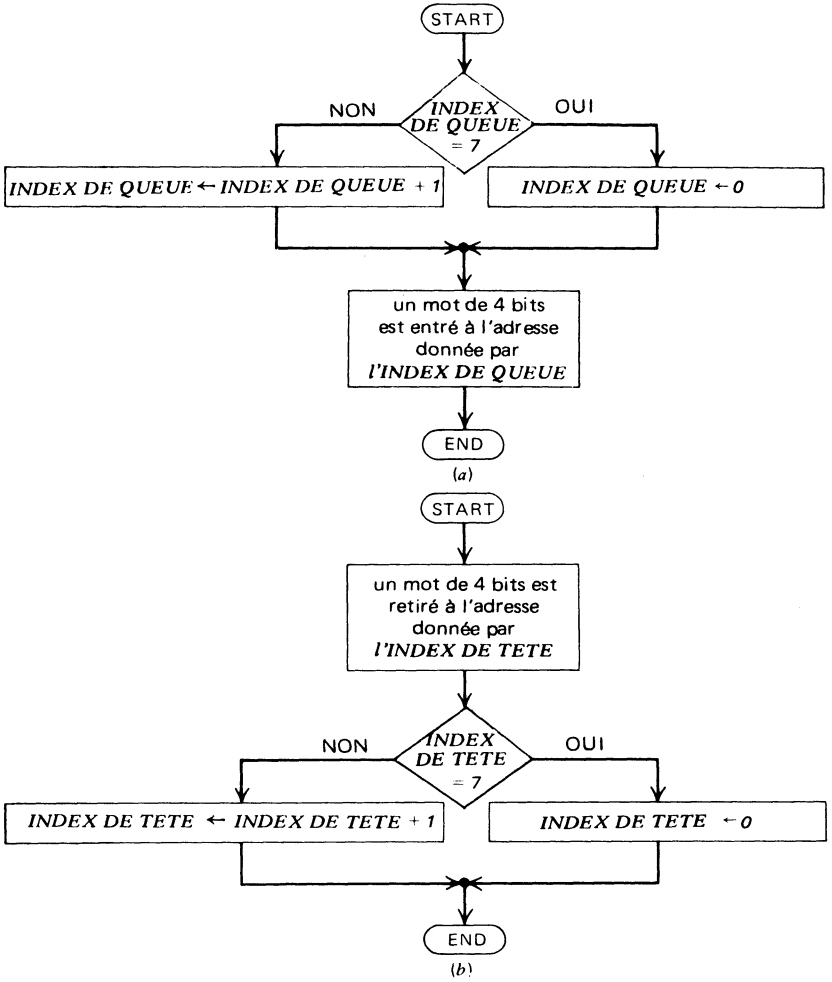


FIG. 9.5. — Programme d'utilisation des files. (*a*) Entrée. (*b*) Retrait.

réalisée grâce à des registres de décalage avec entrée en série et sortie en parallèle. Celle de la seconde colonne est rarement utilisée et seulement citée pour être exhaustif. Dans la représentation de la troisième colonne, le contenu n'est pas décalé. Cette représentation peut être réalisée dans une mémoire centrale ou une mémoire intermédiaire comme le montre la colonne de droite. Remarquons que l'addition d'un mot et le retrait d'un autre décale la file d'un emplacement mémoire. Ainsi, la file pourrait sortir de sa zone mémoire même quand elle n'est qu'en partie remplie. Ceci est évité par un bouclage, c'est-à-dire, en déclarant l'adresse 0111 ($= 7_{10}$) contiguë à l'adresse 0, comme le montrent les programmes de la figure 9.5. Ces programmes font fonctionner la file correctement tant que ne se produit aucun débordement ou tant que la file n'est pas vide. Le programme doit donc prévoir la détection de ces phénomènes dans les petits systèmes, ce qui est plus délicat que dans le cas des piles.

9.4 LIENS DE SOUS-PROGRAMME

L'appel d'un sous-programme et le retour dans le programme initial nécessite plusieurs opérations réunies sous le terme de *liens de sous-programme*. L'une de ces opérations est le stockage dans une pile du contenu du compteur ordinal qui servira comme adresse de retour. La même pile peut être aussi utilisée pour stocker les contenus des registres et des indicateurs au cas où ces éléments doivent être utilisés par le sous-programme. Les liens de sous-programme permettent aussi de transmettre les paramètres entre le programme principal et les sous-programmes.

Exemple 9.3. Cet exemple illustre l'utilisation des liens de sous-programme et la transmission des paramètres dans le cas d'un programme qui lit la valeur de x à partir d'une bande papier, appelle un sous-programme qui calcule $x + \sqrt{x}$, appelle un sous-programme qui calcule $\sqrt{x}$ et imprime le résultat. Remarquons que le but de ce programme est d'illustrer l'utilisation des liens de sous-programme, donc que le programme n'est pas nécessairement optimal (voir aussi le problème 5 à la fin de ce chapitre). Remarquons aussi que beaucoup des instructions utilisées devraient être des macroinstructions dans le cas d'un minicalculateur.

Le tableau 9.2 montre le programme. Celui-ci est divisé en trois parties qui sont localisées dans trois zones de la mémoire centrale : la *zone programme principal*, la *zone paramètre* et la

TABLEAU 9.2. — **Programme permettant de calculer** $x + \sqrt{x}$.

Emplacement mémoire	*Contenu*
Zone programme principal	
0	Début.
1	Lire le ruban papier et stocker le nombre dans l'emplacement mémoire 16.
2	Appeler le sous-programme calculant $x + \sqrt{x}$ d'adresse 32.
3	Imprimer le contenu de l'emplacement 17.
4	Stop.
Zone paramètre	
16	Entrée de x dans le sous-programme calculant $x + \sqrt{x}$.
17	Sortie du sous-programme calculant $x + \sqrt{x}$.
18	Entrée dans le sous-programme « racine carrée ».
19	Sortie du sous-programme « racine carrée ».
Zone sous-programme	
32	Début du sous-programme $x + \sqrt{x}$.
33	Stocker le contenu de l'emplacement mémoire 16 dans l'emplacement mémoire 35.
34	Aller à l'emplacement mémoire 36.
35	Stockage de l'entrée du sous-programme calculant $x + \sqrt{x}$.
36	Stocker le contenu de l'emplacement mémoire 35 dans l'emplacement mémoire 18.
37	Appeler le sous-programme « racine carrée » d'adresse 48.
38	Ajouter le contenu de l'emplacement mémoire 19 au contenu de l'emplacement 35 et stocker le résultat dans l'emplacement mémoire 40.
39	Aller à l'emplacement mémoire 41.
40	Stockage de $x + \sqrt{x}$.
41	Stocker le contenu de l'emplacement mémoire 40 dans l'emplacement mémoire 17.
42	Retour du sous-programme calculant $x + \sqrt{x}$.
48	Début du sous-programme « racine carrée ».
49	Stocker le contenu de l'emplacement mémoire 18 dans l'emplacement mémoire 51.
50	Aller à l'emplacement mémoire 52.
51	Stockage de l'entrée du sous-programme « racine carrée ».
52 53 54	Ces trois pas calculent la racine carrée du nombre stocké dans l'emplacement mémoire 51. Le résultat est stocké dans l'emplacement mémoire 56.
55	Aller à l'emplacement mémoire 57.
56	Stockage de la sortie du sous-programme « racine carrée ».
57	Stocker le contenu de l'emplacement mémoire 56 dans l'emplacement mémoire 19.
58	Retour du sous-programme « racine carrée ».

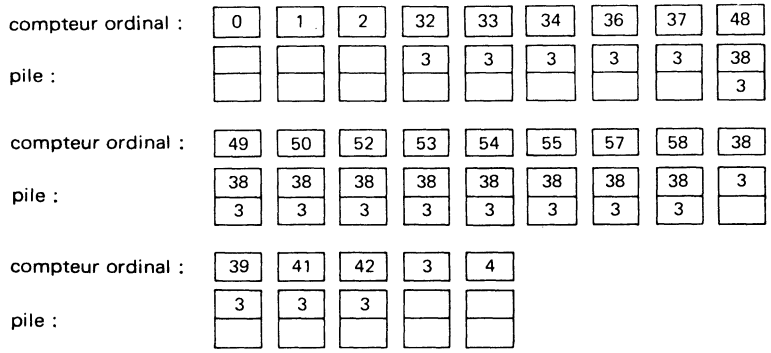


FIG. 9.6. — Calcul de $x + \sqrt{x}$: fonctionnement du compteur ordinal et de la pile.

zone sous-programme. La zone paramètre (souvent placée dans la page de base) doit être adressable à la fois à partir de la zone programme principal et à partir de la zone sous-programme. Cependant, aucun adressage n'est nécessaire entre la zone programme principal et la zone sous-programme. La figure 9.6 montre le fonctionnement du compteur ordinal et de la pile ; cette dernière est représentée comme une pile refoulante à deux mots du type indiqué dans la colonne de gauche de la figure 9.2.

Le programme démarre à l'emplacement mémoire 0 avec la pile vide. Après l'exécution de l'instruction de lecture et de stockage qui se trouve à l'emplacement 1, le sous-programme d'adresse 32 est appelé ; en conséquence, l'adresse de retour (adresse 3) est entrée dans la pile et le compteur ordinal prend la valeur de l'adresse du sous-programme appelé, c'est-à-dire 32.

Alors l'exécution du sous-programme a lieu. Le paramètre x qui avait été stocké par le programme principal à l'adresse 16 est maintenant placé dans l'emplacement mémoire *35 au* moyen de l'instruction d'adresse 33. En prévision de l'appel du sous-programme « racine carrée », l'instruction d'adresse 36 charge x dans l'emplacement 18 de la zone paramètre. L'instruction d'adresse 37 appelle le sous-programme « racine carrée » d'adresse 48. En conséquence, l'adresse de retour 38 est entrée dans la pile et le compteur ordinal prend la valeur 48.

Le sous-programme « racine carrée » est exécuté ensuite. Il

se termine par l'instruction de retour d'adresse 58 qui entre l'adresse de retour 38 dans le compteur ordinal. Alors l'exécution du premier sous-programme reprend et continue jusqu'à l'instruction de retour d'adresse 42 qui entre l'adresse de retour 3 dans le compteur ordinal. L'instruction d'impression d'adresse 3 du programme principal est exécutée ensuite et le programme se termine par l'instruction STOP d'adresse 4.

9.5 SIMULATION

La bonne marche d'un système sur minicalculateur devrait être vérifiée dans des conditions réelles de fonctionnement. Cependant, il est souvent préférable d'effectuer un test simulé qui n'exige pas l'utilisation d'unités I/O externes.

Exemple 9.4. Le minicalculateur contrôleur de circulation de l'exemple 2.2 est vérifié au moyen d'un test spécial effectué par un calculateur plus grand et qui n'utilise ni capteurs ni véhicules. Les données obtenues donnent les caractéristiques du contrôleur dans différents cas simulés de circulation.

La simulation peut être aussi utilisée dans la conception d'un système sur minicalculateur. La bonne marche d'un programme peut être vérifiée en simulant le fonctionnement du minicalculateur grâce à un programme de simulation qui est exécuté dans un autre calculateur.

Exemple 9.5. Un minicalculateur est choisi pour être utilisé dans un système de contrôle. Ensuite un programme de contrôle est écrit en accord avec les exigences du système de contrôle et les règles de programmation du minicalculateur. Ce programme et un programme de simulation sont traités dans un gros calculateur qui fournit les caractéristiques du programme de contrôle et du minicalculateur dans des conditions réelles de fonctionnement.

9.6 PARTAGE HARDWARE

Jusqu'à présent, nous supposions tacitement que le minicalculateur et ses unités périphériques n'effectuaient qu'une seule tâche et n'exécutait qu'un programme. Cependant ce n'est pas obligatoirement le cas. Ainsi

une unité périphérique utilisée par intermittence peut être partagée entre plusieurs minicalculateurs.

> ***Exemple 9.6.*** Dans une usine chimique, trois processus sont contrôlés par trois minicalculateurs indépendants mais situés au même endroit. Les programmes sont conçus ailleurs et sont disponibles sur bande perforée. Un programme est entré dans un minicalculateur donné au moyen d'un lecteur de ruban unique utilisable par n'importe lequel des trois minicalculateurs.

Dans d'autres exemples une unité mémoire unique peut être partagée entre plusieurs programmes *(multiprogrammation)* ou entre plusieurs minicalculateurs.

9.7 SYSTÈMES D'EXPLOITATION

En plus des entrées-sorties, de la CPU, de la mémoire et du programme, le fonctionnement d'un minicalculateur exige aussi du *hardware de fonctionnement* et du *logiciel de fonctionnement*. Le hardware de fonctionnement peut se composer des commandes du panneau de contrôle ainsi que des unités périphériques supplémentaires servants d'interface entre le minicalculateur et les utilisateurs. Le logiciel de fonctionnement souvent appelé *système d'exploitation* ou *programme d'exécution* fonctionne de concert avec le hardware de fonctionnement. Il peut se composer d'un *éditeur de programme* ou *éditeur de texte* qui facilite les modifications dans un programme avant son assemblage et d'un *programme de mise au point* qui fonctionne pendant l'exécution du programme de l'utilisateur pour l'aider à détecter les erreurs. Le système d'exploitation permet aussi la mise en séquence des opérations d'assemblage et de chargement et il régit le partage hardware et le déroulement des programmes.

> ***Exemple 9.7.*** Un programme pour un minicalculateur est écrit dans un langage d'assemblage. Le résultat est appelé *code source* (ou *module source* ou *fichier source*) et peut être un *paquet de cartes source* ou une *bande source*. Le code source est lu dans le minicalculateur et il est traduit en langage machine par l'assembleur. Le résultat est un *code objet* (ou *module objet* ou *fichier objet*) qui peut être un paquet de cartes objet ou une bande objet. Le code objet est lu dans le minicalculateur et les programmes bibliothèque appelés y sont inclus par

l'*éditeur de liens*. Le résultat est un *module de chargement* qui peut être chargé en mémoire centrale par un programme chargeur. Ces opérations sont dirigées par le système d'exploitation et par l'intervention manuelle de l'utilisateur.

PROBLÈMES

1. Généraliser l'organigramme de la figure 9.3 *a* pour obtenir une indication de débordement de pile quand la pile est pleine et que l'entrée d'un mot dans la pile est effectuée.
2. Généraliser l'organigramme des figures 9.5 *a* et 9.5 *b* pour obtenir une indication de débordement de file quand la file est pleine et que l'entrée d'un mot dans la file est effectuée. Nous supposons que la file est initialement vide et que les deux index indiquent l'emplacement 0.
3. Une structure de données simple et courante qui utilise des emplacements mémoire consécutifs est la zone. La zone la plus simple est la zone unidimensionnelle qui est en fait une liste ordonnée. Un exemple d'une telle zone est la liste contenant les heures de coucher du soleil pour les 31 jours d'un mois. Établir l'organigramme qui permet de lire et de stocker chaque *élément* de cette zone. Inclure des tests d'erreur qui permettent de détecter l'accès à un emplacement mémoire extérieur à la zone (par exemple l'accès au jour 32).
4. Généraliser l'organigramme du problème 3 ci-dessus au cas d'une zone *bidimensionnelle* qui contient les heures de lever et de coucher du soleil pour les 31 jours d'un mois.
5. Étudier l'emplacement de la zone paramètre et trouver une simplification des opérations d'adresses 33 à 36 et 49 à 51 du programme du tableau 9.2.

RÉPONSES AUX PROBLÈMES

Chapitre 2

1 : 76.

Chapitre 3

1 : 00000101, 00001011, 11111111. 7 : 51.

Chapitre 4

1 : de droite à gauche.

Chapitre 5

1 : 45 ; 0,71875 ; 6.625. 2 : 1111011 ; 0,01001 ; 11100000,011.
7 : 1 1001 0011,0010 ; 1 0001 0111,0010 0101 ; 1,0111 0110 1001.
9 : E=0 1000, M=0 111101 ; E=0 0101, M=1 01000101 ; E=0 0100.
M=0 101000001 ; E=1 1011, M=1 01011.

Chapitre 7

1 : 32. 5 : $41\,984_{10}$, $42\,075_{10}$.

Chapitre 8

2 : OF_{16}, $C104B5_{16}$ (5 fois), 101_{10}. 6 : Oui, non.
7 : 8 microsecondes, 5 microsecondes. 8 : RECEIVE 1 = 0, SEND 1=1, RECEIVE 2=1, SEND 2=0, RECEIVE 3=1, SEND 3=0.

Chapitre 9

5 : utilisation minimum de l'adressage indirect.

ANNEXE A

TABLES ARITHMÉTIQUES EN BASE 8

TABLE A.1 **Addition**

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

TABLE A.2 **Multiplication**

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

TABLE A.3 **Multiples de** 10_{10}

BASE 10	BASE 8
10	12
20	24
30	36
40	50
50	62
60	74
70	106
80	120
90	132

ANNEXE B

TABLES ARITHMÉTIQUES EN BASE 16

TABLE B.1 **Addition**

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

TABLE B.2 **Multiplication**

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

ANNEXE C

TABLE DES PUISSANCES 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	0.0625						
32	5	0.03125						
64	6	0.01562	5					
128	7	0.00781	25					
256	8	0.00390	625					
512	9	0.00195	3125					
1024	10	0.00097	65625					
2048	11	0.00048	82812	5				
4096	12	0.00024	41406	25				
8192	13	0.00012	20703	125				
16384	14	0.00006	10351	5625				
32768	15	0.00003	05175	78125				
65536	16	0.00001	52587	89062	5			
1 31072	17	0.00000	76293	94531	25			
2 62144	18	0.00000	38146	97265	625			
5 24288	19	0.00000	19073	48632	8125			
10 48576	20	0.00000	09536	74316	40625			
20 97152	21	0.00000	04768	37158	20312	5		
41 94304	22	0.00000	02384	18579	10156	25		
83 88608	23	0.00000	01192	09289	55078	125		
167 77216	24	0.00000	00596	04644	77539	0625		
335 54432	25	0.00000	00298	02322	38769	53125		
671 08864	26	0.00000	00149	01161	19384	76562	5	
1342 17728	27	0.00000	00074	50580	59692	38281	25	
2684 35456	28	0.00000	00037	25290	29846	19140	625	
5368 70912	29	0.00000	00018	62645	14923	09570	3125	
10737 41824	30	0.00000	00009	31322	57461	54785	15625	
21474 83648	31	0.00000	00004	65661	28730	77392	57812	5
42949 67296	32	0.00000	00002	32830	64365	38696	28906	25

INDEX ALPHABÉTIQUE

U

V

Z

IMPRIMERIE BARNÉOUD S. A.
LAVAL (Mayenne)
N° 7278 — 3-1977

Dépôt légal : 1er trim. 1977
N° d'Éditeur : 3196

Imprimé
en France

ÉDITIONS EYROLLES

ALAZARD et LAMOITIER - **Le langage basic et ses extensions** - 216 p., 1973

BONNIN - **Le Cobol A.N.S.** avec exercices et corrigés - 192 p., 1977

— **Les extensions au Cobol A.N.S.** avec exercices et corrigés 200 p., 1976

— **Pratique du PL/1 et programmation sructurée** - 176 p., 1976

— **Les techniques avancées de programmation PL/1** - 168 p., 1976

DÉANGELI - **Programmation de la production des produits de série** - 232 p., 1976

LARRÉCHÉ - **Le basic,** une introduction à la programmation - 120 p., 1977

LORIFERNE - **La conversion analogique-numérique, numérique-analogique** - 110 p., 1976

RADIX - **Introduction au filtrage numérique.** Lissage de données. Estimation de paramètres. Identification de processus. Exercices et solutions - 240 p., 1970

SARZOTTI - **Introduction aux techniques d'évaluation et de mesure des systèmes informatiques** - 288 p., 1977

STEWART et JENSEN - **Solution numérique des problèmes matriciels** - 252 p., 1975

ÉDITIONS EYROLLES