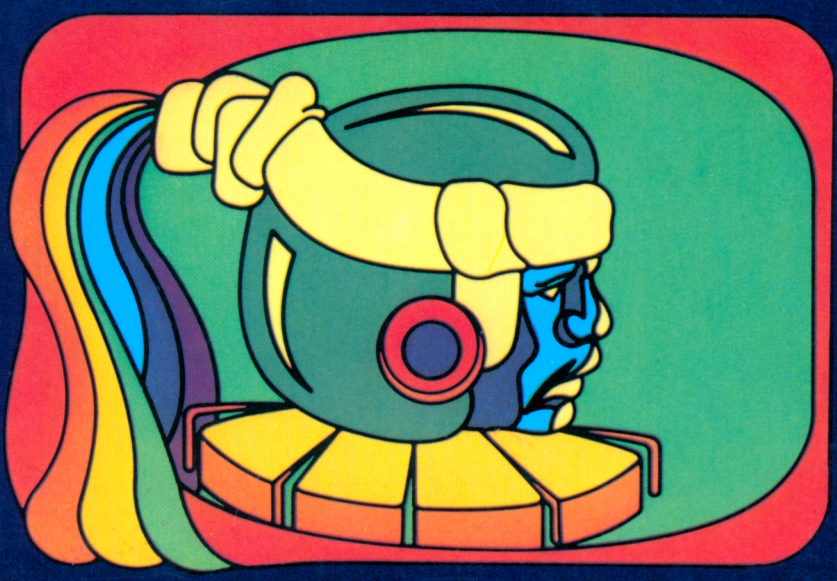
Z80

PROGRAMMATION en LANGAGE ASSEMBLEUR

LANCE A. LEVENTHAL

OSBORNE/McGRAW-HILL



S. E. C. F.
ÉDITIONS RADIO



S. E. C. F.
Editions Radio

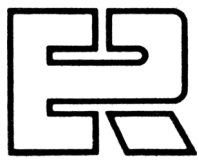
9, rue Jacob, 75006 Paris - Tél. 329.63.70

Z 80
PROGRAMMATION EN LANGAGE ASSEMBLEUR

LANCE A. LEVENTHAL

Z 80
PROGRAMMATION EN LANGAGE ASSEMBLEUR

S. E. C. F.



Éditions Radio
9, RUE JACOB - 75006 PARIS
TÉL. : 329-63-70

L'éditeur tient à remercier tout particulièrement
Monsieur Alain GEORGES,
Ingénieur à la Société ZILOG FRANCE,
qui a effectué le contrôle technique de cet ouvrage.

Copyright © 1979 by Adam Osborne & Associates
Z 80 Assembly language programming
Translation copyright © Adam Osborne & Associates Inc. 1983

Imprimé en France par Berger-Levrault, Nancy
Dépôt légal : mars 1983
Éditeur n° 908. Imprimeur n° 778036
I.S.B.N. 2 7091 0909 3

CHAPITRE 1

INTRODUCTION AU LANGAGE DE PROGRAMMATION ASSEMBLEUR

Cet ouvrage traite de la programmation en langage assembleur. Le lecteur est supposé connaître les principes de fonctionnement des micro-ordinateurs, qui ne sont pas décrits ici : caractéristiques générales, modes d'adressage, jeux d'instructions, etc., exposés dans un livre intitulé « Initiation aux micro-ordinateurs - Niveau 2 », de A. Osborne, édité par SECF-Éditions Radio.

COMMENT LIRE CE LIVRE

On remarquera que le texte de ce livre est imprimé en gras ou en maigre. **Cela, afin d'aider le lecteur à sauter les passages qui lui sont familiers. Le texte en maigre ne fait que développer les informations en gras. De ce fait, on pourra ne lire que ce qui est imprimé en gras ; si l'on trouve un thème sur lequel on souhaite davantage d'explications, on abordera alors la lecture du texte imprimé en maigre.**

SIGNIFICATION DES INSTRUCTIONS

Le jeu d'instructions d'un microprocesseur est l'ensemble des entrées binaires qui produisent des actions définies pendant un cycle d'instruction. Un jeu d'instructions est à un microprocesseur ce qu'une table de fonctions est à un système logique, porte, additionneur ou registre à décalage. Bien évidemment, les commandes exécutées par un microprocesseur en réponse aux instructions qu'il reçoit sont bien plus complexes que celles qu'exécute un dispositif logique.

INSTRUCTIONS BINAIRES **Une instruction est simplement une séquence de bits (entités binaires) — elle doit être disponible à l'entrée des données du microprocesseur à un moment précis afin d'être interprétée comme une instruction.** Par exemple, lorsque le microprocesseur Z80 reçoit le mot binaire 10000000 en entrée, au cours d'une opération de recherche en mémoire (« *fetch* »), cela signifiera :

« additionner le contenu du registre B au contenu de l'accumulateur »

De même, la configuration 00111110 signifie :

« charger l'accumulateur avec le contenu du mot suivant de la mémoire de programme »

Le microprocesseur (tout comme n'importe quel ordinateur) ne reconnaît dans les séquences de bits que des instructions ou des données ; il ne distingue ni les mots ni les nombres, que ce soit en octal, décimal ou héxadécimal.

UN PROGRAMME INFORMATIQUE

Un programme est une série d'instructions qui obligent un ordinateur à exécuter une tâche particulière.

PROGRAMME INFORMATIQUE

En fait, un programme inclut, outre des instructions, des données et des adresses en mémoire utilisées pár le microprocesseur pour accomplir les tâches définies par les instructions. Par exemple, si le microprocesseur doit faire une addition, il doit disposer de deux nombres et d'un endroit où il placera le résultat. Le programme informatique doit déterminer les sources des données et la destination du résultat, ainsi que le type d'opération à exécuter.
Tout microprocesseur exécute les instructions séquentiellement, à moins qu'une des instructions ne modifie la séquence ou n'arrête le microprocesseur. Ainsi, le microprocesseur acquiert toujours l'instruction suivante, sauf si l'instruction en cours en décide autrement.
En dernière analyse, chaque programme est transformé en un ensemble de nombres binaires. Voici, par exemple, le programme qui commandera au Z80 d'additionner le contenu des positions-mémoires 60_{16} et 61_{16} et de ranger le résultat en 62_{16} :

```
00111010
01100000
00000000
01000111
00111010
01100001
00000000
10000000
00110010
01100010
00000000
```

PROGRAMME OBJET

PROGRAMME EN LANGAGE MACHINE

Cette liste est un programme en langage machine, ou encore improprement appelé un programme objet*. Si un tel programme était entré dans la mémoire d'un ordinateur construit autour d'un Z80, le microprocesseur serait capable de l'exécuter directement.

NDT : En fait, on appelle programme objet (ou code objet) le code généré par tout traducteur d'un langage dans un autre. Ainsi, un « assembleur » produit un code objet qui, dans le cas d'ordinateurs quelque peu évolués, devra être traité par un « éditeur de liens » (voir ce mot plus loin) avant de pouvoir être exécuté, et sera donc différent du code exécutable.

LE PROBLÈME DE LA PROGRAMMATION

Nombreuses sont les difficultés associées à la création d'un programme objet, ou programme en langage binaire machine. En voici quelques-unes :

1) Le programme est difficile à comprendre et à mettre au point (tous les nombres binaires se ressemblent, surtout après avoir travaillé avec eux pendant plusieurs heures).
2) L'introduction du programme est longue, car il faut déterminer chaque bit séparément.
3) Le programme ne décrit pas la tâche à accomplir d'une façon évidente, telle que le ferait un texte écrit en français par exemple.
4) Le programme est long et fastidieux à écrire.
5) Le programmeur commet souvent, par manque d'attention, des erreurs très difficiles ensuite à détecter.

Par exemple, **la version suivante du programme d'addition en langage machine comporte une simple interversion de bits. Essayez de la trouver :**

```
00111010
01100000
00000000
01000111
01110010
01100001
00000000
10000000
00110010
01100010
00000000
```

Ainsi, le binaire que l'ordinateur traite avec facilité ne convient pas au programmeur. On trouve généralement les programmes binaires longs, pénibles, prêtant à confusion et sans signification apparente. Un programmeur pourra tenter de se souvenir de quelques codes binaires mais il pourrait investir ses efforts dans des tâches plus productives.

UTILISATION DE L'OCTAL OU DE L'HEXADÉCIMAL

OCTAL OU HEXADÉCIMAL

La situation s'améliore sensiblement si, au lieu d'écrire en binaire, on rédige les instructions en octal ou en hexadécimal. Dans ce livre, on utilisera l'hexadécimal parce que les nombres sont plus courts et parce qu'il s'agit d'un standard de l'industrie des microprocesseurs. Le tableau 1.1 donne l'équivalent hexadécimal et binaire des nombres décimaux. **Le programme d'addition du Z80 devient alors :**

```
3A
60
00
47
3A
61
00
80
32
62
00
```

On constate ainsi que la version hexadécimale est plus courte à écrire et moins pénible à examiner.

Les erreurs sont plus faciles à détecter dans une séquence hexadécimale. La version erronée du programme d'addition devient, en effet, en hexadécimal :

```
3A
60
00
47
72
61
00
80
32
62
00
```

L'erreur est plus facilement localisée.

Que faire, ensuite, avec ce programme hexadécimal ? Le microprocesseur ne comprend que les codes d'instructions binaires. La réponse est qu'il faut convertir les nombres hexadécimaux en nombres binaires. Cette conversion est une tâche répétitive et fastidieuse. Ceux qui l'entreprennent introduisent toutes sortes d'erreurs, par exemple se trompent de ligne, oublient un bit, ou transposent mal les chiffres.
Or, il s'agit là d'une activité parfaite pour un ordinateur. La machine ne se fatigue jamais, n'est jamais saturée, et ne commet jamais de telles erreurs bêtes. L'idée consiste alors à rédiger **un programme qui prend les nombres hexadécimaux, les convertit en nombres binaires, et place ces derniers dans la mémoire du micro-ordinateur. Un tel programme standard est fourni avec de nombreux microprocesseurs ; on l'appelle un chargeur hexadécimal.**

CHARGEUR HEXADÉCIMAL

Le chargeur hexadécimal vaut-il la peine d'être acquis ? Si vous rédigez vos programmes en binaire et si vous les introduisez en binaire dans la machine, alors non, vous n'avez pas besoin d'un chargeur hexadécimal.
Mais si vous choisissez le chargeur hexadécimal, vous devrez le payer. Le chargeur hexadécimal est en lui-même un programme qu'il faut charger en mémoire. De plus, le chargeur hexadécimal occupera de la mémoire — mémoire que vous souhaiteriez voir utilisée à d'autres fins.
Le compromis fondamental réside ainsi entre le prix et les besoins en mémoire pour le chargeur hexadécimal, et l'économie en temps de programmation.
Le chargeur hexadécimal vaut bien son faible prix.
Il ne résoud certainement pas tous les problèmes de programmation. La version hexadécimale d'un programme est encore difficile à lire et à interpréter ; par exemple, on ne distingue pas les instructions des données ou des adresses ; de plus, le listage ne suggère aucunement ce que le programme est censé faire. Que signifient 32, ou 47, ou 3A ? Apprendre par cœur une page pleine de codes est une suggestion peu attractive. De plus, ces codes seront totalement différents avec un autre microprocesseur, et le programme réclamera un effort particulier de documentation.

Tableau 1.1. — Conversion hexadécimale

Chiffre (*) hexadécimal	Equivalent binaire	Equivalent décimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

MNÉMONIQUES DES CODES D'INSTRUCTIONS

Un progrès évident consiste à affecter un nom à chaque code d'instruction. Ce nom est appelé un mnémonique. Ce mnémonique doit évoquer ce que l'instruction est censée faire.

PROBLÈMES AVEC LES MNÉMONIQUES

En fait, chaque fabricant de microprocesseur (ils sont incapables, eux aussi, de se souvenir des codes hexadécimaux) fournit un jeu de mnémoniques correspondant au jeu d'instructions. **Vous n'êtes pas obligé de subir les mnémoniques du fabricant** ; ils ne sont pas sacrés. Néanmoins, ils sont standards pour un microprocesseur donné et de ce fait, compris par tous les utilisateurs. Ce sont les noms d'instructions que vous trouverez dans les manuels, les cartes, les livres, les articles et les programmes. Lors de la sélection des codes mnémoniques, le problème consiste à trouver des noms évidents. C'est le cas pour quelques instructions (par exemple, ADD, AND, OR) ; d'autres sont des contractions de l'anglais (par exemple, XOR pour OR exclusif) ; certains mnémoniques, enfin, ne présentent guère de lien avec leur instruction. C'est le cas de DJNZ, EXX, et même OTDR (essayez, si vous parlez couramment l'anglais, de deviner ce qu'ils signifient !). La plupart des fabricants s'en sortent avec quelques noms raisonnables, et d'autres sans espoir. On constate cependant que les utilisateurs qui préfèrent leurs propres mnémoniques font rarement mieux que les fabricants.
En plus des mnémoniques d'instructions, le fabricant attribue généralement des noms aux registres de l'unité centrale de traitement, ou UCT en abrégé. De même qu'avec les

(*) Chiffre : l'hexadécimal fait intervenir des chiffres, puis des lettres, considérées alors comme des chiffres. On utilise fréquemment le mot « digit », contraction de l'anglais « unité digitale » *(digital unit)* pour désigner toutes sortes de valeurs numériques, quelle que soit la base de la numération. Dans ce livre, on adoptera indifféremment les mots *chiffres* ou *digits*.

mnémoniques d'instructions, certains noms de registres sont évidents (A pour Accumulateur, par exemple), tandis que d'autres n'ont qu'une signification historique. A nouveau, on utilisera les suggestions du fabricant simplement dans le but de promouvoir la standardisation.
Si nous utilisons les instructions standards du Z80 et les mnémoniques des registres tels qu'ils ont été définis par ZILOG, notre programme d'addition devient :

```
LD      A,(60H)
LD      B,A
LD      A,(61H)
ADD     A,B
LD      (62H),A
```

PROGRAMME EN LANGAGE ASSEMBLEUR

Ce programme est encore loin d'être évident mais au moins et pour partie, il est compréhensible. ADD A,B est un progrès considérable sur 80 ; LD suggère qu'on charge la donnée dans un registre ou dans une position mémoire (LD = « load », en anglais : charger). **Un tel programme est un programme en langage assembleur.**

LE PROGRAMME ASSEMBLEUR

ASSEMBLAGE MANUEL

Comment introduire un programme en langage assembleur dans l'ordinateur ? Il nous faut le traduire, soit en hexadécimal, soit en binaire. **Vous pouvez traduire manuellement un programme en langage assembleur,** instruction par instruction ; c'est ce qu'on appelle l'assemblage manuel.

L'assemblage manuel d'une séquence de trois instructions peut être illustré ainsi :

Nom de l'instruction		Equivalent hexadécimal
LD	A,(NN)	3A
LD	B,A	47
ADD	A,B	80
LD	(NN),A	32

Comme dans le cas de la conversion hexadécimale à binaire, l'assemblage manuel est une tâche mécanique, dénuée d'intérêt, répétitive, et sujette à de nombreuses erreurs mineures. Une mauvaise ligne, l'inversion de digits, l'omission d'instructions, des erreurs de lecture des codes, sont quelques-unes des erreurs que vous pouvez commettre. La plupart des microprocesseurs compliquent encore la tâche par le fait qu'ils disposent d'instructions de diverses longueurs. Quelques-unes n'occupent qu'un mot, tandis que d'autres en demandent deux ou trois. Certaines instructions imposent des données dans les second et troisième mots ; d'autres exigent des adresses-mémoires, des références à des registres ou d'autres informations.

ASSEMBLEUR

L'assemblage est ainsi une autre tâche mécanique qu'on peut assigner au micro-ordinateur. Ce dernier ne commet

PROGRAMME SOURCE

PROGRAMME OBJET

jamais d'erreur lorsqu'il traduit des codes ; il sait en permanence combien de mots chaque instruction exige et quel est leur format. Le programme qui exécute cette fonction est l'assembleur. Le programme assembleur traduit le programme utilisateur, ou programme source rédigé à l'aide des mnémoniques, en un programme en langage machine, ou programme-objet*, que le micro-ordinateur peut exécuter. L'entrée, pour l'assembleur, est le programme-source ; sa sortie est le programme-objet. Les compromis dont nous avons discuté à propos du chargeur hexadécimal sont amplifiés dans le cas de l'assembleur. Les assembleurs sont plus chers, occupent davantage de mémoire, exigent plus de périphériques et plus de temps d'exécution que les chargeurs hexadécimaux. Bien que les utilisateurs puissent écrire leur propre chargeur (et souvent le font), peu se préoccupent de rédiger leur propre assembleur.

Les assembleurs disposent de leurs propres règles et vous devez les apprendre. Ces règles comprennent l'utilisation de certains délimiteurs (tels qu'espaces, virgules, points-virgules ou des points), placés à bon escient, le respect des séquences de lettres, le recours à des commandes propres et, ce qui est peut-être le plus important, une disposition correcte des noms et des nombres. Ces règles sont généralement simples et vite apprises.

AUTRES CARACTÉRISTIQUES DES ASSEMBLEURS

Les premiers programmes assembleurs se bornaient surtout à traduire les mnémoniques des instructions et des registres en leurs équivalents binaires. Cependant, la plupart des assembleurs actuels possèdent des caractéristiques supplémentaires telles que celles décrites ci-dessous :

1) Ils offrent à l'utilisateur la possibilité d'assigner des noms à des positions-mémoires, des organes périphériques d'entrée et de sortie, et même à certaines séquences d'instructions.

2) Ils peuvent convertir des données ou des adresses à partir de nombreux systèmes de numération (par exemple, décimal ou hexadécimal) vers le binaire, et aussi des caractères alphanumériques (ou autres) dans leurs codes ASCII ou EBCDIC équivalents.

3) Ils sont en mesure d'exécuter quelques opérations arithmétiques devant intervenir dans le processus d'assemblage.

4) Ils indiquent au programme chargeur (ou éditeur de liens, voir chapitre 2) où loger en mémoire le programme et les données.

5) L'utilisateur peut réserver des zones en mémoire où il stockera temporairement des données, et placer des données fixes dans certaines zones de la mémoire de programme.

---

* NDT : En fait, on appelle programme objet (ou code objet) le code généré par tout traducteur d'un langage dans un autre. Ainsi, un « assembleur » produit un code objet qui, dans le cas d'ordinateurs quelque peu évolués, devra être traité par un « éditeur de liens » (voir ces mots plus loin) avant de pouvoir être exécuté, et sera donc différent du code exécutable.

6) Des informations spéciales peuvent être incluses afin d'introduire dans le programme courant des programmes standards issus de bibliothèques de programmes, ou des programmes rédigés antérieurement.

7) Ils offrent à l'utilisateur le contrôle du format du listage du programme, ainsi que celui des organes périphériques d'entrée et de sortie utilisés.

LE CHOIX D'UN ASSEMBLEUR Toutes ces caractéristiques, bien sûr, imposent un coût et des besoins en mémoire supplémentaires. Les micro-ordinateurs, en général, disposent d'assembleurs bien plus simples que les gros ordinateurs, mais la tendance va toujours vers l'accroissement de la taille des assembleurs. Vous aurez souvent le choix entre plusieurs assembleurs. Le critère important n'est pas tant le nombre de caractéristiques exceptionnelles offertes que le confort d'emploi dans les applications courantes.

INCONVÉNIENTS DU LANGAGE ASSEMBLEUR

Tout comme le chargeur hexadécimal, l'assembleur ne résoud pas tous les problèmes de programmation. L'un de ces problèmes est l'énorme fossé qui existe entre le jeu d'instructions du microprocesseur et les tâches que le micro-ordinateur doit accomplir. Les instructions de l'ordinateur tendent à exécuter des fonctions telles l'addition du contenu de deux registres, le décalage de un bit du contenu de l'accumulateur, ou le chargement d'une nouvelle valeur dans le compteur ordinal. En revanche, l'utilisateur attend de son micro-ordinateur qu'il fasse des choses telles que tester si une valeur analogique lue excède le seuil fixé, surveiller et réagir à une commande particulière d'un télé-imprimeur, ou encore activer un relais en temps utile. Le programmeur utilisant l'assembleur doit traduire ces tâches en des séquences comportant les simples instructions de la machine. Ce travail peut être difficile et long.

De plus, si vous programmez en assembleur, vous devez connaître de façon détaillée les particularités du microprocesseur que vous employez. Vous devez savoir de quels registres et de quelles instructions il dispose, de façon précise comment les instructions affectent les divers registres, quels sont les modes d'adressage disponibles, et connaître une myriade d'autres informations. Aucune de ces informations n'a un rapport quelconque avec la tâche que le micro-ordinateur a à exécuter.

PORTABILITÉ **De plus, les programmes en assembleur ne sont pas portables.** Chaque microprocesseur dispose de son propre langage assembleur qui reflète son architecture. Un programme en assembleur rédigé pour le Z80 ne tournera pas sur le 6800 de Motorola, le F8 de Fairchild ou le Pace de National Semiconducteur. Par exemple, le même programme d'addition écrit pour le 6800 de Motorola serait :

```
LDAA    $60
ADDA    $61
STAA    $62
```

L'absence de portabilité ne signifie pas seulement que vous ne pourrez pas utiliser votre programme en assembleur sur un autre micro-ordinateur mais, de plus, que vous ne serez pas en mesure d'exploiter tout autre programme qui n'aurait pas été écrit spécifiquement

pour le micro-ordinateur que vous employez. C'est là un handicap certain pour les micro-ordinateurs en raison du fait que ces machines sont récentes et qu'il n'existe encore que peu de programmes disponibles. Il en résulte trop fréquemment que vous dépendez totalement de vous-même. Si vous recherchez un programme accomplissant telle fonction particulière, le plus probable est que vous ne le trouverez pas dans les bibliothèques de programmes, des plus réduites, que la plupart des fabricants vous proposent. Vous ne le trouverez probablement pas non plus dans les archives, les publications ou toute autre source de programmes. Par conséquent, vous serez certainement obligé de l'écrire vous-même.

LANGAGES HAUT NIVEAU

COMPILATEUR **La solution aux nombreuses difficultés associées à l'emploi d'un langage assembleur réside dans l'usage d'un langage dit « de haut niveau », ou encore « langage orienté procédure ». Un tel langage permet la description des tâches d'une façon orientée vers le problème plutôt qu'orienté vers l'ordinateur. Chaque instruction d'un programme en langage haut niveau exécute une fonction identifiable ; elle correspond généralement à plusieurs instructions en langage assembleur. Un programme, appelé compilateur, traduit les instructions sources, rédigées en langage haut niveau, en code objet ou en instructions de langage machine.**

FORTRAN Il existe de nombreux langages haut niveau. Si, par exemple, vous pouvez exprimer votre problème en notation algébrique, vous pourrez écrire votre programme en FORTRAN (« FORmula TRANSlation language »), le plus ancien et le plus largement répandu des langages de haut niveau. Ainsi, si vous devez additionner deux nombres, dites simplement à votre ordinateur.

```
SOMME = NOMB 1 + NOMB 2
```

C'est bien plus simple (et plus court) qu'en langage machine ou qu'en assembleur. Les autres langages haut niveau incluent le COBOL (pour les applications à la gestion), l'ALGOL et le PASCAL (autre langage mathématique), PL/1 (une combinaison des FORTRAN, ALGOL et COBOL), ainsi que APL, BASIC, C, ADA, etc.

AVANTAGES DES LANGAGES HAUT NIVEAU

Manifestement, les langages haut niveau rendent les programmes plus faciles et plus rapides à écrire. Une estimation courante montre qu'un programmeur rédige un programme environ dix fois plus vite en langage haut niveau qu'en assembleur. Cela ne concerne que la rédaction du programme et ne comprend pas la définition du problème, la conception du programme, la mise au point, le test ou la documentation, tous points qui deviennent eux aussi plus simples et plus rapides. Par exemple, un programme rédigé en langage haut niveau est partiellement auto-documenté : même si vous ne connaissez pas le FORTRAN, vous pouvez probablement dire ce qu'exécute l'instruction donnée ci-dessus.

INDÉPENDANCE À L'ÉGARD DE LA MACHINE **Les langages de haut niveau résolvent bien d'autres problèmes surgissant lorsqu'on emploie le langage assembleur.** Le langage haut niveau dispose de sa propre syntaxe (généralement définie par des normes, nationales ou internationales).

Il ne se réfère pas au jeu d'instructions, aux registres, ou aux autres caractéristiques d'un microprocesseur particulier. C'est le compilateur qui prend en charge ces détails. Les programmeurs peuvent alors se concentrer sur leur propre travail ; ils n'ont pas besoin d'une connaissance détaillée de l'architecture sous-jacente de l'UCT (unité centrale de traitement) — et en fait, ils n'éprouvent même pas le besoin de savoir quoi que ce soit sur l'ordinateur qu'ils programment.

PORTABILITÉ DES LANGAGES HAUT NIVEAU

Les programmes rédigés en langage haut niveau sont portables — tout au moins en théorie. Ils tourneront sur n'importe quel ordinateur disposant du compilateur standard prévu pour ce langage.

De ce fait, tous les programmes antérieurs écrits en langage haut niveau pour d'autres ordinateurs précédents deviennent disponibles lorsque vous programmez sur un nouvel ordinateur. Cela peut signifier que vous aurez à votre disposition des milliers de programmes si vous choisissez un langage aussi répandu que FORTRAN ou BASIC.

INCONVÉNIENTS DES LANGAGES HAUT NIVEAU

Si tout le bien que nous venons de dire des langages haut niveau est vrai, si vous pouvez écrire des programmes plus vite et s'ils sont portables, pourquoi se préoccuper encore des assembleurs ? Qui souhaite réellement se compliquer l'existence avec des registres, des codes d'instructions, des mnémoniques et tout le reste ! C'est qu'en réalité, un certain nombre d'inconvénients viennent contrebalancer ces avantages.

SYNTAXE DES LANGAGES HAUT NIVEAU

Le premier problème évident consiste à **apprendre les « règles » ou la syntaxe de n'importe quel langage haut niveau** que vous souhaitez employer. Un langage haut niveau dispose d'un ensemble de règles passablement compliquées. Vous trouverez qu'il vous faut déjà un temps très appréciable pour simplement aboutir à un programme dont la syntaxe est correcte (et qui probablement, même dans ce cas, n'exécutera pas encore ce que vous aviez prévu). Un langage haut niveau pour ordinateur est comme une langue étrangère. Si vous avez quelque talent, vous maîtriserez ses règles et serez capable de préparer des programmes que le complitateur acceptera. Cependant, l'acquisition de ces règles et la tentative de faire accepter le programme par le compilateur ne suffisent pas pour que votre travail soit exécuté.

Voici, par exemple, quelques règles applicables au FORTRAN :

— Les *étiquettes* (ou *labels*)* sont uniquement des nombres et doivent être placés dans les cinq premières colonnes des cartes.

— Les instructions doivent commencer à la septième colonne.

— Les variables de type entier doivent commencer par les lettres I, J, K, L, M ou N.

LE PRIX DU COMPILATEUR

Un autre problème réside dans le fait qu'il faut acquérir un compilateur afin de traduire le programme, rédigé en langage haut niveau, en programme en langage machine. Les compilateurs sont onéreux et exigent beaucoup de mémoire. Alors que la plupart des

(*) Etiquettes et labels sont ici synonymes. On utilisera indifféremment ces deux mots.

assembleurs se contentent de 2 à 16 K octets de mémoire (1 K = 1024), les compilateurs en exigent généralement bien davantage.

NOTATION ALGÉBRIQUE

D'autre part, **seuls quelques compilateurs seront en mesure de simplifier l'implémentation de votre tâche** (sa réalisation). FORTRAN, par exemple, est bien adapté aux problèmes pouvant se ramener à des formules mathématiques. Mais si vous avez à piloter une imprimante, éditer une chaîne de caractères ou surveiller un système d'alarme, votre problème se laissera mal réduire à une suite d'équations. En fait, formuler la solution en notation mathématique serait sans doute plus astreignant et plus difficile que de traiter ce problème en langage assembleur. La solution d'un tel problème peut être l'utilisation d'un langage évolué mieux adapté. Il en existe, mais ils sont bien moins répandus et standardisés que FORTRAN. De plus, leur spécialisation leur fait souvent perdre les avantages inhérents aux langages de haut niveau classiques.

INEFFICACITÉ DES LANGAGES ÉVOLUÉS

Le plus souvent, les langages évolués ne produisent pas des programmes très efficaces en langage machine. La raison fondamentale en est que la compilation est un processus automatique criblé de compromis, afin d'aboutir au maximum de possibilités. Le compilateur travaille de façon fort semblable à celle d'un traducteur informatique de langues... parfois, les mots sont corrects mais les sons et les phrases ne le sont pas. Un simple compilateur ne peut savoir si une variable ne doit plus être utilisée et peut être abandonnée, si le passage par un registre est plus intéressant que le passage par une position mémoire. Le programmeur expérimenté peut profiter de « raccourcis » afin de réduire les temps d'exécution ou l'occupation mémoire. Quelques compilateurs, peu nombreux (et connus sous le nom de compilateurs optimisés), sont aussi capables de le faire mais ils sont bien plus gros et bien plus lents que les compilateurs courants.

OPTIMISATION DU COMPILATEUR

Les avantages et les inconvénients généraux des langages de haut niveau sont :

Avantages :

AVANTAGES DES LANGAGES DE HAUT NIVEAU

— Meilleure description des tâches.
— Meilleure productivité du programmeur.
— Documentation plus aisée.
— Syntaxe universelle.
— Indépendance à l'égard de la structure d'un ordinateur particulier.
— Portabilité.
— Disponibilité de bibliothèque et autres programmes.

Inconvénients :

INCONVÉNIENTS DES LANGAGES DE HAUT NIVEAU

— Règles spéciales.
— Exigent un support logiciel et matériel conséquent.
— Orientation mathématique ou gestion des plus communs d'entre eux.
— Programmation non optimale.

— Code difficile à optimiser pour atteindre des objectifs de temps d'exécution et d'occupation mémoire.
— Inaptitude à utiliser avec avantage les caractéristiques particulières à un ordinateur.

LANGAGES HAUT NIVEAU POUR MICROPROCESSEURS

Plusieurs difficultés particulières attendent les utilisateurs de microprocesseurs, travaillant avec des langages haut niveau. Parmi elles :

— Peu de langages haut niveau existent pour les microprocesseurs.
— Peu de langages standards sont largement disponibles.
— Les compilateurs exigent généralement de très larges volumes de mémoire ou même parfois, un ordinateur différent.
— La plupart des applications des microprocesseurs ne sont pas bien adaptées aux langages haut niveau.
— Le prix des mémoires est souvent critique dans les applications des microprocesseurs.

Le manque de langages haut niveau résulte partiellement du fait que les microprocesseurs sont récents et sont des produits de fabricants de semiconducteurs plutôt que de fabricants d'informatique. Très peu de langages haut niveau existent avec les microprocesseurs ; les plus usuels sont les variantes du PL/1 telles que les PL/M d'Intel, MPL de Motorola et PL/Z de ZILOG ainsi que les langages PASCAL, C et surtout BASIC.
La plupart des langages haut niveau existant ne sont d'ailleurs même pas conformes à des standards reconnus et de ce fait, l'utilisateur ne peut guère espérer gagner en portabilité, en accès aux bibliothèques, ou pouvoir utiliser une expérience antérieure en programmation. Le principal avantage qui subsiste consiste dans la réduction de l'effort de programmation et un moindre besoin de connaissance de l'architecture interne de l'ordinateur.

EXIGENCES DES LANGAGES HAUT NIVEAU

Les exigences supplémentaires des langages haut niveau sont considérables pour les microprocesseurs. En effet, les microprocesseurs sont, par essence, mieux adaptés aux commandes et aux applications interactives assez lentes qu'aux traitements de caractères et à l'analyse de langage qu'implique la compilation. Aussi, la plupart des compilateurs pour microprocesseurs ne tourneront pas sur un système à microprocesseur. Ils exigeront un ordinateur bien plus puissant ; aussi existent-ils sous forme de cross-compilateurs plutôt que de compilateurs résidants. L'utilisateur doit non seulement faire face aux dépenses d'un gros ordinateur mais il doit, de plus, transférer physiquement le programme du gros ordinateur vers l'ordinateur à microprocesseur.
Peu de compilateurs résidants sont disponibles. Ils tournent sur des micro-ordinateurs et produisent le code objet. Malheureusement, il leur faut disposer d'un important volume mémoire (16 k ou davantage), en plus des supports logiciel et matériel spécifiques.

INADÉQUATION DES LANGAGES HAUT NIVEAU

De plus, les langages haut niveau ne sont généralement guère adaptés aux applications des microprocesseurs. En effet, la plupart des langages haut niveau ont été étudiés pour résoudre des problèmes scientifiques ou pour traiter des masses de données dans les problèmes de gestion. Or, peu d'applications des microprocesseurs relè-

vent de ces domaines. La plupart des applications des microprocesseurs impliquent des échanges d'informations, données et commandes, avec des dispositifs périphériques, en entrées et en sorties. Souvent, les informations de commandes et d'états se traduiront par quelques chiffres binaires dont la signification est étroitement liée au matériel. Si vous tentez de rédiger un programme typique de contrôle en langage haut niveau, vous vous sentirez souvent dans la position de celui qui veut manger une soupe avec des baguettes. Pour des applications aux équipements de test, aux terminaux, aux systèmes de navigation et aux équipements de gestion, les langages haut niveau conviennent mieux qu'aux applications à l'instrumentation, aux communications, aux périphériques et à l'automobile.

DOMAINES D'APPLICATION DES LANGAGES HAUT NIVEAU

Les applications préférentielles des langages haut niveau sont ainsi celles qui requièrent des mémoires importantes. Si le coût d'un simple circuit mémoire compte, comme c'est le cas lors d'une commande de valve, pour un jeu électronique, une application de commande ou un petit instrument, l'inefficacité des langages haut niveau devient intolérable. Si, d'autre part, le système dispose de plusieurs milliers d'octets de mémoire, ce qui sera le cas d'un terminal ou d'un équipement de test, l'inefficacité du langage haut niveau sera moins importante. Ainsi, la taille du programme et le volume du produit constituent manifestement d'autres facteurs importants. Un programme long accroîtra les avantages des langages haut niveau. Par ailleurs, une application de grande série réduira l'importance des coûts fixes de développement du logiciel par rapport au coût des mémoires pour chaque système.

QUEL NIVEAU UTILISER ?

Tout dépend ici de votre application particulière. Résumons succinctement quelques-uns des facteurs favorisant les divers niveaux :

APPLICATIONS POUR LANGAGE MACHINE

Langage machine

Pratiquement, personne ne programme plus en langage machine. Son usage ne peut se justifier eu égard au bas prix d'un assembleur.

APPLICATIONS POUR LANGAGE ASSEMBLEUR

Langage assembleur

- **Petits à moyens programmes.**
- **Applications pour lesquelles le coût des mémoires intervient.**
- **Commandes en temps réel.**
- **Traitement limité de données.**
- **Applications de grandes séries.**
- **Applications impliquant davantage d'entrées-sorties que de calculs.**

APPLICATIONS POUR LANGAGES HAUT NIVEAU

Langages haut niveau

- **Programmes importants.**
- **Applications de faibles séries demandant des programmes longs.**
- **Applications demandant beaucoup de mémoire.**

— Ou exigeant davantage de calculs que d'entrées-sorties ou de commandes.
— Lorsqu'une compatibilité est recherchée avec des applications similaires sur gros ordinateurs.
— Lorsque des programmes spécifiques sont disponibles en langage haut niveau et peuvent être utilisés.

Bien d'autres facteurs sont également importants : par exemple, la disponibilité d'un gros ordinateur utilisable en développement, l'expérience de tel langage, ou la compatibilité avec d'autres applications.
En dernière analyse, si c'est le matériel qui intervient le plus dans le coût de votre application, ou si la vitesse est critique, vous devez favoriser le langage d'assemblage. Mais préparez-vous alors à investir davantage de temps dans le développement du logiciel en échange d'une économie sur les mémoires et d'une vitesse d'exécution accrue. Si c'est le logiciel qui représente le coût le plus important, vous devez favoriser le langage haut niveau. Mais apprêtez-vous à dépenser davantage pour le matériel et le logiciel de support.
Bien sûr, personne — sauf quelques théoriciens — n'objectera à l'usage simultané de l'assembleur et du langage haut niveau. Vous pouvez rédiger le programme original en langage haut niveau et ensuite développer certaines sections en assembleur. La plupart des utilisateurs préfèrent cependant éviter cette solution en raison des soucis qu'elle fait naître lors de la mise au point, les tests et la documentation.

ET POUR LE FUTUR ?

TENDANCES DES LANGAGES

Nous attendons du futur qu'il tende à favoriser les langages haut niveau pour les raisons suivantes :

— Les programmes semblent toujours accumuler davantage de fonctions et se développer en longueur.
— Le matériel et les mémoires voient leur prix décroître.
— Le logiciel et les programmeurs coûtent de plus en plus chers.
— Les circuits mémoires deviennent disponibles en densités accrues à un prix « par bit » plus bas ; de ce fait, l'économie en mémoire risque d'être moins évidente.
— Des compilateurs de plus en plus nombreux deviennent disponibles.
— Des langages haut niveau mieux adaptés et plus efficaces sont développés.
— Une standardisation plus poussée des langages haut niveau devrait intervenir.

L'assembleur ne deviendra pas pour autant un langage en voie de disparition pour les microprocesseurs, pas plus qu'il ne l'est pour les gros ordinateurs. Mais des programmes plus longs, des mémoires moins chères, et des programmeurs plus coûteux vont attribuer aux coûts logiciels la part la plus grande dans la plupart des applications. Le choix se portera alors sur les langages haut niveau.

POURQUOI CE LIVRE ?

Si l'avenir est aux langages haut niveau, pourquoi un livre sur le langage de programmation assembleur ? En voici les raisons :

1) La plupart des utilisateurs des microprocesseurs courants programment en assembleur (environ les 2/3, si l'on se réfère à une étude récente).

2) De nombreux utilisateurs de micro-ordinateurs continueront à programmer en assembleur parce qu'ils ont besoin des commandes détaillées qu'il procure.

3) Aucun langage haut niveau adapté n'est devenu largement disponible ou standardisé.

4) De nombreuses applications requièrent l'efficacité de l'assembleur.

5) Une bonne compréhension de l'assembleur peut aider à comprendre et utiliser les langages haut niveau.

Le reste de ce livre traitera exclusivement des assembleurs et du langage assembleur de programmation. Cependant, nous souhaitions que les lecteurs sachent que l'assembleur n'est pas la seule solution. Vous devriez être vigilants et envisager de nouveaux développements qui pourraient réduire les coûts de programmation de façon significative, si ces coûts constituent un facteur majeur dans votre application.

RÉFÉRENCES

Quelques comparaisons générales relatives aux temps de programmation de différents types de programmes à des niveaux de langage différents sont données par M. H. Halstead dans *Elements of Software Science,* American Elsevier, New York, 1977, et par V. Schneider, « Prediction of Software Effort and Project Duration - Four New Formulas ». *SIGPLAN Notices* juin 1978, pp. 49-55.

CHAPITRE 2

LES ASSEMBLEURS

Ce chapitre traite des fonctions exécutées par les assembleurs, en commençant par des caractéristiques communes à la majorité des assembleurs et en poursuivant par des possibilités plus élaborées telles que les macros et l'assemblage conditionnel. Vous pouvez sauter ce chapitre maintenant, afin d'y revenir lorsque vous vous sentirez plus à l'aise avec ce thème.

CARACTÉRISTIQUES DES ASSEMBLEURS

Ainsi qu'on l'a mentionné antérieurement, les assembleurs actuels font beaucoup plus que traduire les mnémoniques du langage d'assemblage en codes binaires. Nous allons décrire comment un assembleur exécute cette traduction avant d'examiner ses autres caractéristiques. Pour finir, nous expliquerons comment les assembleurs sont utilisés.

INSTRUCTIONS DE L'ASSEMBLEUR

« CHAMPS » DE L'ASSEMBLEUR

Les instructions du langage assembleur sont divisées en un certain nombre de colonnes (ou champs), comme le montre le tableau 2.1.

Le champ du code opératoire est le seul champ qui ne peut jamais être vide ; il contient toujours soit le mnémonique d'une instruction, soit une directive à l'assembleur, appelée une pseudo-instruction ou pseudo-opération.

Le champ opérandes ou adresses peut contenir une adresse ou une donnée, ou peut être vide.

Tableau 2.1. — Les champs des instructions du langage assembleur.

Champ label	Champ code opératoire (ou mnémonique)	Champ opérande ou adresse	Champ commentaire
DEBUT :	LD	A,VAL1	; CHARGER LE 1er NOMBRE DANS A
	LDV	B,A	; LE TRANSFERER DANS B
	LD	A,VAL2	; CHARGER LE 2e NOMBRE DANS A
	ADD	A,B	; LUI AJOUTER LE PREMIER
	LD	(SOMME),A	; RANGER LE RESULTAT
SUITE :	?	?	; INSTRUCTION SUIVANTE
VAL1 :	DEFS	1	
VAL2 :	DEFS	1	
SOMME :	DEFS	1	

FORMAT **Naturellement, l'assembleur doit offrir le moyen d'indiquer la fin d'un champ et le début du suivant.** Les assembleurs utilisant des cartes perforées demandent souvent qu'un champ débute dans une colonne spécifique de la carte. C'est un format fixe. Cependant, le format fixe n'est pas pratique et gêne les programmeurs. L'alternative réside dans un format libre, les champs pouvant apparaître n'importe où sur la ligne.

SEPARATEURS (DÉLIMITEURS) Si l'assembleur ne peut pas se référer à une position fixe sur la ligne pour définir un champ, il doit utiliser autre chose. **La plupart des assembleurs recourent à des symboles spéciaux, appelés séparateurs, ou délimiteurs, au début ou à la fin de chaque champ.** Le délimiteur le plus commun est un espace. Les virgules, points-virgules, deux points, barres obliques, points d'interrogation et autres caractères qui, autrement, ne trouveraient pas d'emploi dans les langages assembleurs, peuvent aussi servir de délimiteurs. Le tableau 2.2 donne la liste des délimiteurs standards de l'assembleur Zilog Z80.

Tableau 2.2. — Séparateurs usuels du Z80

:	après un label (une étiquette).
« espace »	entre le code opératoire et l'adresse.
,	entre des opérandes dans le champ adresses.
;	avant un commentaire

Il vous faudra prendre quelques précautions avec les séparateurs. Certains assembleurs se révèlent sensibles à la présence d'espaces supplémentaires ou à l'apparition de délimiteurs dans les commentaires ou labels. Un assembleur bien conçu se jouera de ces problèmes mineurs, mais la plupart des assembleurs ne sont pas parfaitement rédigés. Notre recommandation est alors simple : évitez les problèmes potentiels dans la mesure du possible. Les règles suivantes vous aideront :

1) N'employez pas d'espaces supplémentaires en frappant vos textes, en particulier après les virgules séparant les opérandes.
2) N'utilisez pas de caractères de délimitation dans les noms ou les labels.
3) Utilisez les séparateurs usuels même si votre assembleur ne les demande pas. Votre programme sera alors acceptable par n'importe quel assembleur.

LABELS

CHAMP LABEL **Le champ « label » est le premier champ d'une instruction en langage assembleur** ; il peut être vide. Si un label est présent, l'assembleur lui attribue la valeur de l'adresse mémoire où le premier octet du programme objet résultant de cette instruction sera logé. Vous pourrez ainsi employer ce label comme une adresse ou comme une donnée dans le champ adresse d'une autre instruction. L'assembleur remplacera le label par la valeur qui lui a été assignée, et ce au moment de la création du programme objet.

LABELS DANS LES INSTRUCTIONS DE BRANCHEMENT **Le plus souvent, les labels sont utilisés dans les instructions de branchements** (JUMP) et d'appel (CALL). Ces instructions placent une nouvelle valeur dans le compteur ordinal et altèrent ainsi le déroulement séquentiel normal des

instructions. Ainsi, JUMP 150_{16} signifie « placer la valeur 150_{16} dans le compteur ordinal ». La prochaine instruction à être exécutée sera celle se trouvant en mémoire à l'adresse 150_{16}. L'instruction JUMP DEBUT signifie « placer la valeur attribuée au label DEBUT dans le compteur ordinal ». L'instruction suivante qui sera exécutée sera celle se trouvant en mémoire à l'adresse correspondant au label DEBUT. Le tableau 2.3 en propose un exemple.

Tableau 2.3. — Utilisation d'un label

PROGRAMME EN ASSEMBLEUR	
DEBUT	LOAD ACCUMULATOR 100 • (Charger 100 dans l'accumulateur) • • (PROGRAMME PRINCIPAL) • • • JUMP DEBUT
Lorsque la version en langage machine de ce programme est exécutée, l'instruction JUMP DEBUT provoque le chargement de l'adresse de l'instruction étiquetée DEBUT dans le compteur ordinal. Cette instruction est ensuite exécutée.	

Pourquoi utiliser un label ? En voici quelques raisons :

1) Un label permet un repérage plus aisé à l'intérieur du programme.

2) Un label peut être déplacé pour corriger un programme. Vous n'avez à modifier aucune des instructions qui l'utilisent ; l'assembleur exécutera toutes les modifications nécessaires.

CONSTANTE DE TRANSLATION

3) L'assembleur ou le chargeur peut translater le programme entier en ajoutant une constante (une constante de translation, ou encore de « relocation » selon la terminologie américaine) à chaque adresse à laquelle un label a été assigné. Ainsi, on peut translater un programme afin de permettre son insertion dans d'autres programmes, ou simplement de ré-organiser la mémoire.

4) Le programme se prête mieux à un archivage dans une bibliothèque de programmes ; par exemple, il est plus facile à une tierce personne de prendre votre programme et de l'ajouter à des programmes totalement différents.

5) Vous n'avez pas à calculer les adresses mémoires. Le calcul des adresses mémoires est particulièrement difficile avec les microprocesseurs disposant d'instructions de longueurs variables.

Il est opportun d'attribuer un label à une instruction à laquelle vous souhaitez vous référer ultérieurement.

CHOIX DU LABEL **La question suivante est : quel label utiliser ? L'assembleur apporte généralement quelques restrictions** sur le nombre de caractères (usuellement, 5 ou 6), le caractère de tête (qui doit souvent être une lettre), et les caractères suivants (qui doivent souvent être des lettres, des chiffres, ou l'un des quelques caractères spéciaux). Dans le cadre de ces impératifs, le choix reste vôtre.
Notre propre préférence va vers des labels suggestifs, par exemple, des mnémoniques évoquant leur usage. Des exemples typiques sont TEMPS pour temporisation, ADDM pour l'addition de 2 mots, NCLES pour un sous-programme contenant le nombre de clés, ou de touches frappées. On se souvient plus facilement d'un label significatif et il contribue à documenter le programme. Certains programmeurs préfèrent l'utilisation de formats standards et par exemple commençant avec L0000. Ces labels sont auto-séquencés (on peut sauter quelques nombres afin de réserver la place pour d'autres insertions), mais ils ne contribuent pas à documenter le programme.

RÈGLES D'« ÉTIQUETAGE » **Quelques règles de sélection des labels préviendront toute difficulté. Nous vous recommandons les suivantes :**

1) N'utilisez pas de labels qui soient des noms de codes opératoires ou d'autres mnémoniques. La plupart des assembleurs ne vous y autoriseront pas ; d'autres les acceptent, mais c'est une source de confusion.

2) N'utilisez pas de labels plus longs que ce que l'assembleur permet. Les assembleurs appliquent diverses règles de coupure.

3) Évitez les caractères spéciaux (ni numériques, ni alphabétiques) ainsi que les lettres minuscules. Certains assembleurs les refuseront, d'autres en permettront quelques-uns. Une pratique simple recommande de s'en tenir aux lettres capitales et aux chiffres.

4) Débutez chaque label par une lettre ; ils seront toujours acceptables.

5) N'utilisez pas de labels qui peuvent être confondus avec d'autres labels. Evitez les lettres I, O, Z et les chiffres 0, 1 et 2. Evitez aussi des labels tels que XXXX et XXXXX car on ne les différencie pas aisément (rappelez-vous les fameuses « lois de Murphy »).

6) Si vous n'êtes pas certain qu'un label est autorisé, ne l'utilisez pas. Vous n'aurez aucun bénéfice réel à explorer ce que l'assembleur accepte exactement.
Il s'agit là de recommandations et non de règles. Vous n'êtes pas obligé de les suivre : mais ne nous blâmez pas ensuite si vous perdez votre temps à des questions stupides.

CODES OPERATOIRES (MNÉMONIQUES) DE L'ASSEMBLEUR

La principale tâche de l'assembleur consiste à traduire les mnémoniques dans leurs codes-binaires équivalents. L'assembleur exécute cette tâche en se servant d'un tableau fixe, très proche de votre façon de procéder si vous faisiez l'assemblage à la main.
L'assembleur doit, cependant, faire plus que simplement traduire en codes opératoires. Il **doit aussi, d'une manière ou d'une autre, déterminer combien l'instruction exige d'opérandes et de quels types ils sont.** Ce processus peut être assez complexe, certaines instruc-

tions (telles que « Halt ») ne disposant d'aucun opérande, d'autres (telles qu'une addition ou un JUMP) en ont un, alors que d'autres, enfin, en demandent deux. Certaines instructions peuvent même autoriser des alternatives ; ainsi, quelques instructions de micro-ordinateurs (telles que SHIFT ou CLEAR) peuvent s'appliquer à l'accumulateur ou à une position mémoire. Nous ne discuterons pas de la façon dont l'assembleur procède mais nous noterons simplement qu'il doit tenir compte de ces distinctions.

PSEUDO-OPÉRATIONS

PSEUDO-OPÉRATIONS

DIRECTIVES DE L'ASSEMBLEUR

Certaines instructions de l'assembleur ne sont pas directement converties en instructions en langage machine. Il s'agit de directives pour l'assembleur ; elles attribuent certaines zones en mémoire au programme, définissent des symboles, désignent les portions de la RAM qui recevront des données transitoires, des tableaux, ou d'autres informations, et exécutent des fonctions d'intendance.

Pour utiliser ces directives, ou pseudo-opérations, le programmeur place leurs mnémoniques dans le champ du code opératoire et une adresse ou une donnée dans le champ adresses si la pseudo-opération visée le requiert.

Les pseudo-opérations les plus communes sont :

DATA (donnée)
EQUATE (équivalence), ou DEFINE
ORIGIN (origine)
RESERVE
ENTRY (point d'entrée)
EXTERNAL

Des assembleurs différents affecteront des noms différents à ces opérations, mais leurs fonctions restent les mêmes. Les pseudo-opérations d'intendance comprennent :

END (fin)
LIST
NAME (nom)
PAGE
SPACE (espace)
TITLE (titre)

Nous expliciterons brièvement ces pseudo-opérations, car leurs fonctions sont généralement évidentes.

PSEUDO-OPÉRATIONS SUR DONNÉES (DATA)

La pseudo-opération DATA permet au programmeur l'introduction de données fixes dans la mémoire de programme. Ces données peuvent inclure des :

— Tables de conversion d'informations.
— Tables de conversion de codes.
— Messages.
— Temps de fréquences caractéristiques.
— Adresses de sous-programmes.
— Identifications de clés.

— Séquences de synchronisation.
— Seuils.
— Noms.
— Coefficients d'équations.
— Commandes.
— Facteurs de conversion.
— Facteurs de pondération.
— Séquences de tests.
— Séquences de génération de caractères.
— Séquences d'identification.
— Tableaux de taxes.
— Structures standards.
— Réseaux de masques.
— Tables de transition d'états.

La pseudo-opération DATA traite les données comme des constantes permanentes du programme.
Le format d'une pseudo-opération DATA est généralement très simple. Une instruction telle que

```
VALDZ    DATA    12
```

placera le nombre 12 dans la prochaine cellule mémoire disponible et attribuera à cette position le nom VALDZ. Généralement, une pseudo-opération DATA dispose d'un label, sauf lorsqu'elle participe à une série de pseudo-opérations. Les données et les labels peuvent prendre toutes formes autorisées par l'assembleur.
La plupart des assembleurs autorisent des instructions DATA plus élaborées, traitant un grand nombre de données d'un seul coup, par exemple :

```
EMESS    DATA    'ERREUR'
CARRE    DATA    1, 4, 9, 16, 25
```

Une seule instruction peut remplir plusieurs cellules mémoires, la limite dépendant uniquement de la longueur de la ligne. Remarquez que, si vous ne pouvez loger toutes les données dans une ligne, vous pouvez toujours faire suivre une instruction DATA par une autre :

```
MESSG    DATA    'HEUREUX'
         DATA    'QUI COMME'
         DATA    'ULYSSE'
         DATA    'A FAIT UN'
         DATA    'LONG VOYAGE'
```

Le nom des pseudo-opérations DATA varient parfois. Ainsi, DEFINE BYTE (définition d'un « *byte* » ou « *octet* ») ou FORM CONSTANT BYTE ou plus simplement BYTE traitent des nombres sur un octet ; DEFINE WORD (« word » signifie **mot**) ou FORM CONSTANT WORD ou WORD traite des mots ou des adresses de 16 bits. D'autres pseudo-opérations particulières pourront traiter de données en caractères codés.

PSEUDO-OPÉRATION EQUATE (ou DEFINE)

DÉFINITIONS DE NOMS

La pseudo-opération EQUATE (pour « équivalence ») permet au programmeur d'attribuer des noms à des adresses ou des données. Cette pseudo-opération est presque toujours codée EQU en mnémonique. Les noms peuvent se rapporter à des adresses de périphéries, des données numériques, des adresses de départ, des adresses fixes, etc.
La pseudo-opération Equate attribue la valeur numérique de son champ opérande au

nom placé dans son champ label. Voici deux exemples :

```
TTY     EQU     5
FIN     EQU     5000
```

La plupart des assembleurs autorisent la définition d'un label à partir d'un autre :

```
DER     EQU     FIN
ST1     EQU     DEBUT + 1
```

Dans ce cas, le label du champ opérande doit, bien entendu, avoir été défini au préalable. Le champ opérande pourra contenir des expressions plus complexes, ainsi que nous le verrons. Des noms doubles (deux noms attribués à une même donnée ou adresse) peuvent être utiles lorsqu'on veut fondre en un plusieurs programmes qui, précisément, utilisent des noms différents (ou différentes orthographes de ce qui pourrait être le même nom).

TABLE DE SYMBOLES **Remarquez que la pseudo-opération EQU n'impose pas à l'assembleur de ranger quelque chose en mémoire. L'assembleur se borne à placer un nom supplémentaire dans une table (appelée table de symboles).** Cette table, à l'inverse de la table des mnémoniques, doit obligatoirement être en RAM puisqu'elle change avec les programmes. Le programme assembleur devra toujours disposer d'un espace en RAM pour la table des symboles ; plus il dispose d'espace et plus grand est le nombre de symboles qu'il peut accepter. Cette zone en RAM vient s'ajouter au reste de l'espace RAM nécessaire pour des rangements temporaires.

USAGE DE NOMS **Quand utiliser un nom ?** La réponse est : chaque fois que vous disposez d'un paramètre que vous pourriez souhaiter modifier ou qui possède une signification intrinsèque en dehors de sa valeur numérique ordinaire. Nous attribuons typiquement des noms à des constantes de temps, des adresses de dispositifs, des masques, des facteurs de conversion, etc. Des noms tels que DELAI, TTY, CLAVR ou DEBUT ne rendent pas seulement le paramètre plus aisément modifiable mais ajoutent encore de la documentation au programme. Nous attribuons aussi des noms à des emplacements en mémoire dont l'objectif est particulier ; ils peuvent stocker des données, marquer le démarrage d'un programme, ou servir à des stockages temporaires.

CHOIX DES NOMS **Quel nom utiliser ?** Les meilleures règles sont encore celles qui s'appliquent aux labels, si ce n'est qu'ici, la signification des noms est primordiale. Pourquoi ne pas appeler le téléimprimeur TTY plutôt que X15, une durée DELAI plutôt que WW et la touche GO de votre clavier CLEGO plutôt que CHEVAL ? Ce conseil semble aller de soi ; pourtant, beaucoup de programmeurs ne le suivent pas.

EMPLACEMENT DES DÉFINITIONS **Où placer les pseudo-opérations EQUATE ? La meilleure place est au tout début d'un programme,** après des commentaires appropriés d'en-tête tels que ADRESSES D'ENTREE/ SORTIE, ZONES DE MEMOIRE TEMPORAIRES, CONSTANTES DE TEMPS, ou TABLE DES ADRESSES. Ceci rendra les définitions faciles à retrouver si vous voulez les modifier. De plus, un autre utilisateur pourra prendre connaissance de

toutes les définitions en un seul bloc. A l'évidence, cette habitude améliore la documentation et rend le programme plus facile à utiliser.
Les définitions n'intervenant que dans des sous-programmes spécifiques devront être placées au début de ces sous-programmes.

PSEUDO-OPÉRATION ORIGINE

La pseudo-opération origine (presque toujours abrégée en ORG) permet au programmeur d'assembler ses programmes, sous-programmes ou données n'importe où en mémoire. Les programmes et les données peuvent être rangés dans diverses zones de la mémoire, en fonction de la configuration de celle-ci. Les sous-programmes d'initialisation, de service d'interruptions, ou autres, peuvent être répartis dans l'ensemble de la mémoire à des adresses pratiques.

COMPTEUR DE POSITION **L'assembleur tient à jour un compteur de positions (ou « compteur de location »), comparable au compteur ordinal du microprocesseur, qui contient la position en mémoire de la prochaine instruction ou donnée à traiter.** Une pseudo-opération ORG oblige l'assembleur à charger une nouvelle valeur dans ce compteur de positions, de façon fort semblable à celle dont une instruction de JUMP impose à l'UCT (unité centrale de traitement, ou CPU) de placer une nouvelle adresse dans le compteur ordinal. La sortie de l'assembleur ne contient pas seulement des instructions et des données, mais doit aussi indiquer au programme chargeur où il doit ranger ces instructions et ces données en mémoire.
Les programmes pour microprocesseurs contiennent souvent plusieurs instructions d'origine ORG, pour les applications suivantes :

Adresses de redémarrage, « reset », départ
Adresses de service des interruptions
Adresses de trappes
Mémoire de sauvegarde
Mémoire de pile
Programme principal
Sous-programmes
Adresses en mémoire réservées pour les dispositifs d'entrées-sorties ou des fonctions spéciales.

Enfin, d'autres instructions ORG pourront réserver des emplacements pour des insertions ultérieures, servir à placer des tables ou des données en mémoire, ou attribuer des emplacements libres en RAM à des buffers de données (cellules-tampons pour données). Avec les micro-ordinateurs, les programmes et les données en mémoire peuvent se voir affecter des adresses largement distribuées afin de simplifier le montage matériel des circuits.

Des instructions d'ORIGINE typiques sont :

```
ORG          RESET
ORG          1000
ORG          INT3
```

Certains assembleurs attribuent par défaut une origine zéro si le programmeur n'a pas introduit une pseudo-opération ORG au début du programme. L'intérêt est maigre ; nous conseillons plutôt d'inclure un ORG pour éviter tout confusion.

PSEUDO-OPÉRATION RESERVE

ATTRIBUTION DE MEMOIRE

La pseudo-opération RESERVE sert au programmeur à réserver des emplacements en RAM à diverses fins : tableaux, stockages temporaires, adressage indirect, pile, etc.

En utilisant la pseudo-opération RESERVE, vous attribuerez un nom à l'espace mémoire réservé et déclarerez le nombre de cellules nécessaires répondant à ce nom. Voici quelques exemples :

```
CLE       RESERVE  1
TEMP      RESERVE  50
VOLTS     RESERVE  80
BUFR      RESERVE  100
```

Vous pouvez utiliser la pseudo-opération RESERVE pour réserver de la mémoire dans la mémoire de programme ou de données ; cependant, elle se révèle plus significative lorsqu'elle porte sur la mémoire de données.
En réalité, tout ce que la pseudo-opération RESERVE fait consiste à ajouter au contenu du compteur de position le nombre de cellules déclarées dans le champ opérande. L'assembleur ne produira en effet aucun code objet.

Notez les caractéristiques suivantes de Reserve :

1) Le label de la pseudo-opération Reserve est attribué à la première adresse réservée. Par exemple, la séquence :

```
          ORG       3000
BUF1      RESERVE   100
BUF2      RESERVE   50
VOLTS     RESERVE   5
```

réserve 155 octets et attribue la valeur 3000 à BUF1, la valeur 3100 à BUF2, et 3150 à VOLTS.

2) On doit spécifier le nombre d'emplacements à réserver. Aucune valeur n'est prise par défaut.

3) Aucune donnée n'est placée dans les cellules réservées. Une donnée qui s'y trouverait par hasard y resterait.

INITIALISATION DE LA RAM

Certains assembleurs permettent au programmeur de loger des valeurs initiales dans la RAM. Nous vous conseillons fortement de ne pas utiliser cette possibilité ; elle suppose que le programme (avec ces valeurs initiales) sera chargé à partir d'un dispositif externe (par exemple, une bande perforée ou une disquette) chaque fois qu'il sera lancé. Or, la plupart des programmes pour microprocesseurs sont stockés dans une mémoire non

volatile ROM et démarrent dès la mise sous tension. Dans ce cas, la RAM ne conserve pas son contenu et n'est pas rechargée. Incluez toujours dans votre programme des instructions d'initialisation pour la RAM.

PSEUDO-OPÉRATIONS DE RÉFÉRENCE EXTERNE

RÉFÉRENCES EXTERNES

On souhaite souvent que des instructions d'un programme ou d'un sous-programme utilisent des noms définis ailleurs. De tels noms sont appelés des références externes ; un programme spécial éditeur de liens sera nécessaire pour introduire ces valeurs et déterminer si certains noms sont restés non définis, ou ont été définis plus d'une fois.
La pseudo-opération EXTERNAL (« externe »), signifie que le nom est défini ailleurs.
La pseudo-opération ENTRY (« entrée ») signifie que le nom utilisé est disponible pour un emploi ailleurs, bien qu'il soit défini dans ce programme.
La manière exacte dont ces pseudo-opérations de référence externe sont exécutées varie d'un assembleur à l'autre. Nous ne nous référerons plus à de telles pseudo-opérations, mais nous noterons qu'elles sont très utiles dans les applications réelles.

PSEUDO-OPÉRATIONS D'INTENDANCE

Des pseudo-opérations d'intendance variées marquent surtout les opérations de l'assembleur et le listage qu'il fournit, plutôt que la sortie du programme lui-même. Ces pseudo-opérations incluent :

- **END, qui marque la fin de l'assemblage du programme source.**
- **LIST, qui indique à l'assembleur qu'il doit commander l'impression du programme source. Certains assembleurs autorisent des variantes telles que « NO LIST » (pas de listage) ou « LIST SYMBOL TABLE » pour éviter des listages longs et répétitifs.**
- **NAME et TITLE (nom et titre) pour l'impression d'un nom en tête de chaque page du listage.**
- **PAGE ou SPACE, pour passer à la page suivante ou à la ligne suivante respectivement, améliorant ainsi l'aspect du listage et le rendent plus facile à lire.**
- **PUNCH, pour transférer le code objet sur bande perforée. Cette pseudo-opération peut, dans certains cas, constituer l'option par défaut et devient alors inutile.**

LABELS AVEC PSEUDO-OPÉRATIONS

Les utilisateurs se demandent souvent s'ils doivent, et quand, assigner un label à une pseudo-opération. Voici nos recommandations :

- Toutes les pseudo-opérations d'équivalence EQU doivent disposer de leur label. Sinon, elles sont inutiles puisque leur fonction consiste précisément à définir ce label.
- Les pseudo-opérations DATA et RESERVE disposent généralement de labels. Ces labels identifient la première cellule mémoire utilisée ou réservée.
- Les autres pseudo-opérations ne devraient pas disposer de labels. Quelques assembleurs leur permettent d'en avoir un, mais nous conseillons d'éviter cette pratique car il n'existe pas de méthode standard pour les interpréter.

CHAMPS OPÉRANDES ET ADRESSES

La plupart des assembleurs laissent au programmeur une grande liberté pour le contenu du champ opérandes et adresses. Mais souvenez-vous que les assembleurs disposent déjà des noms des registres du microprocesseur et de ses instructions, et peuvent également stocker d'autres noms par construction.
Quelques options courantes pour le champ opérande sont :

DONNÉES OU ADRESSES DÉCIMALES

1) Nombres décimaux
La plupart des assembleurs supposent que tous les nombres sont décimaux, sauf spécification contraire. Ainsi :

```
ADD          A,100
```

signifie : « ajouter la valeur 100 (en décimal) au contenu de l'accumulateur ».

SYSTÈMES DE NUMÉRATION NON DÉCIMAUX

2) Autres systèmes de numération
La plupart des assembleurs acceptent le binaire, l'octal et l'hexadécimal. Mais on doit identifier ces systèmes de numération d'une façon ou d'une autre, par exemple à l'aide d'un suffixe ou d'un préfixe accolé au nombre et servant d'identificateur, sous forme d'une lettre ou d'un autre caractère. Voici quelques identificateurs courants :

B pour le binaire.
O, @ , Q ou C pour l'octal (nous évitons généralement la lettre O en raison d'une confusion possible avec zéro).
H pour l'hexadécimal (ou le DCB standard).
D pour le décimal. D peut être omis, le décimal constituant le choix de l'assembleur par défaut de spécification*.

Parfois, l'assembleur exigera que les nombres hexadécimaux commencent par un digit numérique (par exemple, 0A36 au lieu de A36) afin de les différencier des noms et des labels. Une bonne habitude consiste à introduire les nombres dans la base qui leur confère la plus claire des significations ; par exemple, des constantes décimales en décimal ; des adresses et des valeurs DCB en hexadécimal ; des masques ou des sorties de bits en binaire s'ils sont courts, en hexadécimal s'ils sont longs.

3) Noms symboliques
Des noms peuvent figurer dans le champ opérandes ; ils seront traités comme les données qu'ils représentent. Souvenez-vous cependant qu'il y a une différence entre données et adresses. La séquence :

```
CINQ         EQU          5
             ADD          A,CINQ
```

additionnera la valeur 5 au contenu de l'accumulateur A.

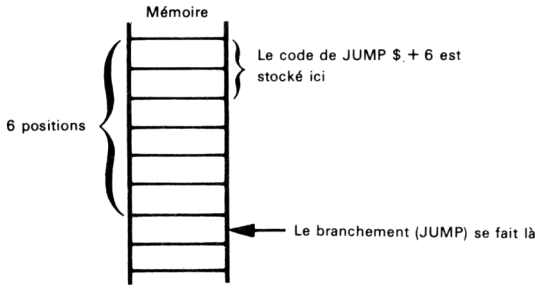
---

* NDT : Notons toutefois que ces conventions n'ont rien d'absolu et qu'elles peuvent varier d'un assembleur à l'autre.

4) La valeur courante du compteur de position (généralement indiquée par * ou $).
Cette notation est utile principalement avec les instructions de JUMP ; par exemple :

```
JUMP        $ + 6
```

provoque un branchement (JUMP, « saut ») à la sixième position mémoire suivant celle qui contient le premier octet de cette même instruction :



La plupart des microprocesseurs disposent de nombreuses instructions à deux ou trois octets. De ce fait, vous éprouverez certaines difficultés pour déterminer les différences réelles entre les caractéristiques de deux langages assembleurs. Aussi, l'usage de tels déplacements par rapport au contenu du compteur de position crée fréquemment des erreurs que vous éviterez en utilisant de préférence des labels.

CARACTÈRES ASCII

5) Codage des caractères
La plupart des assembleurs permettent l'introduction de caractères sous forme de chaînes en code ASCII. Ces chaînes peuvent être identifiées en les entourant d'apostrophes ou de guillemets ; elles sont encore parfois marquées par un symbole tel que A ou C placé en préfixe ou en suffixe. Peu d'assembleurs autorisent aussi l'entrée de chaînes EBCDIC.
Nous vous recommandons l'usage de chaînes de caractères pour tous vos textes, afin d'améliorer la clarté et la lisibilité de vos programmes.

EXPRESSIONS ARITHMÉTIQUES ET LOGIQUES

6) Combinaisons de 1) à 5) avec les opérateurs arithmétiques, logiques et spéciaux.
Presque tous les assembleurs autorisent des combinaisons arithmétiques simples telles que DEBUT +1. Certains permettent aussi des multiplications, divisions, fonctions logiques, décalages, etc., dans des expressions. On notera que les expressions sont évaluées par l'assembleur pendant l'assemblage. Même si une expression, dans le champ opérande, inclut des multiplications, vous n'aurez peut-être pas le droit d'utiliser des multiplications dans la logique de votre propre programme — à moins de rédiger un sous-programme pour cette séquence spécifique.
Les assembleurs diffèrent par les expressions qu'ils acceptent et par la façon de les interpréter. Des expressions complexes rendent un programme difficile à lire et à comprendre. Nous vous avons présenté quelques recommandations au cours de cette section, mais nous les répèterons et nous en ajouterons d'autres. **En général, mettez l'accent sur la clarté et la simplicité.** Il n'y a aucun avantage à devenir un expert des subtilités des assem-

bleurs ou à user d'expressions très complexes là où elles ne sont pas indispensables. **Nous vous proposerons les approches suivantes :**

1) — Utilisez les systèmes de numération ou les codes de caractères les plus évidents. Rédiger les masques et les nombres BCD en décimal, les caractères ASCII en octal, et les constantes numériques ordinaires en hexadécimal n'offre aucun intérêt et devra donc être évité.
2) — Ne pas confondre les données et les adresses.
3) — N'utilisez pas de décalage par rapport au compteur de position.
4) — Rédigez des expressions simples et évidentes. Ne vous appuyez pas sur quelque caractéristique obscure de l'assembleur.

ASSEMBLAGE CONDITIONNEL

Certains assembleurs permettent l'inclusion ou l'exclusion de parties du programme source, en fonction des conditions existant durant l'assemblage. C'est ce qu'on appelle l'assemblage conditionnel ; il confère à l'assembleur la souplesse d'un compilateur. **La plupart des assembleurs pour microprocesseurs ne disposent que de capacités réduites en assemblage conditionnel.**
Une forme typique d'assemblage conditionnel sera :

```
IF COND
.
. Programme conditionnel
.
.
ENDIF
```

Si l'expression COND est vraie, au moment de l'assemblage, les instructions entre IF et ENDIF (END... IF : fin de « si », « if »), qui sont ici des pseudo-opérations, sont incluses dans le programme.
Les applications typiques de l'assemblage conditionnel sont :

1) L'inclusion ou l'exclusion de variables supplémentaires.

2) L'introduction de diagnostics dans les lancements pour test.

3) Permettre l'intervention de données de longueurs (en bits) diverses.

4) Créer des versions différentes à partir d'un programme original.
Malheureusement, l'assemblage conditionnel tend à embrouiller les programmes et les rend difficiles à lire. Ne l'utilisez donc que lorsqu'il est nécessaire.

MACROS

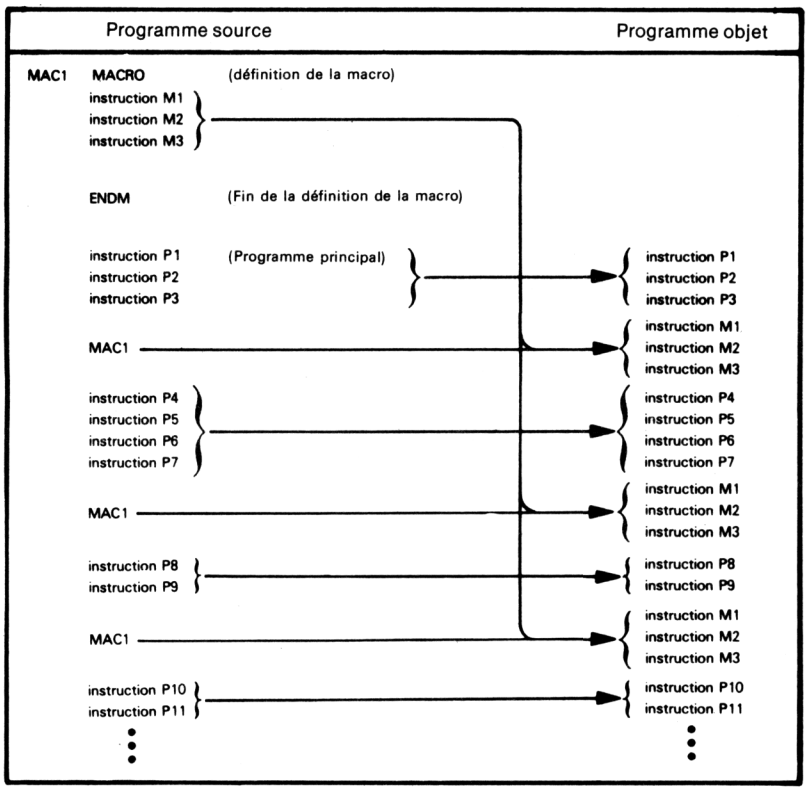
DÉFINITION D'UNE SÉQUENCE D'INSTRUCTIONS

On remarquera souvent que certaines séquences d'instructions reviennent plusieurs fois dans un programme source. Ces séquences peuvent refléter les besoins logiques de votre programme, ou encore viennent compenser les déficiences

du jeu d'instructions de votre microprocesseur. Vous pouvez éviter d'avoir à les ré-écrire à chaque fois en utilisant une macro.

Les macros vous permettent d'affecter un nom à une séquence d'instructions. Vous utilisez ensuite ce nom dans votre programme source au lieu de la séquence répétitive. L'assembleur remplacera ce nom par ladite séquence d'instructions.

Ce qui peut s'illustrer de la façon suivante :



Macros et sous-programmes ne sont pas semblables. Un sous-programme est développé une fois dans un programme, l'exécution de celui-ci l'appelant au moment opportun. Une macro est développée à chaque fois qu'elle doit intervenir ; de ce fait, la macro ne dépend d'aucun appel ou branchement.

Avantages des macros :

AVANTAGES DES MACROS

1) Programmes source plus courts.

2) Meilleure documentation des programmes.

3) Utilisation de séquences d'instructions bien mises au point — une fois que la macro a été mise au point : vous êtes certain que cette séquence ne comportera pas d'erreurs.

4) Modifications aisées. Changez la définition de la macro et l'assembleur introduira cette nouvelle version chaque fois qu'elle sera utilisée.

5) Inclusion de commandes, de mots-clés ou d'autres instructions dans le jeu d'instructions de base de votre microprocesseur. Vous utiliserez les macros pour développer ou clarifier le jeu d'instructions.

Inconvénients des macros :

INCONVÉNIENTS DES MACROS

1) Répétition des mêmes séquences d'instructions puisque la macro est développée à chacun de ses appels.

2) Une simple macro peut créer une quantité d'instructions.

3) Le manque de standardisation rend les programmes difficiles à lire et à comprendre.

4) Effets possibles sur les registres et les indicateurs non clairement déterminés.

VARIABLES LOCALES OU GLOBALES

L'un des problèmes réside dans le fait que les variables intervenant dans les macros ne sont connues que dans celles-ci (ce qui signifie qu'elles sont locales et non globales). C'est là souvent une source de grande confusion, sans aucun bénéfice en contrepartie.
Pensez à ce problème en utilisant des macros.

COMMENTAIRES

Tous les assembleurs vous permettent l'introduction de commentaires dans vos programmes source. Ces commentaires n'ont aucun effet sur le code objet, mais ils vous aident à lire, comprendre et documenter vos programmes. De bons commentaires constituent une part essentielle de l'écriture de programmes en langage assembleur ; sans commentaires, les programmes sont très difficiles à comprendre.

TECHNIQUES DE COMMENTAIRES

Nous discuterons de l'art de commenter les programmes avec la documentation dans un chapitre ultérieur, mais voici qui devrait vous guider en un premier temps :

1) Utilisez les commentaires pour montrer quelle fonction d'application votre programme exécute, et non comment votre microprocesseur exécute l'instruction.
Les commentaires doivent préciser des choses telles que « LA LIMITE SUPERIEURE DE TEMPERATURE EST-ELLE ATTEINTE ? », « LIGNE SUIVANTE », ou « EXAMEN DE LA CHARGE DU COMMUTATEUR ».
Mais ils ne devraient pas dire des choses telles que : « ADDITIONNER 1 A L'ACCUMULATEUR », « BRANCHER A START », ou « EXAMINEZ LA RETENUE ».

Vous devrez décrire comment le programme affecte le système, car les effets internes sur l'UCT sont rarement intéressants.
2) Soyez brefs et précis. Les détails interviendront ailleurs dans la documentation.

3) Commentez tous les points-clés.

4) Ne commentez pas des instructions standard ou des séquences qui modifient les compteurs et pointeurs. Prêtez toute votre attention aux instructions dont la signification n'est pas évidente.

5) N'utilisez pas d'abréviations obscures.

6) Vos commentaires doivent être clairs et lisibles.

7) Commentez toutes les définitions, décrivez leur objet. Marquez également toutes les tables et les zones de rangement des données.

8) Commentez aussi bien les sections de votre programme que les instructions particulières.

9) Soyez cohérent avec votre terminologie. Vous pouvez (devez) être répétitif, vous ne devez pas avoir besoin de consulter une encyclopédie.

10) Notez les points qui risquent d'introduire des confusions, par exemple : « LE CARRY A ETE POSITIONNE PAR LA DERNIERE INSTRUCTION ». Vous pourrez omettre ceci dans la documentation finale.
On travaille aisément avec un programme bien documenté. Vous récupérerez plusieurs fois le temps investi dans la rédaction des commentaires. Nous essayerons de vous proposer un bon style de commentaires dans les exemples de programmes, mais nous serons souvent amenés à les sur-commenter pour des raisons pédagogiques.

TYPES D'ASSEMBLEURS

Bien que tous les assembleurs exécutent les mêmes tâches, ils diffèrent considérablement. Nous n'allons pas décrire tous les types d'assembleurs existants ; nous allons simplement vous en présenter les appellations et indiquer quelques choix.

CROSS-ASSEMBLEUR

Un cross-assembleur (ou « assembleur croisé », pour certains) **est un assembleur qui tourne sur un ordinateur autre que celui auquel le programme assemblé est destiné.**

L'ordinateur sur lequel tourne le cross-assembleur est généralement une grosse machine avec un fort support logiciel et des périphériques rapides — tels qu'un IBM 370, un Univac 1108 ou un Burroughs 6700. L'ordinateur sur lequel tournera le programme assemblé est typiquement réalisé avec un microprocesseur tel que le Z80. La plupart des cross-assembleurs sont écrits en FORTRAN de façon à les rendre portables*.

* NDT : A l'heure actuelle, la tendance est également d'utiliser des langages de type structuré, tel le langage C qui permet la rédaction du logiciel de base (assembleur, compilateur, éditeur de texte, etc.) jusqu'au niveau du système d'exploitation. Ces langages, étant de type évolué, assurent au même titre que FORTRAN une portabilité facile.

ASSEMBLEUR RÉSIDANT Un assembleur résidant, ou auto-assembleur, est un assembleur qui tourne sur l'ordinateur auquel est destiné le programme assemblé. L'assembleur résidant exigera de la mémoire et des périphériques et pourra tourner assez lentement.

MACRO-ASSEMBLEUR **Un macro-assembleur est un assembleur vous permettant de définir des séquences d'instructions sous forme de macros.**

MICRO-ASSEMBLEUR **Un micro-assembleur, ou assembleur de microprogramme, est un assembleur utilisé pour rédiger des microprogrammes servant à définir le jeu d'instructions d'un micro-ordinateur. La microprogrammation n'a rien de spécifique à voir avec les microprocesseurs.**

MÉTA-ASSEMBLEUR **Un méta-assembleur est un assembleur capable de traiter plusieurs jeux d'instructions.** L'utilisateur doit définir le jeu d'instructions utilisé à un instant donné.

ASSEMBLEUR MONO-PASSE **Un assembleur mono-passe est un assembleur qui examine le programme en langage d'assemblage une seule fois.** Il devra pouvoir résoudre le problème des références ultérieures, par exemple, le renvoi à des JUMP dont le label n'a pas encore été défini.

ASSEMBLEUR À DEUX PASSES **Un assembleur à deux passes traite le programme source en langage d'assemblage en le parcourant deux fois. A la première lecture, il collecte simplement et définit tous les symboles ; à la seconde, il remplace les références par leur définition. De ce fait, il n'éprouve généralement pas de difficulté avec des références ultérieures. Cependant, le développement des macros et l'assemblage conditionnel peuvent être source de problèmes. Sur quelques grosses machines, 7 passes ou plus peuvent être nécessaires pour résoudre toutes les références. Un assembleur deux passes peut être très lent s'il ne dispose pas d'une mémoire de masse rapide (telle qu'une disquette) ; en effet, il devra alors lire deux fois le programme source à partir d'un support lent (tel qu'un lecteur de bande perforée de télé-imprimeur). La plupart des assembleurs pour microprocesseurs sont à deux passes.**

ERREURS

Les assembleurs délivrent normalement des messages d'erreurs, consistant souvent en une simple lettre codée. Quelques erreurs typiques sont :

1) Nom non défini (souvent mal orthographié, ou carrément oublié).

2) Caractère illégal (par exemple, un 2 dans un mot binaire).

3) Format illégal (délimiteur erroné ou opérande incorrect).

4) Expression illégale (par exemple, deux opérateurs dans une rangée).

5) Valeur illégale (souvent excessive).

6) Opérateur manquant.

7) Double définition (par exemple, deux valeurs différentes attribuées à un nom).

8) Label illégal (par exemple, un label sur une pseudo-opération qui ne peut en disposer).

9) Label manquant.

10) Mnémonique non défini.
En interprétant les erreurs indiquées par l'assembleur, vous devez conserver présent à l'esprit que l'assembleur a pu être dirigé sur une mauvaise voie s'il a trouvé une lettre égarée, un espace superflu ou une ponctuation incorrecte. La plupart des assembleurs se fourvoieront alors dans l'interprétation des instructions suivantes et émettront des messages d'erreur sans signification. Attachez toujours la plus grande importance à la première erreur ; les autres peuvent en découler. Collez toujours aux formats standard, cela vous évitera bien des ennuis.

CHARGEURS

Le chargeur est un programme qui reçoit la sortie de l'assembleur (le code objet) et le place en mémoire. La gamme des chargeurs couvre du chargeur le plus simple au plus complexe. En voici quelques types :

CHARGEUR BOOTSTRAP — ***Le chargeur « bootstrap » est un programme qui utilise ses propres premières instructions pour se charger complètement ou charger un autre programme chargeur en mémoire.***
Le chargeur bootstrap peut se trouver en ROM, ou alors vous pouvez avoir à l'introduire en mémoire à l'aide des commutateurs fixés sur le panneau avant de la machine. L'assembleur peut placer un chargeur bootstrap au départ du programme objet qu'il produit.

CHARGEUR TRANSLATEUR — **Un *chargeur translateur* (on dit aussi : relayeur, de l'américain « relocating ») peut charger un programme n'importe où dans la mémoire.** Généralement, il charge chaque programme à la suite du précédent. Le programme doit, dans ce cas, être capable de subir des déplacements : il doit être translatable. Un chargeur absolu, au contraire, placera toujours le programme au même endroit de la mémoire.

CHARGEUR DE LIENS — **Un *chargeur de liens* (« linking loader ») charge les programmes et les sous-programmes qui ont été assemblés séparément ; il résoud les références croisées** — c'est-à-dire les problèmes d'instructions d'un programme qui se réfèrent au label d'un autre. Le programme objet chargé par un chargeur de liens doit être créé par un assembleur qui permet et qui note les références croisées.

ÉDITEUR DE LIENS — Une autre approche consiste à séparer les fonctions d'établissement des liens et de chargement, et de faire traiter les relations entre programmes (liens) par un programme appelé **éditeur de liens.**

RÉFÉRENCES

Une monographie complète sur les macros, de M. Campbell-Kelly « An Introduction to Macros », *American Elsevier,* New York, 1973.

A. Osborne : *Initiation aux micro-ordinateurs : niveau 2,* SECF-Editions Radio, Paris, 1980.

La microprogrammation est décrite sous forme conceptuelle dans INITIATION AUX MICRO-ORDINATEURS NIVEAU 2 (traduction de *An Introduction to Microcomputers : Volume 1 — Basic Concepts),* chapitre 4. Une description plus technique est donnée par A.K. Agrawala et T.G. Rauscher dans *Foundations of Microprogramming,* Academic Press, New York, 1976.

Il est possible de trouver des descriptions plus détaillées des assembleurs et des chargeurs dans les ouvrages de D.W. Barron « *Assemblers and Loaders* », American Elsevier, New York, 1972, et de C.W. Gear Computer Organization and Programming, McGraw-Hill, New York, 1974.

CHAPITRE 3

JEU D'INSTRUCTIONS DU Z80 LANGAGE ASSEMBLEUR

Nous sommes maintenant prêts à rédiger nos premiers programmes en langage assembleur. Nous commencerons dans ce chapitre par définir chacune des instructions (mnémoniques) du Z80, ainsi que les règles syntaxiques de l'assembleur ZILOG.

Nous ne traiterons pas de la partie « matériel » (signaux, interfaçage, etc.) du microprocesseur dans ce livre. Tout cela est décrit en détail dans « An Introduction to Microcomputers : Vol 2 - Some Real Microprocessors » et « Vol 3 - Some Real Support Devices ».

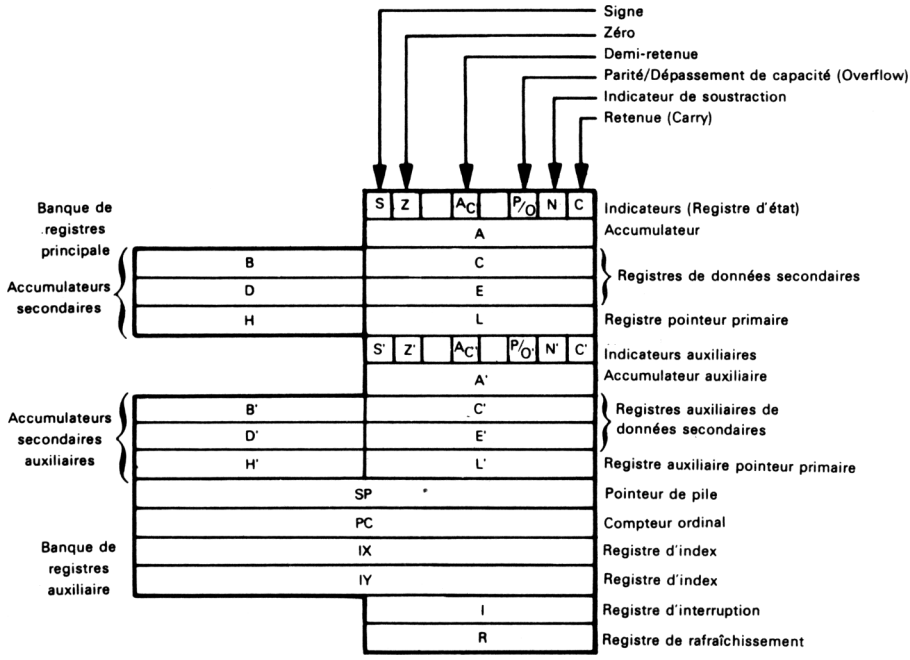
Dans ce livre, nous verrons par contre quelles sont les techniques d'utilisation du langage assembleur dont dispose le programmeur, pour qui les broches et signaux produits sont inintéressants.

Les interruptions, l'accès direct mémoire (DMA) et l'architecture de la pile du Z80 seront décrits dans les chapitres qui suivent, ainsi que dans les méthodes de programmation en assembleur concernant les mêmes sujets.
Le chapitre comprend une définition détaillée de chaque instruction de l'assembleur. Ces instructions sont décrites de la même façon qu'au chapitre 6 de « Z80 Programming for Logic Design ».

La description détaillée de chaque instruction est précédée d'un exposé général sur le jeu d'instructions du Z80, en distinguant les instructions qui sont beaucoup utilisées de celles qui le sont moyennement ou peu. Pour un programmeur expérimenté, cette distinction n'est pas importante — et sans doute même discutable. Pour un programmeur débutant, par contre, il est recommandé de n'utiliser au départ que les instructions classées comme étant « les plus souvent utilisées ». Une fois les concepts de programmation bien maîtrisés, on peut alors examiner les autres instructions et... les utiliser si nécessaire.

LES REGISTRES ET LES INDICATEURS (FLAGS)

On peut schématiser la structure interne du Z80 comme indiqué ci-dessous :



L'accumulateur est la source (primaire) et la destination des instructions à un ou deux opérandes.

Par exemple, les transferts de données les plus rapides entre unité centrale et organes d'E/S sont exécutés au moyen de l'accumulateur. En outre, il existe plus d'instructions transférant des données entre mémoire et accumulateur qu'entre n'importe quel autre registre et la mémoire. De même, toutes les instructions arithmétiques ou logiques sur 8 bits trouvent leur premier opérande dans l'accumulateur et y stockent leur résultat. Ce qui signifie que l'**on doit toujours charger l'accumulateur avant d'exécuter quelque opération arithmétique ou logique que ce soit.**

Les registres B, C, D, E, H et L sont tous des registres secondaires. Les données stockées dans n'importe lequel de ces 6 registres peuvent être traitées avec la même facilité ; elles peuvent être transférées dans n'importe quel registre, ou utilisées comme opérandes secondaires dans les instructions à 2 opérandes.

Il y a, cependant, quelques différences importantes dans les fonctions des registres B, C, D, E, H et L.

Les registres H et L forment un registre pointeur primaire pour le Z80. Cela signifie qu'on utilise généralement ces deux registres pour contenir l'adresse (16 bits) de la donnée à laquelle on désire accéder. Des données peuvent ainsi être transférées entre n'importe quel registre et l'emplacement mémoire pointé par H et L. Puisque HL est un pointeur primaire, l'utiliser comme premier opérande permet souvent de « consommer » moins de cycles instructions et moins d'octets de code objet. Le programmeur Z80 devra

donc s'efforcer d'adresser la mémoire avec H et L chaque fois que possible. **Dans vos programmes, réservez toujours les registres H et L pour contenir une adresse mémoire. Les registres B, C, D et E fournissent un stockage de donnée secondaire** ; fréquemment, le second opérande des instructions à 2 opérandes est chargé dans l'un de ces 4 registres (le premier étant l'accumulateur, qui est également le lieu de destination du résultat).
Il existe un **nombre limité d'instructions** qui **traitent les registres B, C, D et E comme des pointeurs (16 bits) de données.** Mais ces instructions ne permettent des transferts qu'entre mémoire et accumulateur.
Dans vos programmes, utilisez surtout les registres B, C, D et E comme des registres tampons pour des données ou des adresses.
Les registres IX et IY sont des registres d'index. Ils permettent de traiter la notion d'indice, telle que décrite dans « Initiation aux micro-ordinateurs - Niveau 2 ». (Nous y reviendrons plus en détail lorsque nous traiterons les différents modes d'adressage.)
Les registres auxiliaires F', A', B', C', D', E', H' et L' forment une copie de la banque de registres usuels. Deux simples instructions d'échanges (codées sur un octet) sélectionnent l'un ou l'autre ensemble de registres : une instruction échange AF avec la paire de registres auxiliaires AF', et l'autre instruction échange BC, DE et HL avec respectivement BC', DE' et HL'. Toutes les opérations registres s'effectuent sur la banque principale, l'utilisation des instructions d'échange permet de sélectionner la banque de registres à activer.
Les registres auxiliaires peuvent ainsi être réservés pour le traitement rapide d'interruptions (changement de contexte)... ou selon le bon plaisir du programmeur pour toute autre application.
Certaines instructions du Z80 permettent le traitement de données 16 bits. Ces instructions font référence à des paires de registres, comme indiqué ci-après :

F	et	A
B	et	C
D	et	E
H	et	L
F'	et	A'
B'	et	C'
D'	et	E'
H'	et	L'
Octet le plus significatif		Octet le moins significatif

L'ensemble accumulateur + registre d'état, considéré comme unité 16 bits, est utilisé uniquement pour les opérations sur pile, ou d'échanges. Les opérations arithmétiques utilisent BC, DE ou HL comme opérandes 16 bits.
L'indicateur de retenue (Carry), est positionné, en dehors des opérations arithmétiques, dans les instructions de décalage ; il est mis à zéro par les instructions booléennes.
L'indicateur de soustraction (N) est testé de façon interne par l'unité centrale. Ce bit est initialisé à 1 par toutes les instructions de soustraction et à 0 par les instructions d'addition.
L'indicateur de Parité/Overflow (P/O) a plusieurs significations, selon le type d'opération exécutée. Pour les opérations arithmétiques, c'est un indicateur de dépassement de capacité (overflow). Pour les instructions d'entrées, logiques et de rotations, c'est un bit de parité : 1 pour parité paire, et 0 pour impaire.
Pendant l'exécution des instructions manipulant des blocs d'octets, ce bit reste à 1 tant

que le compteur d'octets est non nul, et passe à 0 sinon. Il est aussi initialisé à la valeur de la bascule d'interruption (IFF2) lors d'une instruction LD A,I ou LD A,R.
L'indicateur de zéro (Z) est mis à 1 dès qu'une opération arithmétique ou logique produit un résultat nul, à zéro dans tout autre cas.
L'indicateur de signe (S) acquiert la valeur du bit le plus significatif du résultat de toute instruction arithmétique ou logique.
L'indicateur de demi-retenue contient toute retenue du bit 3 vers le bit 4 résultant d'une instruction arithmétique. Le rôle de ce bit est de simplifier l'utilisation des opérations sur opérandes DCB (décimaux codés en binaire).
Tous les indicateurs décrits ci-dessus conservent leurs valeurs jusqu'à ce qu'une instruction qui les modifie soit exécutée.
Changer la valeur de l'accumulateur ne modifiera pas forcément la valeur du registre d'état.
Par exemple, si l'indicateur de zéro (Z) est égal à 1, et si un chargement de l'accumulateur est effectué, le fait que l'accumulateur contienne une valeur non nulle ne modifiera pas l'indicateur Z.
Le pointeur de pile (16 bits) permet l'implantation d'une pile n'importe où dans l'espace mémoire adressable. La taille de la pile est limitée uniquement par la taille mémoire. En fait, on se sert rarement de plus de 256 octets pour la pile, laquelle est utilisée pour les chaînages de sous-programmes et les traitements d'interruptions. Il est préférable de ne pas se servir de la pile pour passer des paramètres, le jeu d'instructions du Z80 n'ayant pas été conçu pour cette application. La pile du Z80 commence à son adresse la plus haute, l'instruction PUSH décrémentant le contenu du pointeur, à l'inverse de l'instruction POP.

Le registre d'interruption et le registre de rafraîchissement sont des registres spécialisés qui ne sont généralement pas utilisés par le programmeur.
Le registre d'interruption contient l'adresse de la page mémoire d'une routine de traitement d'interruption, le déplacement dans la page étant fourni par le périphérique qui a généré l'interruption. Cette philosophie permet de changer l'adresse de gestion de l'interruption tout en fournissant une réponse rapide au périphérique. Le registre de rafraîchissement consiste en un compteur 7 bits.
Ce compteur est incrémenté automatiquement après chaque instruction et fournit une adresse de rafraîchissement à d'éventuelles mémoires dynamiques. Le bit le plus significatif de ce registre reste inchangé par rapport à la valeur chargée par la dernière instruction LD R,A exécutée.

LES MODES D'ADRESSAGE DU Z80

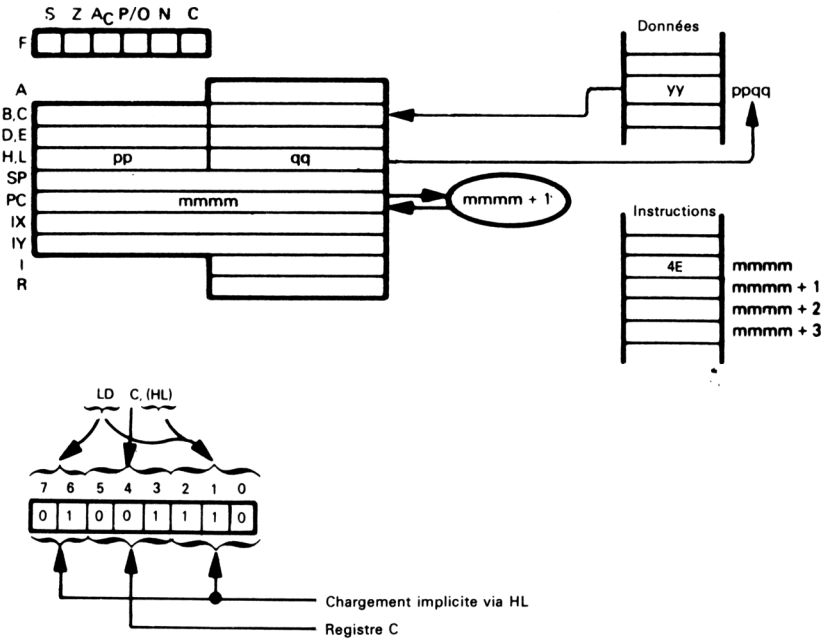
Le Z80 comprend un ensemble assez étendu de modes d'adressage, à savoir :
- **Implicite**
- **Implicite avec auto-incrémentation/décrémentation**
- **Implicite avec accès pile**
- **Indexé**
- **Direct**
- **Relatif**
- **Basé**
- **Indirect**
- **Immédiat**

Adressage implicite (ou adressage indirect)

Dans ce type d'adressage, les registres H et L contiennent l'adresse de l'octet mémoire à utiliser. Des données peuvent ainsi être transférées entre l'adresse précisée et n'importe lequel des registres A, B, C, D, E, H ou L. Par exemple, l'instruction :

LD C,(HL)

charge le registre C avec le contenu de la mémoire pointée par HL. Ce qui peut être illustré de la façon suivante :



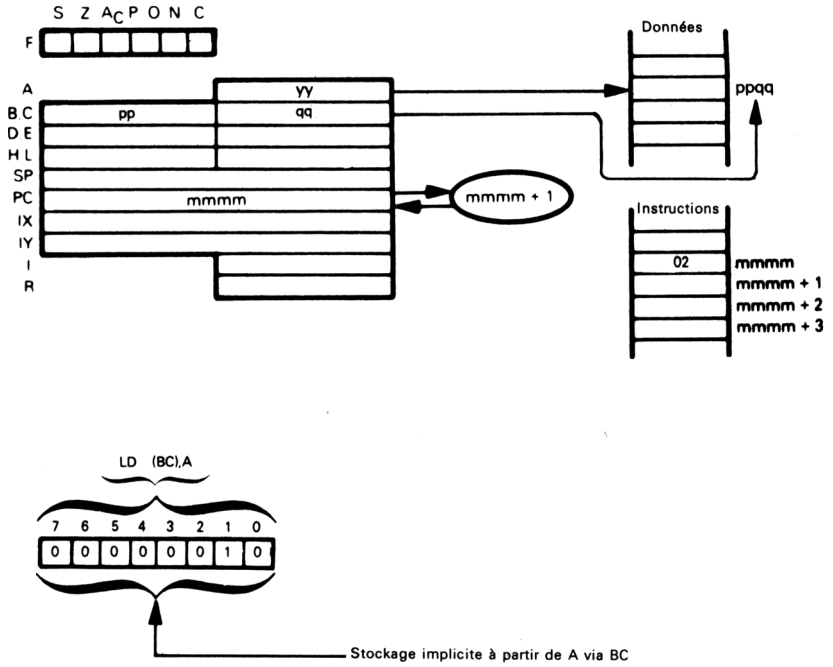
Note : on appelle parfois ce type d'adressage « *adressage indirect* ».

Un nombre limité d'instructions utilise les registres B,C ou D,E comme pointeurs. Ces instructions transfèrent des données entre accumulateur et octet mémoire adressé par B,C ou D,E. L'instruction :

LD (BC),A

stocke le contenu de A dans la mémoire adressée par la paire de registres BC, comme indiqué page 44.

NDT : Par abus de langage, et surtout pour alléger la description de certaines instructions, on emploiera indifféremment les mots « contenu d'une adresse » pour « contenu d'un octet de mémoire adressé ». Ainsi, par exemple, le « contenu de l'adresse spécifiée par HL » correspondra en fait à la valeur de l'octet contenu à l'emplacement mémoire dont l'adresse est contenue dans le double registre HL.



Adressage implicite avec auto-incrémentation/décrémentation (transferts de blocs)

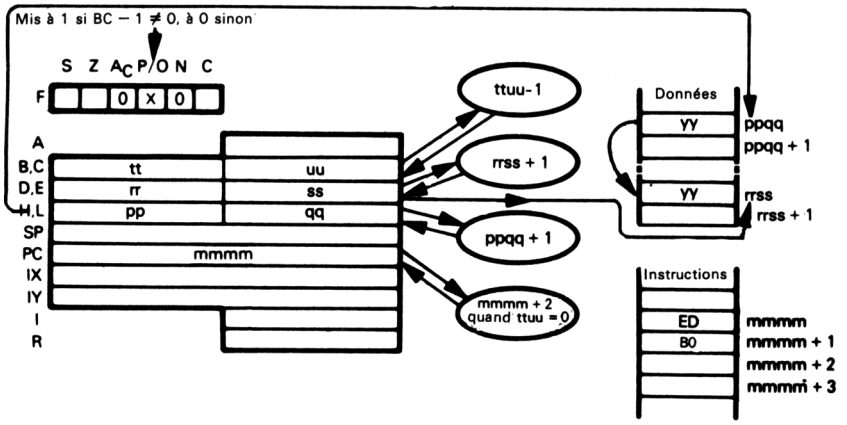
Ces instructions opèrent sur un bloc de données dont la taille est initialisée par le programmeur à une valeur contenue dans BC. Dans cette forme d'adressage, un octet est transféré de la mémoire adressée par HL vers la mémoire adressée par DE ; HL et DE sont ensuite incrémentés et BC est décrémenté. L'opération continue jusqu'à ce que BC atteigne la valeur 0, auquel cas l'instruction est terminée. Il existe un certain nombre de variations telles que :

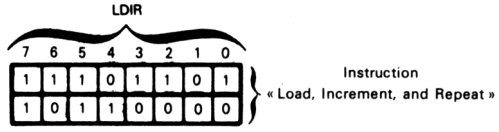
— branchement en fin de boucle non automatique, de manière à pouvoir effectuer d'autres opérations avant de remonter dans la boucle ;
— auto-décrémentation de HL et DE (au lieu d'auto-incrémentation) ;
— possibilité d'utiliser des instructions de recherches, qui comparent l'octet mémoire adressé par HL avec le contenu de A, et positionnent un indicateur s'il y a correspondance.

L'instruction :

LDIR

correspond au traitement suivant :





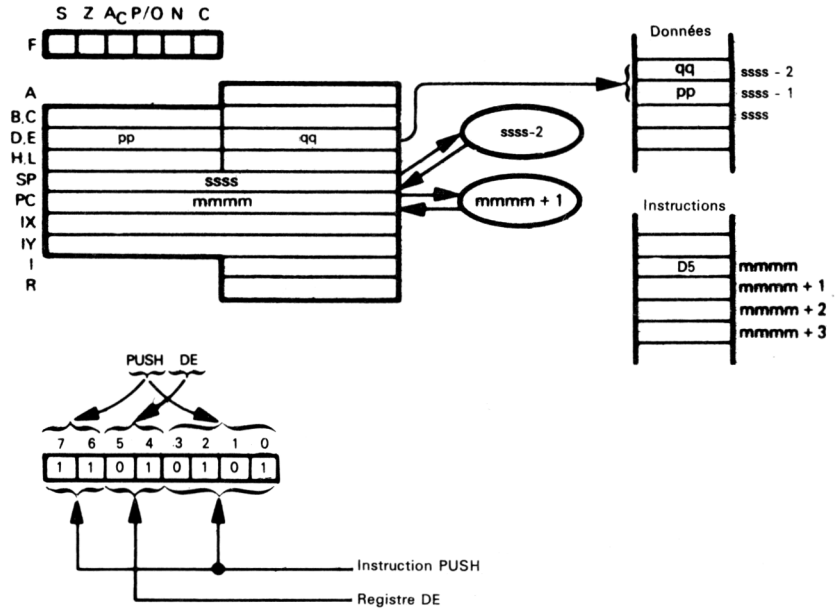
Un groupe similaire d'instructions d'entrées/sorties existe, permettant des transferts de blocs de données entre mémoire et organe d'E/S. Le numéro de port est contenu dans C, tandis que B est utilisé comme compteur. La mémoire est adressée par HL.

Adressage implicite avec accès pile

Puisque la pile fait partie de l'espace mémoire, on doit considérer les instructions avec accès à la pile comme des instructions à référence mémoire. **Les instructions PUSH et POP transfèrent 2 octets de donnée entre une paire de registres et la mémoire pointée par SP,** c'est-à-dire le sommet de la pile. L'adresse de la pile du Z80 est décrémentée par chaque instruction PUSH et incrémentée par chaque instruction POP. L'instruction :

PUSH DE

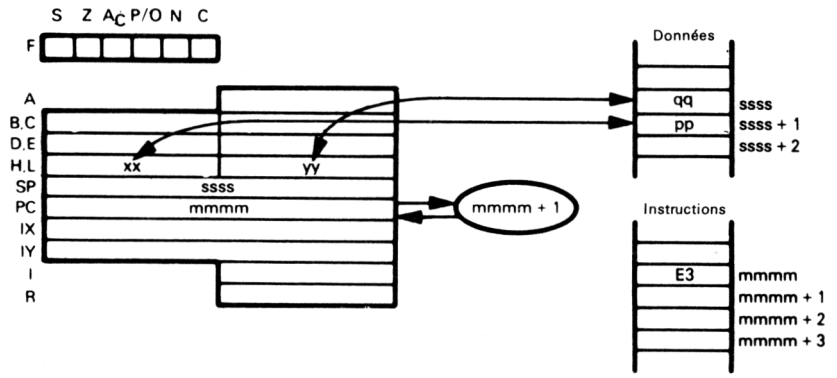
correspond au traitement suivant :



Le Z80 comporte également des instructions qui échangent les 2 octets en sommet de pile avec un registre 16 bits (HL, IX, IY). L'instruction :

EX (SP),HL

correspond au traitement suivant :



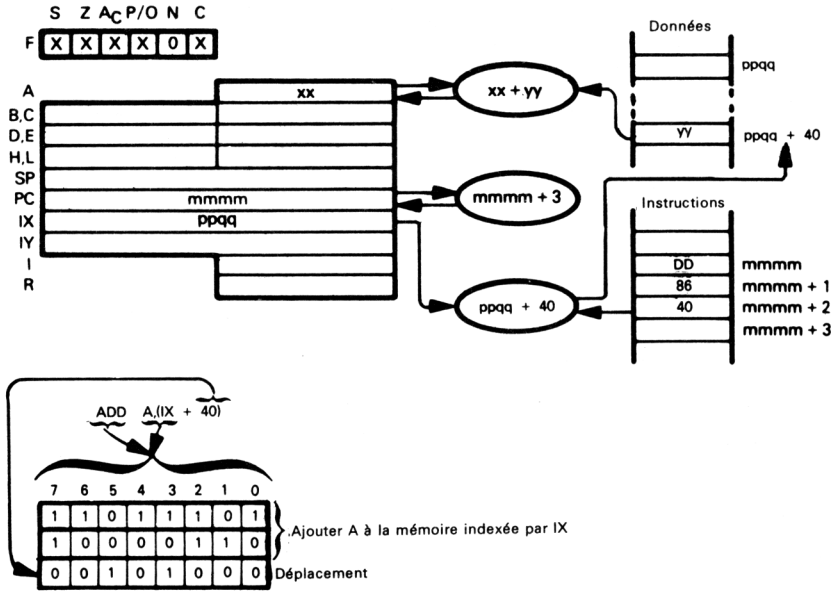
Adressage indexé

Le Z80 possède 2 registres d'index 16 bits, appelés IX et IY. Ils ont tous deux les mêmes fonctions. Toutes les références mémoire pour lesquelles (HL) peut être utilisé, peuvent

également utiliser un adressage indexé. La différence entre l'utilisation de (HL) et de IX (ou IY) est que l'**adresse indexée comporte un déplacement qui est ajouté à la valeur de l'index.** Dans l'instruction :

ADD A,(IX+4ØH)

l'adresse mémoire est égale à la somme du contenu du registre IX et de la valeur 4Ø hexadécimale. Soit :



Adressage direct

L'adressage direct peut être utilisé pour charger l'accumulateur avec n'importe quelle valeur 8 bits contenue en mémoire, charger BC, DE, HL, SP, IX ou IY avec n'importe quelle valeur 16 bits contenue en mémoire, et se brancher ou appeler des sous-programmes à n'importe quelle adresse mémoire. L'adresse 16 bits est stockée dans les 2 derniers octets de l'instruction, dans l'ordre octet bas-octet haut (ce qui est l'inverse de la notation classique haut-bas). L'instruction :

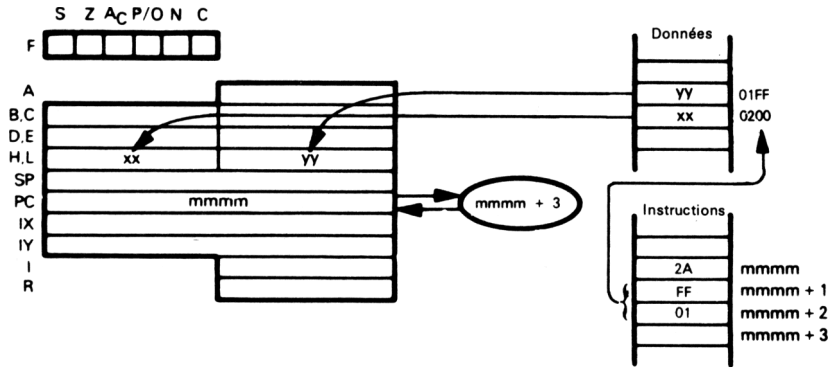
LD A,(NETX)

charge le registre A avec le contenu de la mémoire dont l'adresse est NETX. L'instruction :

LD HL,(1FFH)

charge le registre L avec le contenu de la mémoire d'adresse $1FF_{16}$ et le registre H avec le contenu de la mémoire d'adresse 200_{16}.

Ce qui correspond au traitement suivant :



LD HL,(1FFH)

7	6	5	4	3	2	1	0	
0	0	1	0	1	0	1	0	Instruction « Load HL direct »
1	1	1	1	1	1	1	1	Adresse directe (octet bas)
0	0	0	0	0	0	0	1	Adresse directe (octet haut)

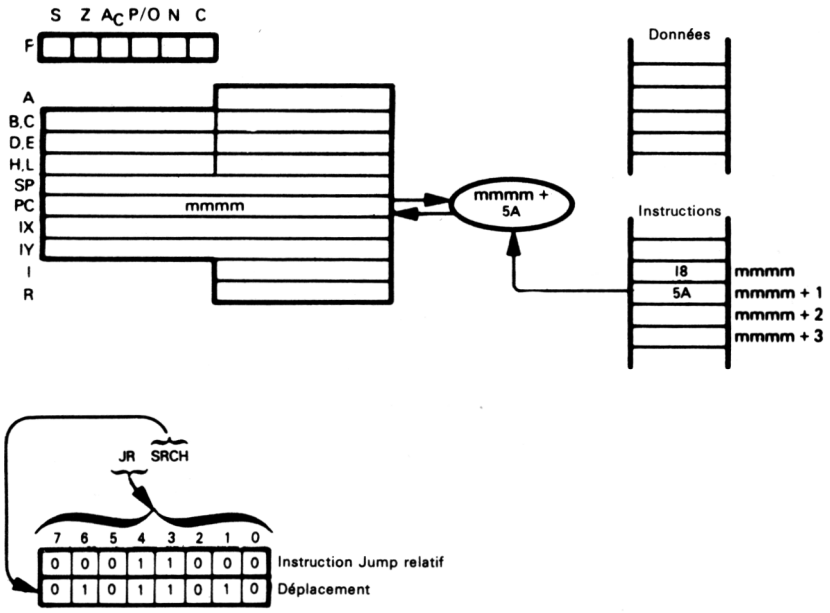
Les instructions JUMP directs permettent des branchements et des appels de sous-programmes, conditionnels ou non. Ce sont toutes des instructions à 3 octets, l'adresse directe étant contenue dans les deuxième et troisième octets de l'instruction.
Il existe 3 autres modes d'adressage utilisés dans les instructions de débranchement du Z80 : relatif, basé et indirect. En général, ils prennent moins de place et/ou moins de temps d'exécution que les adressages directs, mais ont souvent des capacités d'adressage plus restreintes.

Adressage relatif

Les instructions JUMP à adressage relatif permettent des sauts (branchements) à des adresses situées au plus 126 octets avant ou 129 octets après le 1er octet de l'instruction JUMP. Ces instructions sont toutes des instructions à 2 octets, la valeur signée du déplacement étant stockée dans le 2e octet. **Il existe des sauts relatifs conditionnels et inconditionnels, ainsi que l'instruction de fin de boucle DJNZ (Decrement and Jump if Not Zero).**
Prenons l'instruction :

JR SRCH

où SRCH est une étiquette adressant la mémoire située 90 octets (valeur hexa. 5A) après le 1er octet de l'instruction suivante.
Cette opération peut être illustrée comme suit :



Adressage basé

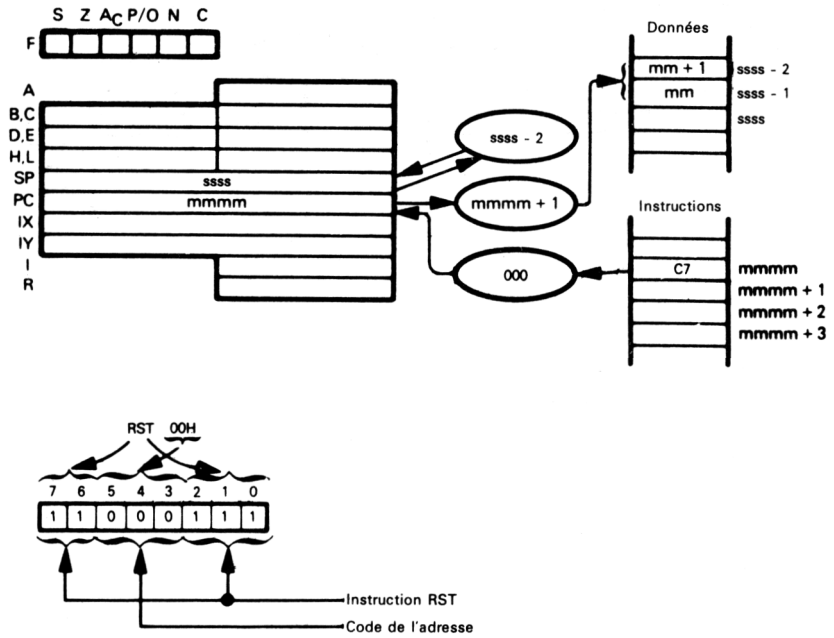
Le Z80 possède un type d'**adressage particulier** pour les instructions RST (« Restart »). Ce sont des instructions qui permettent, avec un **seul octet de code opératoire, d'appeler un sous-programme stocké dans la partie basse de la mémoire.** Elles peuvent ainsi appeler 8 sous-programmes stockés à des adresses bien définies, et précisées par 3 bits du code opératoire :

Code 3 bits	Adresse du sous-programme appelé
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

L'adresse décodée est chargée dans l'octet bas du compteur ordinal (PC), l'octet haut étant mis à zéro. Par exemple, l'instruction :

RST 00H

peut être illustrée comme suit :



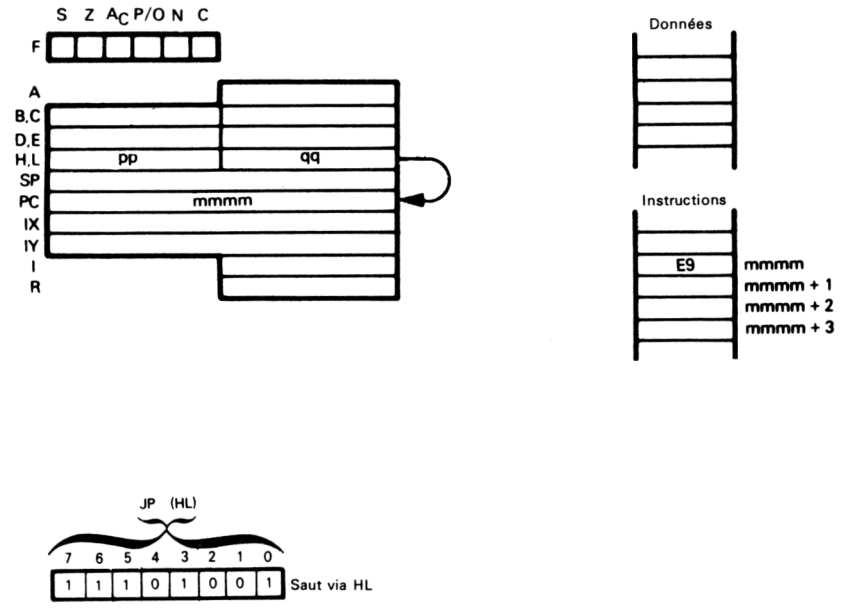
Adressage indirect

Dans ce mode d'adressage, un registre contient l'adresse d'un opérande, et l'instruction spécifie le registre contenant cette adresse. Notons que ce type d'adressage peut se comparer, par exemple pour les instructions de chargement (LD), à l'adressage dit implicite. Cependant, **le Z80 possède des instructions de saut (JUMP) qui permettent un branchement vers une mémoire dont l'adresse est contenue dans un registre (16 bits).**

Cette forme d'adressage indirect est décrite séparément, étant donné que la plupart des microprocesseurs possède un mode d'adressage implicite, alors que très peu peuvent utiliser des branchements indirects. L'instruction :

JP (HL)

implique un saut (branchement) à la mémoire dont l'adresse est contenue dans HL. Soit :



Adressage immédiat

Une instruction « immédiate » est une instruction à 2, 3 ou 4 octets dans laquelle le dernier (ou les deux derniers) octet(s) contient (contiennent) la donnée à charger dans un registre ou la mémoire. **Le Z80 possède des instructions immédiates pour :**

— le chargement d'une donnée 8 bits dans un registre 8 bits ;
— le chargement d'une donnée 16 bits dans un registre 16 bits ;
— le stockage d'une donnée 8 bits en mémoire en utilisant l'adressage implicite ou indexé ;
— l'exécution d'opérations arithmétiques ou logiques utilisant l'accumulateur et une valeur 8 bits immédiate. L'instruction :

```
LD BC,ØBCH
```

charge la valeur immédiate BC_{16} dans la paire de registres BC. Soit :

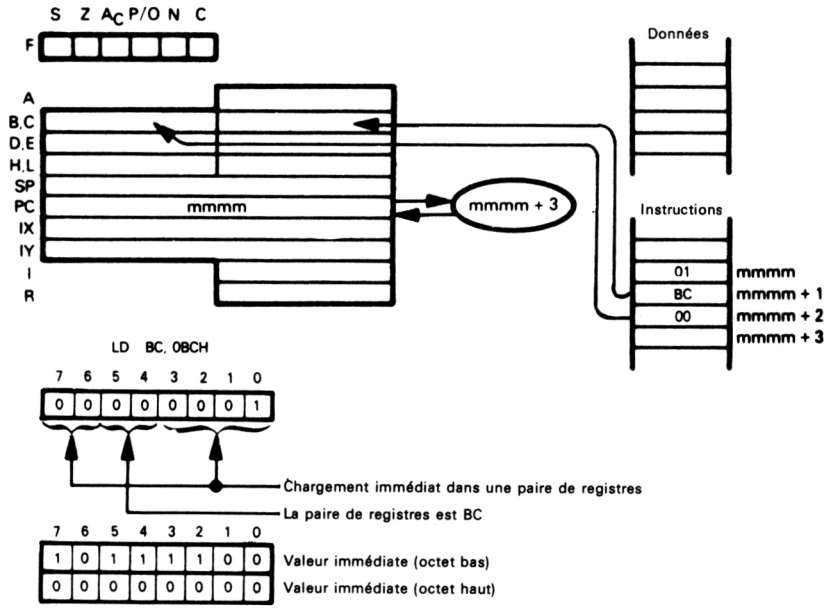


Tableau 3.1. — Instructions du Z80 les plus utilisées

Mnémonique	Signification (en anglais)	Effet
ADC A	Add with Carry to Accumulator	Addition avec retenue
ADD	Add	Addition
AND	Logical AND	ET logique
CALL addr	Call Subroutine	Appel de sous-progr.
CALL cond,addr	Call Conditional	Appel conditionnel
CP	Compare	Comparaison
DEC	Decrement	Décrémentation
DJNZ	Decrement and Jump If Not Zero	Décrémentation et test
IN	Input	Entrée
INC	Increment	Incrémentation
JR	Jump Relative	Saut relatif
JR cond,addr	Jump Relative Conditional	Saut relatif conditionnel
LD reg,(HL)	Load Register	Chargement de registre
LD A,(addr)	Load Accumulator Direct	Chargement direct de l'acc.
LD data	Load Immediate	Chargement immédiat
LD (HL),reg	Store Register	Sauvegarde de registre
LD (addr),A	Store Accumulator Direct	Sauvegarde directe de l'acc.
LD dst,src	Move Register-to-Register	Transfert registre à registre
OUT	Output	Sortie
POP	Pop from Stack	Dépilage
PUSH	Push to Stack	Empilage
RET	Return from Subroutine	Retour de sous-programme
RET cond	Return Conditional	Retour conditionnel
RLA	Rotate Accumulator Left Through Carry	Rotation à gauche
RRA	Rotate Accumulator Right Through Carry	Rotation à droite
SLA	Shift Left Arithmetic	Décalage à gauche
SRL	Shift Right Logical	Décalage à droite
SUB	Subtract	Soustraction

Tableau 3.2. — Instructions moyennement utilisées

Mnémonique		Signification (en anglais)	Effet
BIT		Test Bit	Test sur bit
CPD, CPDR		Compare, Decrement, (Repeat)	Comp., Décr. (Boucle)
CPI, CPIR		Compare, Increment, (Repeat)	Comp., Incr. (Boucle)
CPL		Complement Accumulator	Complément à 1
DAA		Decimal Adjust Accumulator	Ajustement BCD
DI		Disable Interrupts	Masquage des interruptions
EI		Enable Interrupts	Autorisation des interruptions
EX		Exchange	Echange
HALT		Halt	Halte
IND, INDR		Input, Decrement, (Repeat)	Entrée, décrém. (Boucle)
INI, INIR		Input, Increment, (Repeat)	Entrée, incrém. (Boucle)
JP	addr	Jump	Saut
JP	cond,addr	Jump Conditional	Saut conditionnel
LD	A,(BC) or (DE)	Load Accumulator Secondary	Chargement accumulateur
LD	HL,(addr)	Load HL Direct	Chargement direct de HL
LD	reg,(xy+disp)	Load Register Indexed	Chargement indexé
LD	rp,(addr)	Load Register Pair Direct	Chargement direct 16 bits
LD	xy,(addr)	Load Index Register Direct	Chargement d'index
LD	(BC) or (DE),A	Store Accumulator Secondary	Stockage dans accumulateur
LD	(addr),HL	Store HL Direct	Stockage direct dans HL
LD	(xy+disp),reg	Store Register Indexed	Stockage indexé
LD	(addr),rp	Store Register Pair Direct	Stockage direct 16 bits
LD	(addr),xy	Store Index Register Direct	Stockage d'index
LD	(HL),data	Store Immediate to Memory	Stockage immédiat en mémoire
LD	(xy+disp),data	Store Immediate to Memory Indexed	Stockage en mémoire indexée
LDD, LDDR		Load, Decrement, (Repeat)	Chargement, décr. (Boucle)
LDI, LDIR		Load, Increment, (Repeat)	Chargement, incr. (Boucle)
NEG		Negate (Twos Complement) Accumulator	Complément à 2
NOP		No Operation	Sans effet
OR		Logical OR	OU logique
OUTD, OTDR		Output, Decrement, (Repeat)	Sortie, décrém. (Boucle)
OUTI, OTIR		Output, Increment, (Repeat)	Sortie, incrém. (Boucle)
RES		Reset Bit	Mise à zéro d'un bit

Tableau 3.2. — Instructions moyennement utilisées (suite)

Mnémonique	Signification (en anglais)	Effet
RETI	Return from Interrupt	Retour de traitement d'IT
RL	Rotate Left Through Carry	Rotation à gauche
RLC	Rotate Left Circular	Rotation circulaire gauche
RLCA	Rotate Accumulator Left Circular	Idem avec l'accumulateur
RR	Rotate Right Through Carry	Rotation à droite
RRC	Rotate Right Circular	Rotation circulaire droite
RRCA	Rotate Accumulator Right Circular	Idem avec l'accumulateur
SET	Set Bit	Mise à 1 d'un bit
SRA	Shift Right Arithmetic	Division par 2 (décalage)
XOR	Logical Exclusive OR	OU exclusif

Tableau 3.3. — Instructions les moins usitées

Mnémonique	Signification (en anglais)	Effet
ADC HL,rp	Add Register Pair with Carry to HL	Addition 16 bits avec HL
CCF	Complement Carry Flag	Complémentation indicateur C
EXX	Exchange Register Pairs and Alternatives	Echanges avec registres auxil.
IM n	Set Interrupt Mode	Choix du mode d'interruption
RETN	Return from Non-Maskable Interrupt	Retour d'IT non masquable
RLD	Rotate Accumulator and Memory Left Decimal	Rotation gauche BCD
RRD	Rotate Accumulator and Memory Right Decimal	Rotation droite BCD
RST	Restart	Appelle sous-prog. privilégié
SBC	Subtract with Carry (Borrow)	Soustraction avec retenue
SCF	Set Carry Flag	Mise à 1 de l'indicateur C
LD A,I	Load Accumulator from Interrupt Vector Register	Transfert I vers acc.
LD A,R	Load Accumulator from Refresh Register	Transfert R vers acc.
LD I,A	Store Accumulator to Interrupt Vector Register	Transfert acc. vers I
LD R,A	Store Accumulator to Refresh Register	Transfert acc. vers R
LD SP,HL	Move HL to Stack Pointer	Chargement HL vers pointeur
LD SP,xy	Move Index Register to Stack Pointer	Chargement IX ou IY vers pointeur

Abréviations utilisées dans ce chapitre :

A,F,B,C,D,E,H,L	Registres 8 bits. A est l'accumulateur, F le registre d'état
AF',BC',DE',HL'	Paires de registres auxiliaires
addr	Adresse 16 bits
x(b)	Bit n° b d'un registre ou d'une mémoire x
Cond	Condition de débranchement. A savoir :

NZ	- Non-Zéro (Z = 0)
Z	- Zéro (Z = 1)
NC	- Non-Carry (C = 0)
C	- Carry (C = 1)
PO	- Parité impaire (P = 0)
PE	- Parité paire (P = 1)
P	- Positif (S = 0)
M	- Négatif (S = 1)

data	Valeur 8 bits
data16	Valeur 16 bits
disp	Déplacement
xx(HI)	Octet haut d'une valeur 16 bits
xx(LO)	Octet bas d'une valeur 16 bits
I	Registre d'interruption
IX,IY	Registres d'index (16 bits chacun)
label	Etiquette (adresse 16 bits)
LSB	Bit le moins significatif (Bit 0)
MSB	Bit le plus significatif (Bit 7)
PC	Compteur ordinal
port	Adresse d'un port (8 bits)
pr	N'importe laquelle des paires de registres BC, DE, HL ou AF
R	Registre de rafraîchissement
reg	N'importe lequel des registres 8 bits A, B, C, D, E, H ou L
rp	N'importe laquelle des paires de registres BC, DE, HL ou SP
SP	Pointeur de pile (16 bits)
xy	IX ou IY (registres d'index)

Code objet :

ccc	code condition :	000 = Non-Zéro 001 = Zéro 010 = Non-Carry (pas de retenue dans C) 011 = Carry (retenue dans C) 100 = Parité impaire 101 = Parité paire 110 = Signe positif 111 = Signe négatif
ddd	registre destination	(cf. rrr)
ppqq	adresse 16 bits	
rrr	code de registre :	111 = A 000 = B 001 = C 010 = D 011 = E 100 = H 101 = L
sss	registre source	(cf. rrr)
x	registre d'index	0 = IX 1 = IY
xx	paire de registres	00 = BC 01 = DE 10 = HL 11 = SP (rp) ou AF (pr)
xxx	code de restart	(000 à 111) cf. instruction RST
yy	donnée 8 bits	
yyyy	donnée 16 bits	

Indicateurs (registre F)

C	- indication de retenue
Z	- indication de résultat nul
S	- indication de signe
P/O	- indication de parité ou de dépassement
AC	- indication de demi-retenue
N	- indication de soustraction

Les symboles suivants sont utilisés dans les colonnes indicateurs :

X	- l'indicateur est positionné en fonction du résultat
(espace)	- l'indicateur reste inchangé
1	- l'indicateur est positionné (à 1)
0	- l'indicateur est ré-initialisé (à 0)
U	- l'indicateur prend une valeur aléatoire
P	- l'indicateur donne la parité du résultat
O	- l'indicateur signale un éventuel dépassement de capacité
I	- l'indicateur signale le masquage éventuel d'interruptions

Adressage

[[ ]] 1) Contenu de la mémoire dont l'adresse est contenue dans le registre spécifié.
2) Port d'E/S dont l'adresse est contenue dans le registre spécifié.

	[ ]	Contenu d'un registre ou d'une mémoire Exemples : [[HL]] ← [[HL]] + 1 indique que le contenu de la mémoire adressée par HL et incrémenté. [HL] ← [HL] + 1 indique que le contenu de HL est incrémenté.
Opérations	Λ	ET logique
	V	OU logique
	⊻	OU exclusif
	←	Transfert de donnée dans le sens de la flèche
	←→	Echange de données

LES INSTRUCTIONS EN ASSEMBLEUR (MNÉMONIQUES)

Le tableau 3.4 présente le jeu d'instructions du Z80. La colonne MNÉMONIQUE donne la liste des mnémoniques (IN, OUT, LD,...). La colonne OPÉRANDE donne le type des éventuels opérandes pour l'instruction correspondante. La partie fixe d'une instruction est notée en MAJUSCULES. La partie variable (valeur immédiate, n° de port d'E/S, nom de registre, étiquette ou adresse) est notée en minuscules.
La liste des différents opérandes sera donnée sans répéter le mnémonique. Par exemple,

```
LD rp,(addr)
   xy,(addr)
```

correspondent à des instructions comme :

```
LD BC,(DATZ)
LD IX,(MEM)
```

LES INSTRUCTIONS EN CODE OBJET

Le code objet et la longueur en octets correspondant à chaque instruction sont donnés dans le tableau 3.4. Le tableau 3.5 donne la liste des codes objets classés par ordre numérique. Pour les codes n'ayant pas de partie variable, la valeur de l'octet est donnée sous la forme de 2 chiffres hexadécimaux (par ex. 3F).
Pour les codes dont l'un des chiffres hexadécimaux varie, la valeur est donnée sous la forme de 4 chiffres binaires plus un chiffre hexadécimal (par ex. 11x1D).
Pour les autres codes variables, la valeur est représentée par 8 chiffres binaires (par ex. 01ss001).

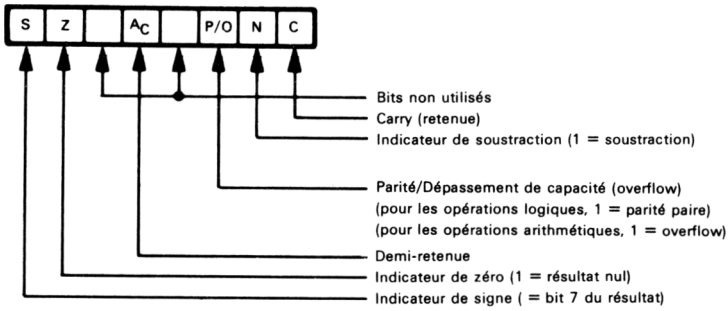
LES TEMPS D'EXECUTION

Le tableau 3.4 donne la liste des temps d'exécution en nombre de périodes d'horloge. Le temps réel sera obtenu en divisant le nombre donné par la fréquence de l'horloge. Soit, pour une instruction qui nécessite 7 périodes, un temps d'exécution de 1,75 μsec. à une fréquence de 4 MHz.

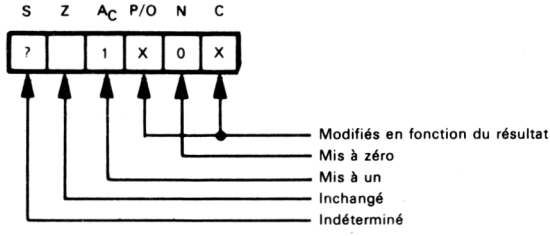
Quand deux temps d'exécution sont donnés (par ex. 5/11), cela signifie que le nombre de périodes dépend en fait des indicateurs (flags) de condition. La première valeur correspond à « Condition non remplie », la seconde à « Condition remplie ».

LES INDICATEURS

Les 6 indicateurs sont positionnés dans le registre d'état (F) comme suit :



Pour la description de chaque instruction, l'effet de l'exécution sur le registre d'état est illustré comme suit :



CHANGEMENTS D'ETATS FONCTIONS DE L'INSTRUCTION EXECUTEE

Tableau 3.4. — Jeu d'instructions du Z80

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état						Effet
						C	Z	S	P/O	A_C	N	
E/S	IN	A,(port)	DB yy	2	10							[A] ← [port] Bus d'adresses : $A_0 - A_7$ = port $A_8 - A_{15}$ = [A]
	IN	reg.(C)	ED 01ddd000	2	11		X	X	P	X	0	[reg] ← [[C]] Bus d'adresses : $A_0 - A_7$ = [C] $A_8 - A_{15}$ = [B]
	INIR		ED B2	2	20/15**		1	?	?	?	1	[[HL]] ← [[C]] [B] ← [B − 1] [HL] ← [HL] + 1 Bus d'adresses : $A_0 - A_7$ = [C] $A_8 - A_{15}$ = [B]
	INDR		ED BA	2	20/15**		1	?	?	?	1	Répéter jusqu'à ce que [B] = 0 : [[HL]] ← [[C]] [B] ← [B] − 1 [HL] ← [HL] − 1 Bus d'adresses : $A_0 - A_7$ = [C] $A_8 - A_{15}$ = [B]
	INI		ED A2	2	15		X	?	?	?	1	[[HL]] ← [[C]] [B] ← [B] − 1 [HL] ← [HL] + 1 Bus d'adresses : $A_0 - A_7$ = [C] $A_8 - A_{15}$ = [B]

** Temps d'exécution pour 1 itération.

Tableau 3.4. — Jeu d'instructions du Z80 (suite)

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état						Effet
						C	Z	S	P/O	A_C	N	
E/S (suite)	IND		ED AA	2	15		X	?	?	?	1	[[HL]] ← [[C]] [B] ← [B] − 1 [HL] ← [HL] − 1 Bus d'adresses : $A_0 - A_7$ = [C] $A_8 - A_{15}$ = [B]
	OUT	(port),A	D3 yy	2	11							[port] ← [A] Bus d'adresses : $A_0 - A_7$ = port $A_8 - A_{15}$ = [A]
	OUT	(C),reg	ED 01sss001	2	12							[[C]] ← reg Bus d'adresses : $A_0 - A_7$ = [C] $A_8 - A_{15}$ = [B]
	OTIR		ED B3	2	20/15**		1	?	?	?	1	Répéter jusqu'à ce que [B] = 0 : [[C]] ← [[HL]] [B] ← [B] − 1 [HL] ← [HL] + 1 Bus d'adresses : $A_0 - A_7$ = [C] $A_8 - A_{15}$ = [B]
	OTDR		ED BB	2	20/15**		1	?	?	?	1	Répéter jusqu'à ce que [B] = 0 [[C]] ← [[HL]] [B] ← [B] − 1 [HL] ← [HL] − 1 Bus d'adresses : $A_0 - A_7$ = [C] $A_8 - A_{15}$ = [B]

** Temps d'exécution pour 1 itération.

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état						Effet
						C	Z	S	P/O	A_C	N	
E/S (suite)	OUTI		ED A3	2	15		X	?	?	?	1	[[C]] ← [[HL]] [B] ← [B] - 1 [HL] ← [HL] + 1 Bus d'adresses : $A_0 - A_7$ = [C] $A_8 - A_{15}$ = [B]
	OUTD		ED AB	2	15		X	?	?	?	1	[[C]] ← [[HL]] [B] ← [B] - 1 [HL] ← [HL] - 1 Bus d'adresses : $A_0 - A_7$ = [C] $A_8 - A_{15}$ = [B]
Transferts à référence mémoire	LD	A,(addr)	3A ppqq	3	13							[A] ← [addr]
	LD	HL,(addr)	2A ppqq	3	16							[H] ← [addr + 1], [L] ← [addr]
	LD	rp,(addr)	ED 01xx1011 ppqq	4	20							[rp(HI)] ← [addr + 1], [rp(LO)] ← [addr] or
		xy,(addr)	11x11101 2A ppqq	4	20							[xy(HI)] ← [addr + 1], [xy(LO)] ← [addr]
	LD	(addr),A	32 ppqq	3	13							[addr] ← [A]
	LD	(addr),HL	22 ppqq	3	16							[addr + 1] ← [H], [addr] ← [L]
	LD	(addr),rp	ED 01xx0011 ppqq	4	20							[addr + 1] ← [rp(HI)], [addr] ← [rp(LO)] or
		(addr),xy	11x11101 22 ppqq	4	20							[addr + 1] ← [xy(HI)], [addr] ← [xy(LO)]
	LD	A,(BC)	0A	1	7							[A] ← [[BC]] or [A] ← [[DE]]
		A,(DE)	1A	1	7							

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état C	Z	S	P/O	A_C	N	Effet
Transferts mémoire (suite)	LD	reg,(HL)	01ddd110	1	7							[reg] ← [[HL]]
	LD	(BC),A	02	1	7							[[BC]] ← [A] or [[DE]] ← [A]
		(DE),A	12	1	7							
	LD	(HL),reg	01110sss	1	7							[[HL]] ← [reg]
	LD	reg,(xy+disp)	11x11101 01ddd110 disp	3	19							[reg] ← [[xy] + disp]
	LD	(xy+disp),reg	11x11101 01110sss disp	3	19							[[xy] + disp] ← [reg]
Transferts ou scrutations de blocs	LDIR		ED B0	2	20/16**				0	0	0	Répéter jusqu'à ce que [BC] = 0 : [[DE]] ← [[HL]] [DE] ← [DE] + 1 [HL] ← [HL] + 1 [BC] ← [BC] - 1
	LDDR		ED B8	2	20/16**				0	0	0	Répéter jusqu'à ce que [BC] = 0 : [[DE]] ← [[HL]] [DE] ← [DE] - 1 [HL] ← [HL] - 1 [BC] ← [BC] - 1

** Temps d'exécution pour 1 itération.

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état						Effet
						C	Z	S	P/O	A_C	N	
Transferts ou scrutations de blocs (suite)	LDI		ED A0	2	16				X	0	0	[[DE]] ← [[HL]] [DE] ← [DE] + 1 [HL] ← [HL] + 1 [BC] ← [BC] - 1
	LDD		ED A8	2	16				X	0	0	[[DE]] ← [[HL]] [DE] ← [DE] - 1 [HL] ← [HL] - 1 [BC] ← [BC] - 1
	CPIR		ED B1	2	20/16**		X	X	X	X	1	Répéter jusqu'à ce que [A] = [[HL]] ou [BC] = 0 [A] - [[HL]] (seuls les indicateurs sont modifiés) [HL] ← [HL] + 1 [BC] ← [BC] - 1
	CPDR		ED B9	2	20/16**		X	X	X	X	1	Répéter jusqu'à ce que [A] = [[HL]] ou [BC] = 0 [A] - [[HL]] (seuls les indicateurs sont modifiés) [HL] ← [HL] - 1 [BC] ← [BC] - 1

** Temps d'exécution pour 1 itération.

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état						Effet
						C	Z	S	P/O	A_C	N	
	CPI		ED A1	2	16		X	X	X	X	1	[A] - [[HL]] (seuls les indicateurs sont modifiés) [HL] ← [HL] + 1 [BC] ← [BC] - 1
	CPD		ED A9	2	16		X	X	X	X	1	[A] - [[HL]] (seuls les indicateurs sont modifiés) [HL] ← [HL] - 1 [BC] ← [BC] - 1
Opérations arithmétiques et logiques	ADD	A,(HL) A,(xy +disp)	86 11x11101 86 disp	1 3	7 19	X	X	X	O	X	0	[A] ← [A] + [[HL]] or [A] ← [A] + [[xy] + disp]
	ADC	A,(HL) A,(xy +disp)	8E 11x11101 8E disp	1 3	7 19	X	X	X	O	X	0	[A] ← [A] + [[HL]] + C or [A] ← [A] + [[xy] + disp] + C
	SUB	(HL) (xy + disp)	96 11x11101 96 disp	1 3	7 19	X	X	X	O	X	1	[A] ← [A] - [[HL]] or [A] ← [A] - [[xy] + disp]
	SBC	A,(HL) A,(xy+disp)	9E 11x11101 9E disp	1 3	7 19	X	X	X	O	X	1	[A] ← [A] - [[HL]] - C or [A] ← [A] - [[xy] + disp] - C
	AND	(HL) (xy + disp)	A6 11x11101 A6 disp	1 3	7 19	0	X	X	P	1	0	[A] ← [A] $\wedge$ [[HL]] or [A] ← [A] $\wedge$ [[xy] + disp]
	OR	(HL) (xy + disp)	B6 11x11101 B6 disp	1 3	7 19	0	X	X	P	1	0	[A] ← [A] V [[HL]] or [A] ← [A] V [[xy] + disp]

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

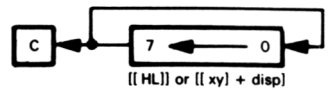
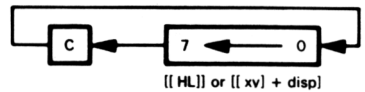
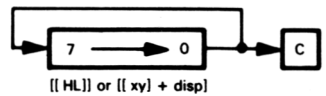
Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état C	Z	S	P/O	A_C	N	Effet
	XOR	(HL)	AE	1	7	0	X	X	P	1	0	[A] ← [A] ⊻ [[HL]] or [A] ← [A] ⊻ [[xy] + disp]
		(xy + disp)	11x11101 AE disp	3	19							
	CP	(HL)	BE	1	7	X	X	X	O	X	1	[A] - [[HL]] or [A] - [[xy] + disp]
		(xy + disp)	11x11101 BE disp	3	19							
												Seuls les indicateurs (flags) sont modifiés
	INC	(HL)	34	1	11		X	X	O	X	0	[[HL]] ← [[HL]] + 1 or [[xy] + disp] ← [[xy] + disp] + 1
		(xy + disp)	11x11101 34 disp	3	23							
	DEC	(HL)	35	1	11		X	X	O	X	1	[[HL]] ← [[HL]] - 1 or [[xy] + disp] ← [[xy] + disp] - 1
		(xy + disp)	11x11101 35 disp	3	23							
Décalages et rotations	RLC	(HL)	CB 06	2	15	X	X	X	P	0	0	 [[HL]] or [[xy] + disp]
		(xy + disp)	11x11101 CB disp 06	4	23							
	RL	(HL)	CB 16	2	15	X	X	X	P	0	0	 [[HL]] or [[xv] + disp]
		(xy + disp)	11x11101 CB disp 16	4	23							
	RRC	(HL)	CB 0E	2	15	X	X	X	P	0	0	 [[HL]] or [[xy] + disp]
		(xy + disp)	11x11101 CB disp 0E	4	23							

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

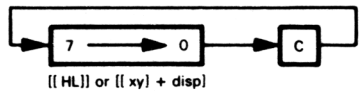
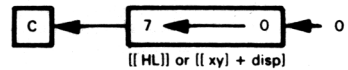
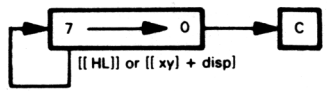
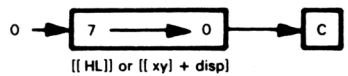
Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état C	Z	S	P/O	A_C	N	Effet
Décalages et rotations (suite)	RR	(HL) (xy + disp)	CB 1E 11x11101 CB disp 1E	2 4	15 23	X	X	X	P	0	0	
	SLA	(HL) (xy + disp)	CB 26 11x11101 CB disp 26	2 4	15 23	X	X	X	P	0	0	 Décalage arithmétique
	SRA	(HL) (xy + disp)	CB 2E 11x11101 CB disp 2E	2 4	15 23	X	X	X	P	0	0	 Décalage arithmétique
	SRL	(HL) (xy + disp)	CB 3E 11x11101 CB disp 3E	2 4	15 23	X	X	X	P	0	0	 Décalage logique

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état C	Z	S	P/O	A_C	N	Effet
Immédiat	LD	reg,data	00ddd110 yy	2	7							[reg] ← data
	LD	rp,data16	00xx0001 yyyy	3	10							[rp] ← data16 or [xy] ← data16
		xy,data16	11x11101 21 yyyy	4	14							
	LD	(HL),data	36 yy	2	10							[[HL]] ← data or [[xy] + disp] ← data
		(xy + disp), data	11x11101 36 disp yy	4	19							
Saut	JP	label	C3 ppqq	3	10							[PC] ← label
	JR	disp	18 (disp-2)	2	12							[PC] ← [PC] + 2 + (disp-2)
	JP	(HL)	E9	1	4							[PC] ← [HL] or [PC] ← [xy]
		(xy)	11x11101 E9	2	8							
Sous-programmes	CALL	label	CD ppqq	3	17							[[SP] - 1] ← [PC(HI)] [[SP] - 2] ← [PC(LO)] [SP] ← [SP] - 2 [PC] ← label
	CALL	cond,label	11ccc100 ppqq	3	10/17							Idem que l'instruction précédente si la condition est satisfaite. Branchement en séquence sinon.
	RET		C9	1	10							[PC(LO)] ← [[SP]] [PC(HI)] ← [[SP] + 1] [SP] ← [SP] + 2
	RET	cond	11ccc000	1	5/11							Idem que l'instruction précédente si la condition est satisfaite. Branchement en séquence sinon.

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état						Effet
						C	Z	S	P/O	A_C	N	
Opérations immédiates	ADD	A,data	C6 yy	2	7	X	X	X	O	X	0	[A] ← [A] + data
	ADC	A,data	CE yy	2	7	X	X	X	O	X	0	[A] ← [A] + data + C
	SUB	data	D6 yy	2	7	X	X	X	O	X	1	[A] ← [A] - data
	SBC	A,data	DE yy	2	7	X	X	X	O	X	1	[A] ← [A] - data - C
	AND	data	E6 yy	2	7	0	X	X	P	1	0	[A] ← [A] $\wedge$ data
	OR	data	F6 yy	2	7	0	X	X	P	1	0	[A] ← [A] $\vee$ data
	XOR	data	EE yy	2	7	0	X	X	P	1	0	[A] ← [A] $\veebar$ data
	CP	data	FE yy	2	7	X	X	X	O	X	1	[A] - data Seuls les indicateurs (flags) sont modifiés.
Sauts conditionnels	JP	cond,label	11ccc010 ppqq	3	10							Si condition vraie, alors [PC] ← label
	JR	C,disp	38 (disp-2)	2	7/12							Si C = 1, alors [PC] ← [PC] + 2 + (disp - 2)
	JR	NC,disp	30 (disp-2)	2	7/12							Si C = 0, alors [PC] ← [PC] + 2 + (disp -2)
	JR	Z,disp	28 (disp-2)	2	7/12							Si Z = 1, alors [PC] ← [PC] + 2 + (disp -2)
	JR	NZ,disp	20 (disp-2)	2	7/12							Si Z = 0, alors [PC] ← [PC] + 2 + (disp -2)
	DJNZ	disp	10 (disp-2)	2	8/13							[B] ← [B] - 1 Si B] ≠ 0, alors [PC] + 2 + (disp -2)

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état						Effet
						C	Z	S	P/O	A_C	N	
Transferts registre à registre	LD	dst,src	01dddsss	1	4							[dst] ← [src] src et dst peuvent être A, B, C, D, E, H ou L
	LD	A,I	ED 57	2	9		X	X	I	0	0	[A] ← [I]
	LD	A,R	ED 5F	2	9		X	X	I	0	0	[A] ← [R]
	LD	I,A	ED 47	2	9							[I] ← [A]
	LD	R,A	ED 4F	2	9							[R] ← [A]
	LD	SP,HL	F9	1	6							[SP] ← [HL]
	LD	SP,xy	11x11101 F9	2	10							[SP] ← [xy]
	EX	DE,HL	EB	1	4							[DE] ← → [HL]
	EX	AF,AF'	08	1	4							[AF] ← → [AF']
	EXX		D9	1	4							$\begin{pmatrix}[BC]\\[DE]\\[HL]\end{pmatrix} \leftarrow\rightarrow \begin{pmatrix}[BC']\\[DE']\\[HL']\end{pmatrix}$

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état						Effet
						C	Z	S	P/O	A_C	N	
Opérations registre à registre	ADD	A,reg	10000rrr	1	4	X	X	X	O	X	0	[A] ← [A] + [reg]
	ADC	A,reg	10001rrr	1	4	X	X	X	O	X	0	[A] ← [A] + [reg] + C
	SUB	reg	10010rrr	1	4	X	X	X	O	X	1	[A] ← [A] - [reg]
	SBC	A,reg	10011rrr	1	4	X	X	X	O	X	1	[A] ← [A] - [reg] - C
	AND	reg	10000rrr	1	4	0	X	X	P	1	0	[A] ← [A] Λ [reg]
	OR	reg	10110rrr	1	4	0	X	X	P	1	0	[A] ← [A] V [reg]
	XOR	reg	10101rrr	1	4	0	X	X	P	1	0	[A] ← [A] ⊻ [reg]
	CP	reg	10111rrr	1	4	X	X	X	O	X	1	[A] - [reg] Seuls les indicateurs (flags) sont modifiés.
	ADD	HL,rp	00xx1001	1	11	X				?	0	[HL] ← [HL] + [rp]
	ADC	HL,rp	ED 01xx1010	2	15	X	X	X	O	?	0	[HL] ← [HL] + [rp] + C
	SBC	HL,rp	ED 01xx0010	2	15	X	X	X	O	?	1	[HL] ← [HL] - [rp] - C
	ADD	IX,pp	DD 00xx1001	2	15	X				?	0	HL. [IX] ← [IX] + [pp] (pp = BC, DE, IX, SP).
	ADD	IY,rr	FD 00xx1001	2	15	X				?	0	[IY] ← [IY] + [rr] (rr = BC, DE, IY, SP).

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

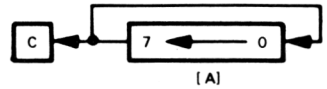
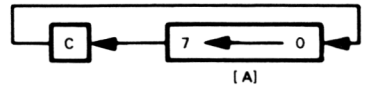
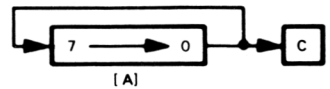
Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état						Effet
						C	Z	S	P/O	A_C	N	
Opérations sur registres	DAA		27	1	4	X	X	X	P	X		Conversion d'un résultat (somme ou différence) sous forme BCD.
	CPL		2F	1	4					1	1	[A] ← [$\overline{A}$] Complémentation à 1
	NEG		ED 44	2	8	X	X	X	O	X	1	[A] ← [$\overline{A}$] + 1 Complémentation à 2
	INC	reg	00rrr100	1	4		X	X	O	X	0	[reg] ← [reg] + 1
	INC	rp	00xx0011	1	6							[rp] ← [rp] + 1 or [xy] ← [xy] + 1
		xy	11x11101 23	2	10							
	DEC	reg	00rrr101	1	4		X	X	O	X	1	[reg] ← [reg] - 1
	DEC	rp	00xx1011	1	6							[rp] ← [rp] - 1 or [xy] ← [xy] - 1
		xy	11x11101 2B	2	10							
Décalage et rotations (registre)	RLCA		07	1	4	X				0	0	
	RLA		17	1	4	X				0	0	
	RRCA		0F	1	4	X				0	0	

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

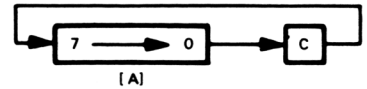
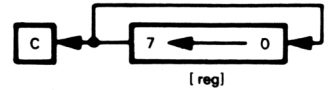
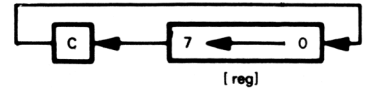
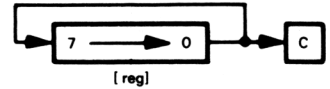
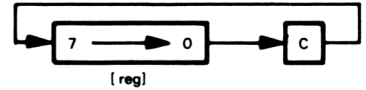
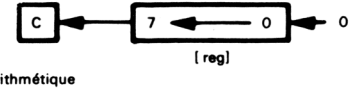
Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état C	Z	S	P/O	A_C	N	Effet
Décalages et rotations (registre)	RRA		1F	1	4	X				0	0	
	RLC	reg	CB 00000rrr	2	8	X	X	X	P	0	0	
	RL	reg	CB 00010rrr	2	8	X	X	X	P	0	0	
	RRC	reg	CB 00001rrr	2	8	X	X	X	P	0	0	
	RR	reg	CB 00011rrr	2	8	X	X	X	P	0	0	
	SLA	reg	CB 00100rrr	2	8	X	X	X	P	0	0	 Décalage arithmétique

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

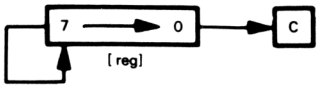
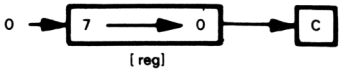
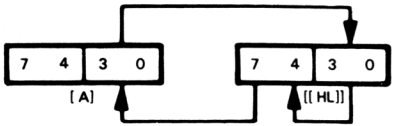
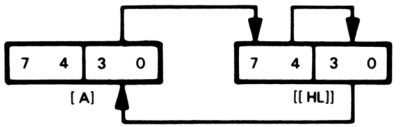
Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état C	Z	S	P/O	A_C	N	Effet
Décalages et rotations (registre)	SRA	reg	CB 00101rrr	2	8	X	X	X	P	0	0	 Décalage arithmétique
	SRL	reg	CB 00111rrr	2	8	X	X	X	P	0	0	 Décalage logique
	RLD		ED 6F	2	18		X	X	P	0	0	 Rotations de chiffres BCD
	RRD		ED 67	2	18		X	X	P	0	0	 Rotations de chiffres BCD

Tableau 3.4. — Jeu d'instructions du Z80 (suite).

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état						Effet
						C	Z	S	P/O	A_C	N	
Manipulations de bit	BIT	b,reg	CB 01bbbrrr	2	8		X	?	?	1	0	Z ← $\overline{\text{reg(b)}}$
	BIT	b,(HL)	CB 01bbb110	2	12		X	?	?	1	0	Z ← $\overline{\text{[[HL]](b)}}$ or Z ← $\overline{\text{[[xy] + disp](b)}}$
		b,(xy + disp)	11x11101 CB disp 01bbb110	4	20							
	SET	b,reg	CB 11bbbrrr	2	8							reg(b) ← 1
	SET	b,(HL)	CB 11bbb110	2	15							[[HL]](b) ← 1 or [[xy] + disp](b) ← 1
		b,(xy + disp)	11x11101 CB disp 11bbb110	4	23							
	RES	b,reg	CB 10bbbrrr	2	8							reg(b) ← 0
	RES	b,(HL)	CB 10bbb110	2	15							[[HL]](b) ← 0 or [[xy] + disp](b) ← 0
		b,(xy + disp)	11x11101 CB disp 10bbb110	4	23							
Pile	PUSH	pr	11xx0101	1	11							[[SP]-1] ← [pr(HI)] [[SP]-2] ← [pr(LO)] [SP] ← [SP]-2
		xy	11x11101 E5	2	15							
	POP	pr	11xx0001	1	10							[pr(LO)] ← [[SP]] [pr(HI)] ← [[SP] + 1] [SP] ← [SP] + 2
		xy	11x11101 E1	2	14							
	EX	(SP),HL	E3	1	19							[H] ←→ [[SP] + 1] [L] ←→ [[SP]]
		(SP),xy	11x11101 E3	2	23							

Tableau 3.4. — Jeu d'instructions du Z80 (suite et fin).

Type	Mnémonique	Opérande	Code objet	Octets	Cycles d'horloge	Registre d'état						Effet
						C	Z	S	P/O	A_C	N	
Interruptions	DI		F3	1	4							Inhibe les interruptions masquables
	EI		FB	1	4							Autorise les interruptions
	RST	n	11xxx111	1	11							[[SP]-1] ← [PC(HI)] [[SP]-2] ← [PC(LO)] [SP] ← [SP]-2 [PC] ← $(8 \cdot n)_{16}$
	RETI		ED 4D	2	14							Retour d'une interruption
	RETN		ED 45	2	14							Retour d'une interruption non masquable
	IM	0	ED 46	2	8							Mode d'interruption utilisée = 0, 1, 2
		1	ED 56	2	8							
		2	ED 5E	2	8							
Flags	SCF		37	1	4	1				0	0	C ← 1
	CCF		3F	1	4	X				?	0	C ← $\overline{C}$
Contrôle	NOP		00	1	4							Sans effet. Les mémoires dynamiques sont rafraîchies.
	HALT		76	1	4							Arrêt de l'unité centrale (en fait, exécution d'instructions NOP pour rafraîchir des mémoires dynamiques).

Tableau 3.5. — Codes objets par ordre numérique

CODE OBJET	INSTRUCTION	
00	NOP	
01 yyyy	LD	BC,data16
02	LD	(BC),A
03	INC	BC
04	INC	B
05	DEC	B
06 yy	LD	B,data
07	RLCA	
08	EX	AF,AF'
09	ADD	HL,BC
0A	LD	A,(BC)
0B	DEC	BC
0C	INC	C
0D	DEC	C
0E yy	LD	C,data
0F	RRCA	
10 disp-2	DJNZ	disp
11 yyyy	LD	DE,data16
12	LD	(DE),A
13	INC	DE
14	INC	D
15	DEC	D
16 yy	LD	D,data
17	RLA	
18 disp-2	JR	disp
19	ADD	HL,DE
1A	LD	A,(DE)
1B	DEC	DE
1C	INC	E
1D	DEC	E
1E yy	LD	E,data
1F	RRA	
20 disp-2	JR	NZ,disp
21 yyyy	LD	HL,data16
22 ppqq	LD	(addr),HL
23	INC	HL
24	INC	H
25	DEC	H
26 yy	LD	H,data
27	DAA	
28 disp-2	JR	Z,disp
29	ADD	HL,HL
2A ppqq	LD	HL,(addr)
2B	DEC	HL
2C	INC	L
2D	DEC	L
2E	LD	L,data
2F	CPL	
30 disp-2	JR	NC,disp
31 yyyy	LD	SP,data16
32 ppqq	LD	(addr),A
33	INC	SP
34	INC	(HL)
35	DEC	(HL)
36 yy	LD	(HL),data
37	SCF	
38	JR	C,disp

CODE OBJET	INSTRUCTION	
39	ADD	HL,SP
3A ppqq	LD	A,(addr)
3B	DEC	SP
3C	INC	A
3D	DEC	A
3E yy	LD	A,data
3F	CCF	
4 0sss	LD	B,reg
46	LD	B,(HL)
4 1sss	LD	C,reg
4E	LD	C,(HL)
5 0sss	LD	D,reg
56	LD	D,(HL)
5 1sss	LD	E,reg
5E	LD	E,(HL)
6 0sss	LD	H,reg
66	LD	H,(HL)
6 1sss	LD	L,reg
6E	LD	L,(HL)
7 0sss	LD	(HL),reg
76	HALT	
7 1sss	LD	A,reg
7E	LD	A,(HL)
8 0rrr	ADD	A,reg
86	ADD	A,(HL)
8 1rrr	ADC	A,reg
8E	ADC	A,(HL)
9 0rrr	SUB	reg
96	SUB	(HL)
9 1rrr	SBC	A,reg
9E	SBC	A,(HL)
A 0rrr	AND	reg
A6	AND	(HL)
A 1rrr	XOR	reg
AE	XOR	(HL)
B 0rrr	OR	reg
B6	OR	(HL)
B 1rrr	CP	reg
BE	CP	(HL)
C0	RET	NZ
C1	POP	BC
C2 ppqq	JP	NZ,addr
C3 ppqq	JP	addr
C4 ppqq	CALL	NZ,addr
C5	PUSH	BC
C6 yy	ADD	A,data
C7	RST	00H
C8	RET	Z
C9	RET	
CA ppqq	JP	Z,addr
CB 0 0rrr	RLC	reg
CB 06	RLC	(HL)
CB 0 1rrr	RRC	reg
CB 0E	RRC	(HL)
CB 1 0rrr	RL	reg
CB 16	RL	(HL)
CB 1 1rrr	RR	reg

Tableau 3.5. — Codes objets par ordre numérique (suite)

CODE OBJET	INSTRUCTION	
CB 1E	RR	(HL)
CB 2 0rrr	SLA	reg
CB 26	SLA	(HL)
CB 2 1rrr	SRA	reg
CB 2E	SRA	(HL)
CB 3 1rrr	SRL	reg
CB 3E	SRL	(HL)
CB 01bbbrrr	BIT	b,reg
CB 01bbb110	BIT	b,(HL)
CB 10bbbrrr	RES	b,reg
CB 10bbb110	RES	b,(HL)
CB 11bbbrrr	SET	b,reg
CB 11bbb110	SET	b,(HL)
CC ppqq	CALL	Z,addr
CD ppqq	CALL	addr
CE yy	ADC	A,data
CF	RST	08H
D0	RET	NC
D1	POP	DE
D2 ppqq	JP	NC,addr
D3 yy	OUT	(port),A
D4 ppqq	CALL	NC,addr
D5	PUSH	DE
D6 yy	SUB	data
D7	RST	10H
D8	RET	C
D9	EXX	
DA ppqq	JP	C,addr
DB yy	IN	A,(port)
DC ppqq	CALL	C,addr
DD 00xx 9	ADD	IX,pp
DD 21 yyyy	LD	IX,data16
DD 22 ppqq	LD	(addr),IX
DD 23	INC	IX
DD 2A ppqq	LD	IX,(addr)
DD 2B	DEC	IX
DD 34 disp	INC	(IX + disp)
DD 35 disp	DEC	(IX + disp)
DD 36 disp yy	LD	(IX + disp),data
DD 01ddd110 disp	LD	reg,(IX + disp)
DD 1 0sss disp	LD	(IX + disp),reg
DD 86 disp	ADD	A,(IX + disp)
DD 8E disp	ADC	A,(IX + disp)
DD 96 disp	SUB	(IX + disp)
DD 9E disp	SBC	A,(IX + disp)
DD A6 disp	AND	(IX + disp)
DD AE disp	XOR	(IX + disp)
DD B6 disp	OR	(IX + disp)
DD BE disp	CP	(IX + disp)
DD CB disp 06	RLC	(IX + disp)
DD CB disp 0E	RRC	(IX + disp)
DD CB disp 16	RL	(IX + disp)
DD CB disp 1E	RR	(IX + disp)
DD CB disp 26	SLA	(IX + disp)
DD CB disp 2E	SRA	(IX + disp)
DD CB disp 3E	SRL	(IX + disp)
DD CB disp 01bbb110	BIT	b,(IX + disp)

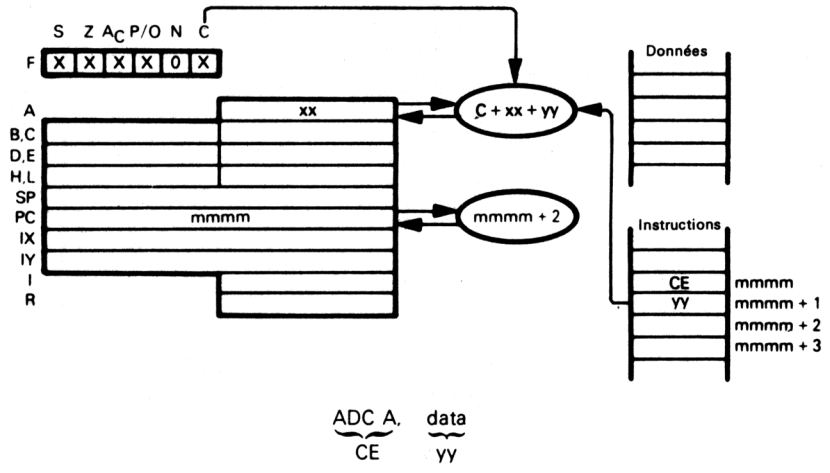
CODE OBJET	INSTRUCTION	
DD CB disp 10bbb110	RES	b,(IX + disp)
DD CB disp 11bbb110	SET	b,(IX + disp)
DD E1	POP	IX
DD E3	EX	(SP),IX
DD E5	PUSH	IX
DD E9	JP	(IX)
DD F9	LD	SP,IX
DE yy	SBC	A,data
DF	RST	18H
E0	RET	PO
E1	POP	HL
E2 ppqq	JP	PO,addr
E3	EX	(SP),HL
E4 ppqq	CALL	PO,addr
E5	PUSH	HL
E6 yy	AND	data
E7	RST	20H
E8	RET	PE
E9	JP	(HL)
EA ppqq	JP	PE,addr
EB	EX	DE,HL
EC ppqq	CALL	PE,addr
ED 01ddd000	IN	reg,(C)
ED 01sss001	OUT	(C),reg
ED 01xx 2	SBC	HL,rp
ED 01xx 3 ppqq	LD	(addr),rp
ED 44	NEG	
ED 45	RETN	
ED 010nn110	IM	n
ED 47	LD	I,A
ED 01xx A	ADC	HL,rp
ED 01xx B ppqq	LD	rp,(addr)
ED 4D	RETI	
ED 4F	LD	R,A
ED 57	LD	A,I
ED 5F	LD	A,R
ED 67	RRD	
ED 6F	RLD	
ED A0	LDI	
ED A1	CPI	
ED A2	INI	
ED A3	OUTI	
ED A8	LDD	
ED A9	CPD	
ED AA	IND	
ED AB	OUTD	
ED B0	LDIR	
ED B1	CPIR	
ED B2	INIR	
ED B3	OTIR	
ED B8	LDDR	
ED B9	CPDR	
ED BA	INDR	
ED BB	OTDR	
EE yy	XOR	data
EF	RST	28H

Tableau 3.5. — Codes objets par ordre numérique (suite et fin)

CODE OBJET	INSTRUCTION	
F0	RET	P
F1	POP	AF
F2 ppqq	JP	P,addr
F3	DI	
F4 ppqq	CALL	P,addr
F5	PUSH	AF
F6 yy	OR	data
F7	RST	30H
F8	RET	M
F9	LD	SP,HL
FA ppqq	JP	M,addr
FB	EI	
FC ppqq	CALL	M,addr
FD 00xx 9	ADD	IY,rr
FD 21 yyyy	LD	IY,data16
FD 22 ppqq	LD	(addr),IY
FD 23	INC	IY
FD 2A ppqq	LD	IY,(addr)
FD 2B	DEC	IY
FD 34 disp	INC	(IY + disp)
FD 35 disp	DEC	(IY + disp)
FD 36 disp yy	LD	(IY + disp),data
FD 01ddd110 disp	LD	reg,(IY + disp)
FD 7 0sss disp	LD	(IY + disp),reg
FD 86 disp	ADD	A,(IY + disp)

CODE OBJET	INSTRUCTION	
FD 8E disp	ADC	A,(IY + disp)
FD 96 disp	SUB	(IY + disp)
FD 9E disp	SBC	A,(IY + disp)
FD A6 disp	AND	(IY + disp)
FD AE disp	XOR	(IY + disp)
FD B6 disp	OR	(IY + disp)
FD BE disp	CP	(IY + disp)
FD CB disp 06	RLC	(IY + disp)
FD CB disp 0E	RRC	(IY + disp)
FD CB disp 16	RL	(IY + disp)
FD CB disp 1E	RR	(IY + disp)
FD CB disp 26	SLA	(IY + disp)
FD CB disp 2E	SRA	(IY + disp)
FD CB disp 3E	SRL	(IY + disp)
FD CB disp 01bbb110	BIT	b,(IY + disp)
FD CB disp 10bbb110	RES	b,(IY + disp)
FD CB disp 11bbb110	SET	b,(IY + disp)
FD E1	POP	IY
FD E3	EX	(SP),IY
FD E5	PUSH	IY
FD E9	JP	(IY)
FD F9	LD	SP,IY
FE yy	CP	data
FF	RST	38H

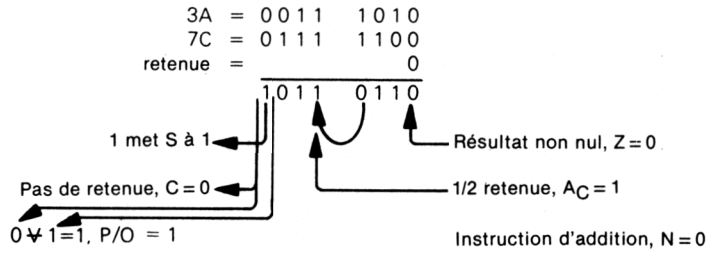
ADC A,data — ADDITION DE LA RETENUE ET D'UNE VALEUR IMMÉDIATE A L'ACCUMULATEUR



Addition du contenu de la donnée spécifiée et de la retenue (c) dans l'accumulateur. Prenons $xx = 3A_{16}$, $yy = 7C_{16}$, et supposons que la retenue = 0. A la fin de l'exécution de l'instruction :

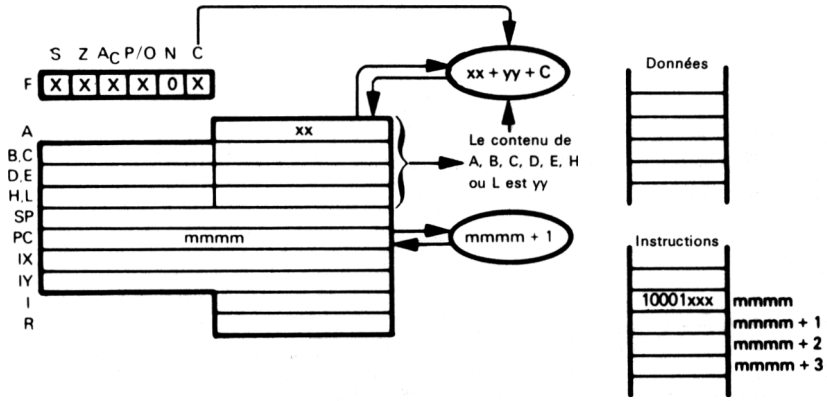
ADC A,7CH

l'accumulateur contient $B6_{16}$:



L'instruction ADC est fréquemment utilisée lors d'une addition sur plusieurs octets pour le second octet et les suivants.

ADC A,reg — ADDITION DE LA RETENUE ET D'UN REGISTRE A L'ACCUMULATEUR



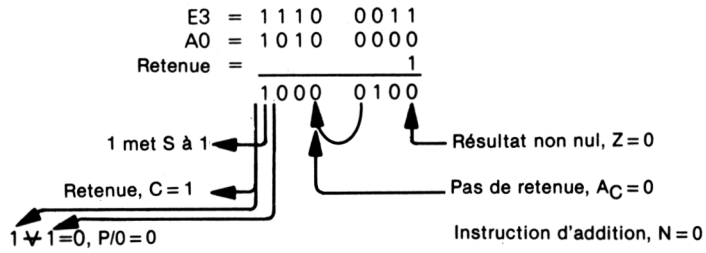
ADC A, reg
10001 xxx

000 pour le registre B
001 pour le registre C
010 pour le registre D
011 pour le registre E
100 pour le registre H
101 pour le registre L
111 pour le registre A

Addition du contenu du registre A, B, C, D, E, H ou L et de la retenue à l'accumulateur. Supposons que $xx = E3_{16}$, le registre E contient $A0_{16}$ et que la retenue = 1. A la fin de l'exécution de l'instruction

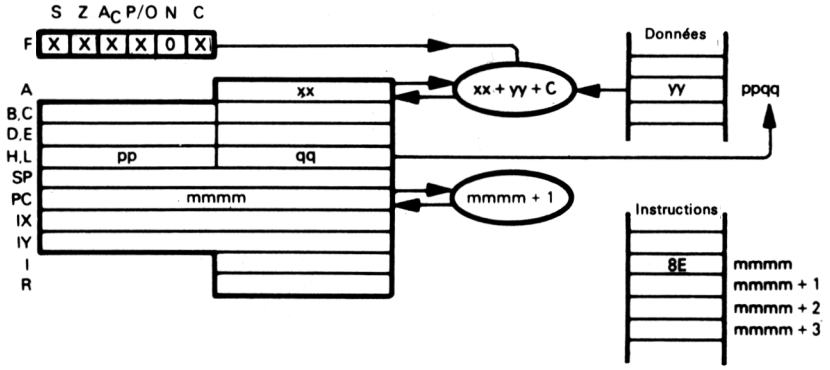
ADC A,E

l'accumulateur contient 84_{16} :



L'instruction ADC sert principalement lors d'addition sur plusieurs octets à partir du second octet.

ADC A,(HL) — ADDITION DE LA RETENUE ET D'UNE DONNÉE
ADC A,(IX + disp) EN MÉMOIRE A L'ACCUMULATEUR
ADC A,(IY + disp)



Le schéma illustre l'exécution de l'instruction ADC A,(HL) :

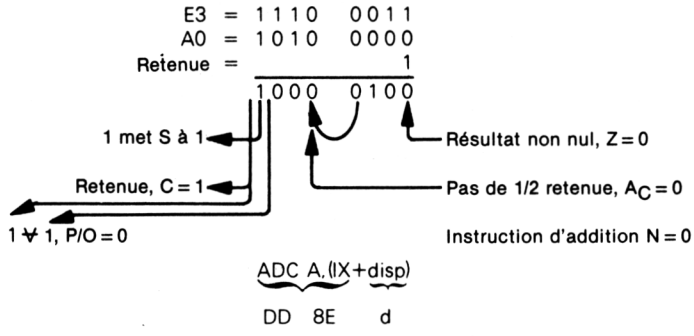
$$\underbrace{\text{ADC A,(HL)}}_{\text{8E}}$$

Addition du contenu de l'adresse (spécifiée par le contenu du double registre HL) et de la retenue à l'accumulateur.

Supposons que $xx = E3_{16}$, $yy = A0_{16}$ et la retenue = 1.
A la fin de l'exécution de l'instruction

ADC A,(HL)

l'accumulateur contient 84_{16} :



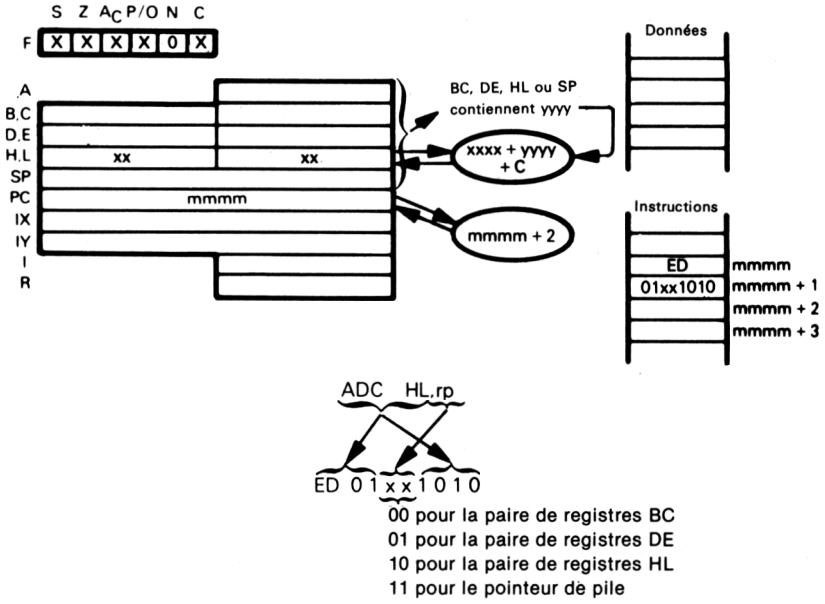
$$\underbrace{\text{ADC A,(IX+disp)}}_{\text{DD 8E d}}$$

Addition du contenu de l'adresse et de la retenue dans l'accumulateur (l'adresse résulte de la somme du contenu du registre IX avec le déplacement d).

$$\underbrace{\text{ADC A,(IY+disp)}}_{\text{FD 8E d}}$$

Cette instruction est identique à la précédente en utilisant IY à la place de IX.
L'instruction ADC sert principalement lors d'additions sur plusieurs octets à partir du second octet.

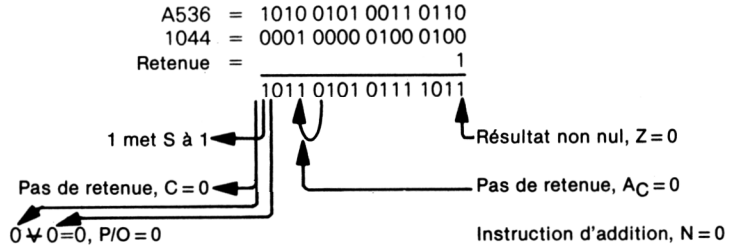
ADC HL,rp — ADDITION D'UNE PAIRE DE REGISTRES ET DE LA RETENUE A H ET L



Addition de la valeur 16 bits et de la retenue au contenu de la paire de registres HL. (La valeur 16 bits est contenue soit dans le pointeur de pile, soit dans la paire de registres BC, DE ou HL).
Supposons que HL contient $A536_{16}$, BC contient 1044_{16} et que la retenue vaut 1. A la fin de l'exécution de l'instruction

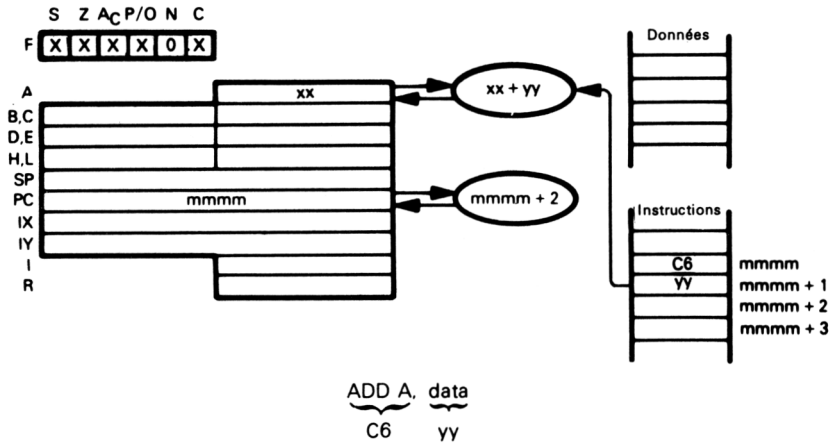
```
ADC HL, BC
```

la paire de registres HL contient :



L'instruction ADC sert principalement à l'addition sur plusieurs octets, à partir du second.

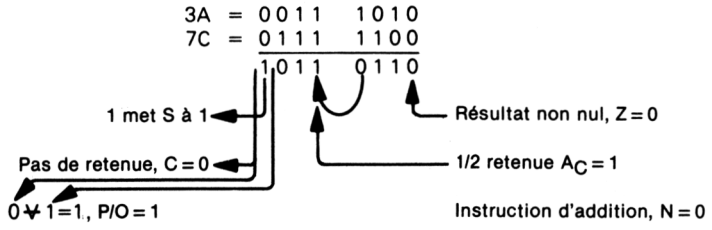
ADD A,data — ADDITION D'UNE VALEUR IMMÉDIATE A L'ACCUMULATEUR



Addition de la donnée spécifiée à l'accumulateur.
Supposons que $xx = 3A_{16}$, $yy = 7C_{16}$ et la retenue = 0.
A la fin de l'exécution de l'instruction

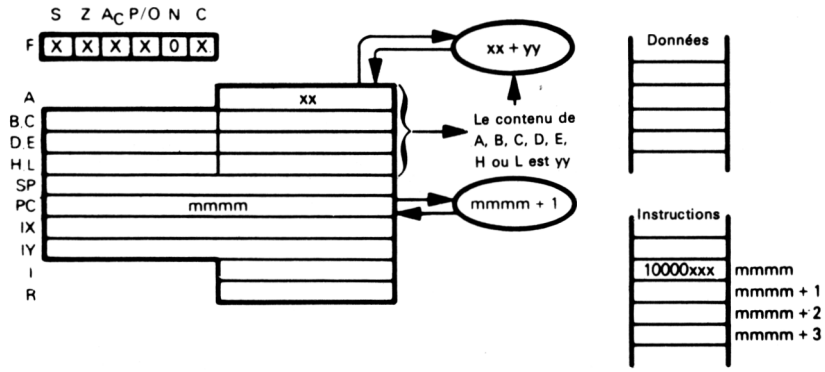
ADD A,7CH

l'accumulateur contient $B6_{16}$:



C'est une instruction classique de manipulation de données.

ADD A,reg — ADDITION DU CONTENU D'UN REGISTRE A L'ACCUMULATEUR



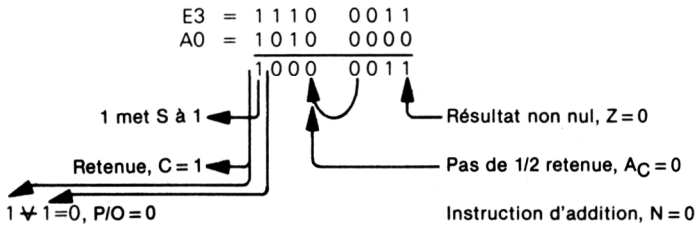
ADD reg
10000 xxx

000 pour le registre B
001 pour le registre C
010 pour le registre D
011 pour le registre E
100 pour le registre H
101 pour le registre L
111 pour le registre A

Addition du contenu du registre A, B, C, D, E, H ou L à l'accumulateur. Supposons que $xx = E3_{16}$, le registre E contient $A0_{16}$. A la fin de l'exécution de l'instruction

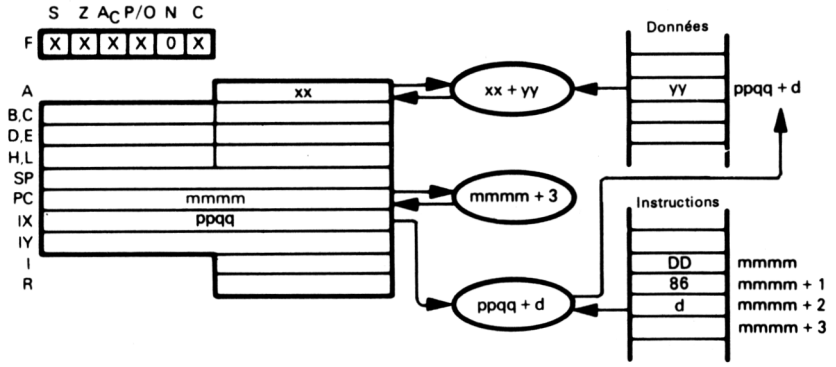
ADD A,E,

l'accumulateur contient 83_{16} :



C'est une instruction classique de manipulation de données.

ADD A, (HL) — ADDITION D'UNE DONNÉE EN MÉMOIRE
ADD A,(IX + disp) A L'ACCUMULATEUR
ADD A,(IY + disp)



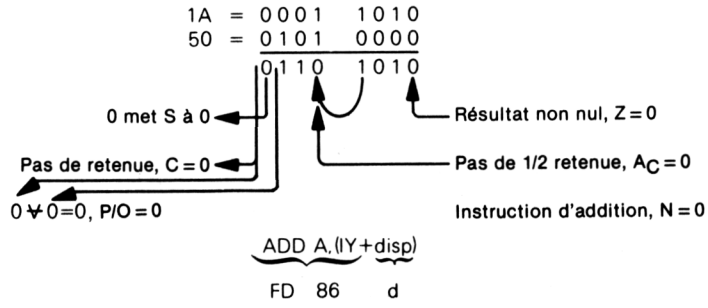
Le schéma illustre l'exécution de l'instruction ADD A,(IX + disp).

ADD A,(IX+disp)
DD 86 d

Addition du contenu de l'adresse (résultant de la somme du contenu du registre IX avec le déplacement d) au contenu de l'accumulateur.
Supposons que ppqq = 4000_{16}, xx = $1A_{16}$ et que l'adresse 4000_{16} contient 50_{16}. A la fin de l'exécution de l'instruction

ADD A,(IX + 0FH)

l'accumulateur contient $6A_{16}$.

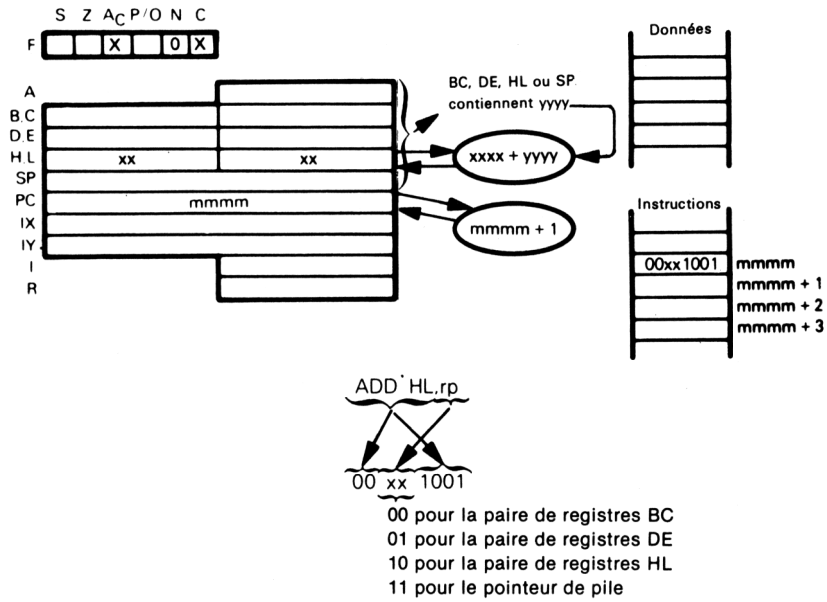


Cette instruction est identique à la précédente, en remplaçant le registre IX par IY.

ADD A,(HL)
86

L'instruction sous cette forme permet d'additionner le contenu de l'adresse, contenue dans le double registre HL, à l'accumulateur.
L'instruction ADD est une instruction classique de manipulation de données.

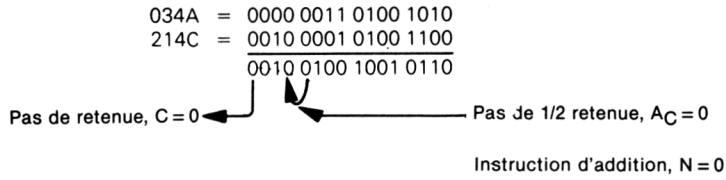
ADD HL,rp — ADDITION D'UNE PAIRE DE REGISTRE A H ET L



Addition de la valeur 16 bits (contenue soit dans le pointeur de pile, soit dans l'une des paires de registres BC, DE ou HL) au contenu du double registre HL.
Supposons que HL contient $034A_{16}$ et BC $214C_{16}$. A la fin de l'exécution de l'instruction :

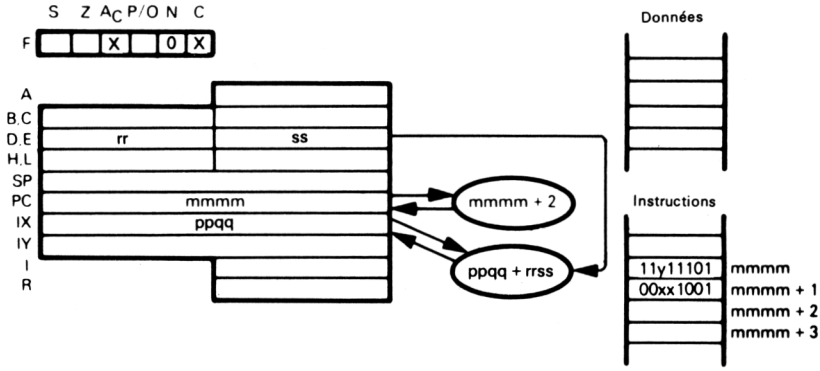
ADD HL,BC

le double registre HL contient 2496_{16}.

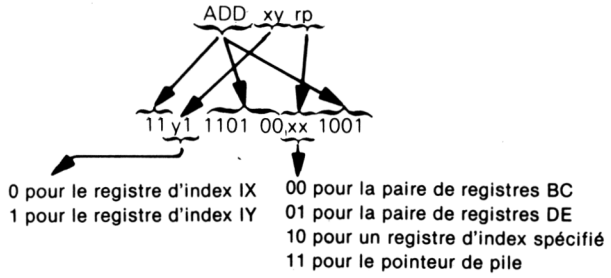


L'instruction ADD HL,HL revient à un décalage à gauche sur 16 bits.

ADD xy,rp — ADDITION D'UNE PAIRE DE REGISTRE A UN REGISTRE D'INDEX



Le schéma illustre l'exécution de l'instruction ADD IX,DE.

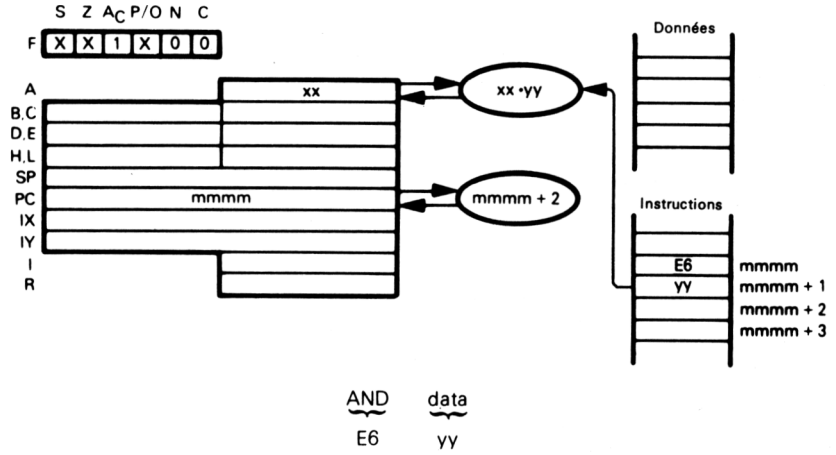


Addition du contenu de la paire de registres spécifiée au contenu du registre d'index indiqué.
Supposons que IY contient $4FF0_{16}$ et BC contient $000F_{16}$. A la fin de l'exécution de l'instruction

ADD IY,BC

le registre d'index IY contient $4FFF_{16}$.

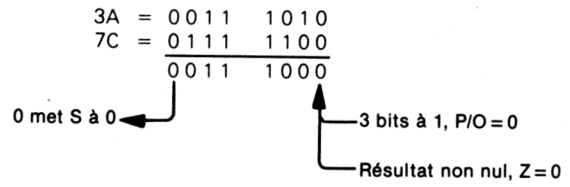
AND data — ET LOGIQUE D'UNE VALEUR IMMÉDIATE AVEC L'ACCUMULATEUR



Cette instruction remplace le contenu de l'accumulateur par le résultat du ET logique entre la donnée spécifiée et le contenu de l'accumulateur.
Supposons que $xx = 3A_{16}$. A la fin de l'exécution de l'instruction

AND 7CH

l'accumulateur contient 38_{16}.

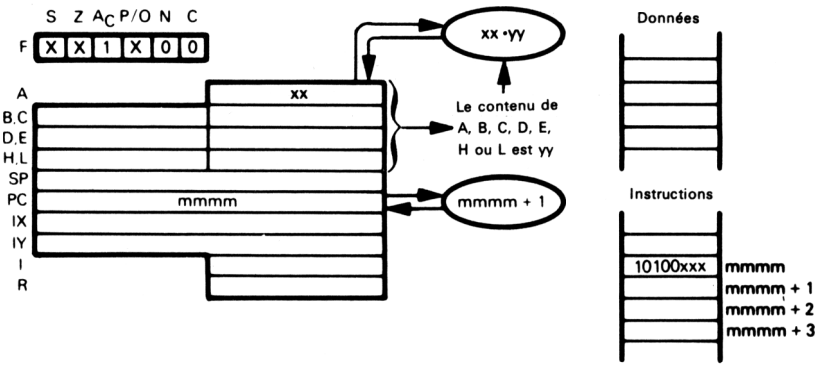


C'est une instruction logique qui sert souvent à mettre à 0 les bits. Par exemple, l'instruction

AND 7FH

met à zéro de façon inconditionnelle le bit de poids fort de l'accumulateur.

AND reg — ET LOGIQUE D'UN REGISTRE AVEC L'ACCUMULATEUR



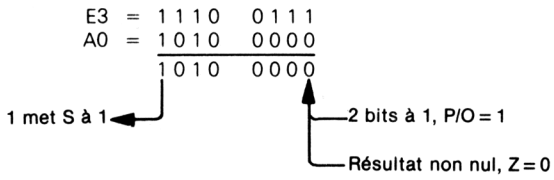
AND (10100) reg (xxx)

000 pour le registre B
001 pour le registre C
010 pour le registre D
011 pour le registre E
100 pour le registre H
101 pour le registre L
111 pour le registre A

Cette instruction remplace le contenu de l'accumulateur par le résultat du ET logique entre les contenus de l'accumulateur et de l'un des registres A, B, C, D, E, H ou L. Supposons que $xx = E3_{16}$ et que le registre E contient $A0_{16}$. A la fin de l'exécution de l'instruction

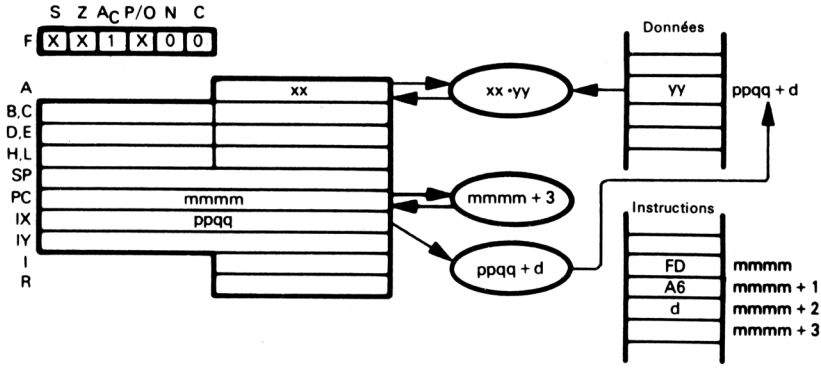
AND E

l'accumulateur contient $A0_{16}$.



L'instruction logique AND est fréquemment utilisée.

AND (HL) — ET LOGIQUE D'UNE DONNEE EN MÉMOIRE
AND (IX + disp) AVEC L'ACCUMULATEUR
AND (IY + disp)



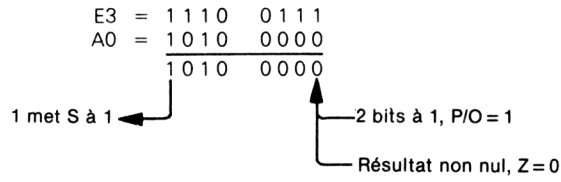
Ce schéma illustre l'exécution de l'instruction AND (IY + disp).

AND (IY+disp)
FD A6 d

Cette instruction effectue le ET logique entre le contenu de l'adresse (résultant de la somme du contenu du registre IY et du déplacement d) et le contenu de l'accumulateur. Supposons que $xx = E3_{16}$, $ppqq = 4000_{16}$ et que l'adresse $400F_{16}$ contient $A0_{16}$. A la fin de l'exécution de l'instruction

AND (IY + 0FH)

l'accumulateur contient $A0_{16}$.



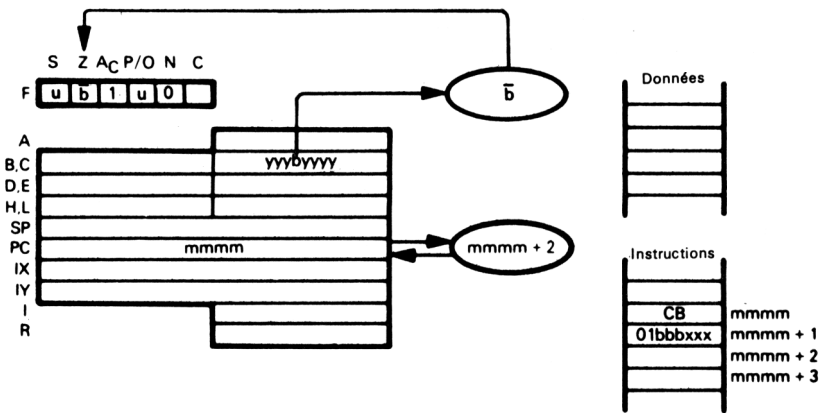
AND (IX+disp)
DD A6 d

Cette instruction est identique à la précédente, si ce n'est qu'elle utilise IX au lieu de IY.

AND (HL)
A6

Cette instruction effectue le ET logique entre le contenu de l'adresse (spécifiée par le double registre HL) et l'accumulateur.
L'instruction logique AND est fréquemment utilisée.

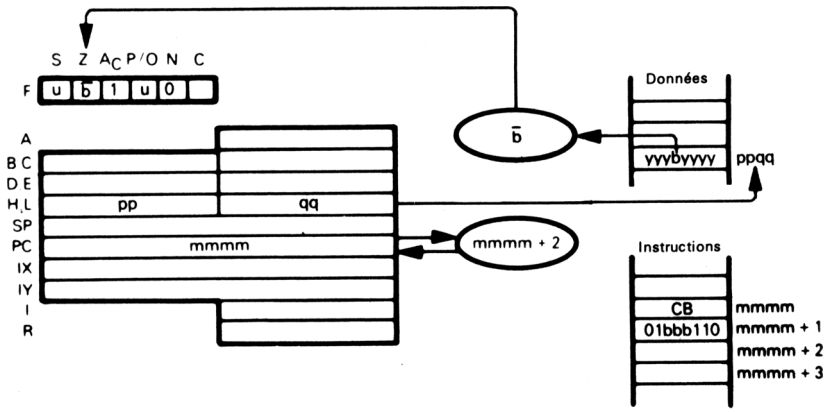
BIT b,reg — TEST DU BIT b DANS LE REGISTRE reg



BIT	b.	reg	
CB 01	bbb	xxx	
Bit testé			Registre
0	000	000	B
1	001	001	C
2	010	010	D
3	011	011	E
4	100	100	H
5	101	101	L
6	110	111	A
7	111		

Le complément du bit spécifié dans le registre précisé est placé dans l'indicateur Z du registre F.
Supposons que le registre C contient 1110 1111. L'instruction BIT 4,C positionne l'indicateur Z à 1 tandis que le bit 4 du registre C reste à 0. (Le bit 0 est le bit de poids faible.)

BIT b,(HL) — TEST DU BIT b DE LA POSITION MÉMOIRE INDIQUÉE
BIT b,(IX + disp)
BIT b,(IY + disp)



Ce schéma illustre l'exécution de l'instruction BIT 4,(HL). Le bit 0 est le bit de poids faible.

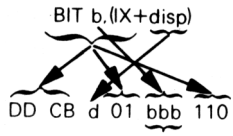
BIT	b.	(HL)
CB 01	bbb	110

Bit testé	bbb
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Cette instruction teste le bit spécifié dans le contenu du double registre HL, et met le complément de ce bit dans l'indicateur Z du registre F.
Supposons que HL contient 4000H et que le bit 3 de l'adresse 4000H contient 1. L'instruction :

```
BIT 3,(HL)
```

met à 0 l'indicateur Z, tandis que le bit 3 de l'adresse 4000H reste à 1.

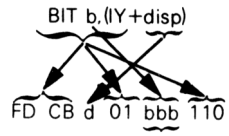


bbb prend les mêmes valeurs que dans l'instruction précédente.

Cette instruction teste le bit spécifié dans la mémoire dont l'adresse est donnée par la somme du registre d'index IX et du déplacement. Le complément du bit est mis dans l'indicateur Z du registre F.
Supposons que le registre d'index IX contient 4000H et que le bit 4 de l'adresse 4004H est 0. L'instruction :

BIT 4,(IX + 4H)

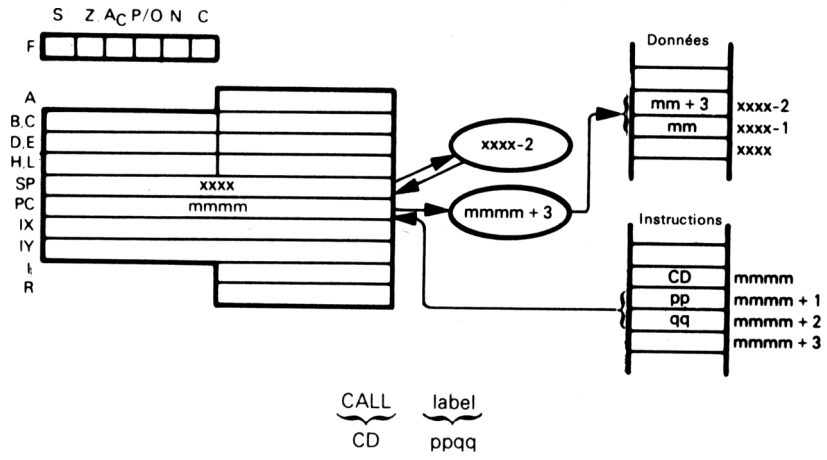
positionne l'indicateur Z à 1 tandis que le bit 4 de l'adresse 4004H reste à 0.



bbb prend les mêmes valeurs que précédemment.

Cette instruction est identique à BIT b,(IX + disp), si ce n'est qu'elle utilise le registre IY à la place de IX.

CALL label — APPEL DU SOUS-PROGRAMME IDENTIFIÉ PAR L'ÉTIQUETTE

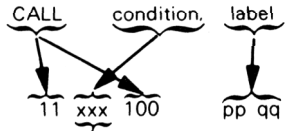


Cette instruction effectue successivement les opérations suivantes :
— elle range en haut de la pile l'adresse de l'instruction suivant le CALL (la pile est adressée par le pointeur de pile) ;
— elle met à jour le pointeur de pile en lui retirant 2 ;
— elle transfère au compteur ordinal l'adresse 16 bits contenue dans les deuxième et troisième octets du code objet de l'instruction CALL. Le second octet de l'instruction CALL constitue la partie basse de l'adresse bas rang et le troisième octet la partie haute.
Soit la séquence d'instructions suivante :

```
        CALL    SUBR
        AND     7CH
        -
        -
        -
SUBR
```

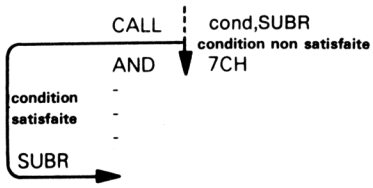
A la fin de l'exécution de l'instruction CALL, l'adresse de l'instruction AND est sauvegardée au sommet de la pile. Le pointeur de pile est décrémenté par 2. L'instruction étiquetée SUBR sera la prochaine instruction exécutée.

CALL condition, label — APPEL DU SOUS-PROGRAMME IDENTIFIÉ PAR L'ÉTIQUETTE SI LA CONDITION EST SATISFAITE



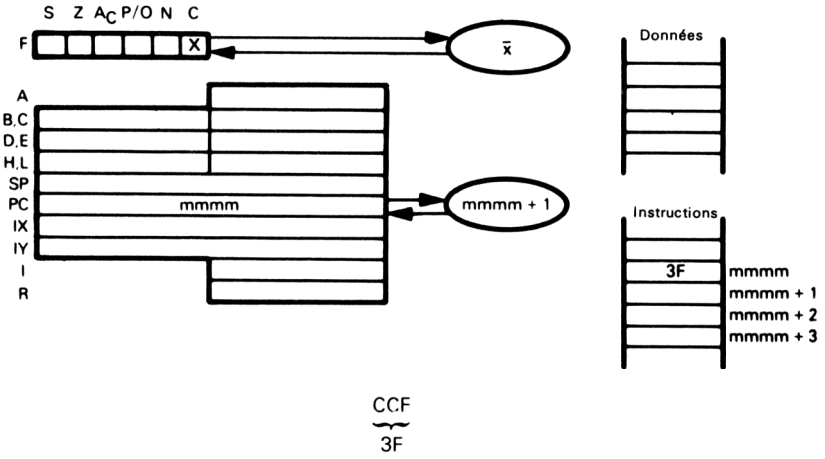
	Condition		Indicateur concerné
000	NZ	non nul ("Non-Zero")	Z
001	Z	nul ("Zero")	Z
010	NC	pas de retenue ("Non-Carry")	C
011	C	retenue ("Carry")	C
100	PO	parité impaire ("Parity Odd")	P/O
101	PE	parité paire ("Parity Even")	P/O
110	P	positif	S
111	M	négatif	S

Cette instruction est identique à l'instruction CALL, si ce n'est que le sous-programme n'est appelé que si la condition est satisfaite ; sinon, l'instruction qui suit le CALL est exécutée.
Soit la séquence d'instructions suivante :



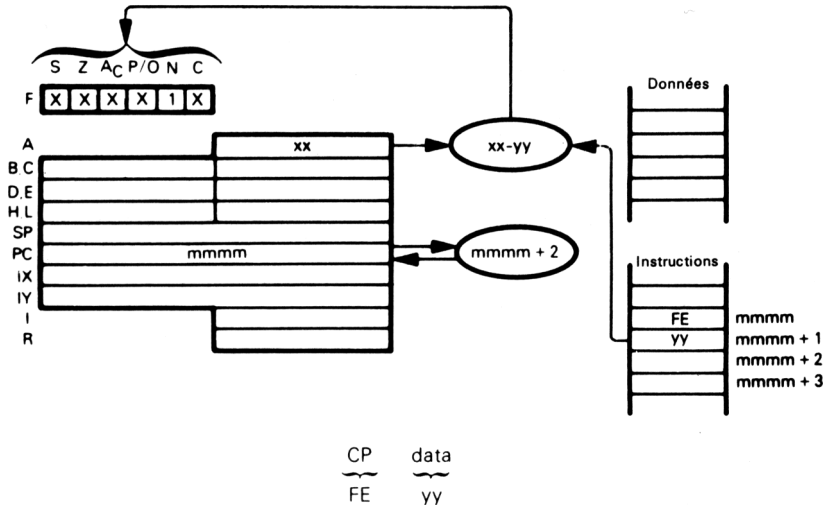
Si la condition n'est pas satisfaite, l'instruction AND sera exécutée après l'exécution de l'instruction CALL cond,SUBR. Si la condition est satisfaite, l'adresse de l'instruction AND est rangée au sommet de la pile et le pointeur de pile est décrémenté de 2. L'instruction dont l'étiquette est SUBR est ensuite exécutée.

CCF — COMPLÉMENTATION DE L'INDICATEUR DE RETENUE (« COMPLEMENT CARRY FLAG »)



Cette instruction effectue la complémentation de l'indicateur de retenue. Aucun autre indicateur, ni registre, n'est modifié.

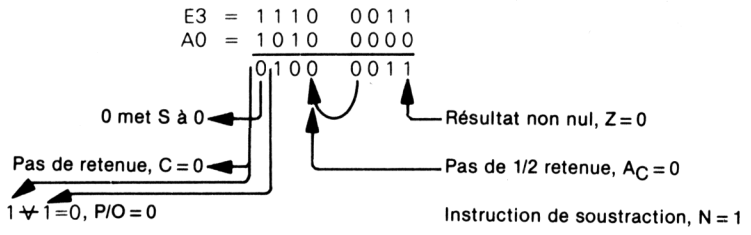
CP data — COMPARAISON D'UNE VALEUR IMMÉDIATE AVEC L'ACCUMULATEUR



Cette instruction soustrait le contenu du second octet du code objet du contenu de l'accumulateur en considérant les deux nombres comme de simples données binaires. Le résultat n'est pas pris en compte : l'accumulateur n'est pas modifié mais les indicateurs d'état reflètent le résultat de la soustraction.
Supposons que $xx = E3_{16}$ et que le second octet du code objet de l'instruction CP contient $A0_{16}$. A la fin de l'exécution de l'instruction

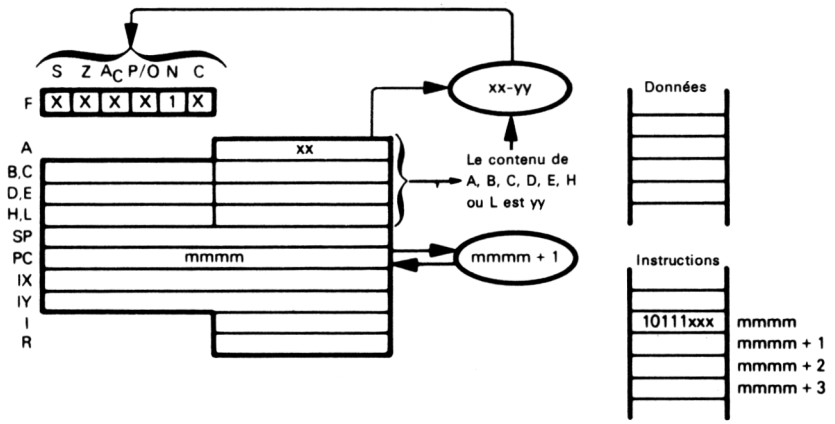
CP 0A0H

l'accumulateur contient toujours $E3_{16}$, mais les indicateurs sont modifiés comme suit :



Remarque : la retenue résultante est complémentée.

CP reg — COMPARAISON D'UN REGISTRE AVEC L'ACCUMULATEUR



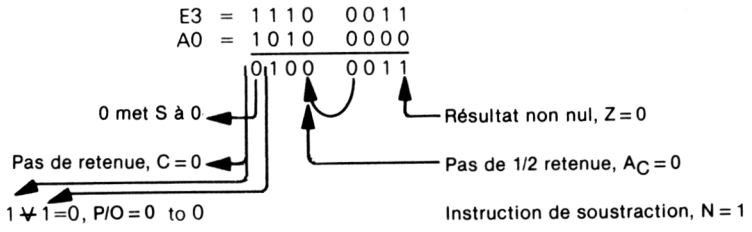
CP reg
10111 xxx

000	pour reg=B
001	pour reg=C
010	pour reg=D
011	pour reg=E
100	pour reg=H
101	pour reg=L
111	pour reg=A

Cette instruction effectue la soustraction du contenu du registre A, B, C, D, E, H ou L du contenu de l'accumulateur en considérant les 2 nombres comme de simples données binaires. Le résultat n'est pas pris en compte : l'accumulateur reste inchangé mais les indicateurs d'état reflètent le résultat de la soustraction.
Supposons que $xx = E3_{16}$ et que le registre B contient $A0_{16}$. A la fin de l'exécution de l'instruction

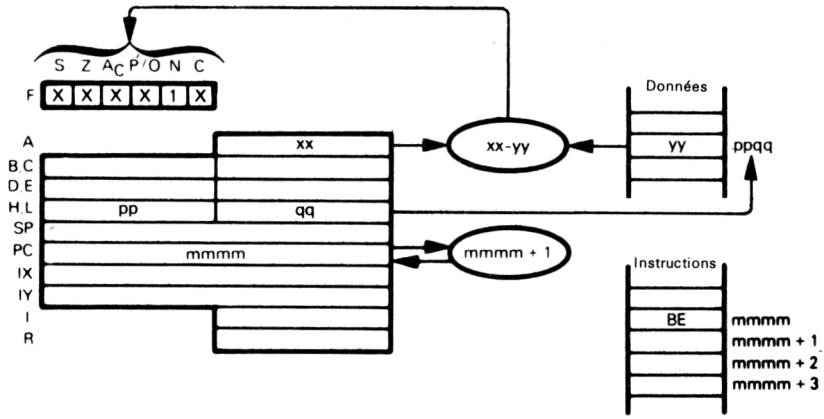
CP B

l'accumulateur contient toujours $E3_{16}$, mais les indicateurs sont modifiés comme suit :

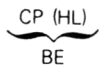


Remarque : la retenue résultante est complémentée.

CP (HL) — COMPARAISON D'UNE DONNÉE EN MÉMOIRE
CP (IX + disp) AVEC L'ACCUMULATEUR
CP (IY + disp)



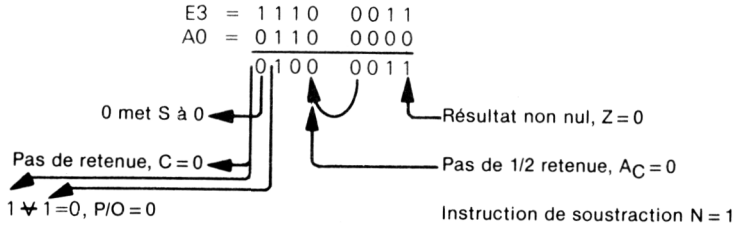
Le schéma illustre l'exécution de l'instruction CP (HL) :



Cette instruction soustrait le contenu de l'adresse (contenu par le double registre HL) du contenu de l'accumulateur, en considérant les deux nombres comme de simples données binaires. Le résultat n'est pas pris en compte : l'accumulateur demeure inchangé mais les indicateurs d'état reflètent le résultat de la soustraction.
Supposons que $xx = E3_{16}$ et $yy = A0_{16}$. A la fin de l'exécution de l'instruction

CP (HL)

l'accumulateur contient toujours $E3_{16}$ mais les indicateurs d'état sont modifiés comme suit :



Remarque : la retenue résultante est complémentée.

CP (IX+disp)

DD BE d

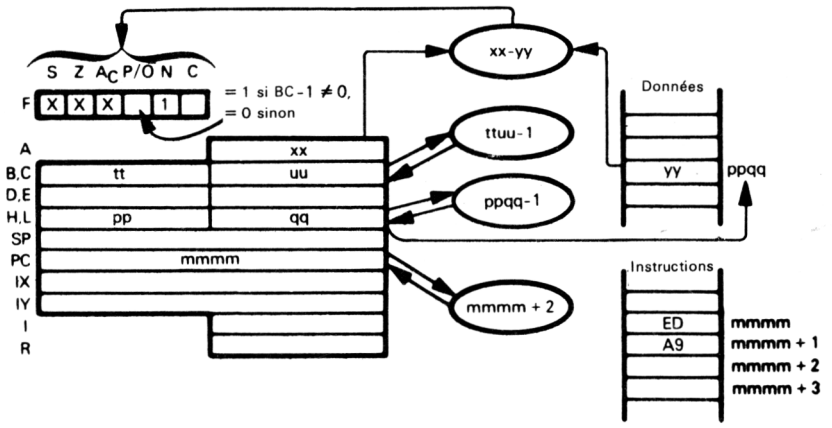
Cette instruction soustrait le contenu de l'adresse (résultant de la somme du contenu du registre IX et du déplacement d) du contenu de l'accumulateur en considérant les deux nombres comme de simples données binaires. Le résultat n'est pas pris en compte : l'accumulateur demeure inchangé mais les indicateurs d'état reflètent le résultat de la soustraction.

CP (IY+disp)

FD BE d

Cette instruction est identique à la précédente, si ce n'est qu'elle utilise le registre IY à la place du registre IX.

CPD — COMPARAISON DE L'ACCUMULATEUR AVEC LA MÉMOIRE DÉCREMENTATION DE L'ADRESSE ET DU COMPTEUR



CPD

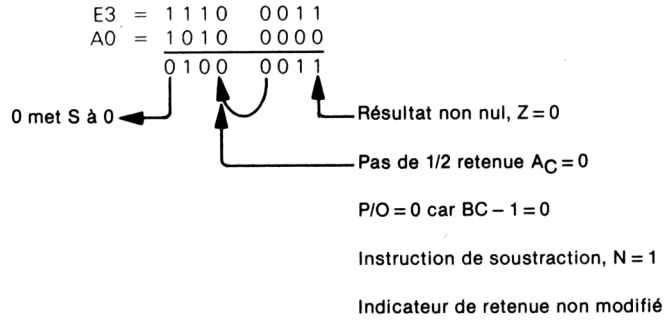
ED A9

Cette instruction compare le contenu de l'accumulateur avec celui de l'adresse (contenu par le double registre HL). Si les 2 contenus sont égaux, l'indicateur Z est positionné à 1. Les deux paires de registres HL et BC sont décrémentées. (BC sert de compteur.)

Supposons que $xx = E3_{16}$, $ppqq = 4000_{16}$, BC contient 0001_{16} et $yy = A0_{16}$. A la fin de l'exécution de l'instruction :

CPD

l'accumulateur contient toujours $E3_{16}$, mais les indicateurs d'état sont modifiés comme suit :



Le double registre HL contient $3FFF_{16}$ et BC = 0.

CPDR — COMPARAISON DE L'ACCUMULATEUR AVEC LA MÉMOIRE DÉCRÉMENTATION DE L'ADRESSE ET DU COMPTEUR RÉPÉTITION JUSQU'A ÉGALITÉ OU COMPTEUR NUL

CPDR
ED B9

Cette instruction est identique à CPD, si ce n'est qu'elle répète les opérations (comparaison, décrémentation) jusqu'à ce qu'il y ait égalité des membres ou que le compteur atteigne zéro. Après chaque transfert de données, les interruptions éventuelles sont prises en compte et deux cycles de rafraîchissement sont exécutés.
Supposons que la paire de registres HL contient 5000_{16}, la paire de registres BC contient $00FF_{16}$, l'accumulateur $F9_{16}$ et que la mémoire est ainsi disposée :

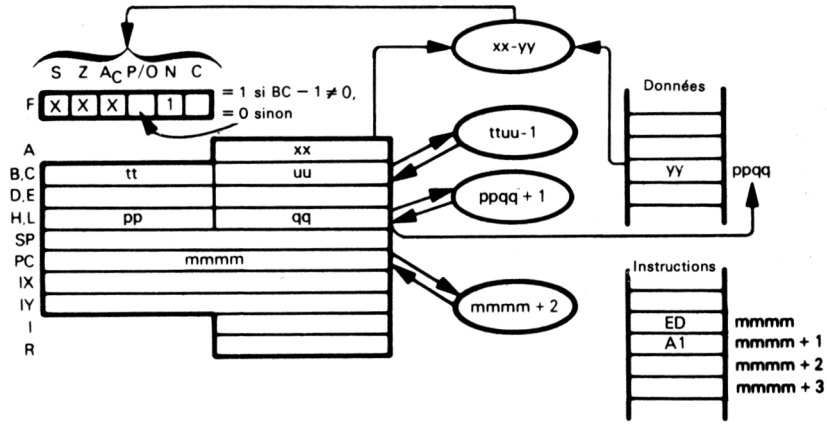
Adresse	Contenu
5000_{16}	AA_{16}
$4FFF_{16}$	BC_{16}
$4FFE_{16}$	19_{16}
$4FFD_{16}$	$7A_{16}$
$4FFC_{16}$	$F9_{16}$
$4FFB_{16}$	DD_{16}

A la fin de l'exécution de l'instruction :

CPDR

l'indicateur P/O est à 1 ainsi que l'indicateur Z, les doubles registres HL et BC contiennent respectivement $4FFB_{16}$ et $00FA_{16}$.

CPI — COMPARAISON DE L'ACCUMULATEUR AVEC LA MÉMOIRE
DÉCREMENTATION DU COMPTEUR
INCRÉMENTATION DE L'ADRESSE

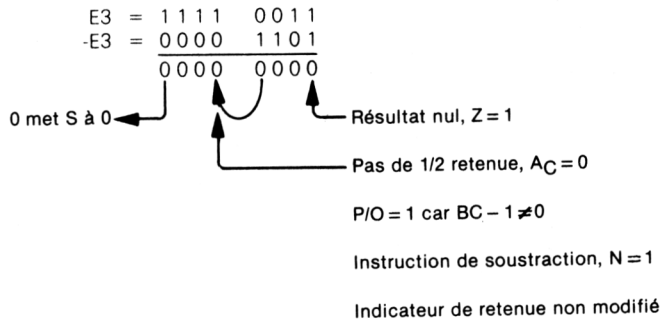


CPI
ED A1

Cette instruction compare le contenu de l'accumulateur avec celui de l'adresse (contenue dans le double registre HL). Si ces contenus sont égaux, l'indicateur Z est positionné à 1. Le double registre HL est incrémenté tandis que le double registre BC est décrémenté. Supposons que $xx = E3_{16}$, $ppqq = 4000_{16}$, BC contient 0032_{16} et $yy = E3_{16}$. A la fin de l'exécution de l'instruction

CPI

l'accumulateur contient $E3_{16}$ mais les indicateurs sont modifiés comme suit :



Les doubles registres HL et BC contiennent respectivement 4001_{16} et 0031_{16}.

CPIR — COMPARAISON DE L'ACCUMULATEUR AVEC LA MÉMOIRE DÉCRÉMENTATION DU COMPTEUR INCRÉMENTATION DE L'ADRESSE RÉPÉTITION JUSQU'A ÉGALITÉ OU COMPTEUR NUL

CPIR
ED B1

Cette instruction est identique à la précédente, si ce n'est qu'elle répète les opérations (comparaison, décrémentation, incrémentation) jusqu'à l'égalité des membres ou l'annulation du compteur. Après chaque transfert de données, les interruptions éventuelles sont prises en compte et deux cycles de rafraîchissement sont exécutés.
Supposons que les doubles registres HL et BC contiennent respectivement 4500_{16} et $00FF_{16}$, l'accumulateur $F9_{16}$ et que la mémoire est ainsi disposée :

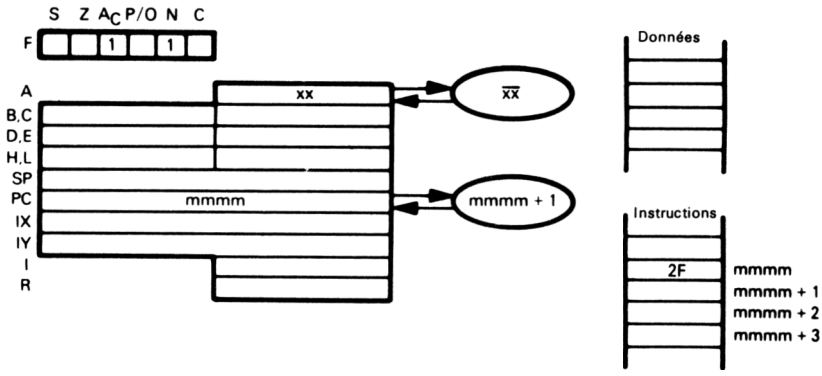
Adresse	Contenu
4500_{16}	AA_{16}
4501_{16}	15_{16}
4502_{16}	$F9_{16}$

A la fin de l'exécution de l'instruction

CPIR

les indicateurs P/O et Z sont tous deux positionnés à 1. Les double registres HL et BC contiennent respectivement 4503_{16} et $00FC_{16}$.

CPL — COMPLÉMENTATION DE L'ACCUMULATEUR



CPL
2F

Cette instruction effectue la complémentation à 1 de l'accumulateur. Aucun autre registre n'est modifié.
Supposons que l'accumulateur contient $3A_{16}$. A la fin de l'exécution de l'instruction

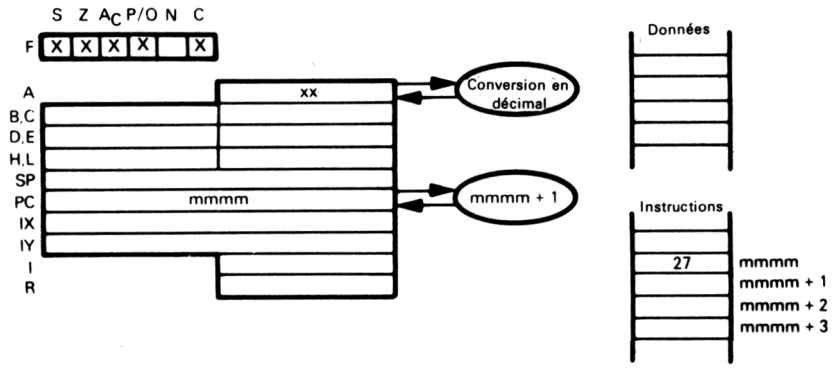
CPL

l'accumulateur contient $C5_{16}$.

3A = 0011 1010
Complément = 1100 0101

CPL est une instruction logique classique. Il n'est pas utile de s'en servir pour des soustractions binaires ; il existe des instructions spéciales à cet effet (SUB, SBC).

DAA — AJUSTEMENT DÉCIMAL DE L'ACCUMULATEUR (BCD)



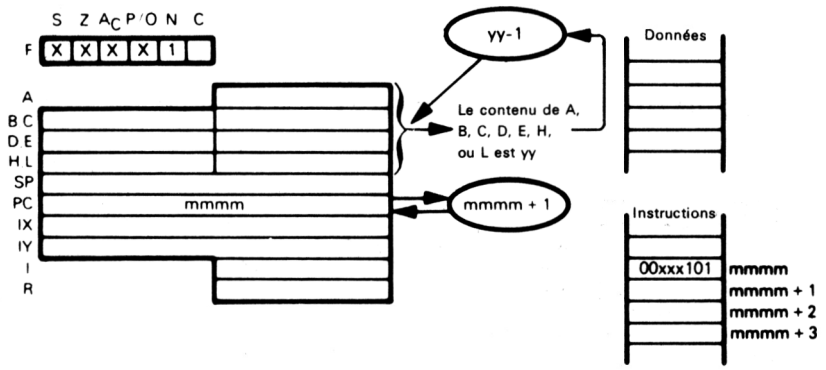
DAA
27

Cette instruction met le contenu de l'accumulateur sous forme DCB (décimal - codé -binaire). Cette instruction ne doit servir qu'après l'addition ou la soustraction de deux nombres DCB, c'est-à-dire que l'on ne doit retrouver DAA que dans les séquences d'instructions suivantes : ADD DAA, ADC DAA, INC DAA, SUB DAA, SBC DAA, DEC DAA et NEG DAA qui sont toutes des opérations arithmétiques décimales opérant sur des nombres DCB et produisant des nombres DCB.
Supposons que l'accumulateur contient 39_{16} et le registre B 47_{16}. A la fin de l'exécution des instructions

ADD B
DAA

l'accumulateur contient 86_{16} et non 80_{16}.
La circuiterie interne du Z80 utilise les valeurs de la retenue, de l'indicateur N et de la 1/2 retenue ainsi que le contenu de l'accumulateur pour exécuter l'instruction DAA.

DEC reg — DÉCRÉMENTATION DU CONTENU D'UN REGISTRE





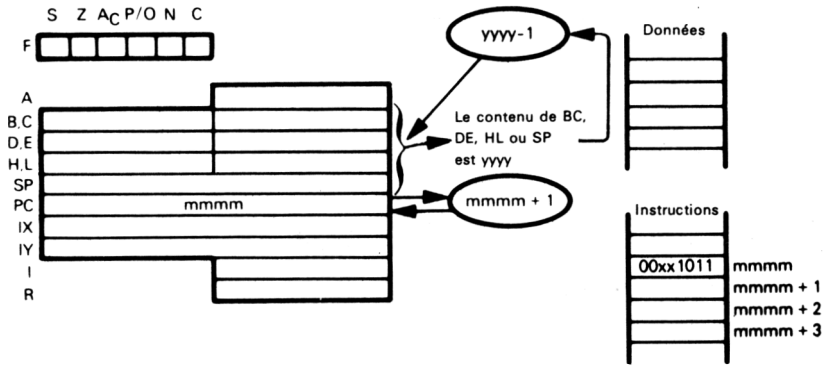
000 pour reg=B
001 pour reg=C
010 pour reg=D
011 pour reg=E
100 pour reg=H
101 pour reg=L
111 pour reg=A

Cette instruction soustrait 1 du contenu du registre spécifié.
Supposons que le registre A contient 50_{16}. A la fin de l'exécution de l'instruction

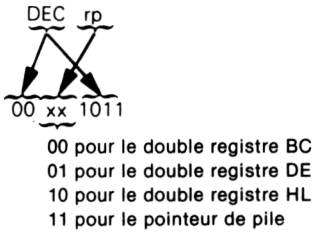
```
DEC A
```

le registre A contient $4F_{16}$.

DEC rp — DÉCRÉMENTATION D'UNE PAIRE DE REGISTRES SPECIFIEE
DEC IX
DEC IY



Le schéma illustre l'exécution de l'instruction, DEC rp :



Cette instruction soustrait 1 à la valeur 16 bits contenue dans le double registre spécifié. Aucun indicateur d'état n'est modifié. Supposons que les registres H et L contiennent $2F00_{16}$. A la fin de l'exécution de l'instruction

```
DEC HL
```

les registres H et L contiennent $2EFF_{16}$.

```
DEC IX
DD 2B
```

Cette instruction soustrait 1 à la valeur 16 bits contenue dans le registre IX.

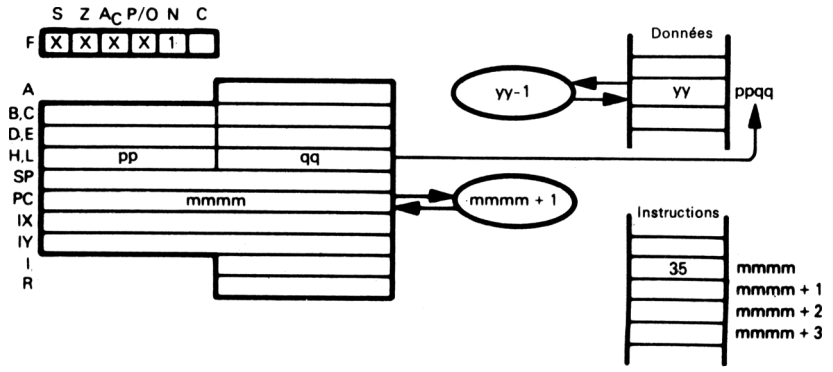
```
DEC IY
FD 2B
```

Cette instruction soustrait 1 à la valeur 16 bits contenue dans le registre IY.

Aucune des instructions DEC rp, DEC IX, DEC IY ne modifie les indicateurs d'état. C'est un défaut du jeu d'instructions du Z80, hérité du 8080. Tandis que l'instruction DEC reg est utilisée dans des boucles avec un compteur dont la valeur ne peut pas dépasser 256, les instructions DEC rp, DEC IX ou DEC IY, au contraire, doivent être utilisées si la valeur du compteur dépasse 256. Puisque l'instruction DEC rp ne modifie aucun indicateur, il faut rajouter quelques instructions pour effectuer le test d'un résultat nul. Ceci constitue une forme typique de boucle :

```
        LD      DE,DATA     ; CHARGER VALEUR INITIALE COMPTEUR 16 BITS
BOUCLE: -                   ; 1re INSTRUCTION BOUCLE
        -
        -
        DEC     DE          ; DECREMENTER COMPTEUR
        LD      A,D         ; TEST SI ZERO, TRANSFERT D VERS A
        OR      E           ; PUIS, OU LOGIQUE ENTRE A ET E
        JP      NZ,BOUCLE   ; REBOUCLE, SI NON NUL
```

DEC (HL) — DÉCRÉMENTATION D'UN CONTENU MÉMOIRE
DEC (IX + disp)
DEC (IY + disp)



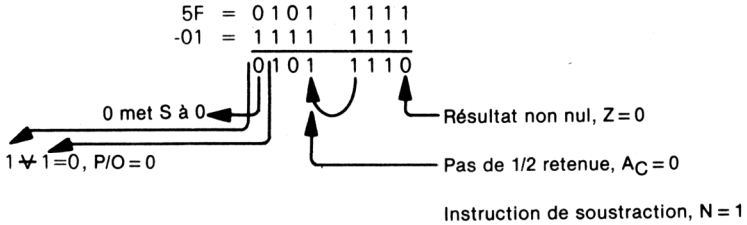
Ce schéma illustre l'exécution de l'instruction DEC (HL) :

$$\underbrace{\text{DEC (HL)}}_{35}$$

Cette instruction soustrait 1 du contenu de l'adresse (contenue dans la paire de registres HL).
Supposons que ppqq = 4500_{16}, yy = $5F_{16}$. A la fin de l'exécution de l'instruction

DEC (HL)

l'adresse 4500_{16} contient $5E_{16}$.



DEC (IX+disp)

DD 35 d

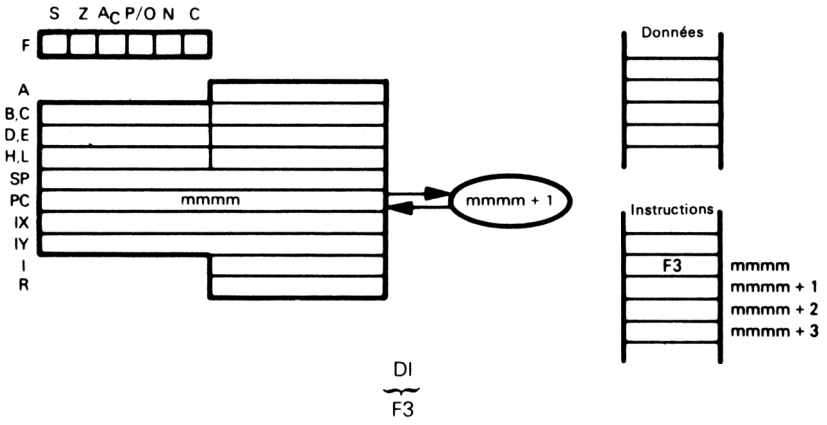
Cette instruction soustrait 1 du contenu de l'adresse (résultant de la somme du contenu du registre IX et du déplacement d).

DEC (IY+disp)

FD 35 d

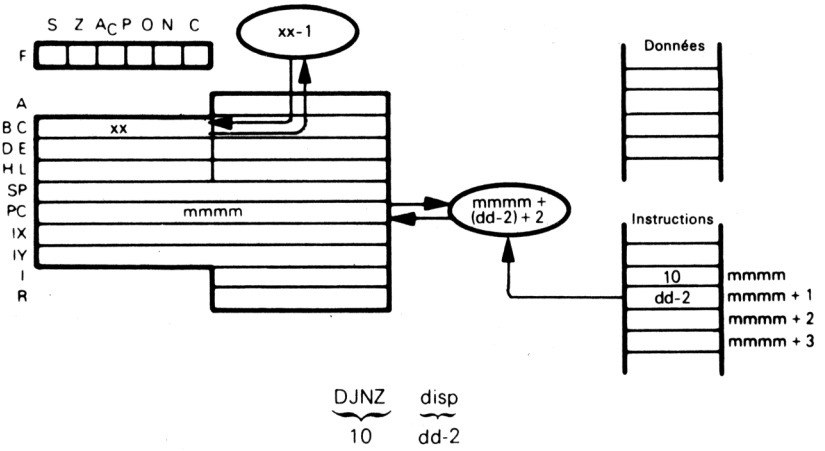
Cette instruction est identique à la précédente, si ce n'est qu'elle utilise IY à la place de IX.

DI — MASQUAGE DES INTERRUPTIONS (« DISABLE INTERRUPTS »)



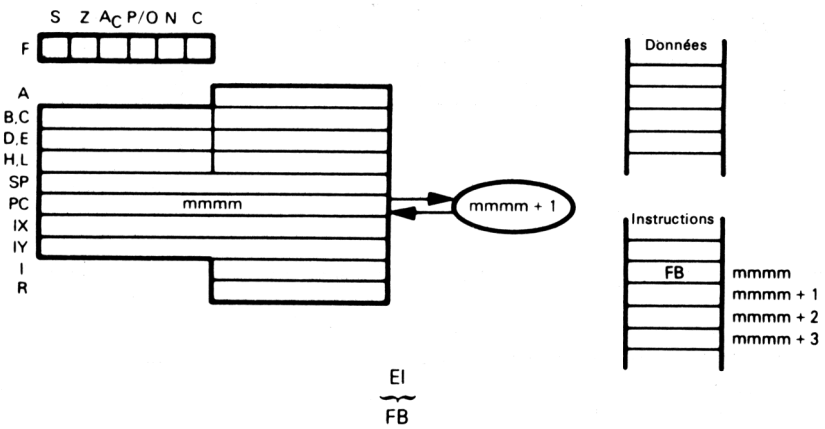
Lorsque cette instruction est exécutée, toute demande d'interruption masquable est inhibée et l'entrée INT de l'UCT est donc ignorée. Rappelons que lorsqu'une interruption est prise en compte, toute interruption masquable est masquée.
La demande d'interruption masquable reste inhibée jusqu'à ce qu'elle soit autorisée par une instruction EI.
Aucun registre ni indicateur n'est modifié par cette instruction.

DJNZ disp — DÉCRÉMENTATION DU REGISTRE ET SAUT RELATIF SI RÉSULTAT NON NUL



Cette instruction commence par décrémenter le registre B. Si le contenu résultant est non nul, elle ajoute le contenu du second octet du code objet de l'instruction DJNZ, +2, au compteur ordinal. Le saut est calculé à partir de l'adresse du code opératoire de l'instruction et a une amplitude de −126 à +129 octets. L'assembleur calcule automatiquement le déplacement relatif au compteur ordinal (cf. instruction JR disp).
Si le contenu de B est nul après la décrémentation, l'instruction qui suit DJNZ est exécutée.
L'instruction DJNZ est particulièrement utile pour toutes les opérations de boucle, puisqu'une seule instruction remplace la séquence d'instructions : « décrémentation puis saut suivant condition ».

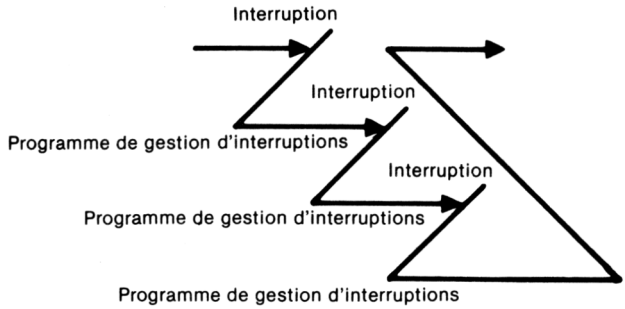
EI — AUTORISATION DES INTERRUPTIONS (« ENABLE INTERRUPTS »)



L'exécution de cette instruction permet de prendre en compte à nouveau les interruptions mais seulement après la fin de l'exécution de l'instruction suivante.
La plupart des programmes de gestion d'interruptions se terminent par :

```
EI          ; INTERRUPTIONS AUTORISEES
RETI        ; RETOUR AU PROGRAMME INTERROMPU
```

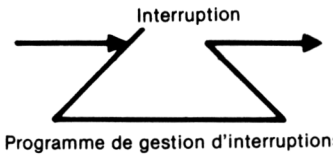
Si les interruptions sont traitées en série, alors tout au long de la durée du programme de gestion d'interruptions, toutes les interruptions masquables sont inhibées. Ce qui veut dire que dans une application multi-interruptions, il y a une grande probabilité pour qu'une ou plusieurs interruptions restent en suspens jusqu'à ce que le programme de gestion d'interruptions achève son exécution.
Si les interruptions étaient reconnues dès l'exécution de l'instruction EI, l'instruction RETI ne serait alors jamais exécutée. Dans ce cas, les retours programme s'empileraient l'un au-dessus de l'autre et prendraient inutilement de la place mémoire. Ce qui peut être illustré comme suit :

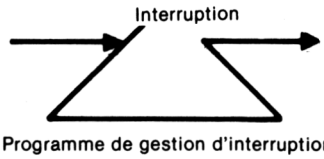


En inhibant l'interruption pendant l'instruction qui suit EI, le Z80 permet que l'instruction RETI soit exécutée dans la séquence :

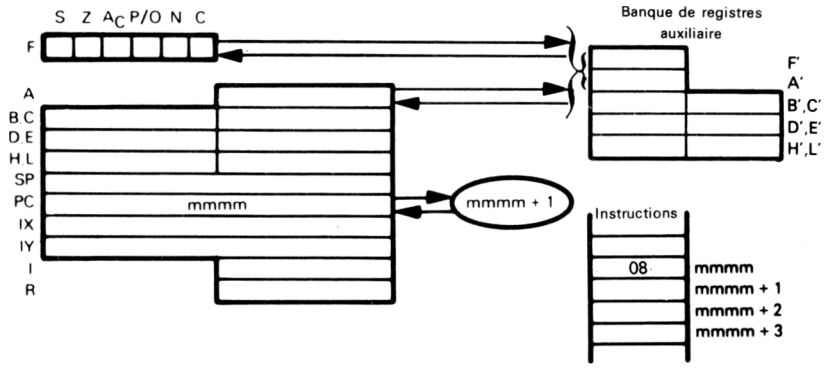
```
—
—
—
EI          ; INTERRUPTIONS AUTORISEES
RETI        ; RETOUR D'INTERRUPTION
```

Il n'est pas rare que les interruptions soient gardées masquées pendant l'exécution d'un programme de gestion d'interruptions. Les interruptions sont alors traitées en série :





EX AF,AF' — ÉCHANGE DE L'ACCUMULATEUR ET DU REGISTRE D'ETAT AVEC LEURS ÉQUIVALENTS AUXILIAIRES



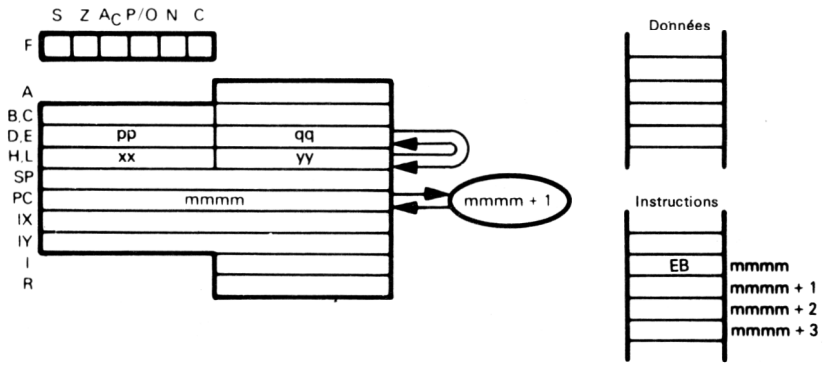
EX AF,AF'
08

Cette instruction échange les contenus sur 2 octets des paires de registres AF et A'F'. Supposons que AF contient $4F99_{16}$ et A'F' contient $10AA_{16}$. A la fin de l'exécution de l'instruction

EX AF,AF'

AF contient $10AA_{16}$ et AF' contient $4F99_{16}$.

EX DE, HL — ÉCHANGE DES CONTENUS DES DOUBLES REGISTRES DE ET HL



EX DE,HL
EB

Cette instruction échange les contenus des paires de registres DE et HL.
Supposons que pp = 03_{16}, qq = $2A_{16}$, xx = 41_{16}, yy = FC_{16}. A la fin de l'exécution de l'instruction

```
EX DE,HL
```

H contient 03_{16}, L : $2A_{16}$, D : 41_{16} et E : FC_{16}.
Les deux instructions :

```
EX DE,HL
LD A,(HL)
```

sont équivalentes à :

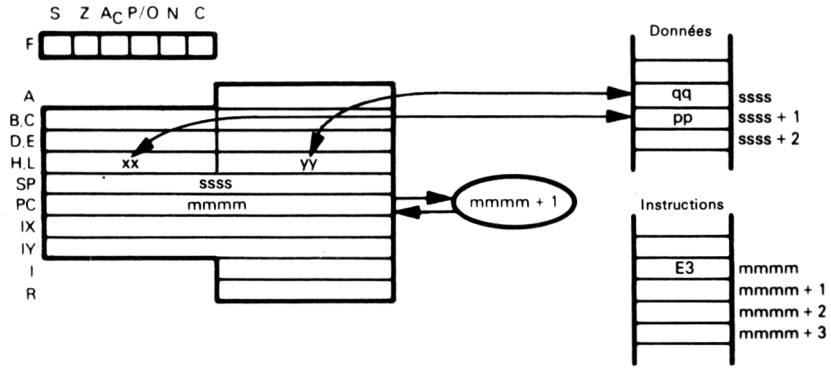
```
LD A,(DE)
```

mais si l'on veut charger dans le registre B la donnée adressée par les registres D et E, les 2 instructions :

```
EX DE,HL
LD B,(HL)
```

ne possèdent pas d'équivalent sur une instruction.

EX (SP),HL — ÉCHANGE DU CONTENU D'UN REGISTRE AVEC
EX (SP),IX LE SOMMET DE PILE
EX (SP),IY



Ce schéma illustre l'exécution de l'instruction EX (SP),HL.

EX (SP),HL
E3

Cette instruction échange le contenu du registre L avec l'octet du sommet de la pile et le contenu du registre H avec l'octet juste en dessous.
Supposons que xx = 21, yy = FA_{16}, pp = $3A_{16}$, qq = $E2_{16}$. A la fin de l'exécution de l'instruction

EX (SP),HL

H contient $3A_{16}$, L $E2_{16}$ et les 2 octets de sommet de pile contiennent respectivement FA_{16} et 21_{16}.
L'instruction EX (SP),HL sert à accéder aux données du sommet de pile et à les traiter.

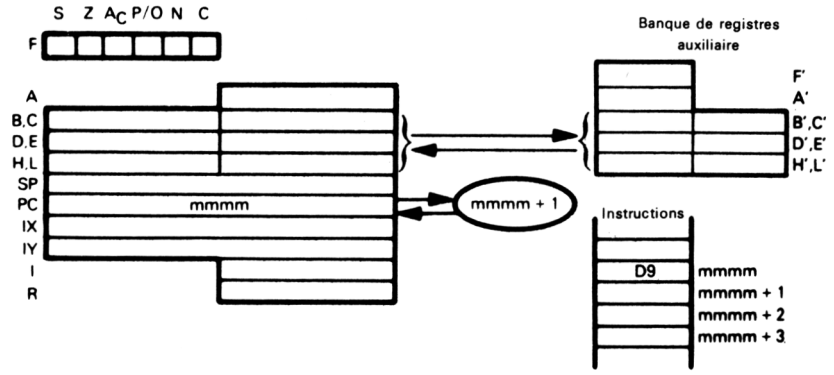
EX (SP),IX
DD E3

Cette instruction échange le contenu de l'octet de poids faible du registre IX avec l'octet du sommet de la pile et le contenu de l'octet de poids fort du registre IX avec celui qui suit l'octet du sommet de pile.

EX (SP),IY
FD E3

Cette instruction est identique à la précédente, si ce n'est qu'elle utilise le registre IY à la place de IX.

EXX — ÉCHANGE DES PAIRES DE REGISTRES AVEC LES PAIRES DES REGISTRES AUXILIAIRES



EXX
D9

Cette instruction échange les contenus des paires respectives de registres BC, DE et HL avec les contenus des paires respectives B'C', D'E' et H'L'.
Supposons que les paires des registres BC, DE et HL contiennent respectivement 4901_{16}, $5F00_{16}$ et 7251_{16} et que les paires des registres B'C', D'E' et H'L' contiennent respectivement 0000_{16}, $10FF_{16}$ et 3333_{16}. A la fin de l'exécution de l'instruction

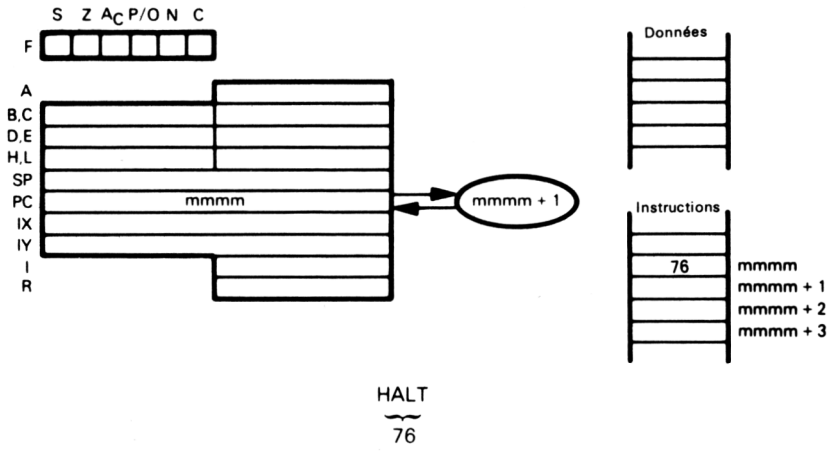
EXX

les registres ont les contenus suivants :

BC: 0000_{16}; DE: $10FF_{16}$; HL: 3333_{16};
B'C': 4901_{16}; D'E': $5F00_{16}$; H'L': 7251_{16}

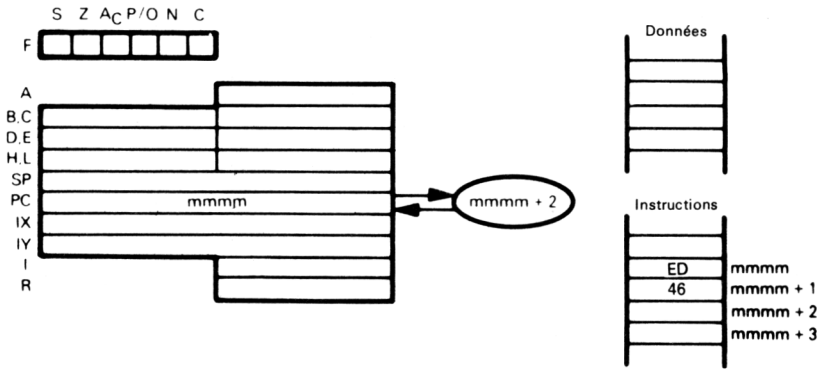
Cette instruction peut servir à échanger les banques des registres pour obtenir des temps de réponse d'interruption très rapides.

HALT



Lorsque l'instruction HALT est exécutée, l'exécution du programme cesse. Une interruption ou un « reset » est alors nécessaire pour que l'UCT reprenne l'exécution. Aucun registre, ni indicateur n'est modifié ; toutefois la logique de rafraîchissement de la mémoire continue d'opérer.

IM 0 — MODE D'INTERRUPTION 0



IM 0
ED 46

Cette instruction met l'UCT en mode d'interruption 0. Dans ce mode, l'organe périphérique qui a généré l'interruption envoie une instruction sur le bus de données et l'UCT exécute ensuite cette instruction. Aucun registre, ni indicateur n'est modifié.

IM 1 — MODE D'INTERRUPTION 1

IM 1
ED 56

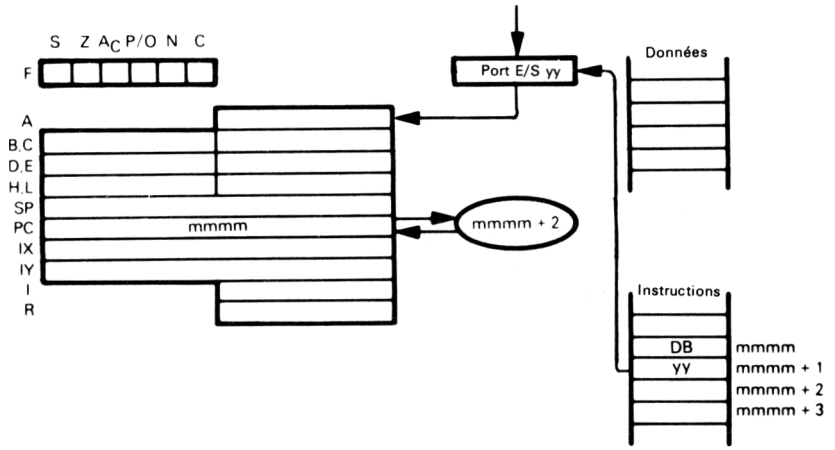
Cette instruction met l'UCT en mode d'interruption 1. Dans ce mode, l'UCT répond à l'interruption en exécutant une instruction RST à l'adresse 0038_{16}.

IM 2 — MODE D'INTERRUPTION 2

IM 2
ED 5E

Cette instruction met l'UCT en mode d'interruption 2. Dans ce mode, l'UCT effectue un appel indirect à n'importe quelle adresse spécifique en mémoire. Les 8 bits de poids fort de cette adresse 16 bits sont donnés par le contenu du registre d'interruption (I) tandis que les 8 bits de poids faible sont fournis par le périphérique qui a envoyé l'interruption. Pour une description complète des modes d'interruption, se reporter au chapitre 12. Aucun registre, ni indicateur, n'est modifié par cette instruction.

IN A,(port) — ENTRÉE VERS L'ACCUMULATEUR



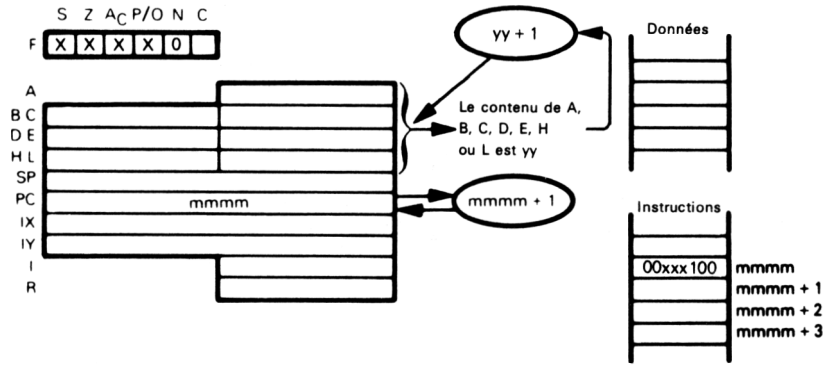
IN A, (port)
DB yy

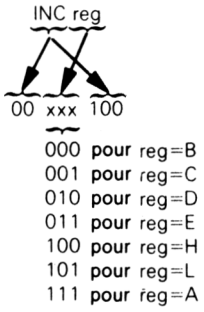
Cette instruction charge un octet de donnée dans l'accumulateur à partir du port d'entrée/sortie (E/S) identifié par le second octet du code objet de l'instruction IN. Supposons que le buffer d'entrée/sortie du port $1A_{16}$ contient 36_{16}. A la fin de l'exécution de l'instruction

IN A,(1AH)

l'accumulateur contient 36_{16}.
L'instruction IN ne modifie pas les indicateurs.
L'utilisation de l'instruction IN dépend essentiellement du matériel. C'est la circuiterie logique externe qui détermine les adresses valides des différents ports d'E/S.
Il est également possible de concevoir une logique externe telle qu'on accède aux ports d'E/S de la même manière qu'à la mémoire, certaines adresses étant réservées à ces ports (et ne pouvant donc pas être utilisées pour la mémoire).

INC reg — INCRÉMENTATION DU CONTENU D'UN REGISTRE



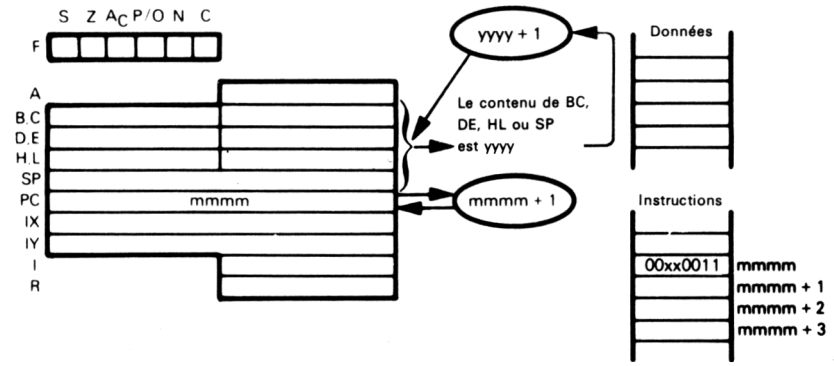


Cette instruction ajoute 1 au contenu du registre spécifié.
Supposons que le registre E contient $A8_{16}$. A la fin de l'exécution de l'instruction

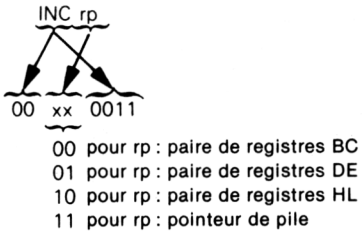
INC E

le registre E contient $A9_{16}$

INC rp — INCRÉMENTATION DU CONTENU DE LA PAIRE DE REGISTRES SPÉCIFIEE
INC IX
INC IY



Ce schéma illustre l'exécution de l'instruction INC rp :



Cette instruction ajoute 1 à la valeur 16 bits contenue dans la paire de registres spécifiée. Aucun indicateur d'état n'est modifié.
Supposons que les registres D et E contiennent $2F7A_{16}$. A la fin de l'exécution de l'instruction INC DE, les registres D et E contiennent $2F7B_{16}$. A la fin de l'exécution de l'instruction

INC DE

les registres D et E contiennent $2F7B_{16}$.

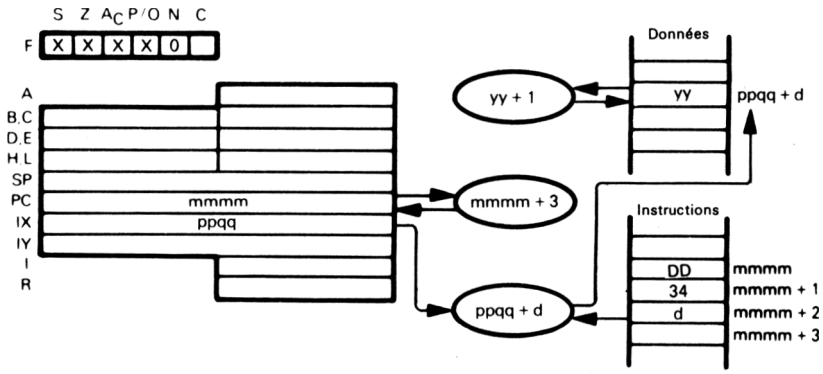
INC IX
DD 23

Cette instruction ajoute 1 à la valeur 16 bits contenue dans le registre IX.

INC IY
FD 23

Cette instruction ajoute 1 à la valeur 16 bits contenue dans le registre IY.
Tout comme les instructions DEC rp, DEC IX et DEC IY, aucune des instructions INC rp, INC IX ou INC IY ne modifie les indicateurs d'état. Ceci est un défaut du jeu d'instructions du Z80 hérité du 8080.

INC (HL) — INCRÉMENTATION D'UN CONTENU MÉMOIRE
INC (IX + disp)
INC (IY + disp)



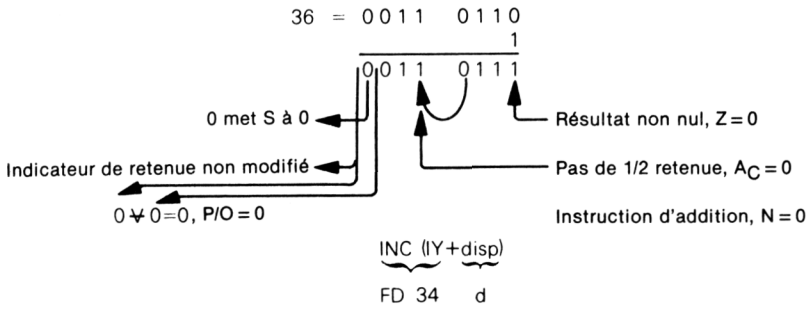
Ce schéma illustre l'exécution de l'instruction INC (IX + d).

INC (IX+disp)
DD 34 d

Cette instruction ajoute 1 au contenu de l'adresse (résultant de la somme du contenu du registre IX et du déplacement d).
Supposons que ppqq = 4000_{16} et que l'adresse $400F_{16}$ contient 36_{16}. A la fin de l'exécution de l'instruction

INC (IX + 0FH)

l'adresse $400F_{16}$ contient 37_{16}.

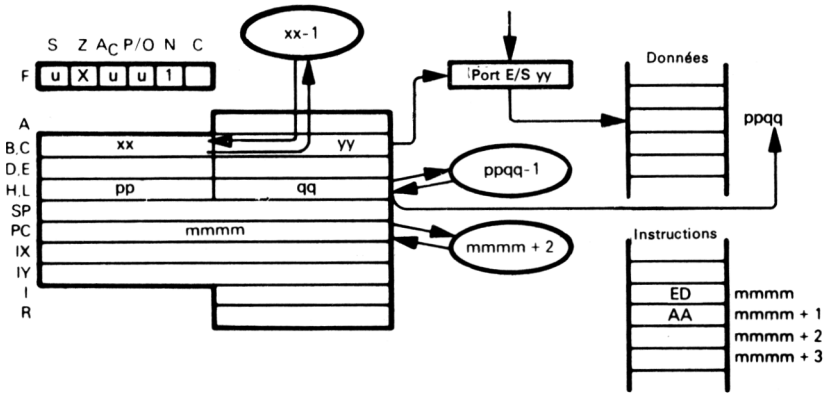


Cette instruction est identique à la précédente, si ce n'est qu'elle utilise le registre IY au lieu du registre IX.

INC (HL)
34

Cette instruction ajoute 1 au contenu de l'adresse (spécifiée par le contenu du double registre HL).

IND — ENTRÉE EN MÉMOIRE ET DÉCRÉMENTATION DU POINTEUR



IND
ED AA

Cette instruction effectue l'entrée à partir du port d'entrée/sortie (E/S), (adressé par le registre C) vers la mémoire dont l'adresse est spécifiée par HL ; puis elle décrémente les registres B et HL.
Supposons que $xx = 05_{16}$, $yy = 15_{16}$, $ppqq = 2400_{16}$ et que le buffer du port d'entrée/sortie (E/S) contient 19_{16}. A la fin de l'exécution de l'instruction

IND

l'adresse 2400_{16} contient 19_{16} ; le registre B 04_{16} et le double registre HL $23FF_{16}$.

INDR — ENTRÉE EN MÉMOIRE ET DÉCRÉMENTATION DU POINTEUR TANT QUE LE COMPTEUR EST NON NUL

INDR
ED BA

L'instruction INDR est identique à IND, mais est répétée jusqu'à ce que le registre B soit égal à 0.
Supposons que le registre B contient 03_{16}, le registre C 15_{16} et HL 2400_{16}. Au port d'entrée/sortie, 15_{16} se présente la série suivante d'octets : 17_{16}, 59_{16} et AE_{16}. A la fin de l'exécution de l'instruction

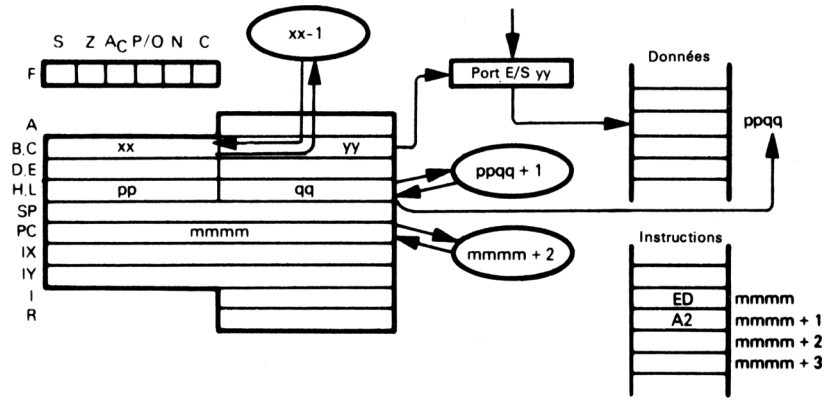
INDR

le double registre HL contient $23FD_{16}$, le registre B zéro et la mémoire est ainsi disposée :

Adresse	Contenu
2400	17_{16}
23FF	59_{16}
23FE	AE_{16}

Cette instruction est extrêmement utile pour le chargement de blocs de données en mémoire à partir d'un périphérique d'entrée.

INI — ENTRÉE EN MÉMOIRE ET INCRÉMENTATION DU POINTEUR



INI
ED A2

Cette instruction effectue l'entrée à partir du port d'entrée/sortie (E/S) (adressé par le registre C), vers la mémoire dont l'adresse est spécifiée par HL ; puis elle décrémente le registre B et incrémente le double registre HL.
Supposons que $xx = 05_{16}$, $yy = 15_{16}$, $ppqq = 2400_{16}$ et que le buffer du port d'entrée/sortie contient 19_{16}. A la fin de l'exécution de l'instruction

INI

l'adresse 2400_{16} contient 19_{16}, le registre B contient 04_{16} et le double registre HL 2401_{16}.

INIR — ENTRÉE EN MÉMOIRE ET INCRÉMENTATION DU POINTEUR TANT QUE LE COMPTEUR EST NON NUL

INIR
ED B2

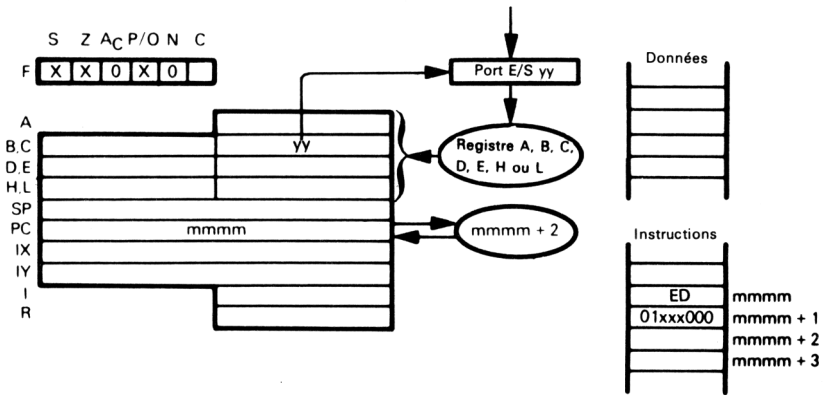
Cette instruction est identique à la précédente mais est répétée jusqu'à ce que le registre B soit nul.
Supposons que le registre B contient 03_{16}, le registre C 15_{16} et que HL contient 2400_{16}. Au port d'entrée/sortie (E/S) 15_{16} se présente la série suivante d'octets : 17_{16}, 59_{16} et AE_{16}. A

la fin de l'exécution de l'instruction INIR, le double registre HL contient 2403_{16}, le registre B zéro et la mémoire est ainsi disposée :

Adresse	Contenu
2400	17_{16}
2401	59_{16}
2402	AE_{16}

Cette instruction est extrêmement utile pour le chargement de blocs de données en mémoire à partir d'un périphérique d'entrée.

IN reg,(C) — ENTRÉE VERS UN REGISTRE





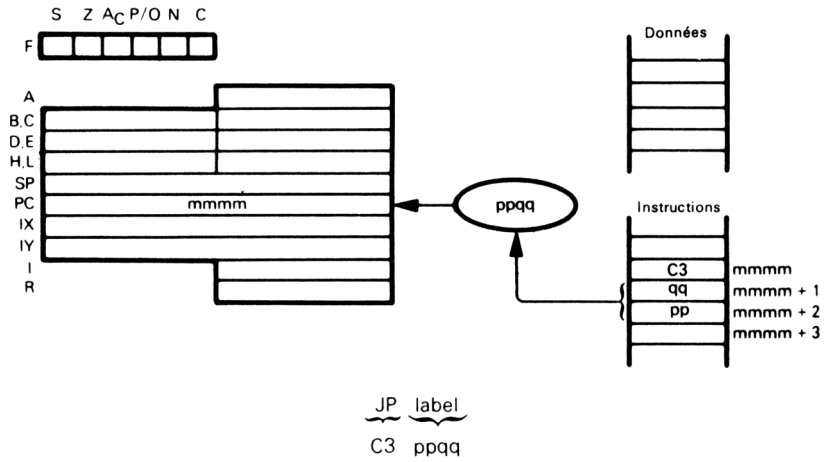
000 pour reg=B
001 pour reg=C
010 pour reg=D
011 pour reg=E
100 pour reg=H
101 pour reg=L
111 pour reg=A
110 pour reg = F

Cette instruction effectue le chargement d'un octet de donnée dans le registre spécifié (reg) à partir du port d'entrée/sortie (identifié par le contenu du registre C).
Supposons que le buffer du port d'entrée/sortie 36_{16} contient 42_{16} et que le registre C contient 36_{16}. A la fin de l'exécution de l'instruction

IN D,(C)

le registre D contient 42_{16}.
Pendant l'exécution de l'instruction, le contenu du registre B est placé sur la partie haute du bus d'adresse, rendant ainsi possible l'extension du nombre de ports d'entrée/sortie adressables.

JP label — SAUT A L'INSTRUCTION IDENTIFIÉE PAR L'ÉTIQUETTE

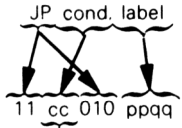


Cette instruction effectue le chargement du contenu des second et troisième octets du code objet de l'instruction JP dans le compteur ordinal ; ce contenu devient l'adresse mémoire de la prochaine instruction à exécuter. Le précédent contenu du compteur ordinal est perdu.
Dans la série d'instructions suivante :

```
        JP      SUIT
        AND     7FH
        —
        —
SUIT:   CPL
```

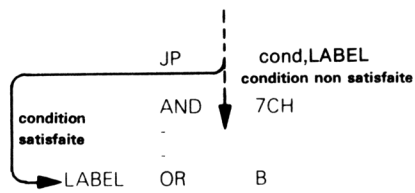
l'instruction CPL est exécutée après l'instruction JP. L'instruction AND n'est jamais exécutée, à moins qu'une instruction de saut dans le reste du programme y opère un branchement.

JP condition, label — SAUT A L'ADRESSE IDENTIFIÉE PAR L'ÉTIQUETTE SI LA CONDITION EST SATISFAITE



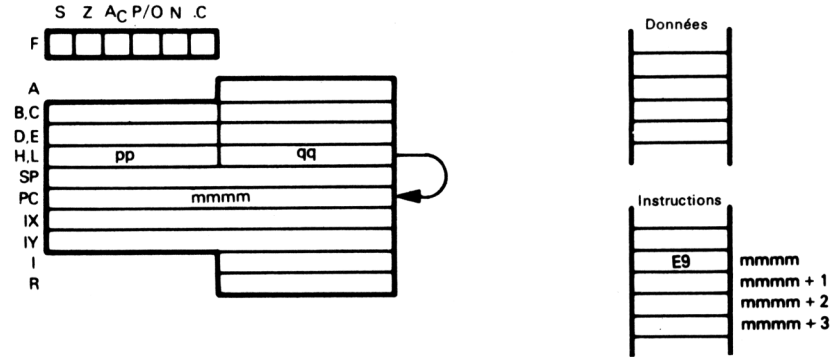
		Condition	Indicateur concerné
000	NZ	non nul (« Non-Zero »)	Z
001	Z	nul (« Zero »)	Z
010	NC	pas de retenue (« No Carry »)	C
011	C	retenue (« Carry »)	C
100	PO	parité impaire (« Parity Odd »)	P/O
101	PE	parité paire (« Parity Even »)	P/O
110	P	signe positif	S
111	M	signe négatif	S

Cette instruction est identique à l'instruction JP, si ce n'est que le saut n'est effectué que si la condition est satisfaite ; sinon, l'instruction qui suit l'instruction JP est exécutée. Considérons la séquence d'instructions suivante :



Après l'exécution de l'instruction JP cond,label, l'instruction OR est exécutée si la condition est satisfaite ; sinon, c'est l'instruction AND, l'instruction suivante, qui est exécutée.

JP (HL) — SAUT A L'ADRESSE SPÉCIFIÉE PAR LE CONTENU
JP (IX) D'UN REGISTRE 16 BITS
JP (IY)



Ce schéma illustre l'exécution de l'instruction JP (HL).

JP (HL)
E9

Cette instruction transfère le contenu du double registre HL dans le compteur ordinal, en conséquence un saut par adressage implicite est utilisé.
Les deux instructions :

```
LD      HL,ADR
JP      (HL)
```

ont exactement le même effet que l'unique instruction :

```
JP      ADR
```

Les deux combinaisons d'instructions spécifient que l'instruction comportant l'étiquette ADR est la prochaine instruction à être exécutée.
L'instruction JP (HL) est utile lorsque l'on veut incrémenter une adresse de retour pour un sous-programme avec plusieurs retours possibles.
Considérons maintenant l'appel de sous-programme suivant :

```
CALL    SPR     ; APPEL SOUS-PROGRAMME
JP      ERR     ; RETOUR AVEC ERREUR
                ; RETOUR SANS ERREUR
```

L'instruction JP ERR est exécutée si l'on utilise l'instruction RET pour revenir du sous-programme SPR ; c'est pourquoi, si SPR s'exécute sans détecter de condition d'erreur, le retour du sous-programme se passe comme suit :

```
POP     HL      ; CHARGEMENT ADRESSE DE RETOUR DANS HL
INC     HL      ; AJOUTER 3 A L'ADRESSE DE RETOUR
INC     HL
INC     HL
JP      (HL)    ; RETOUR
```

JP (IX)
DD E9

Cette instruction est identique à l'instruction JP (HL), si ce n'est qu'elle utilise le registre IX à la place du double registre HL.

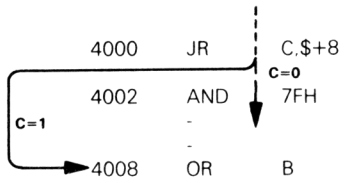
JP (IY)
FD E9

Cette instruction est identique à l'instruction JP (HL), si ce n'est qu'elle utilise le registre IY à la place du double registre HL.

JR C,disp — SAUT RELATIF SI L'INDICATEUR DE RETENUE = 1

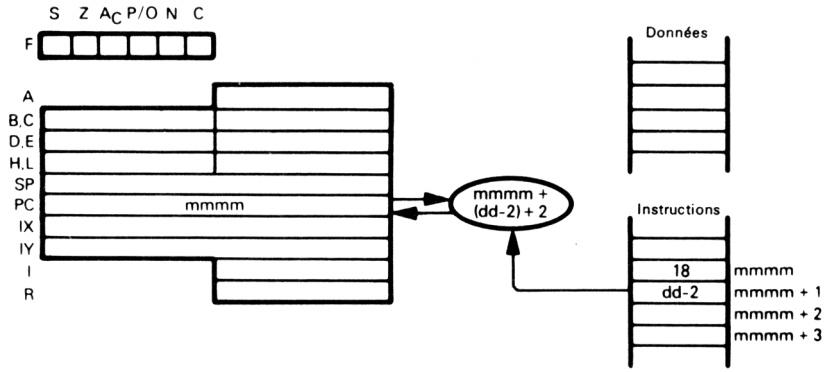
JR C, disp
38 dd-2

Cette instruction est identique à l'instruction JR disp (qui suit), si ce n'est que le saut n'est exécuté que si l'indicateur de retenue vaut 1 ; sinon l'instruction suivante est exécutée.
Soit la séquence d'instructions suivante :



Après l'instruction JR C,$ + 8, l'instruction OR n'est exécutée que si l'indicateur de retenue vaut 1 ; sinon c'est l'instruction AND qui est exécutée.

JR disp — SAUT RELATIF AU CONTENU DU COMPTEUR ORDINAL



JR disp

18 dd-2

Cette instruction fait la somme du second octet du code objet de l'instruction JR, du contenu du compteur ordinal et de la valeur 2. Elle charge cette somme dans le compteur ordinal. Le saut est calculé à partir de l'adresse du code opératoire de l'instruction et a une amplitude de -126 à $+129$ octets. L'assembleur calcule automatiquement le déplacement relatif au compteur ordinal*.
L'instruction qui suit permet de se brancher 4 octets après l'adresse 4000_{16}.

JR $+4

Le résultat de l'instruction est montré ci-dessous :

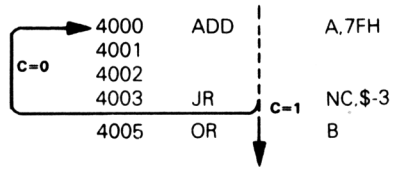
	Adresse	Instruction
	4000	18
	4001	02
[PC] + 0 →	4002	-
[PC] + 1 →	4003	-
Nouvelle valeur du PC = [PC] + 2 →	4004	-

* NDT. En effet, le compteur ordinal (PC) contient en permanence l'adresse de l'instruction qui VA être exécutée, en l'occurence, l'instruction suivant l'instruction JR. Cette dernière comportant 2 octets, la valeur 2 doit donc être ajoutée si l'on calcule le déplacement à partir de l'instruction JR elle-même. Ce travail est généralement pris en charge par l'assembleur pour simplifier la tâche du programmeur.

JR NC,disp — SAUT RELATIF SI L'INDICATEUR DE RETENUE EST NUL

JR NC,disp
30 dd-2

Cette instruction est identique à l'instruction JR disp, si ce n'est que le saut n'est effectué que si l'indicateur de retenue vaut 0 ; sinon l'instruction suivante est exécutée.
Soit la séquence d'instructions suivante :

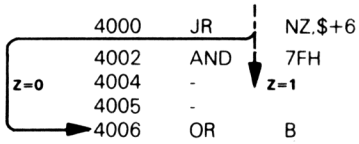


Après l'instruction JR NC,$-3, l'instruction OR n'est exécutée que si l'indicateur de retenue vaut 1. S'il est nul, c'est l'instruction ADD qui est exécutée.

JR, NZ,disp — SAUT RELATIF SI L'INDICATEUR DE ZÉRO EST NUL

JR NZ,disp
20 dd-2

Cette instruction est identique à l'instruction JR disp, si ce n'est que le saut n'est effectué que pour la valeur nulle de l'indicateur de zéro ; dans le cas contraire, l'instruction suivante est exécutée.
Soit la séquence d'instructions suivante :



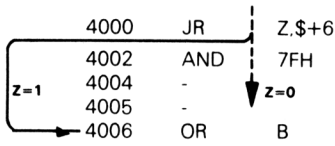
Après l'instruction JR NZ,$+6, l'instruction OR n'est exécutée que si l'indicateur de zéro est nul. L'instruction AND est exécutée si l'indicateur de zéro vaut 1.

NDT. Rappelons que l'indicateur de zéro est positionné à 1 (et non à 0) après un résultat nul.

JR Z,disp — SAUT RELATIF SI L'INDICATEUR DE ZÉRO = 1

JR Z,disp
28 dd-2

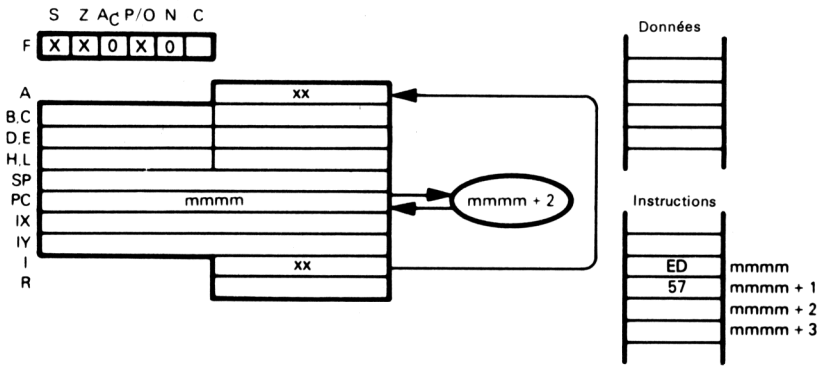
Cette instruction est identique à l'instruction JR disp, si ce n'est que le saut n'est exécuté que si l'indicateur de zéro vaut 1 ; sinon, l'instruction suivante est exécutée.
Soit la séquence d'instructions suivante :



Après l'instruction JR Z,$ + 6, l'instruction OR n'est exécutée que si l'indicateur de zéro vaut 1. L'instruction AND est exécutée s'il est nul.

NDT. Rappelons que l'indicateur de zéro est positionné à 1 (et non à 0) après un résultat nul.

LD A,I — TRANSFERT DU CONTENU DU REGISTRE D'INTERRUPTIONS OU LD A,R DU REGISTRE DE RAFRAÎCHISSEMENT VERS L'ACCUMULATEUR



Le schéma illustre l'exécution de l'instruction :

LD A,I
ED 57

Cette instruction transfère le contenu du registre d'interruptions vers l'accumulateur et assigne la valeur du masque d'interruption à l'indicateur P/O.
Supposons que le registre d'interruptions contient $7F_{16}$ et que les interruptions sont inhibées. A la fin de l'exécution de l'instruction

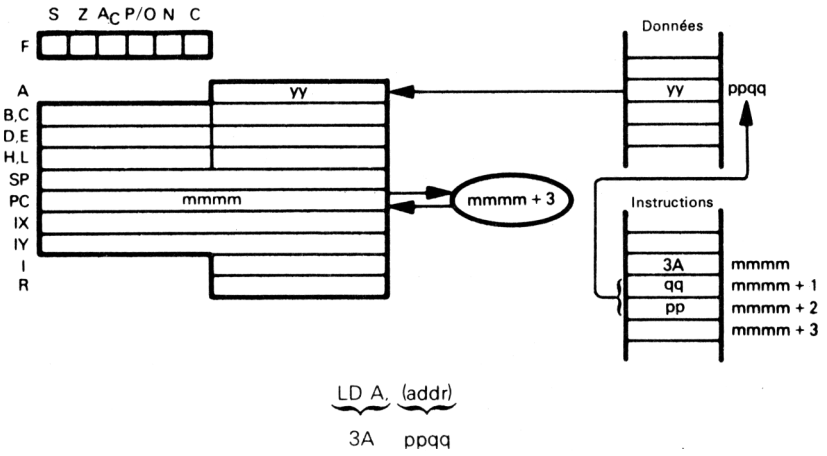
```
LD A,I
```

le registre A contient $7F_{16}$ et P/O = 0.

```
LD A,R
ED 5F
```

Cette instruction transfère le contenu du registre de rafraîchissement vers l'accumulateur. La valeur du masque d'interruption apparaît dans l'indicateur P/O.

LD A,(addr) — CHARGEMENT DE L'ACCUMULATEUR A PARTIR DE LA MÉMOIRE A L'AIDE DE L'ADRESSAGE DIRECT



Cette instruction effectue le chargement dans l'accumulateur de l'octet mémoire (adressé directement par les second et troisième octets du code objet de l'instruction LD A,(addr). Supposons que l'octet mémoire $084A_{16}$ contient 20_{16}. A la fin de l'exécution de l'instruction

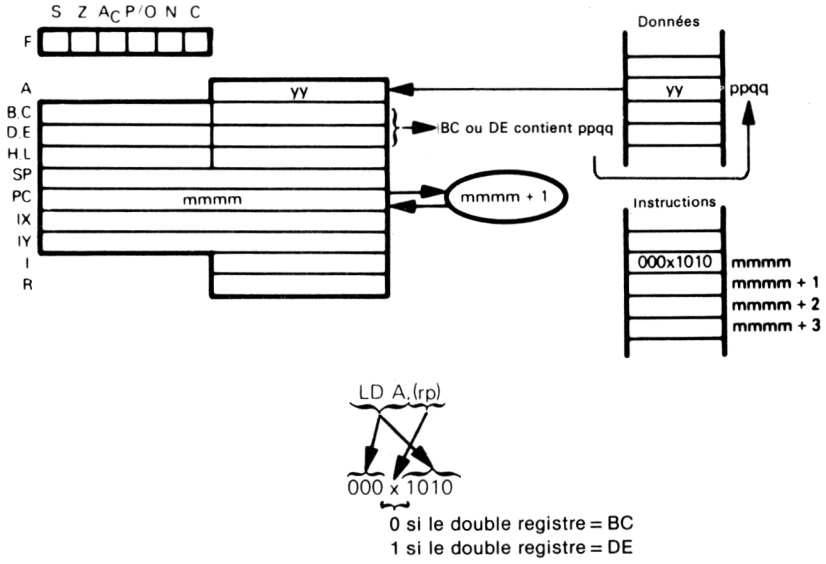
```
label   EQU     084AH
-
-
-
LD      A,(label)
```

l'accumulateur contient 20_{16}.
Il faut se rappeler que EQU est une directive assembleur et non une instruction. Elle indique à l'assembleur d'utiliser la valeur 16 bits $084A_{16}$ chaque fois que l'étiquette label apparaît.
L'instruction LD A,(label) est équivalente aux 2 instructions :

```
LD      HL,label
LD      A,(HL)
```

Lorsqu'on charge une seule valeur à partir de la mémoire, il est préférable d'utiliser l'instruction LD A,(label) ; cela prend une instruction et 3 octets de code objet pour faire ce que la séquence LD HL,label – LD A,(HL) réalise en 2 instructions et 4 octets de code objet. De plus, la séquence des 2 instructions utilise les registres H et L, ce qui n'est pas le cas de l'instruction LD A,(label).

LD A,(rp) — CHARGEMENT DE L'ACCUMULATEUR A PARTIR DE L'EMPLACEMENT MÉMOIRE ADRESSE PAR UN DOUBLE REGISTRE



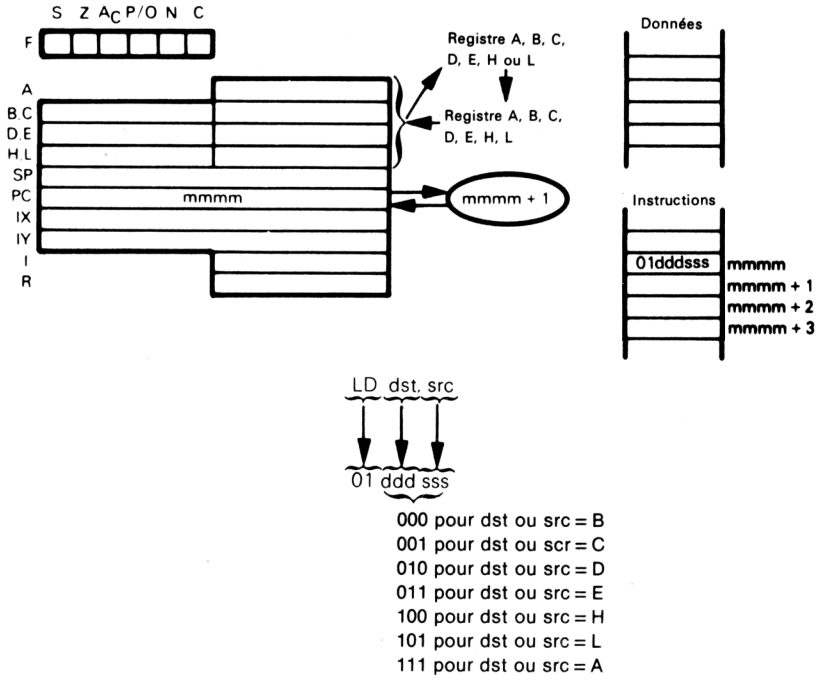
Cette instruction charge dans l'accumulateur le contenu de l'octet memoire (adressé par le double registre BC ou DE).
Supposons que le registre B contient 08_{16}, le registre C $4A_{16}$ et que l'octet de mémoire $084A_{16}$ contient $3A_{16}$. A la fin de l'exécution de l'instruction

LD A,(BC)

l'accumulateur contient $3A_{16}$.
Habituellement, les instructions LD A,(rp) et LD rp,data sont utilisées ensemble, puisque l'instruction LD rp,data charge une adresse 16 bits dans les registres BC ou DE de la façon suivante :

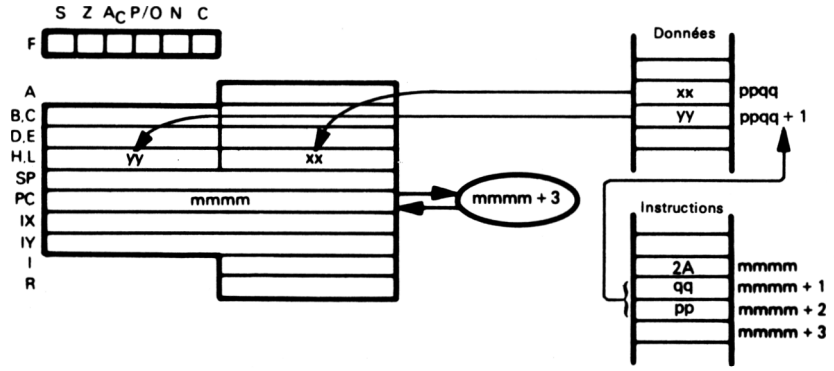
```
LD      BC,084AH
LD      A,(BC)
```

LD dst,src — TRANSFERT DU CONTENU D'UN REGISTRE SOURCE VERS UN REGISTRE DESTINATION



Cette instruction permet de charger le contenu de n'importe quel registre dans n'importe quel autre registre.
Par exemple : l'instruction LD A,B charge le contenu du registre B dans le registre A, l'instruction LD L,D charge le contenu du registre D dans le registre L,
l'instruction LD C,C est sans effet puisque le registre C désigne à la fois le registre source et destination.

LD HL,(addr) — CHARGEMENT D'UNE PAIRE DE REGISTRES OU
LD rp,(addr) D'UN REGISTRE D'INDEX A PARTIR DE LA MÉMOIRE
LD IX,(addr) A L'AIDE DE L'ADRESSAGE DIRECT
LD IY,(addr)



Ce schéma illustre l'exécution de l'instruction LD HL,(ppqq).

LD HL,(addr)
2A ppqq

Cette instruction charge le double registre HL à partir de l'emplacement mémoire adressé directement.
Supposons que l'adresse 4004_{16} contient AD_{16} et que l'adresse 4005_{16} contient 12_{16}. A la fin de l'exécution de l'instruction

LD HL,(4004H)

le double registre HL contient $12AD_{16}$.

LD rp, (addr)
ED 01 dd 1011 ppqq

00 pour rp = BC
01 pour rp = DE
10 pour rp = HL
11 pour rp = Pointeur de pile

Cette instruction charge un double registre à partir de la mémoire adressée directement.
Supposons que l'adresse $49FF_{16}$ contient BE_{16} et que l'adresse $4A00_{16}$ contient 33_{16}. A la fin de l'exécution de l'instruction

LD DE,(49FFH)

le double registre DE contient $33BE_{16}$.

LD IX,(addr)
DD 2A ppqq

Cette instruction charge le registre IX à partir de la mémoire adressée directement. Supposons que l'adresse $D111_{16}$ contient FF_{16} et que l'adresse $D112_{16}$ contient 56_{16}. A la fin de l'exécution de l'instruction

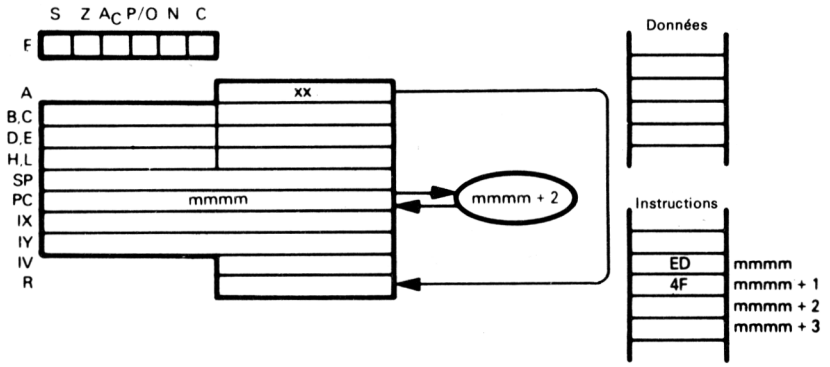
LD IX,(D111H)

le registre IX contient $56FF_{16}$.

LD IY,(addr)
FD 2A ppqq

Cette instruction charge le registre IY à partir de la mémoire adressée directement. Elle modifie le registre IY à la place de IX ; elle est sur tout autre point identique à l'instruction LD IX,(addr).

LD I,A — CHARGEMENT DU REGISTRE D'INTERRUPTION OU LD R,A DU REGISTRE DE RAFRAÎCHISSEMENT A PARTIR DE L'ACCUMULATEUR



Ce schéma illustre l'exécution de l'instruction LD R,A.

LD R,A
ED 4F

Cette instruction charge le registre de rafraîchissement à partir de l'accumulateur.

Supposons que l'accumulateur contient $7F_{16}$. A la fin de l'exécution de l'instruction

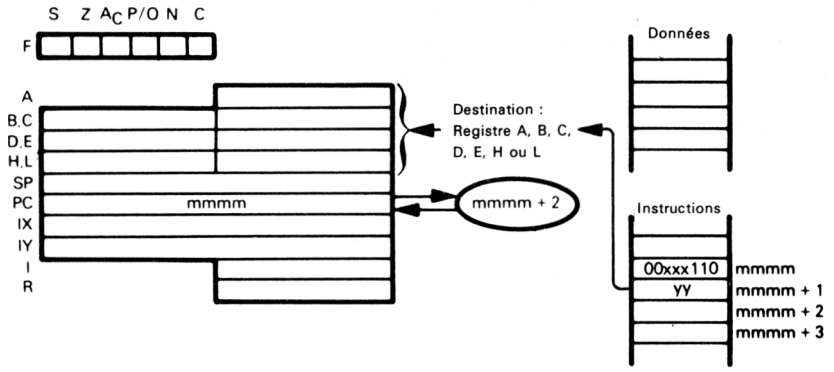
LD R,A

le registre de rafraîchissement contient $7F_{16}$.

LD I,A
ED 47

Cette instruction charge le registre d'interruption à partir de l'accumulateur.

LD reg,data — CHARGEMENT IMMÉDIAT DANS UN REGISTRE



LD reg,data

00 xxx 110 yy

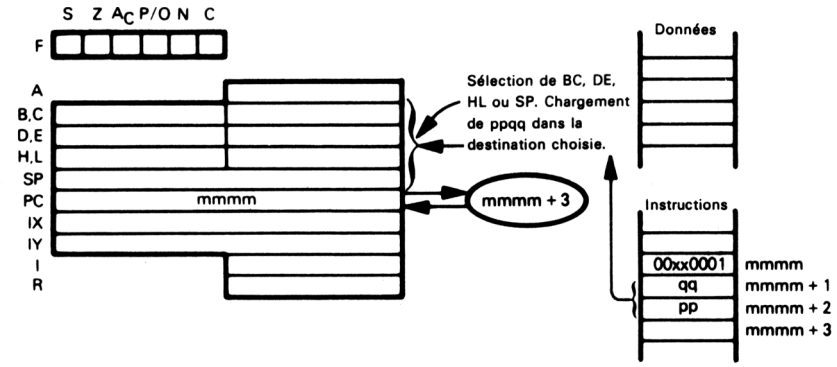
000 pour reg=B
001 pour reg=C
010 pour reg=D
011 pour reg=E
100 pour reg=H
101 pour reg=L
111 pour reg=A

Cette instruction charge le contenu du second octet du code objet dans l'un des registres. A la fin de l'exécution de l'instruction

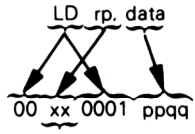
LD A,2AH

l'accumulateur contient $2A_{16}$.

LD rp,data — CHARGEMENT IMMÉDIAT D'UNE DONNÉE 16 BITS
LD IX,data DANS UN REGISTRE
LD IY,data



Ce schéma illustre l'exécution de l'instruction LD rp,data.



00 pour rp = BC
01 pour rp = DE
10 pour rp = HL
11 pour rp : Pointeur de pile

Cette instruction effectue le chargement des second et troisième octet du code objet dans le double registre choisi. A la fin de l'exécution de l'instruction

LD SP,217AH

le pointeur de pile contient 217AH.

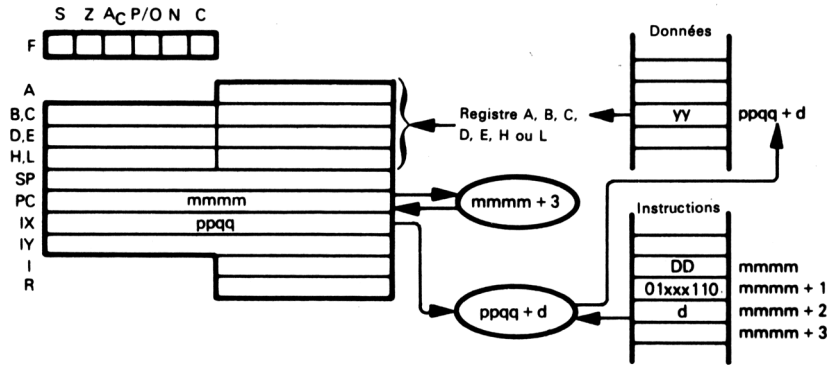
LD IX, data
DD 21 ppqq

Cette instruction charge le contenu des second et troisième octets du code objet dans le registre d'index IX.

LD IY, data
FD 21 ppqq

Cette instruction charge le contenu des second et troisième octets du code objet dans le registre d'index IY.
Remarque : l'instruction LD rp,data équivaut à 2 instructions LD reg,data.
Par exemple : LD HL,032AH est équivalente aux 2 instructions LD H,03H et LD L,2AH.

LD reg,(HL) — CHARGEMENT D'UN REGISTRE A PARTIR DE LA MÉMOIRE
LD reg,(IX + disp)
LD reg,(IY + disp)

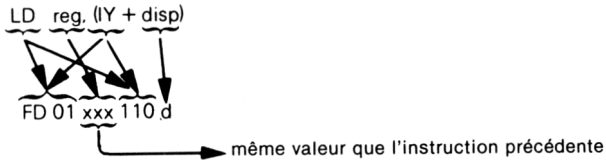


Ce schéma illustre l'exécution de l'instruction :

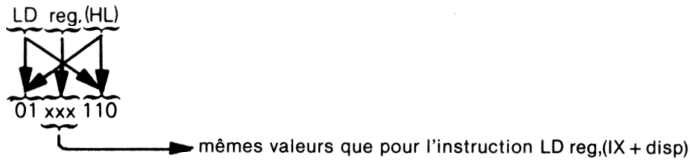
LD reg, (IX + disp)
DD 01 xxx 110 d

000 pour reg=B
001 pour reg=C
010 pour reg=D
011 pour reg=E
100 pour reg=H
101 pour reg=L
111 pour reg=A

Cette instruction effectue le chargement du registre spécifié à partir de l'adresse (résultant de la somme du contenu du registre IX et du déplacement d).
Supposons que ppqq = 4004_{16} et que l'adresse 4010_{16} contient FF_{16}. A la fin de l'exécution de l'instruction LD B,(IX + 0CH), le registre B contiént FF_{16}.

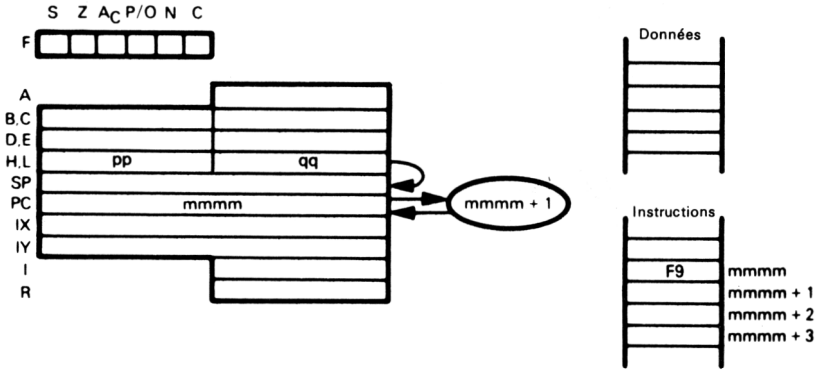


Cette instruction est identique à LD reg,(IX + disp), si ce n'est qu'elle utilise le registre IY au lieu du registre IX.

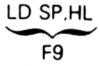


Cette instruction charge le registre spécifié à partir de l'adresse contenue dans le double registre HL.

LD SP,HL — TRANSFERT DU CONTENU DE HL OU D'UN REGISTRE
LD SP,IX D'INDEX VERS LE POINTEUR DE PILE
LD SP,N



Ce schéma illustre l'exécution de l'instruction LD SP,HL :



Cette instruction charge le contenu de HL dans le pointeur de pile. Supposons que $pp = 08_{16}$ et $qq = 3F_{16}$. A la fin de l'exécution de l'instruction

LD SP,HL

le pointeur de pile contient $083F_{16}$.

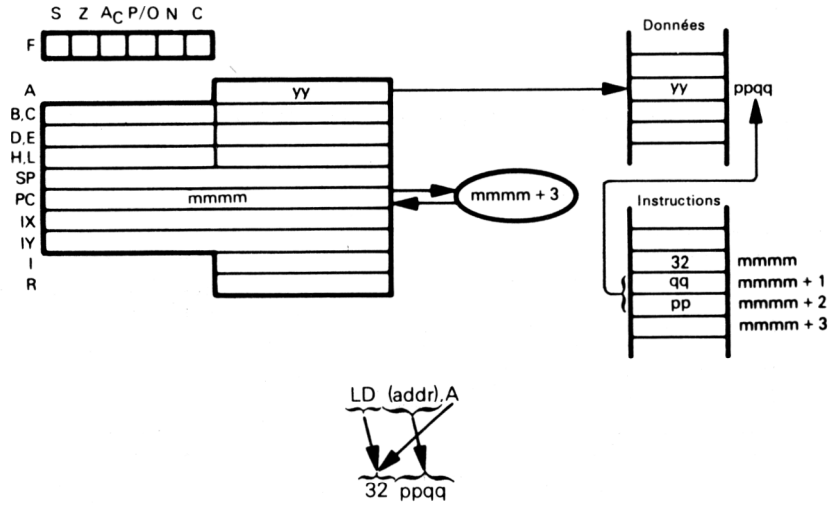
LD SP,IX
DD F9

Cette instruction charge le contenu du registre d'index IX dans le pointeur de pile.

LD SP,IY
FD F9

Cette instruction charge le contenu du registre IY dans le pointeur de pile.

LD (addr),A — RANGEMENT DE L'ACCUMULATEUR EN MÉMOIRE PAR ADRESSAGE DIRECT



Cette instruction permet de ranger le contenu de l'accumulateur à l'octet mémoire adressé directement par les second et troisième octets du code objet de l'instruction LD (addr),A.

Supposons que l'accumulateur contient $3A_{16}$. A la fin de l'exécution de l'instruction

```
label   EQU     084AH
        -
        -
        LD      (label),A
```

l'emplacement mémoire d'adresse 084AH contient $3A_{16}$.
Il faut se rappeler que EQU est une directive assembleur et non une instruction ; elle indique à l'assembleur d'utiliser la valeur 16 bits 084AH chaque fois que le mot « label » apparaît.

L'instruction

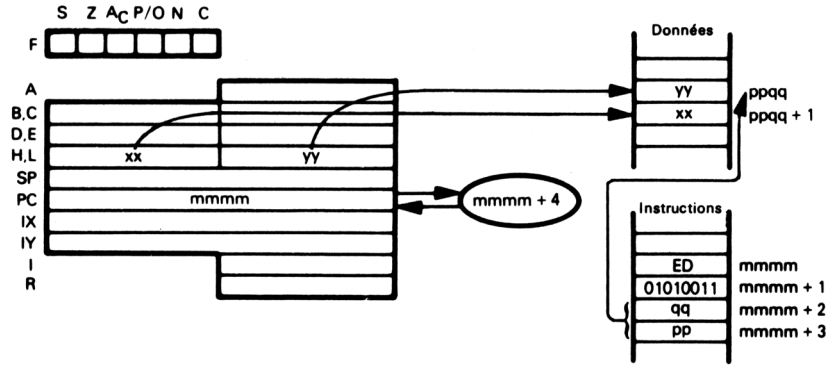
```
LD (addr),A
```

est équivalente aux 2 instructions

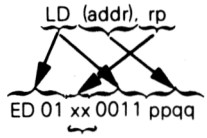
```
LD HL,label
LD (HL),A
```

Lorsque l'on range une seule donnée en mémoire, il est préférable d'utiliser l'instruction LD (label),A ; cela prend une instruction et 3 octets de code objet au lieu de 2 instructions et 4 octets de code objet pour la séquence : LD HL,label – LD (HL),A. De plus, cette séquence utilise les 2 registres H et L, contrairement à l'instruction LD (label),A.

LD (addr),HL — RANGEMENT EN MÉMOIRE DU CONTENU D'UN DOUBLE
LD (addr),rp REGISTRE OU D'UN REGISTRE D'INDEX PAR ADRESSAGE
LD (addr),xy DIRECT



Ce schéma illustre l'exécution de l'instruction LD (ppqq),rp.

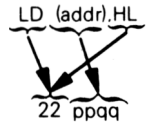


00 pour rp = BC
01 pour rp = DE
10 pour rp = HL
11 pour rp = SP (pointeur de pile)

Cette instruction effectue le rangement en mémoire du contenu de la paire de registres spécifiée. Les troisième et quatrième octets du code objet donnent l'adresse de l'emplacement mémoire où l'octet de bas rang doit être écrit. L'octet de haut rang est écrit à l'emplacement mémoire suivant.
Supposons que la paire de registres BC contient $3C2A_{16}$. A la fin de l'exécution de l'instruction

```
label   EQU   084AH
-
-
-
LD    (label),BC
```

l'octet mémoire $084A_{16}$ contient $2A_{16}$. L'octet mémoire $084B_{16}$ contient $3C_{16}$.
Il faut se rappeler que EQU est une directive assembleur et non une instruction ; elle indique à l'assembleur d'utiliser la valeur 16 bits $084A_{16}$, chaque fois que le mot « label » intervient.



Cette instruction est la version sur 3 octets de l'instruction LD (addr),rp qui précise HL comme double registre source.

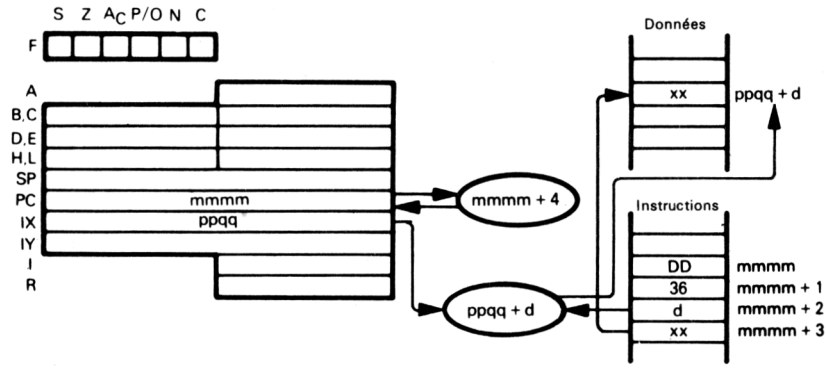


Cette instruction range en mémoire le contenu du registre d'index IX. Les troisième et quatrième octets du code objet donnent l'adresse de l'emplacement mémoire où l'octet de bas rang doit être écrit. L'octet de haut rang est écrit à l'adresse suivante.

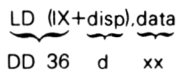


Cette instruction est identique à LD (addr),IX, si ce n'est qu'elle utilise le registre IY à la place du registre IX.

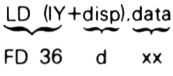
LD (HL),data — CHARGEMENT D'UNE VALEUR IMMÉDIATE EN MÉMOIRE
LD (IX + disp),data
LD (IY + disp),data



Ce schéma illustre l'exécution de l'instruction LD (IX + d),xx.



Cette instruction effectue un chargement immédiat à l'adresse désignée par adressage relatif à une base.
Supposons que ppqq = 5400_{16}. A la fin de l'exécution de l'instruction LD (IX + 9),0FAH, l'adresse 5409_{16} contient FA_{16}.

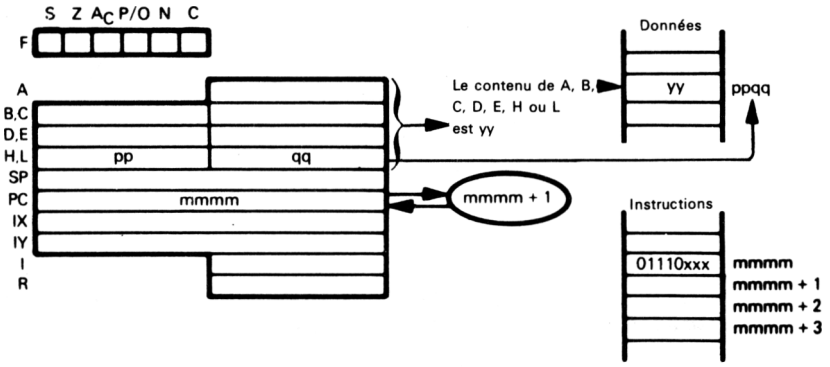


Cette instruction est identique à la précédente, si ce n'est qu'elle utilise le registre IY au lieu du registre IX.

LD (HL),data
36 xx

Cette instruction effectue le chargement immédiat à l'adresse contenue par le double registre HL.
Les instructions de chargement immédiat en mémoire sont beaucoup moins utilisées que les instructions de chargement immédiat dans les registres.

LD (HL),reg — CHARGEMENT EN MÉMOIRE A PARTIR D'UN REGISTRE
LD (IX + disp),reg
LD (IY + disp),reg



Ce schéma illustre l'exécution de l'instruction LD (HL),reg :

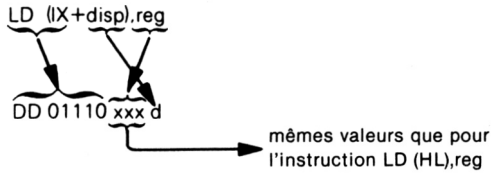
LD (HL),reg
01110 xxx

000 pour reg=B
001 pour reg=C
010 pour reg=D
011 pour reg=E
100 pour reg=H
101 pour reg=L
111 pour reg=A

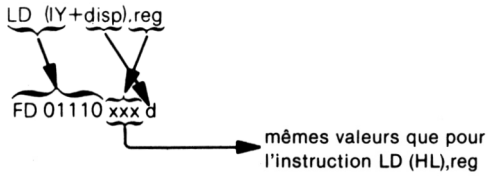
Cette instruction effectue le chargement du contenu de l'adresse (contenue dans le double registre HL) à partir du registre spécifié.
Supposons que ppqq = 4500_{16} et que le registre C contient $F9_{16}$. A la fin de l'exécution de l'instruction

LD (HL),C

l'adresse 4500_{16} contient $F9_{16}$.

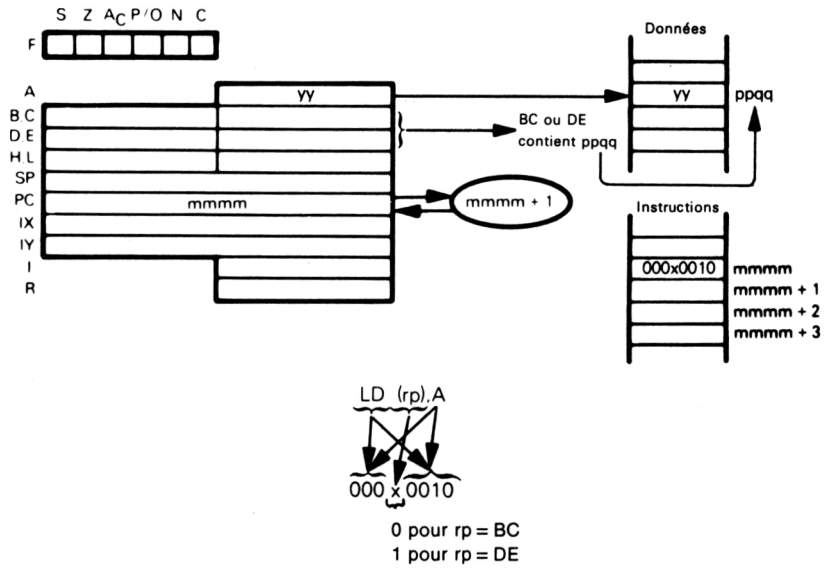


Cette instruction effectue le chargement du contenu de l'adresse (résultant de la somme du contenu du registre IX et du déplacement d), à partir du registre spécifié.



Cette instruction est identique à LD (IX + disp),reg, si ce n'est qu'elle utilise le registre IY à la place du registre IX.

LD (rp),A — CHARGEMENT DE L'ACCUMULATEUR A L'EMPLACEMENT MÉMOIRE ADRESSÉ PAR UN DOUBLE REGISTRE



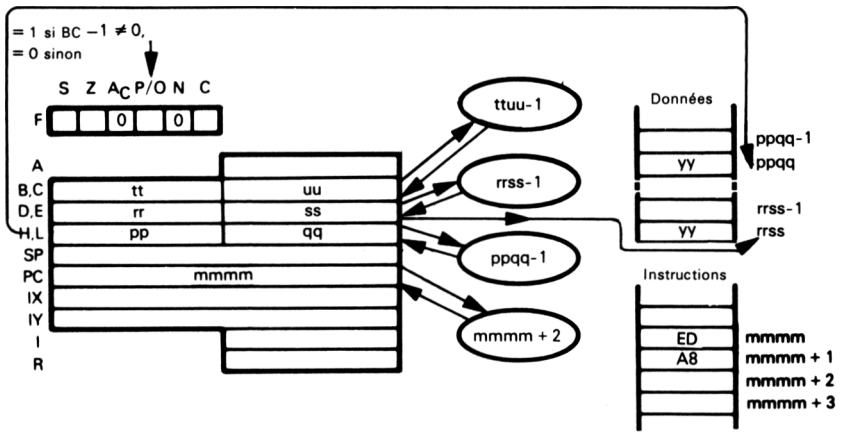
Cette instruction effectue le rangement du contenu de l'accumulateur dans l'octet mémoire adressé par le double registre BC ou DE.
Supposons que le double registre BC contient $084A_{16}$ et que l'accumulateur contient $3A_{16}$. A la fin de l'exécution de l'instruction

```
LD (BC),A
```

l'octet mémoire $084A_{16}$ contient $3A_{16}$.
Les instructions LD (rp),A et LD rp,data sont habituellement utilisées ensemble puisque la dernière instruction charge une adresse 16 bits dans les registres BC ou DE comme ci-dessous :

```
LD BC,084AH
LD (BC),A
```

LDD — TRANSFERT DE DONNÉES MÉMOIRE A MÉMOIRE DÉCRÉMENTATION DES ADRESSES SOURCE ET DESTINATION



LDD
ED A8

Cette instruction transfère un octet de donnée à partir de l'emplacement mémoire adressé par le double registre HL vers l'emplacement mémoire adressé par le double registre DE. Elle décrémente ensuite le contenu des doubles registres BC, DE et HL.
Supposons que le double registre BC contient $004F_{16}$, DE 4545_{16}, HL 2012_{16} et que l'adresse 2012_{16} contient 18_{16}. A la fin de l'exécution de l'instruction

```
LDD
```

l'adresse 4545_{16} contient 18_{16}, le double registre BC contient $004E_{16}$, DE 4544_{16} et HL 2011_{16}.

LDDR — TRANSFERT DE DONNÉES MÉMOIRE A MÉMOIRE TANT QUE LE COMPTEUR EST NON NUL. DÉCRÉMENTATION DES ADRESSES SOURCE ET DESTINATION

LDDR
ED B8

Cette instruction est identique à l'instruction LDD, si ce n'est qu'elle se répète tant que le double registre BC est non nul. Après chaque transfert de données, les interruptions (éventuelles) sont prises en compte et deux cycles de rafraîchissement sont exécutés.

Supposons que les contenus de la mémoire et des doubles registres sont ainsi disposés :

Registre/Contenu		Adresse/Contenu	
HL	2012_{16}	2012_{16}	18_{16}
DE	4545_{16}	2011_{16}	AA_{16}
BC	0003_{16}	2010_{16}	25_{16}

A la fin de l'exécution de l'instruction

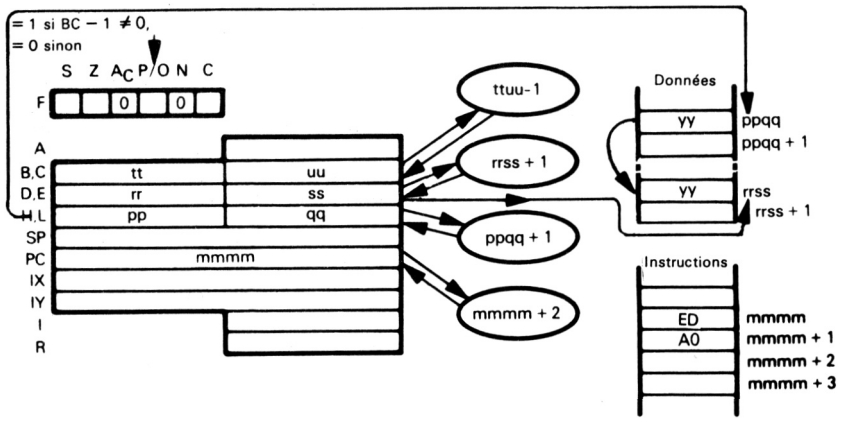
LDDR

les paires de registres et les adresses présentent les contenus suivants :

Registre/Contenu		Adresse/Contenu		Adresse/Contenu	
HL	2009_{16}	2012_{16}	18_{16}	4545_{16}	18_{16}
DE	4542_{16}	2011_{16}	AA_{16}	4544_{16}	AA_{16}
BC	0000_{16}	2010_{16}	25_{16}	4543_{16}	25_{16}

Cette instruction est extrêmement utile pour le transfert de blocs de données d'une zone mémoire à une autre.

LDI — TRANSFERT DE DONNÉES MÉMOIRE A MÉMOIRE INCRÉMENTATION DES ADRESSES SOURCE ET DESTINATION



LDI
ED A0

Cette instruction effectue le transfert d'un octet de donnée à partir de l'emplacement mémoire adressé par le double registre HL vers l'emplacement mémoire adressé par le double registre DE. Elle incrémente ensuite le contenu des doubles registres HL et DE et décrémente celui du double registre BC.
Supposons que le double registre BC contient $004F_{16}$, DE contient 4545_{16}, HL contient 2012_{16} et que l'adresse 2012_{16} contient 18_{16}. A la fin de l'exécution de l'instruction

LDI

l'adresse 4545_{16} contient 18_{16}, le double registre BC contient $004E_{16}$, DE 4546_{16} et HL 2013_{16}.

LDIR — TRANSFERT DE DONNÉES MÉMOIRE A MÉMOIRE TANT QUE LE COMPTEUR EST NON NUL INCRÉMENTATION DES ADRESSES SOURCE ET DESTINATION

LDIR
ED B0

Cette instruction est identique à LDI, si ce n'est qu'elle se répète tant que le double registre BC n'est pas nul. Après chaque transfert de données, les interruptions (éventuelles) sont prises en compte et deux cycles de rafraîchissement sont exécutés.

Supposons que les contenus de la mémoire et des registres sont ainsi disposés :

Registre/Contenu		Adresse/Contenu	
HL	2012_{16}	2012_{16}	18_{16}
DE	4545_{16}	2013_{16}	CD_{16}
BC	0003_{16}	2014_{16}	$F0_{16}$

A la fin de l'exécution de l'instruction

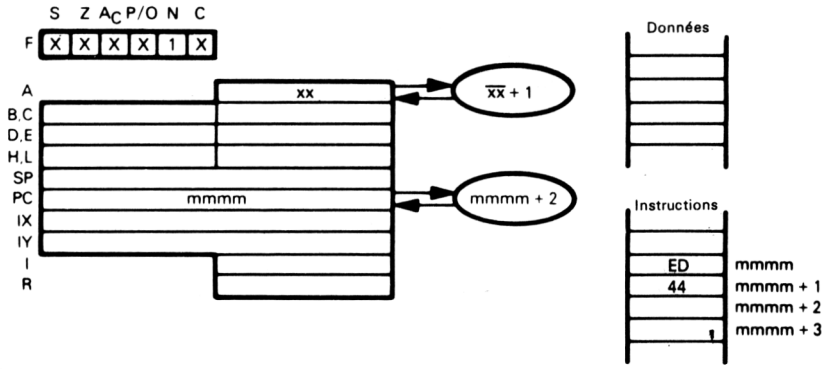
LDIR

les doubles registres et les adresses présentent les contenus suivants :

Registre/Contenu		Adresse/Contenu		Adresse/Contenu	
HL	2015_{16}	2012_{16}	18_{16}	4545_{16}	18_{16}
DE	4548_{16}	2013_{16}	CD_{16}	4546_{16}	CD_{16}
BC	0000_{16}	2014_{16}	$F0_{16}$	4547_{16}	$F0_{16}$

Cette instruction est extrêmement utile pour le transfert de blocs de données d'une zone mémoire à une autre.

NEG — CHANGEMENT DE SIGNE DU CONTENU DE L'ACCUMULATEUR (« NEGATE »)



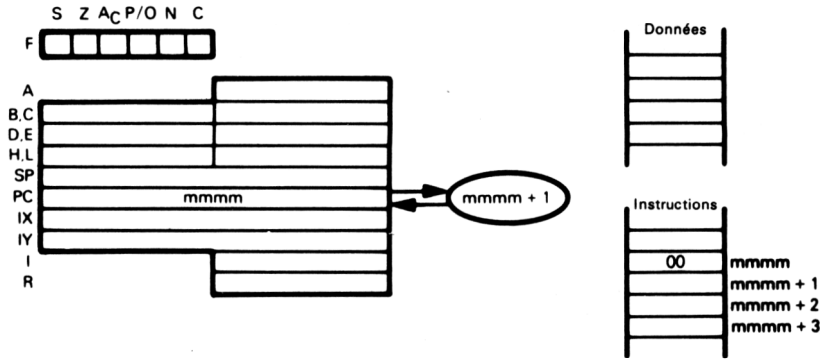
Cette instruction change le signe de l'accumulateur. Ceci revient à soustraire le contenu de l'accumulateur de zéro. Le résultat donne la complémentation à 2.
80H reste inchangé par cette instruction.
Supposons que $xx = 5A_{16}$. A la fin de l'exécution de l'instruction

NEG

l'accumulateur contient $A6_{16}$.

5A = 0101 1010
Complément à 2 = 1010 0110

NOP — PAS D'OPÉRATION (« NO OPERATION »)



NOP
00

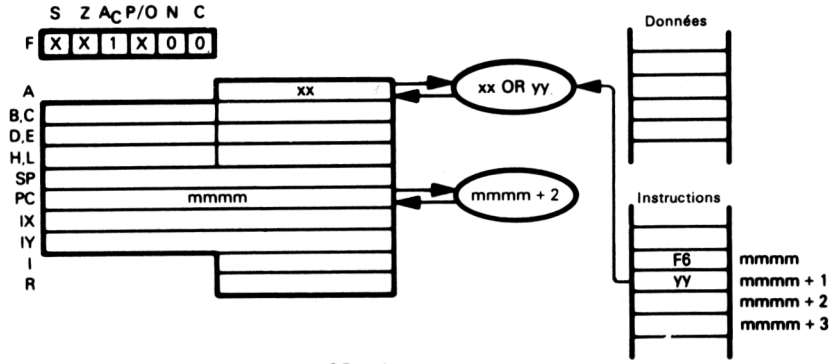
C'est une instruction d'un octet qui n'effectue aucune opération, si ce n'est l'incrémentation du compteur ordinal (le rafraîchissement de la mémoire ne cesse pas non plus). Cette instruction existe pour plusieurs raisons :

1) Une erreur de programme qui cherche un code objet à partir d'une mémoire qui n'existe pas, trouvera 00. C'est bien de savoir que l'erreur de programme la plus banale n'aura pas d'effet.
2) L'instruction NOP permet de donner une étiquette à un octet du code objet :

```
ICI : NOP
```

3) Elle permet de régler de façon précise les temporisations. Chaque instruction NOP ajoute 4 cycles d'horloge à un retard.
L'instruction NOP n'est ni très utile, ni fréquemment utilisée.

OR data — OU LOGIQUE AVEC L'ACCUMULATEUR

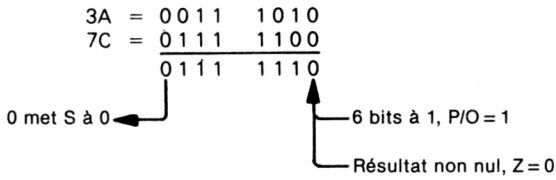


OR data
F6 yy

Cette instruction effectue un OU logique du contenu de l'accumulateur avec le contenu du second octet du code objet de l'instruction.
Supposons que xx = $3A_{16}$. A la fin de l'exécution de l'instruction

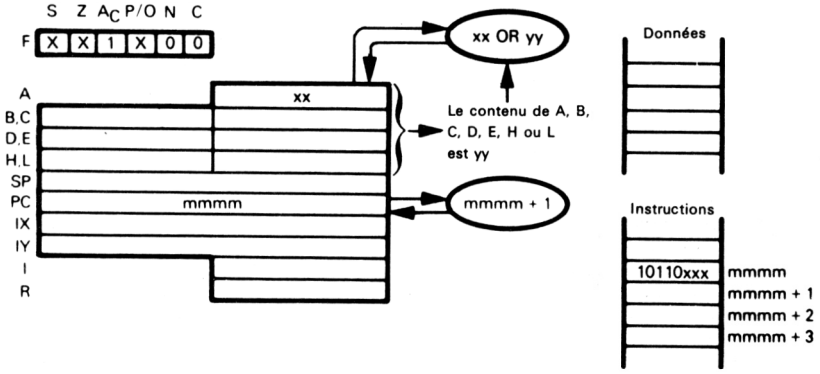
```
OR 7CH
```

l'accumulateur contient $7E_{16}$.



C'est une instruction logique qui sert souvent à mettre les bits à 1.
Par exemple, l'instruction OR 80H met inconditionnellement à 1 le bit de poids fort de l'accumulateur.

OR reg — OU LOGIQUE D'UN REGISTRE AVEC L'ACCUMULATEUR

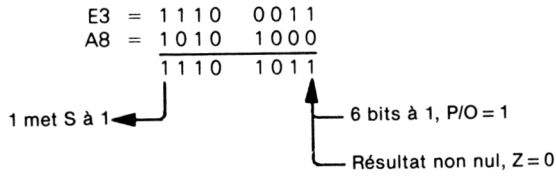


OR reg
10110 xxx
000 pour reg=B
001 pour reg=C
010 pour reg=D
011 pour reg=E
100 pour reg=H
101 pour reg=L
111 pour reg=A

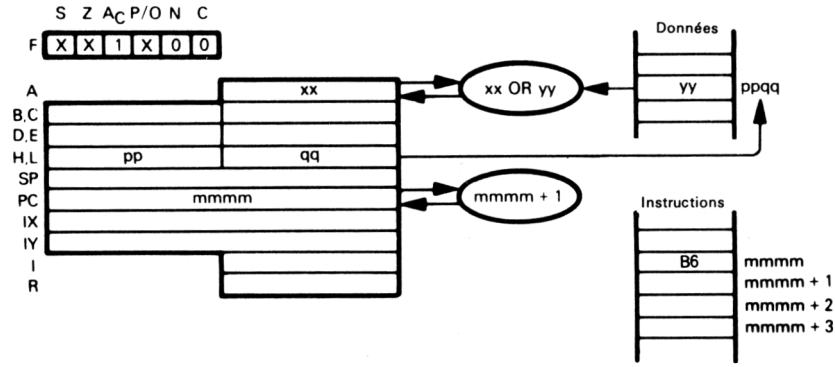
Cette instruction effectue le OU logique du contenu de l'accumulateur avec le contenu du registre A, B, C, D, E, H ou L. Le résultat est rangé dans l'accumulateur.
Supposons que $xx = E3_{16}$ et que le registre E contient $A8_{16}$. A la fin de l'exécution de l'instruction

OR E

l'accumulateur contient EB_{16}.



OR (HL) — OU LOGIQUE DE LA MÉMOIRE AVEC L'ACCUMULATEUR
OR (IX + disp)
OR (IY + disp)



Ce schéma illustre l'exécution de l'instruction OR (HL) :

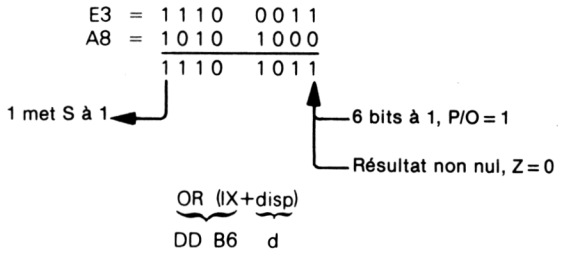
OR (HL)
B6

Cette instruction effectue un OU logique qui entre le contenu de l'adresse (spécifiée par le contenu du double registre HL) et celui de l'accumulateur.

Supposons que $xx = E3_{16}$, $ppqq = 4000_{16}$ et l'adresse 4000_{16} contient $A8_{16}$. A la fin de l'exécution de l'instruction

OR (HL)

l'accumulateur contient EB_{16}.

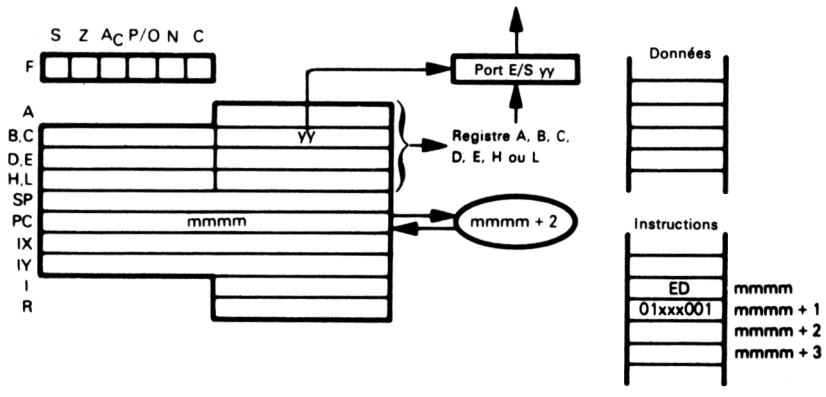


Cette instruction effectue un OU logique entre le contenu de l'adresse (résultant de la somme du contenu du registre IX avec le déplacement d) et le contenu de l'accumulateur.

OR (IY+disp)
FD B6 d

Cette instruction est identique à OR (IX + disp), si ce n'est qu'elle utilise le registre IY à la place de IX.

OUT (C),reg — SORTIE A PARTIR D'UN REGISTRE





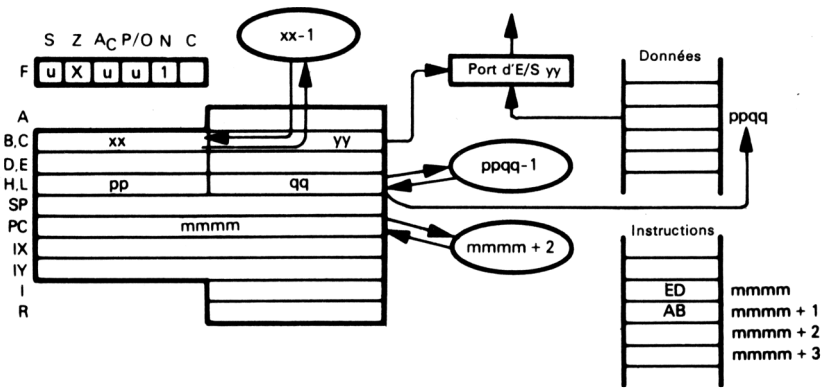
000 pour reg=B
001 pour reg=C
010 pour reg=D
011 pour reg=E
100 pour reg=H
101 pour reg=L
111 pour reg=A

Supposons que yy = $1F_{16}$ et que le registre H contient AA_{16}. A la fin de l'exécution de l'instruction

OUT (C),H

le buffer du port d'entrée/sortie (E/S) $1F_{16}$ contient AA_{16}.

OUTD — SORTIE A PARTIR DE LA MÉMOIRE DÉCRÉMENTATION DE L'ADRESSE



OUTD
ED AB

Cette instruction effectue la sortie d'une adresse contenue par le double registre HL vers le port d'entrée/sortie (E/S) adressé par le registre C. Les registres B et HL sont décrémentés.

Supposons que xx = $0A_{16}$, yy = FF_{16}, ppqq = 5000_{16} et que l'adresse 5000_{16} contient 77_{16}. A la fin de l'exécution de l'instruction

OUTD

le buffer du port d'E/S FF_{16} contient 77_{16}, le registre B contient 09_{16} et le double registre HL $4FFF_{16}$.

OTDR — SORTIE A PARTIR DE LA MÉMOIRE, DÉCRÉMENTATION DE L'ADRESSE TANT QUE LE REGISTRE B EST NON NUL

OTDR
ED BB

Cette instruction est identique à OUTD, mais elle se répète tant que le registre B n'est pas nul.

Supposons que le registre B contient 03_{16}, le registre C FF_{16} et le registre HL 5000_{16}. Les adresses $4FFE_{16}$ à 5000_{16} contiennent :

Adresse/Contenu	
$4FFE_{16}$	CA_{16}
$4FFF_{16}$	$1B_{16}$
5000_{16}	$F1_{16}$

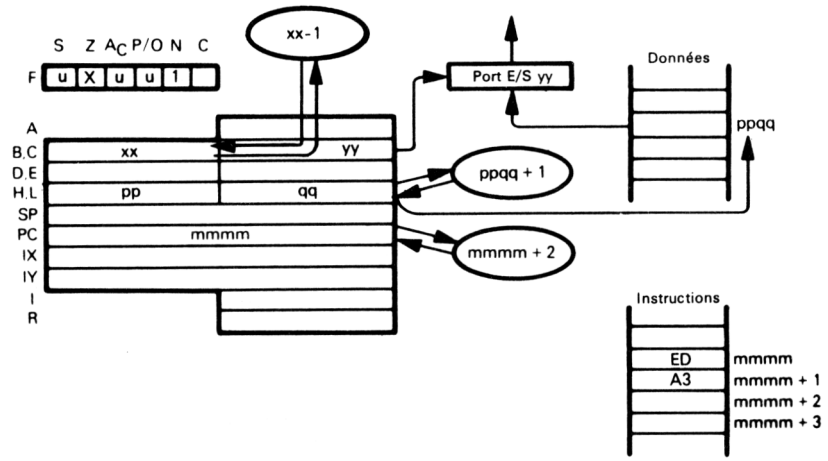
A la fin de l'exécution de l'instruction

OTDR

le double registre HL contient $4FFD_{16}$, le registre B est nul et la séquence $F1_{16}$, $1B_{16}$, CA_{16} a été inscrite sur le port d'E/S FF_{16}.

Cette instruction est très utile pour les transferts de blocs de données de la mémoire vers un périphérique de sortie.

OUTI — SORTIE A PARTIR DE LA MÉMOIRE INCRÉMENTATION DE L'ADRESSE



OUTI
ED A3

Cette instruction effectue la sortie du contenu de l'adresse spécifiée par HL vers le port d'entrée/sortie (E/S) adressé par le registre C. Le registre B est décrémenté tandis que le double registre HL est incrémenté.

Supposons que $xx = 0A_{16}$, $yy = FF_{16}$, $ppqq = 5000_{16}$ et que l'adresse 5000_{16} contient 77_{16}. A la fin de l'exécution de l'instruction

OUTI

le buffer du port d'E/S FF_{16} contient 77_{16}. Le registre B contient 09_{16} et le double registre HL contient 5001_{16}.

OTIR — SORTIE A PARTIR DE LA MÉMOIRE, INCRÉMENTATION DE L'ADRESSE TANT QUE LE REGISTRE B EST NON NUL

OTIR
ED B3

Cette instruction est identique à OUTI, si ce n'est qu'elle se répète tant que le registre B est non nul.

On suppose que le registre B contient 04_{16}, le registre C FF_{16} et que HL contient 5000_{16}. La mémoire est disposée de la façon suivante :

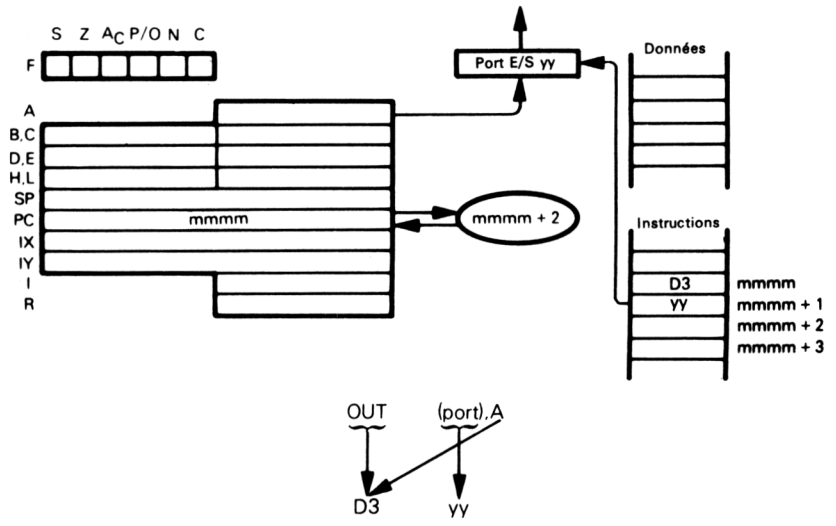
Adresse/Contenu	
5000_{16}	CA_{16}
5001_{16}	$1B_{16}$
5002_{16}	$B1_{16}$
5003_{16}	AD_{16}

A la fin de l'exécution de l'instruction

OTIR

le double registre HL contient 5004_{16}, le registre B est nul, et la séquence des valeurs CA_{16}. $1B_{16}$, $B1_{16}$ et AD_{16} a été inscrite sur le port d'E/S FF_{16}.
Cette instruction est très utile pour transférer des blocs de données à partir de la mémoire vers un périphérique de sortie.

OUT (port),A — SORTIE A PARTIR DE L'ACCUMULATEUR

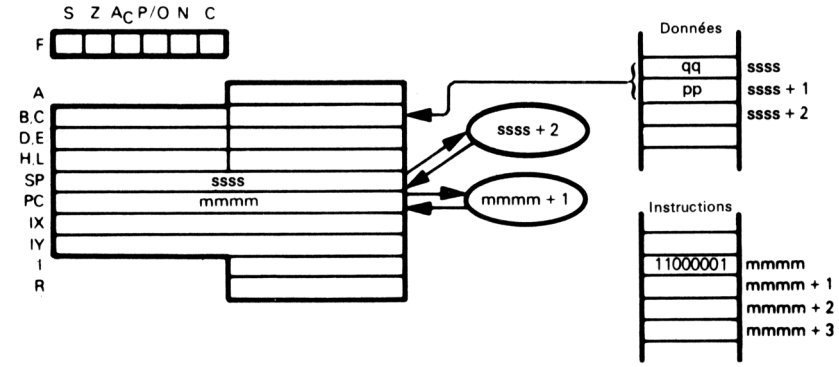


Cette instruction effectue la sortie du contenu de l'accumulateur vers le port d'entrée/sortie (E/S) identifié par le second octet du code objet de l'instruction OUT. Supposons que l'accumulateur contient 36_{16}. A la fin de l'exécution de l'instruction

```
OUT (1AH),A
```

le buffer du port d'E/S $1A_{16}$ contient 36_{16}.
L'instruction OUT ne modifie aucun indicateur. L'emploi de cette instruction dépend essentiellement du matériel. C'est la circuiterie logique externe qui détermine les adresses valides des différents ports d'E/S. Il est également possible de concevoir une logique externe telle qu'on accède aux ports d'E/S de la même manière qu'à la mémoire, certaines adresses étant réservées à ces ports (et ne pouvant donc pas être utilisées pour la mémoire). Les instructions OUT sont couramment utilisées de diverses façons pour contrôler la logique du microcalculateur externe à l'unité centrale.

POP rp — LECTURE DU SOMMET DE PILE
POP IX
POP IY



Ce schéma illustre l'exécution de l'instruction POP rp :



00 pour rp = BC
01 pour rp = DE
10 pour rp = HL
11 pour rp = AF

Cette instruction met les 2 octets du sommet de pile dans le double registre désigné. Supposons que qq = 01_{16} et pp = $2A_{16}$. L'exécution de l'instruction

POP HL

permet de charger 01_{16} dans le registre L et $2A_{16}$ dans le registre H. L'exécution de l'instruction

POP AF

permet de charger 01_{16} dans le registre d'état et $2A_{16}$ dans l'accumulateur. Ainsi, l'indicateur de retenue est à 1, les autres indicateurs à 0.

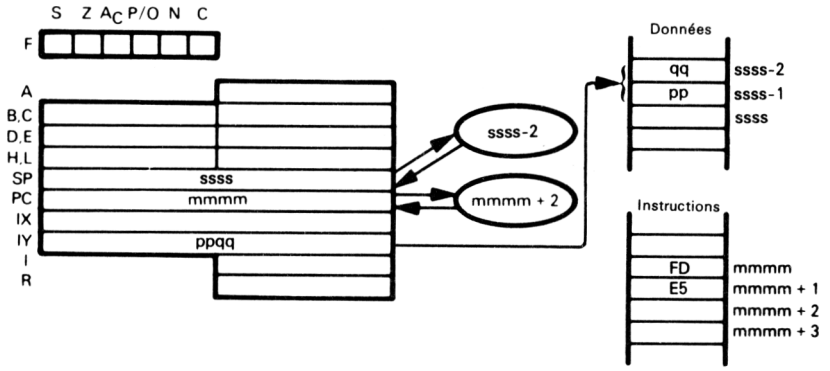
POP IX
DD E1

Cette instruction met les 2 octets du sommet de pile dans le registre IX.

POP IY
FD E1

Cette instruction met les 2 octets du sommet de pile dans le registre IY.
L'instruction POP est utilisée très couramment pour remettre en place le contenu d'un registre ou d'indicateur sauvegardé sur la pile (par exemple, pendant une interruption).

PUSH rp — ÉCRITURE EN SOMMET DE PILE
PUSH IX
PUSH IY



Ce schéma illustre l'exécution de l'instruction PUSH IY :

PUSH IY
FD E5

Cette instruction charge le contenu du registre IY sur le sommet de la pile. Supposons que le registre IY contient $45FF_{16}$. L'exécution de l'instruction

PUSH IY

charge 45_{16}, puis FF_{16} au sommet de la pile.

PUSH IX
DD E5

Cette instruction charge le contenu du registre IX au sommet de la pile.

PUSH rp
11 xx 0101

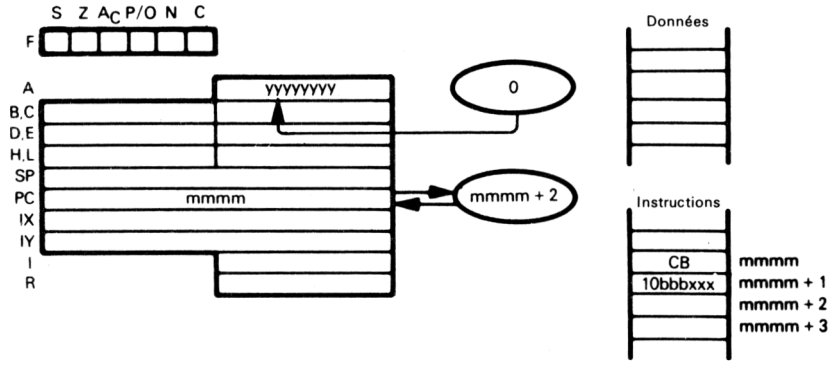
00 pour rp = BC
01 pour rp = DE
10 pour rp = HL
11 pour rp = AF

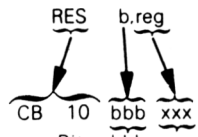
Cette instruction charge le contenu du registre désigné au sommet de la pile. L'exécution de l'instruction

PUSH AF

permet de charger l'accumulateur et le registre d'état au sommet de la pile.
L'instruction PUSH est utilisée de façon la plus courante pour sauvegarder les contenus des registres et des indicateurs ; (par exemple avant une interruption).

RES b,reg — MISE A ZÉRO D'UN BIT DE REGISTRE (« RESET »)





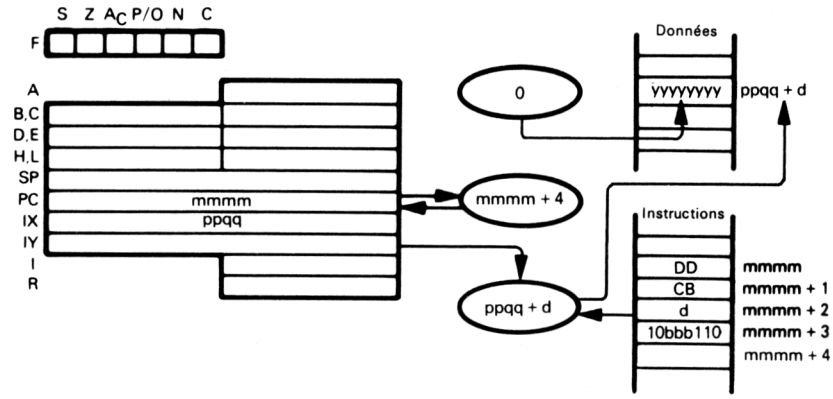
Bit	bbb	xxx	Registre
0	000	000	B
1	001	001	C
2	010	010	D
3	011	011	E
4	100	100	H
5	101	101	L
6	110	111	A
7	111		

Cette instruction remet à zéro le bit indiqué du registre spécifié. A la fin de l'exécution de l'instruction

```
RES 6,H
```

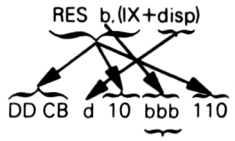
le bit 6 du registre H est mis à 0 (le bit 0 est le bit de poids faible).

RES b,(HL) — MISE A ZÉRO D'UN BIT EN MÉMOIRE (« RESET »)
RES b,(IX + disp)
RES b,(IY + disp)



Ce schéma illustre l'exécution de l'instruction RES b,(IX + d).

Le bit 0 est le bit de poids faible.



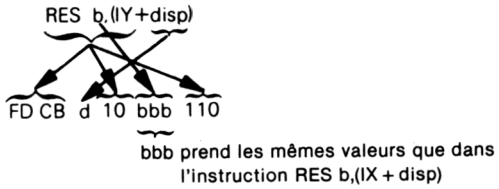
bbb	n° bit
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Cette instruction met à zéro le bit indiqué dans le contenu de l'adresse résultant de la somme du contenu du registre IX et du déplacement.

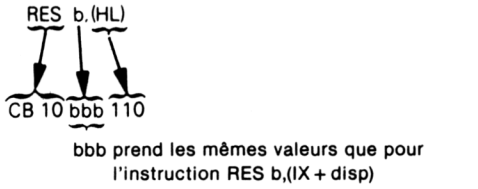
Supposons que IX contient 4110_{16}. A la fin de l'exécution de l'instruction

RES 0,(IX + 7)

le bit 0 du contenu de l'adresse 4117 est à 0.



Cette instruction est identique à RES b,(IX + disp), si ce n'est qu'elle utilise le registre IY à la place de IX.

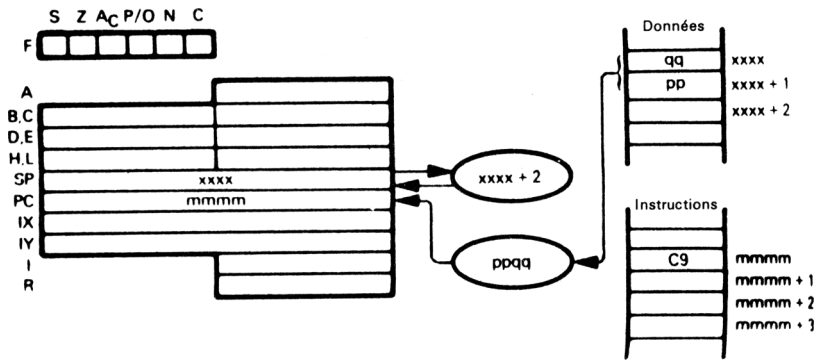


Cette instruction met à zéro le bit indiqué du contenu de l'adresse spécifiée par HL. Supposons que HL contient 4444_{16}. A la fin de l'exécution de l'instruction

RES 7,(HL)

le bit 7 du contenu de l'adresse 4444_{16} est à zéro.

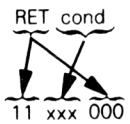
RET — RETOUR DE SOUS-PROGRAMME



RET
C9

Cette instruction transfère le contenu des octets du sommet de pile dans le compteur ordinal ; ces deux octets fournissent l'adresse de la prochaine instruction devant être exécutée. Le précédent contenu du compteur ordinal est perdu. Le pointeur de pile est incrémenté de 2 pour pouvoir adresser le nouveau sommet de pile.
Chaque sous-programme doit contenir au moins une instruction de retour (éventuellement conditionnelle) ; c'est la dernière instruction exécutée dans le sous-programme, elle provoque le retour au programme appelant.

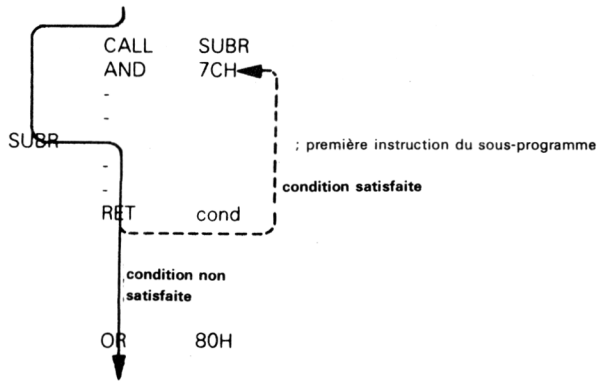
RET cond — RETOUR DE SOUS-PROGRAMME SI LA CONDITION EST SATISFAITE



		Condition	Indicateur concerné
000	NZ	non nul (« Non-Zero »)	Z
001	Z	nul (« Zero »)	Z
010	NC	pas de retenue (« Non-Carry »)	C
011	C	retenue (« Carry »)	C
100	PO	parité impaire (« Parity Odd »)	P/O
101	PE	parité paire (« Parity Even »)	P/O
110	P	signe positif	S
111	M	signe négatif	S

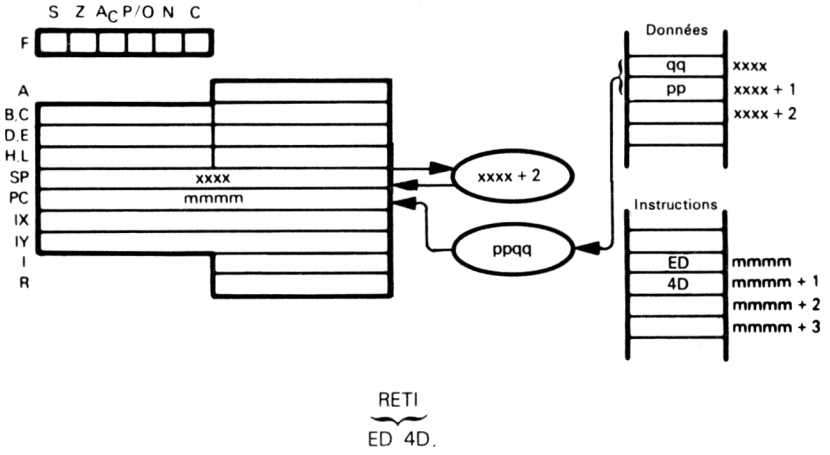
Cette instruction est identique à l'instruction RET, si ce n'est que le retour n'est exécuté que si la condition est satisfaite ; si elle ne l'est pas, l'instruction qui suit l'instruction RET cond est exécutée.

Soit la séquence d'instructions :



Après l'exécution de l'instruction RET cond, si la condition est satisfaite, l'exécution revient à l'instruction AND qui suit le « CALL ». Si la condition n'est pas satisfaite, l'instruction OR est exécutée.

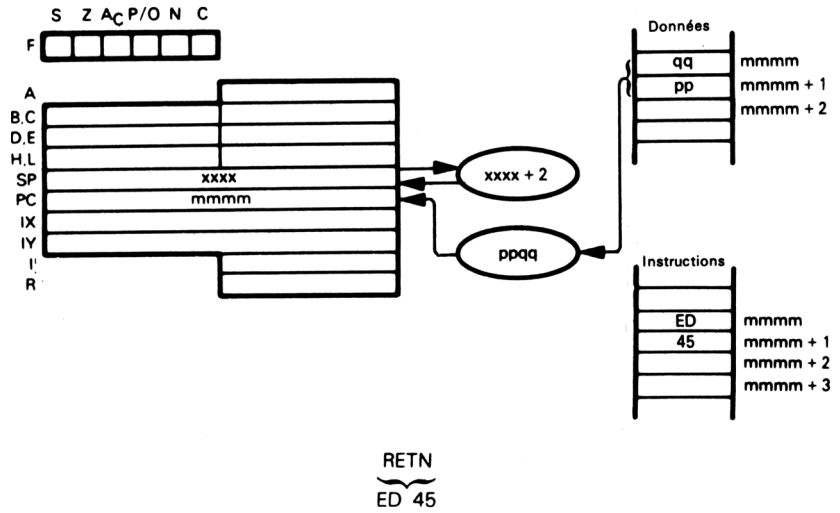
RETI — RETOUR APRÈS INTERRUPTION



Cette instruction transfère le contenu des 2 octets du sommet de pile dans le compteur ordinal ; ces 2 octets fournissent l'adresse de l'instruction suivante devant être exécutée. Le contenu précédent du compteur ordinal est perdu. Le pointeur de pile est incrémenté de 2, pour pouvoir adresser le nouveau sommet de pile.

Cette instruction est utilisée à la fin d'un programme de gestion d'interruptions et, en plus du retour au programme interrompu, elle permet d'indiquer à un périphérique d'entrée/sortie que le programme d'interruption est terminé. Ce périphérique d'entrée/sortie doit comporter la logique nécessaire pour comprendre le code opératoire de l'instruction : se reporter à *An Introduction to Microcomputer :* Volume 2 pour la description de l'instruction RETI avec la famille de périphériques du Z80.

RETN — RETOUR D'UNE INTERRUPTION NON MASQUABLE

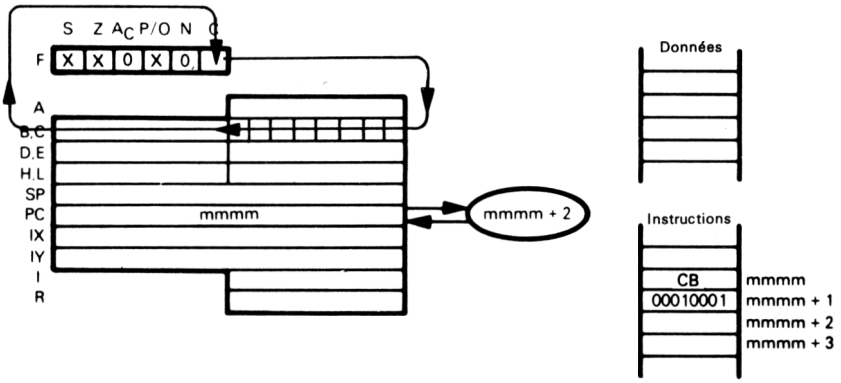


Cette instruction transfère le contenu des 2 octets du sommet de pile dans le compteur ordinal ; ces 2 octets fournissent l'adresse de l'instruction suivante devant être exécutée.

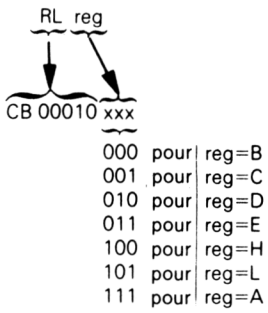
Le contenu précédent du compteur ordinal est perdu. Le pointeur de pile est incrémenté de 2 pour pouvoir adresser le nouveau sommet de pile. Enfin, cette instruction repositionne le masque d'interruptions dans l'état où il était avant que l'interruption non masquable n'arrive.

Cette instruction est utilisée à la fin d'un programme de gestion d'interruption non masquable et provoque le retour au programme interrompu.

RL reg — ROTATION A GAUCHE DU CONTENU D'UN REGISTRE A TRAVERS LA RETENUE



Ce schéma illustre l'exécution de l'instruction : RL reg.

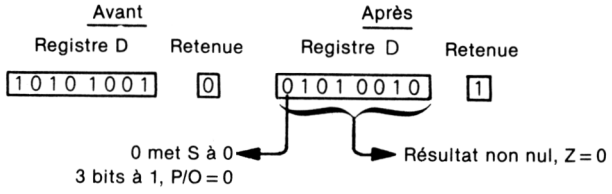


Cette instruction effectue la rotation d'un bit à gauche à travers la retenue (Carry) du contenu du registre spécifié.

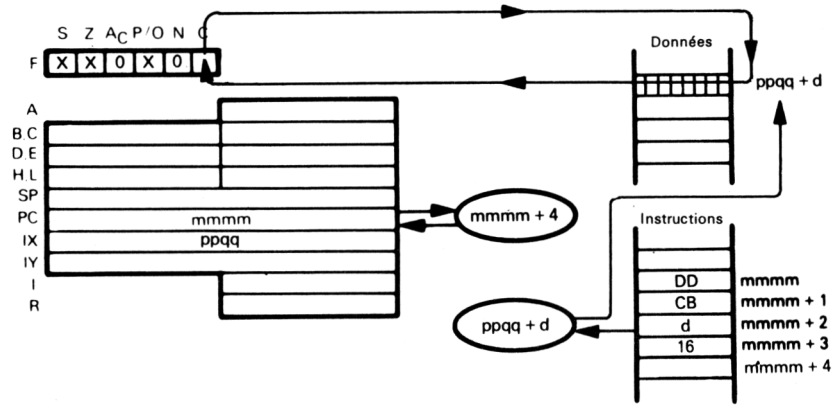
Supposons que D contient $A9_{16}$ et que la retenue vaut 0. A la fin de l'exécution de l'instruction

RL D

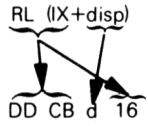
le registre D contient 52_{16} et la retenue vaut 1.



RL (HL) — ROTATION A GAUCHE A TRAVERS LA RETENUE
RL (IX + disp) DU CONTENU D'UNE ADRESSE
RL (IY + disp)



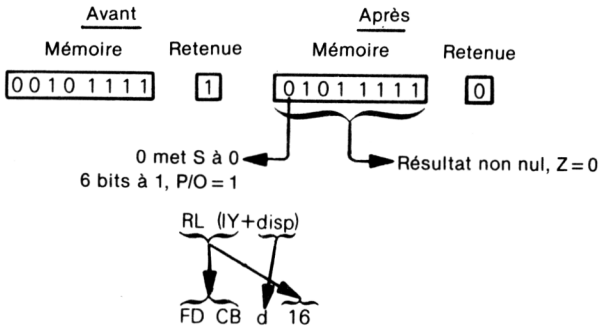
Ce schéma illustre l'exécution de l'instruction RL (IX + d).



Cette instruction effectue la rotation d'un bit à gauche à travers la retenue (Carry) du contenu de l'adresse (résultant de la somme du contenu du registre d'index IX et du déplacement d).
Supposons que le registre IX contient 4000_{16}, l'adresse 4007_{16} contient $2F_{16}$ et que la retenue est positionnée à 1. A la fin de l'exécution de l'instruction

RL (IX + 7)

l'adresse 4007_{16} contient $5F_{16}$ et la retenue est nulle :

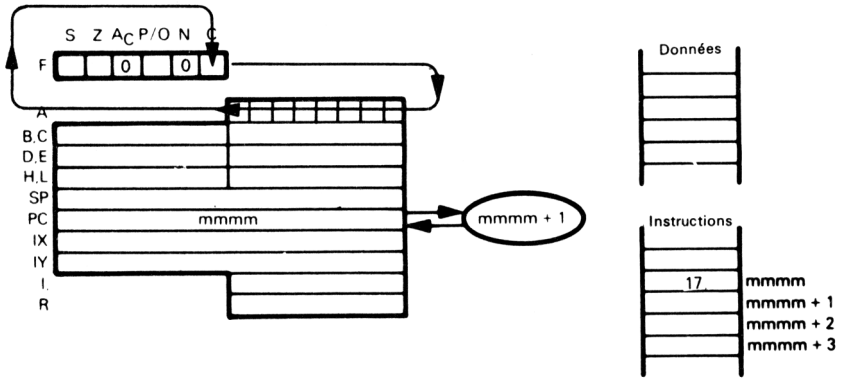


Cette instruction est identique à RL (IX + disp), si ce n'est qu'elle utilise le registre IY à la place de IX.

RL (HL)
CB 16

Cette instruction effectue la rotation d'un bit à gauche à travers la retenue (Carry) du contenu de l'adresse spécifiée par le double registre HL.

RLA — ROTATION A GAUCHE A TRAVERS LA RETENUE DU CONTENU DE L'ACCUMULATEUR



RLA
17

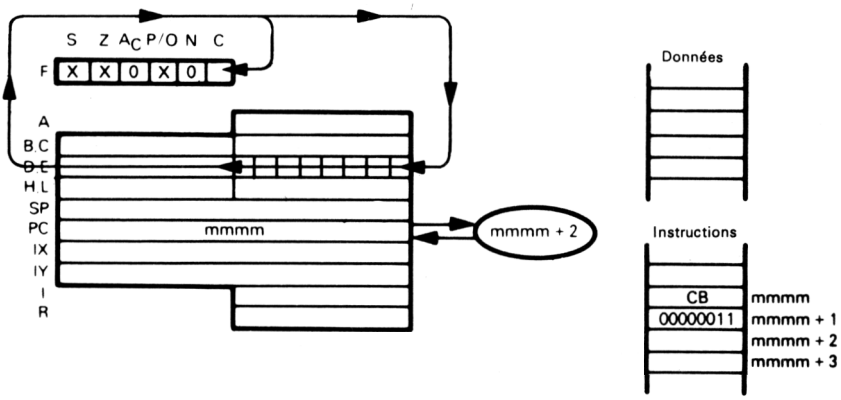
Cette instruction effectue la rotation d'un bit à gauche à travers la retenue du contenu de l'accumulateur.
Supposons que l'accumulateur contient $2A_{16}$ et que l'indicateur de retenue est positionné à 1. A la fin de l'exécution de l'instruction

RLA

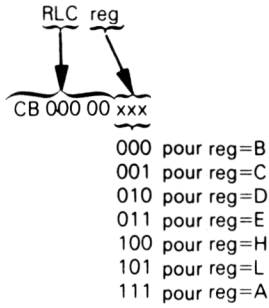
l'accumulateur contient $F5_{16}$ et l'indicateur de retenue est positionné à 0.

Avant		Après	
Accumulateur	Retenue	Accumulateur	Retenue
0 1 1 1 1 0 1 0	1	1 1 1 1 0 1 0 1	0

RLC reg — ROTATION CIRCULAIRE A GAUCHE DU CONTENU D'UN REGISTRE



Ce schéma illustre l'exécution de l'instruction RLC E.

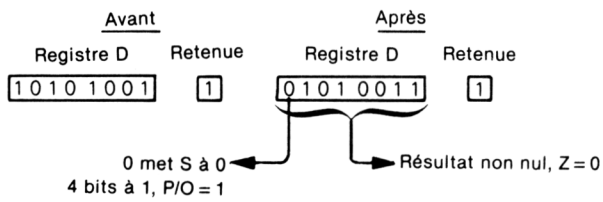


Cette instruction effectue la rotation d'un bit à gauche du contenu du registre spécifié, en mettant le bit 7 dans l'indicateur de retenue.

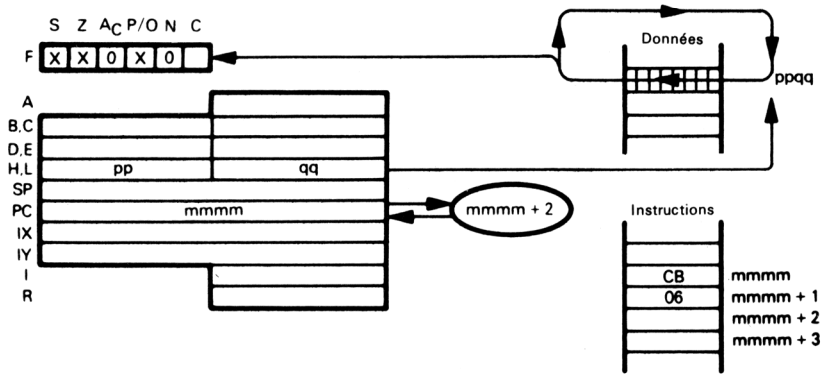
Supposons que le registre D contient $A9_{16}$ et que la retenue vaut 1. A la fin de l'exécution de l'instruction

RLC D

le registre D contient 53_{16} et la retenue est positionnée à 1.



RLC (HL) — ROTATION CIRCULAIRE A GAUCHE DU CONTENU
RLC (IX + disp) D'UNE ADRESSE
RLC (IY + disp)



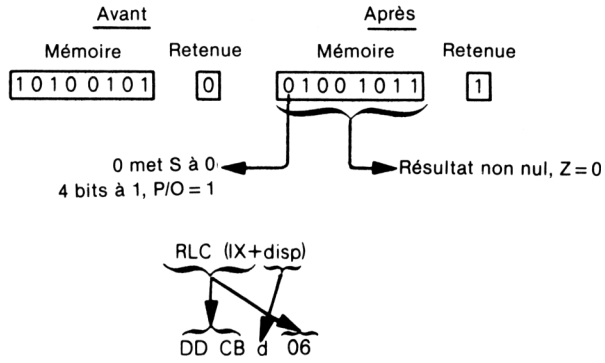
Ce schéma illustre l'exécution de l'instruction RLC (HL).

RLC (HL)
CB 06

Cette instruction effectue la rotation d'un bit à gauche, du contenu de l'adresse spécifiée par le double registre HL. (Le bit 7 est recopié dans la retenue.)
Supposons que le double registre HL contient $54FF_{16}$. L'adresse $54FF_{16}$ contient $A5_{16}$ et la retenue vaut 0. A la fin de l'exécution de l'instruction

RLC (HL)

l'adresse $54FF_{16}$ contient $4B_{16}$ et la retenue est positionnée à 1.

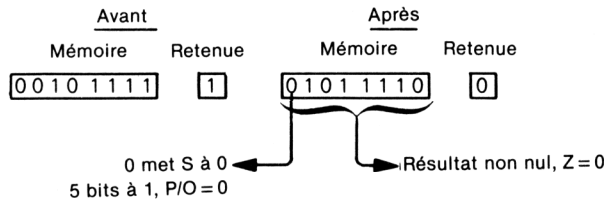


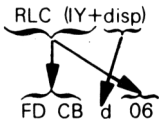
Cette instruction effectue la rotation d'un bit à gauche du contenu de l'adresse (résultant de la somme du contenu du registre d'index IX et du déplacement d). Le bit 7 est recopié dans la retenue.

Supposons que le registre d'index IX contient 4000_{16}, la retenue est à 1, et que l'adresse 4007_{16} contient $2F_{16}$. A la fin de l'exécution de l'instruction

RLC (IX + 7)

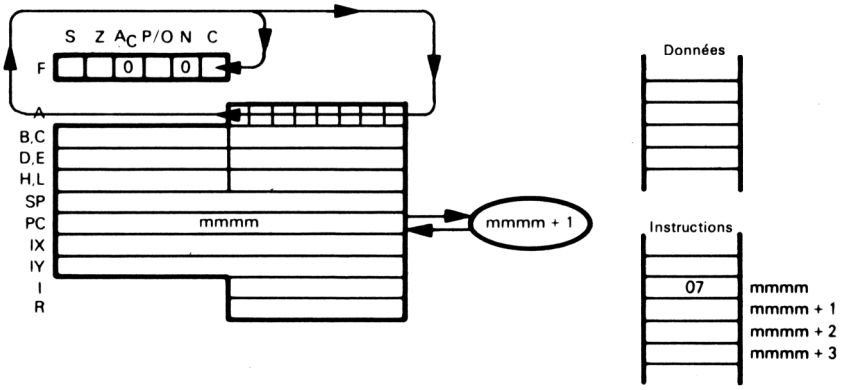
l'adresse 4007_{16} contient $5E_{16}$ et la retenue est positionnée à 0.





Cette instruction est identique à l'instruction RLC (IX + disp), mais elle utilise le registre IY au lieu du registre IX.

RLCA — ROTATION CIRCULAIRE A GAUCHE DE L'ACCUMULATEUR



RLCA
07

Cette instruction effectue la rotation d'un bit à gauche du contenu de l'accumulateur. Le bit 7 est recopié dans la retenue.

Supposons que l'accumulateur contient $7A_{16}$ et que l'indicateur de retenue est positionné à 1. A la fin de l'exécution de l'instruction

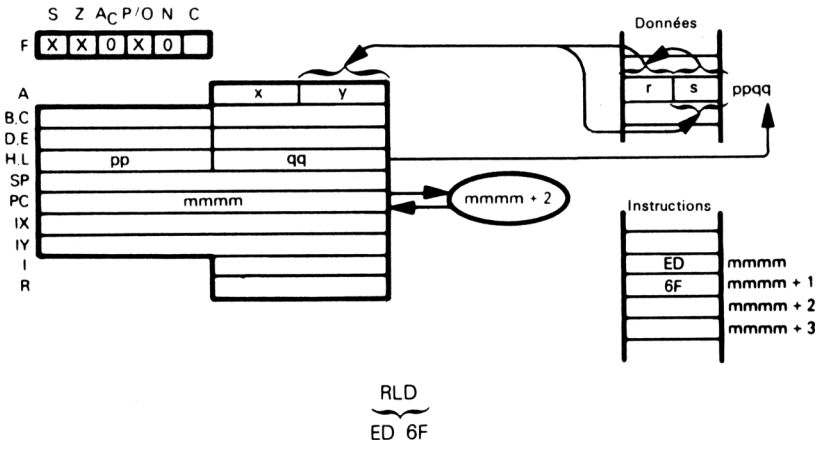
RLCA

l'accumulateur contient $F4_{16}$ et l'indicateur de retenue est remis à zéro.

Avant		Après	
Accumulateur	Retenue	Accumulateur	Retenue
0 1 1 1 1 0 1 0	1	1 1 1 1 0 1 0 0	0

RLCA est généralement utilisée en tant qu'instruction logique.

RLD — ROTATION A GAUCHE D'UN CHIFFRE BCD ENTRE L'ACCUMULATEUR ET UN EMPLACEMENT MÉMOIRE

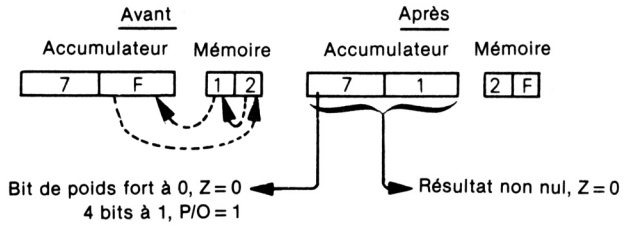


Cette instruction copie les 4 bits de poids faible d'une mémoire (adressée par la paire de registres HL) dans les 4 bits de poids fort de cette même mémoire. Le précédent contenu des 4 bits de poids fort de cette mémoire est transféré dans les 4 bits de poids faible de l'accumulateur. Le précédent contenu des 4 bits de poids faible de l'accumulateur est transféré dans les 4 bits de poids faible de la mémoire spécifiée.

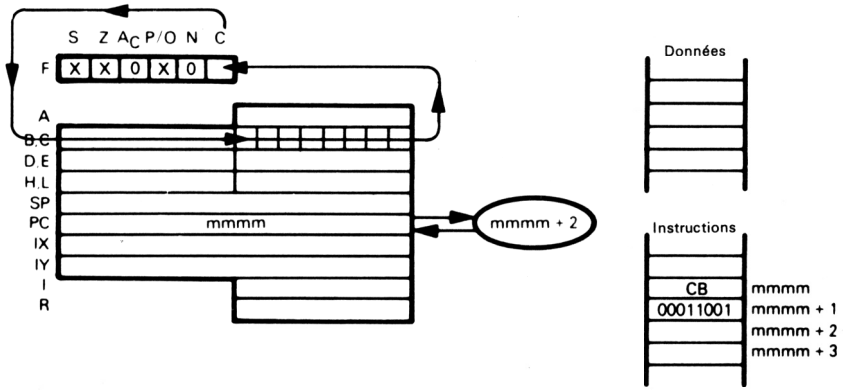
Supposons que l'accumulateur contient $7F_{16}$, le double registre HL contient 4000_{16} et la mémoire d'adresse 4000_{16} contient 12_{16}. A la fin de l'exécution de l'instruction

RLD

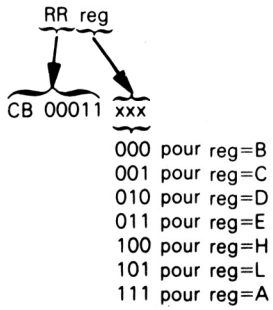
l'accumulateur contient 71_{16} et l'adresse 4000_{16} contient $2F_{16}$.



RR reg — ROTATION A DROITE D'UN REGISTRE, A TRAVERS LA RETENUE



Le schéma illustre l'exécution de l'instruction RRC :

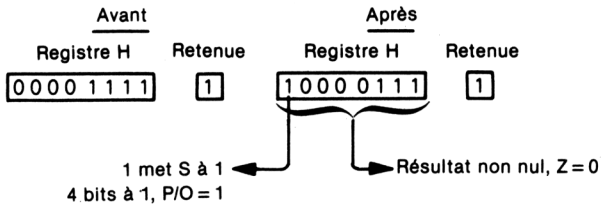


Cette instruction effectue la rotation d'un bit à droite, à travers la retenue, du contenu du registre spécifié.

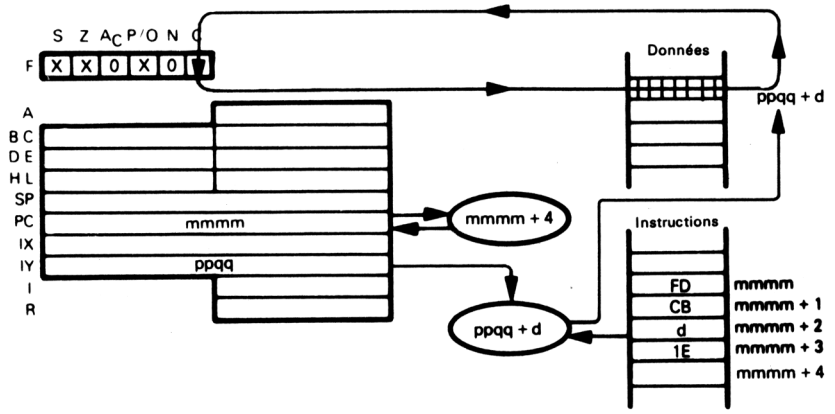
Supposons que le registre H contient $0F_{16}$ et que la retenue est positionnée à 1. A la fin de l'exécution de l'instruction

RR H

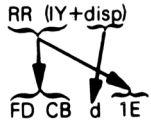
le registre H contient 87_{16} et la retenue vaut 1 :



RR (HL) — ROTATION A DROITE D'UN OCTET MÉMOIRE A TRAVERS
RR (IX + disp) LA RETENUE
RR (IY + disp)



Ce schéma illustre l'exécution de l'instruction RR (IY + d) :

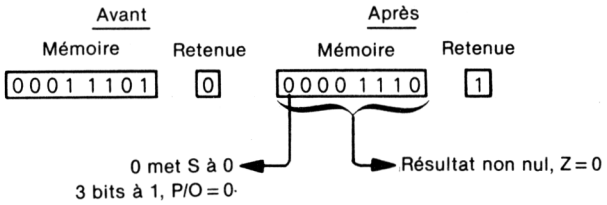


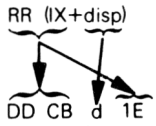
Cette instruction effectue la rotation d'un bit à droite, à travers la retenue, du contenu de l'adresse (résultant de la somme du contenu du registre IY avec le déplacement d).

Supposons que le registre IY contient 4500_{16}, l'adresse $450F_{16}$ contient $1D_{16}$ et la retenue est nulle. Après l'exécution de l'instruction

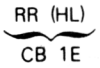
RR (IY + 0FH)

l'adresse $450F_{16}$ contient $0E_{16}$ et la retenue est positionnée à 1 :



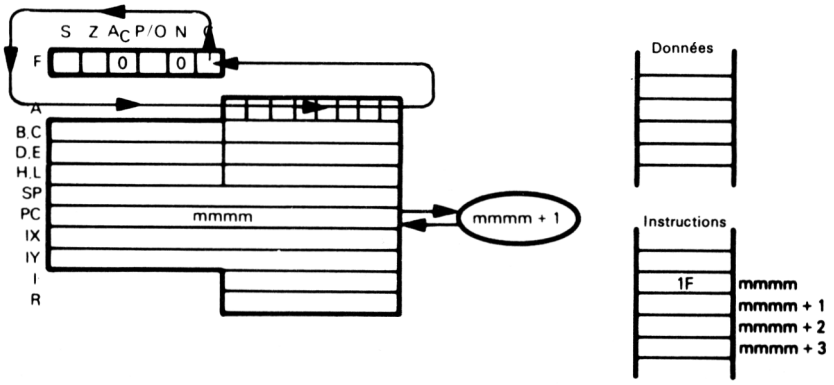


Cette instruction est identique à RR (IY + disp), si ce n'est qu'elle utilise le registre IX à la place de IY.



Cette instruction effectue la rotation d'un bit à droite, à travers la retenue, du contenu de l'adresse spécifiée par le double registre HL.

RRA — ROTATION A DROITE DE L'ACCUMULATEUR A TRAVERS LA RETENUE



RRA
1F

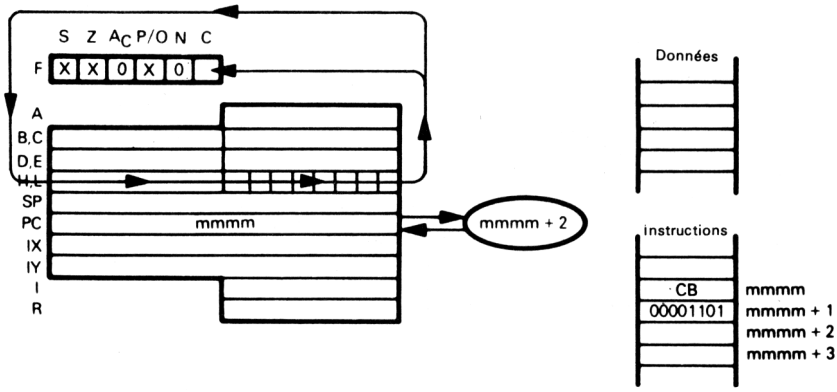
Cette instruction effectue la rotation d'un bit à droite, à travers l'indicateur de retenue, du contenu de l'accumulateur.
Supposons que l'accumulateur contient $7A_{16}$ et que l'indicateur de retenue est positionné à 1. A la fin de l'exécution de l'instruction

RRA

l'accumulateur contient BD_{16} et l'indicateur de retenue est remis à 0.

Avant		Après	
Accumulateur	Retenue	Accumulateur	Retenue
0 1 1 1 1 0 1 0	1	1 0 1 1 1 1 0 1	0

RRC reg — ROTATION CIRCULAIRE A DROITE DU CONTENU D'UN REGISTRE



Ce schéma illustre l'exécution de l'instruction RRC reg :

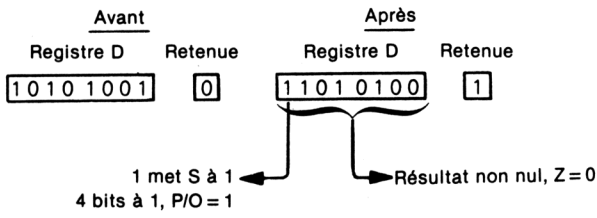
RRC reg
CB 00001 xxx

000 pour reg=B
001 pour reg=C
010 pour reg=D
011 pour reg=E
100 pour reg=H
101 pour reg=L
111 pour reg=A

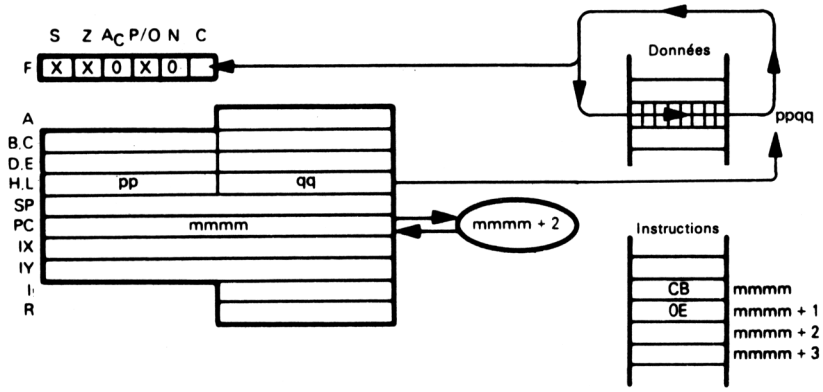
Cette instruction effectue la rotation circulaire d'un bit à droite du contenu du registre spécifié. Le bit 0 est copié dans l'indicateur de retenue.
Supposons que le registre D contient $A9_{16}$ et que la retenue est nulle. A la fin de l'exécution de l'instruction

RRC D

le registre D contient $D4_{16}$ et la retenue est positionnée à 1 :



RRC (HL) — ROTATION CIRCULAIRE A DROITE DU CONTENU
RRC (IX + disp) D'UN OCTET MÉMOIRE
RRC (IY + disp)



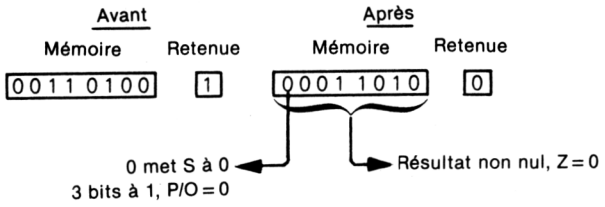
Ce schéma illustre l'exécution de l'instruction RRC (HL) :

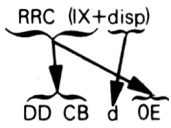
RRC (HL)
CB 0E

Cette instruction effectue la rotation circulaire d'un bit à droite du contenu de l'adresse spécifiée par le double registre HL ; le bit 0 est copié dans l'indicateur de retenue.
Supposons que le double registre HL contient 4500_{16}, l'adresse 4500_{16} contient 34_{16} et que la retenue est positionnée à 1. A la fin de l'exécution de l'instruction

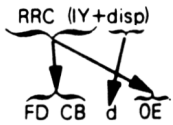
RRC (HL)

l'adresse 4500_{16} contient $1A_{16}$ et la retenue est nulle.



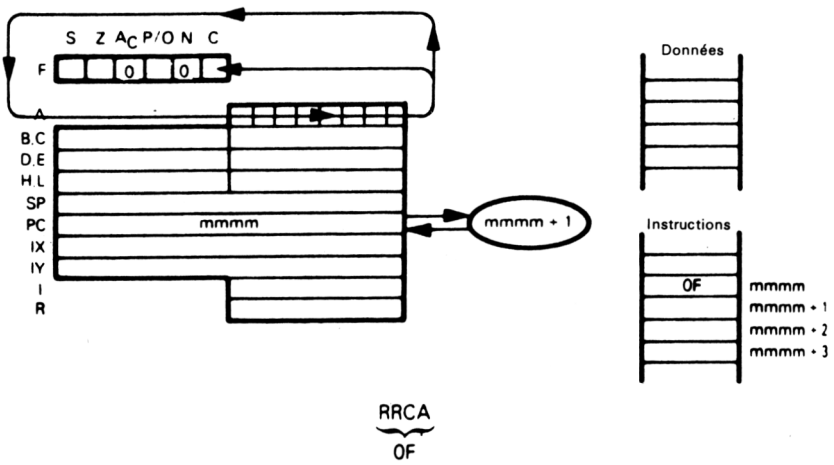


Cette instruction effectue la rotation circulaire d'un bit à droite du contenu de l'adresse (résultant de la somme du contenu du registre IX et du déplacement d) ; le bit 0 est copié dans l'indicateur de retenue.



Cette instruction est identique à RRC (IX + disp) si ce n'est qu'elle utilise le registre IY au lieu du registre IX.

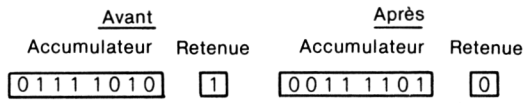
RRCA — ROTATION CIRCULAIRE A DROITE DE L'ACCUMULATEUR



Cette instruction effectue une rotation circulaire d'un bit à droite du contenu de l'accumulateur, le bit 0 est copié dans l'indicateur de retenue.
Supposons que l'accumulateur contient $7A_{16}$ et que l'indicateur de retenue est positionné à 1. A la fin de l'exécution de l'instruction

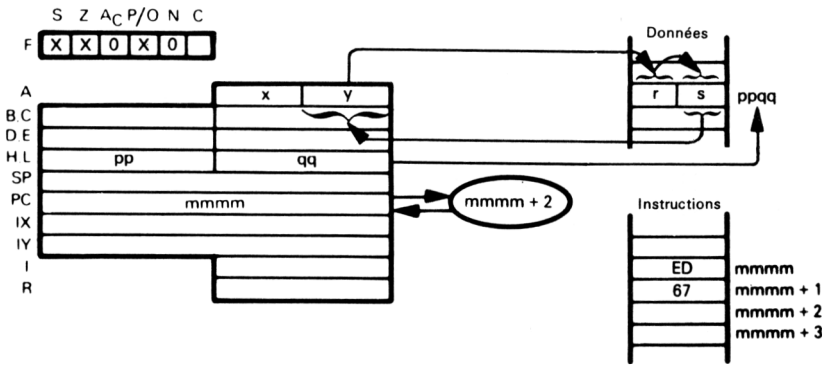
RRCA

l'accumulateur contient $3D_{16}$ et l'indicateur de retenue est remis à zéro :



RRCA est généralement utilisée en tant qu'instruction logique.

RRD — ROTATION A DROITE D'UN CHIFFRE BCD ENTRE L'ACCUMULATEUR ET UN EMPLACEMENT MÉMOIRE



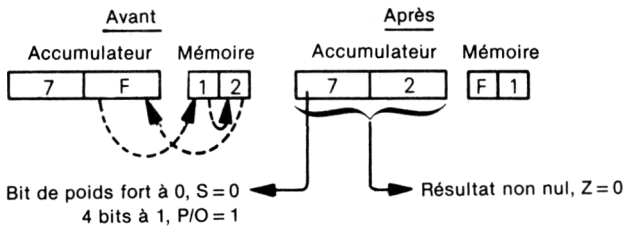
RRD
ED 67

Cette instruction effectue les opérations de transfert suivantes :

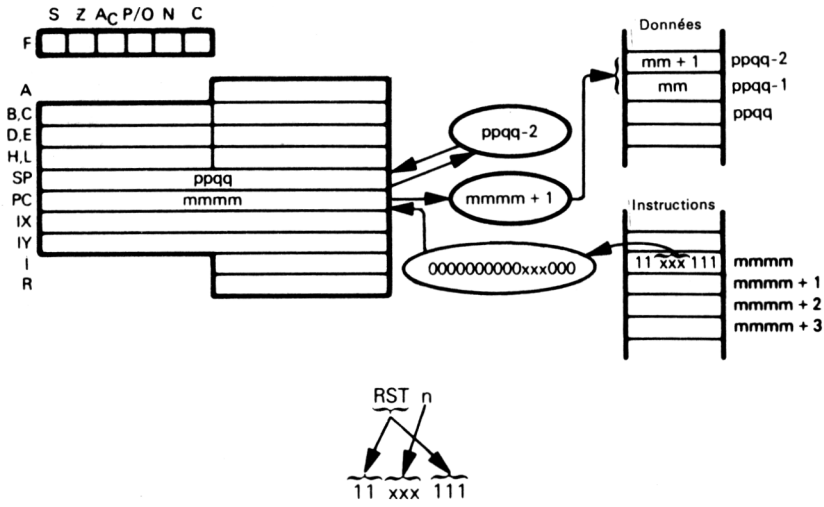
— les 4 bits de poids fort de la mémoire adressée par le double registre HL sont copiés dans les 4 bits de poids faible de cette même mémoire,
— les précédents 4 bits de poids faible sont copiés dans les 4 bits de poids faible de l'accumulateur,
— les précédents 4 bits de poids faible de l'accumulateur sont copiés dans les 4 bits de poids fort de la mémoire spécifiée.
Supposons que l'accumulateur contient $7F_{16}$, le double registre HL contient 4000_{16} que l'adresse 4000_{16} contient 12_{16}. A la fin de l'exécution de l'instruction

RRD

l'accumulateur contient 72_{16} et l'adresse 4000_{16} contient $F1_{16}$:



RST n — INSTRUCTION DE RELANCE (« RESTART »)



Cette instruction effectue l'appel au sous-programme qui commence à l'adresse mémoire spécifiée par n.
A la fin de l'exécution de l'instruction

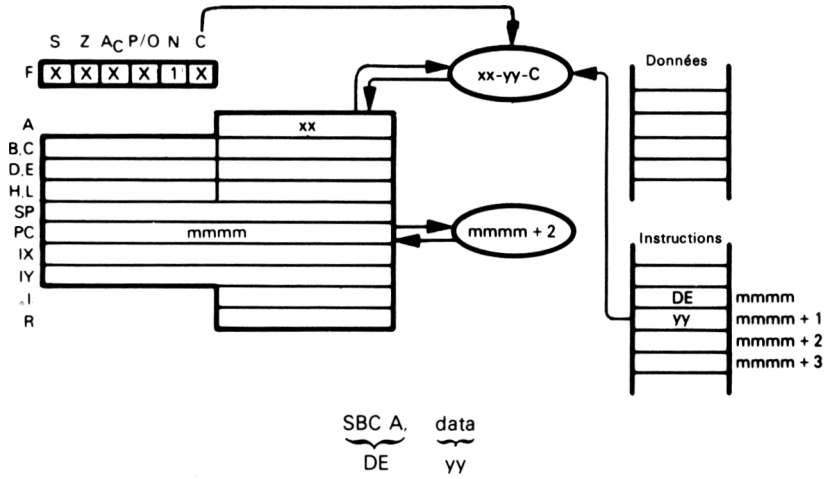
RST 18H

le sous-programme commençant à l'adresse 0018_{16} est appelé. Le précédent contenu du compteur ordinal est chargé en sommet de pile.
Habituellement, l'instruction RST est utilisée en conjonction avec un traitement d'interruptions. comme cela est décrit dans le chapitre 12.

APPEL DE SOUS-PROGRAMME AVEC RST

Si votre application ne se sert pas de toutes les instructions RST pour gérer les interruptions, ne négligez pas la possibilité d'appeler des sous-programmes à l'aide des instructions RST. Les sous-programmes les plus souvent utilisés pourront ainsi être appelés par les instructions RST (un seul octet) au lieu d'utiliser l'instruction CALL (3 octets).

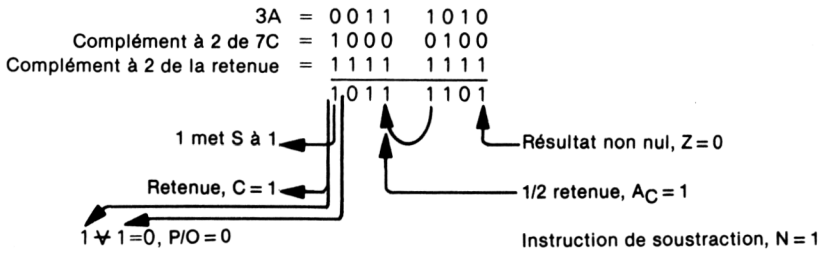
SBC A,data — SOUSTRACTION AVEC RETENUE D'UNE DONNÉE A PARTIR DE L'ACCUMULATEUR



Cette instruction soustrait du contenu de l'accumulateur l'indicateur de retenue ainsi que le contenu du second octet du code de l'instruction.
Supposons que xx = $3A_{16}$ et que la retenue est positionnée à 1. A la fin de l'exécution de l'instruction

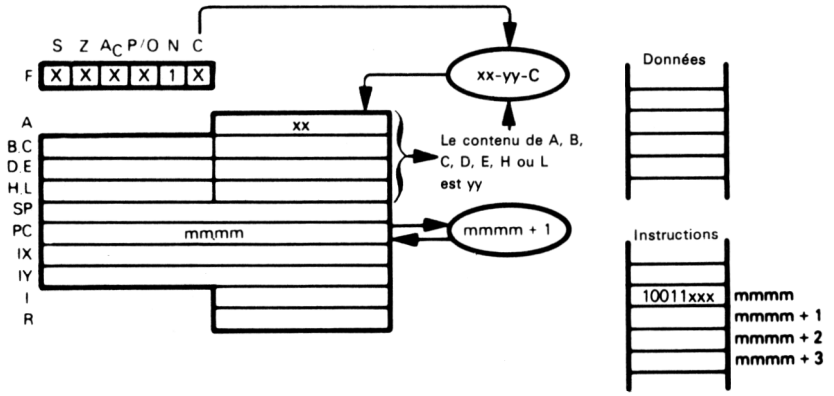
```
SBC A,7CH
```

l'accumulateur contient BD_{16}.



L'indicateur de retenue (Carry) est positionné à 1 quand il y a une retenue et à zéro sinon.

SBC A,reg — SOUSTRACTION AVEC RETENUE DU CONTENU D'UN REGISTRE A PARTIR DE L'ACCUMULATEUR



SBC A, reg

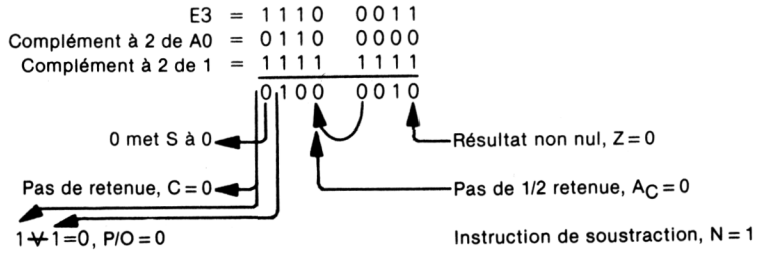
10011 xxx

000 pour reg=B
001 pour reg=C
010 pour reg=D
011 pour reg=E
100 pour reg=H
101 pour reg=L
111 pour reg=A

Cette instruction soustrait de l'accumulateur le contenu du registre spécifié et l'indicateur de retenue.
Supposons que xx = $E3_{16}$, le registre E contient $A0_{16}$ et que l'indicateur de retenue = 1. A la fin de l'exécution de l'instruction

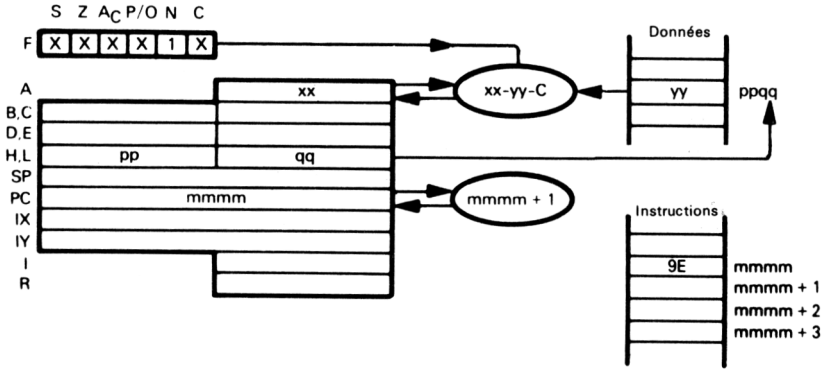
SBC A,E

l'accumulateur contient 42_{16}.



L'indicateur de retenue (Carry) est positionné à 1 pour une retenue et remis à zéro, sinon.

SBC A,(HL) — SOUSTRACTION AVEC RETENUE D'UN OCTET MÉMOIRE
SBC A,(IX + disp) A PARTIR DE L'ACCUMULATEUR
SBC A,(IY + disp)



Ce schéma illustre l'exécution de l'instruction SBC A,(HL) :

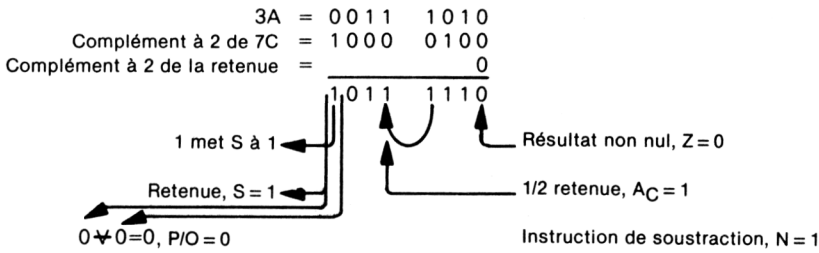
$$\underbrace{\text{SBC A,(HL)}}_{\text{9E}}$$

Cette instruction soustrait de l'accumulateur le contenu de l'adresse (spécifiée par le contenu du double registre HL) et l'indicateur de retenue.

On suppose que la retenue = 0, ppqq = 4000_{16}, xx = $3A_{16}$ et que l'adresse 4000_{16} contient $7C_{16}$. A la fin de l'exécution de l'instruction

SBC A,(HL)

l'accumulateur contient BE_{16} :



L'indicateur de retenue (Carry) est mis à 1 pour une retenue, à 0 sinon.

$$\underbrace{\text{SBC A,(IX+disp)}}_{\text{DD 9E}} \quad \underbrace{\text{d}}$$

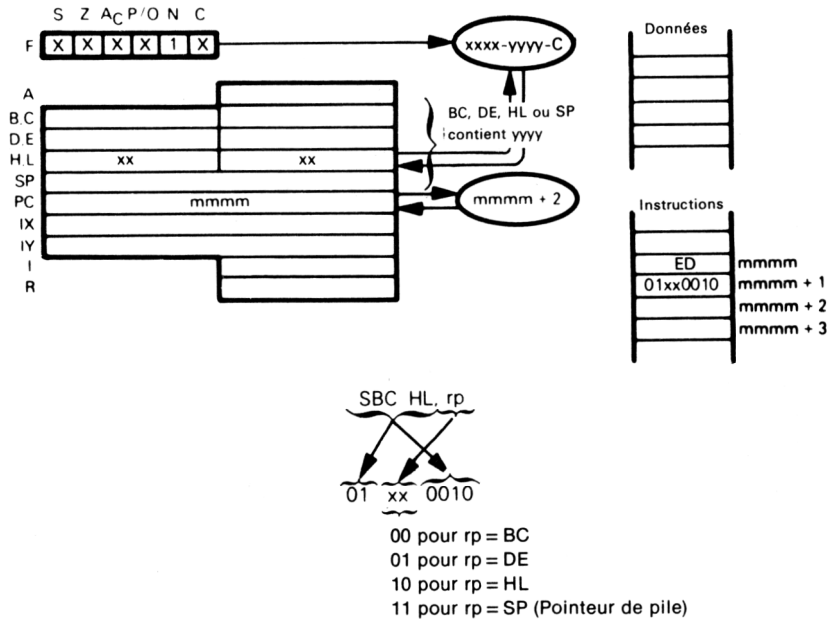
Cette instruction soustrait de l'accumulateur le contenu de l'adresse (résultant de la somme du contenu du registre IX et du déplacement d) et l'indicateur de retenue.

SBC A,(IY+disp)

FD 9E d

Cette instruction est identique à la précédente, si ce n'est qu'elle utilise IY à la place de IX.

SBC HL,rp — SOUSTRACTION AVEC RETENUE DU CONTENU D'UNE PAIRE DE REGISTRES A PARTIR DE H ET DE L



SBC HL, rp

01 xx 0010

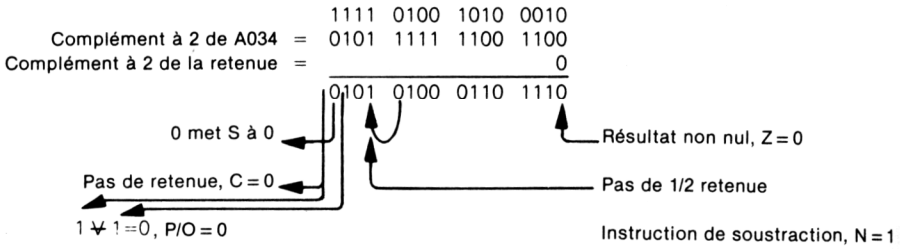
00 pour rp = BC
01 pour rp = DE
10 pour rp = HL
11 pour rp = SP (Pointeur de pile)

Cette instruction soustrait du contenu du double registre HL le contenu de la paire de registres désignée et l'indicateur de retenue.

Supposons que HL contient $F4A2_{16}$, BC contient $A034_{16}$ et que l'indicateur de retenue est à zéro. A la fin de l'exécution de l'instruction

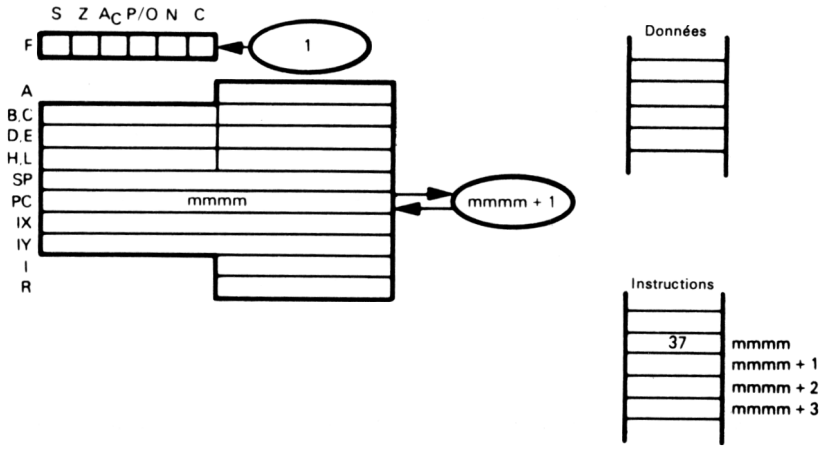
SBC HL,BC

le double registre HL contient $546E_{16}$:



L'indicateur de retenue (Carry) est positionné à 1 en cas de retenue, à zéro sinon.

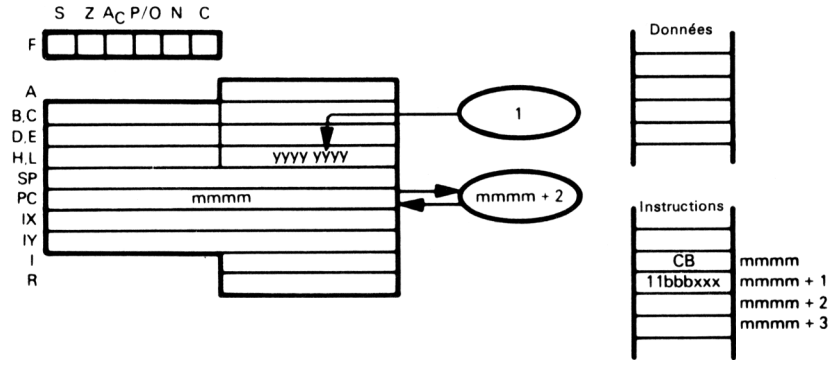
SCF — POSITIONNEMENT A 1 DE L'INDICATEUR DE RETENUE (« SET CARRY FLAG »)

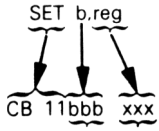


SCF
37

Lorsque cette instruction est exécutée, l'indicateur de retenue est inconditionnellement mis à 1. Aucun autre indicateur ni registre n'est modifié.

SET b,reg — POSITIONNEMENT A 1 D'UN BIT DE REGISTRE





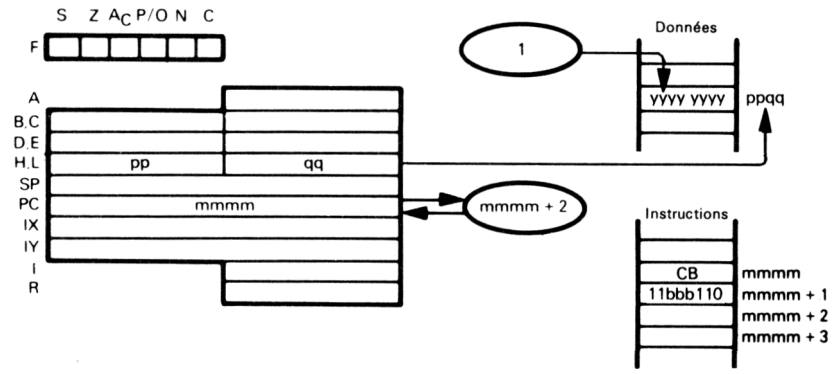
Bit	bbb	xxx	Registre
0	000	000	B
1	001	001	C
2	010	010	D
3	011	011	E
4	100	100	H
5	101	101	L
6	110	111	A
7	111		

Cette instruction positionne à 1 le bit indiqué (b) du registre spécifié (reg). A la fin de l'exécution de l'instruction

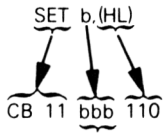
SET 2,L

le bit 2 du registre L est positionné à 1. (Le bit 0 est le bit le moins significatif.)

SET b,(HL) — POSITIONNEMENT A 1 DU BIT b DE L'OCTET MÉMOIRE
SET b,(IX + disp) INDIQUÉ
SET b,(IY + disp)



Ce schéma illustre l'exécution de l'instruction SET b,(HL). Le bit 0 est le moins significatif.

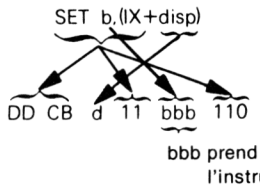


n° bit	bbb
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Cette instruction positionne à 1 le bit indiqué (b) de l'emplacement mémoire adressé par la paire de registres HL.
Supposons que HL contient 4000_{16}. A la fin de l'exécution de l'instruction

SET 5,(HL)

le bit 5 du contenu de l'adresse 4000_{16} vaut 1.

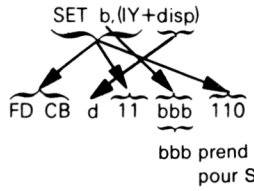


Cette instruction met à 1 le bit indiqué (b) du contenu de l'adresse résultant de la somme du contenu du registre IX et du déplacement d.

On suppose que le registre IX contient 4000_{16}. A la fin de l'exécution de l'instruction

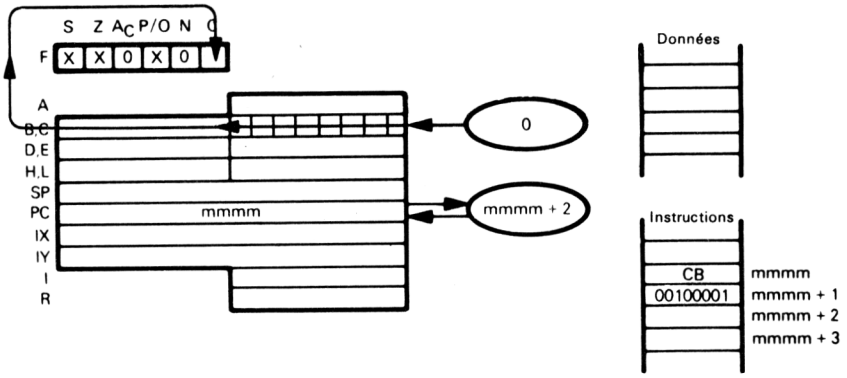
SET 6,(IX + 5H)

le bit 6 du contenu de l'adresse 4005_{16} vaut 1.

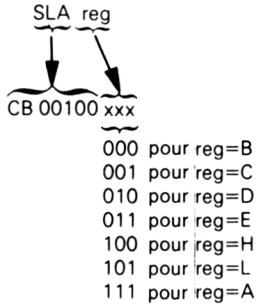


Cette instruction est identique à SET b,(IX + disp), si ce n'est qu'elle utilise le registre IY à la place du registre IX.

SLA reg — DÉCALAGE ARITHMÉTIQUE A GAUCHE DU CONTENU D'UN REGISTRE



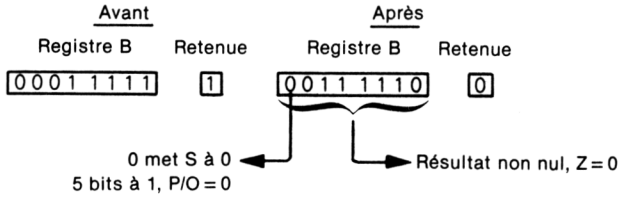
Ce schéma illustre l'exécution de l'instruction SLA C :



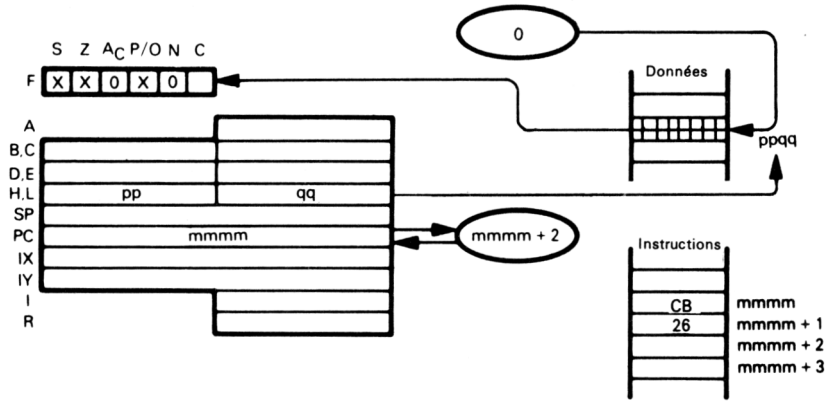
Cette instruction effectue le décalage à gauche d'un bit du contenu du registre spécifié, en mettant à 0 le bit le moins significatif.
Supposons que le registre B contient $1F_{16}$ et que l'indicateur de retenue vaut 1. A la fin de l'exécution de l'instruction

SLA B

le registre B contient $3E_{16}$ et l'indicateur de retenue vaut 0.



SLA (HL) — DÉCALAGE ARITHMÉTIQUE A GAUCHE D'UN OCTET MÉMOIRE
SLA (IX + disp)
SLA (IY + disp)



Ce schéma illustre l'exécution de l'instruction SLA (HL) :

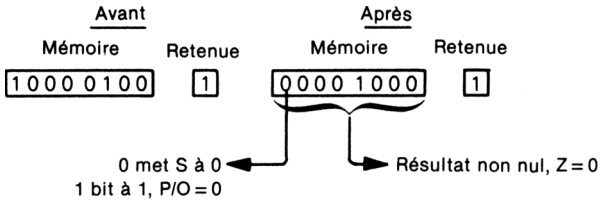


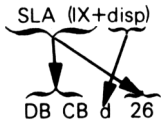
Cette instruction effectue le décalage d'un bit à gauche du contenu de l'adresse (spécifiée par le double registre HL), en mettant le bit le moins significatif à zéro.

Supposons que le double registre HL contient 4500_{16}, que l'adresse 4500_{16} contient 84_{16} et que l'indicateur de retenue = 0. A la fin de l'exécution de l'instruction

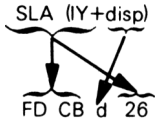
SLA (HL)

l'adresse 4500_{16} contient 08_{16} et l'indicateur de retenue = 1.



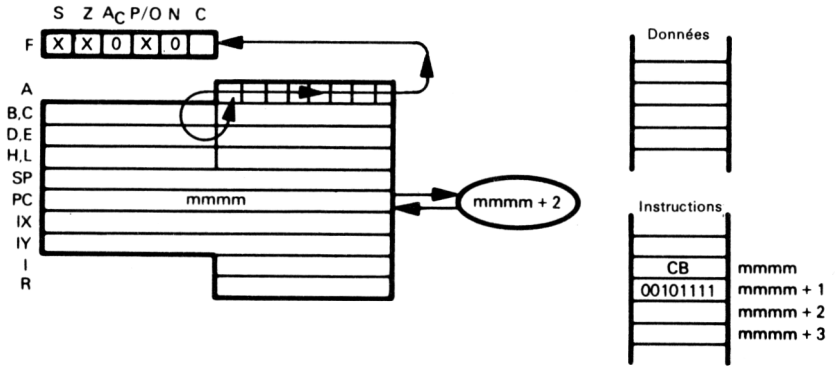


Cette instruction effectue le décalage arithmétique d'un bit à gauche du contenu de l'adresse (résultant de la somme du contenu du registre IX et du déplacement d), en mettant à 0 le bit le moins significatif.

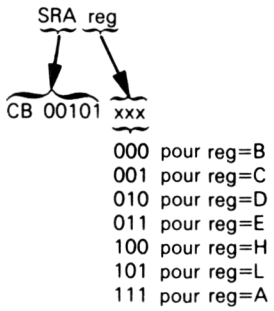


Cette instruction est identique à SLA (IX + disp) ; si ce n'est qu'elle utilise le registre IY à la place de IX.

SRA reg — DÉCALAGE ARITHMÉTIQUE A DROITE DU CONTENU D'UN REGISTRE



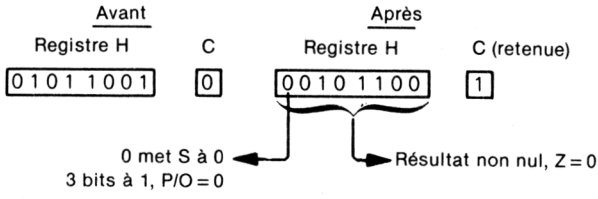
Ce schéma illustre l'exécution de l'instruction SRA A :



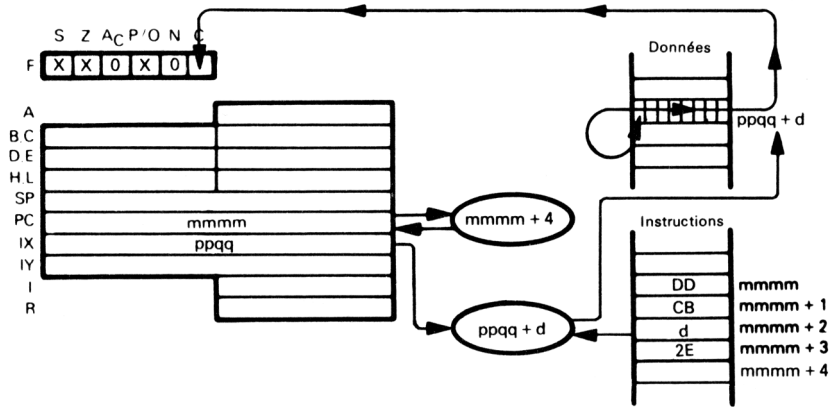
Cette instruction effectue le décalage arithmétique d'un bit à droite du contenu du registre spécifié. Le bit le plus significatif n'est pas modifié.
Supposons que le registre H contient 59_{16} et que l'indicateur de retenue = 0. A la fin de l'exécution de l'instruction

SRA H

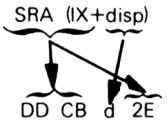
le registre H contient $2C_{16}$ et l'indicateur de retenue est à 1.



SRA (HL) — DÉCALAGE ARITHMÉTIQUE A DROITE D'UN OCTET MÉMOIRE
SRA (IX + disp)
SRA (IY + disp)



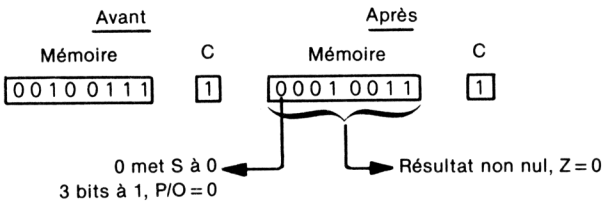
Ce schéma illustre l'exécution SRA (IX + d) :

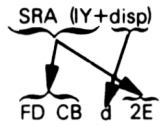


Cette instruction effectue le décalage arithmétique d'un bit à droite du contenu de l'adresse résultant de la somme du contenu du registre IX et du déplacement d. Le bit le plus significatif n'est pas modifié.
Supposons que le registre IX contient 3400_{16}, que l'adresse $34AA_{16}$ contient 27_{16} et que l'indicateur de retenue = 1. A la fin de l'exécution de l'instruction

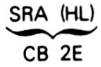
SRA (IX + 0AAH)

l'adresse 34AAH contient 13_{16} et l'indicateur de retenue = 1.



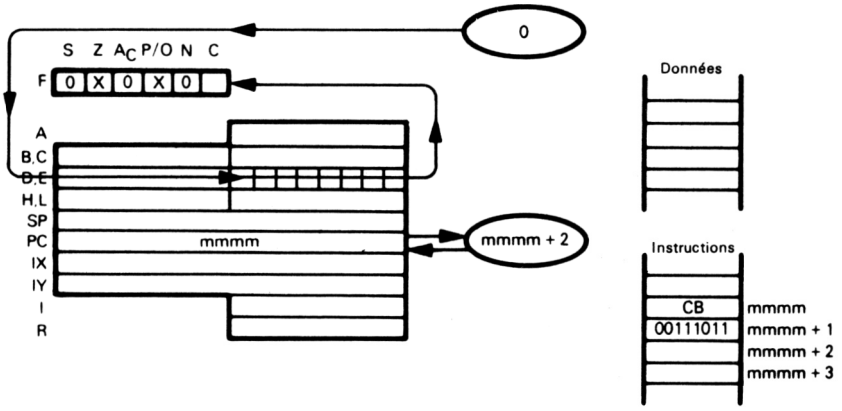


Cette instruction est identique à la précédente, si ce n'est qu'elle utilise le registre IY à la place de IX.

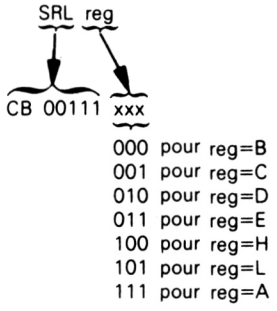


Cette instruction effectue le décalage d'un bit à droite du contenu de l'adresse spécifiée par le double registre HL. Le bit le plus significatif n'est pas modifié.

SRL reg — DÉCALAGE LOGIQUE A DROITE DU CONTENU D'UN REGISTRE



Ce schéma illustre l'exécution de l'instruction SRL E :

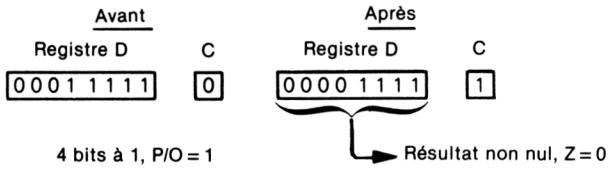


Cette instruction effectue le décalage logique d'un bit à droite du contenu du registre spécifié. Le bit le plus significatif est mis à zéro.

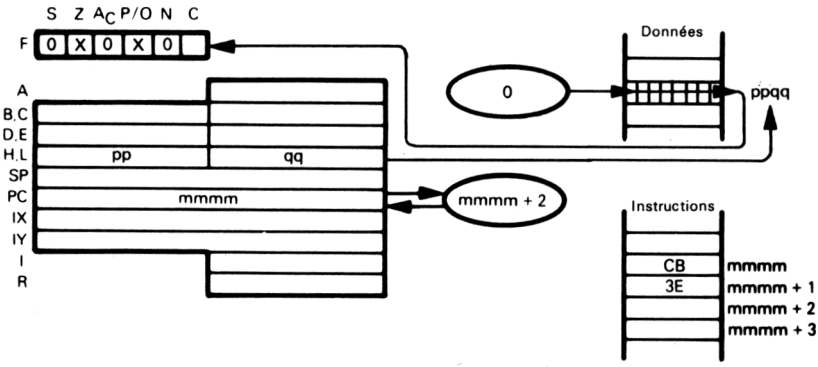
Supposons que le registre D contient $1F_{16}$ et que l'indicateur de retenue = 0. A la fin de l'exécution de l'instruction

SRL D

le registre D contient $0F_{16}$ et l'indicateur de retenue = 1.



SRL (HL) — DÉCALAGE LOGIQUE A DROITE D'UN OCTET MÉMOIRE
SRL (IX + disp)
SRL (IY + disp)



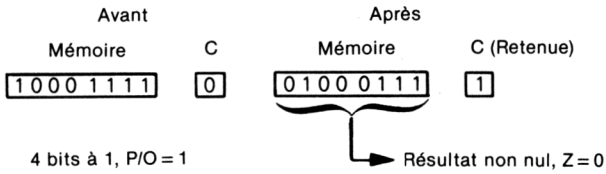
Ce schéma illustre l'exécution de l'instruction SRL (HL) :

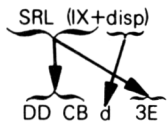
SRL (HL)
CB 3E

Cette instruction effectue le décalage logique d'un bit à droite de la mémoire adressée par le double registre HL. Le bit le plus significatif est mis à 0.
Supposons que le double registre HL contient 2000_{16}, l'adresse 2000_{16} contient $8F_{16}$ et que l'indicateur de retenue = 0. A la fin de l'exécution de l'instruction

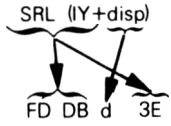
SRL (HL)

l'adresse 2000_{16} contient 47_{16} et l'indicateur de retenue = 1.



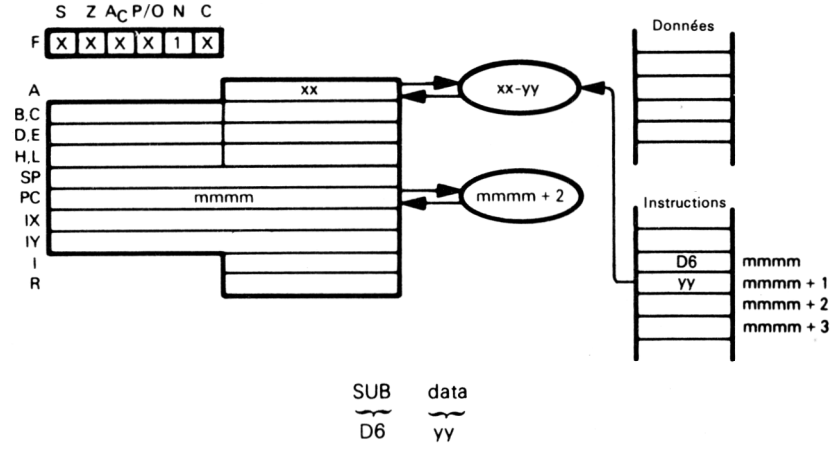


Cette instruction effectue le décalage logique d'un bit à droite du contenu de l'adresse (résultant de la somme du registre IX et du déplacement d). Le bit le plus significatif est mis à 0.

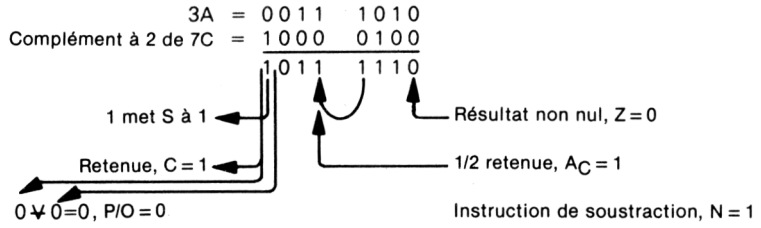


Cette instruction est identique à la précédente SRL (IX + disp), si ce n'est qu'elle utilise le registre IY à la place de IX.

SUB data — SOUSTRACTION D'UNE DONNÉE A PARTIR DE L'ACCUMULATEUR

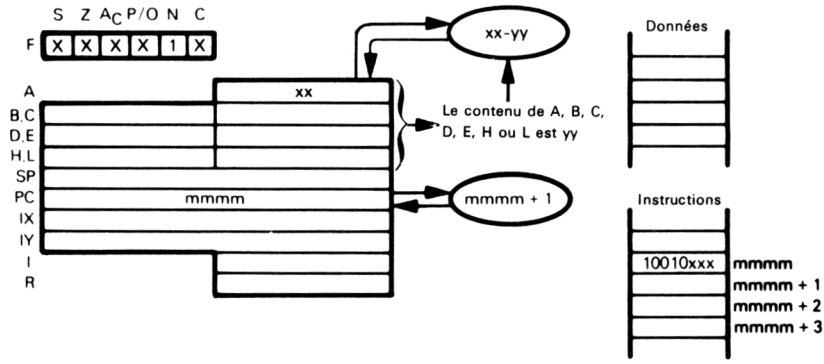


Cette instruction soustrait de l'accumulateur le contenu du second octet du code objet. Supposons que $xx = 3A_{16}$. A la fin de l'exécution de l'instruction SUB 7CH, l'accumulateur contient BE_{16}.



Remarque : La retenue résultante est complémentée.

SUB reg — SOUSTRACTION DU CONTENU D'UN REGISTRE A PARTIR DE L'ACCUMULATEUR



SUB reg
10010 xxx

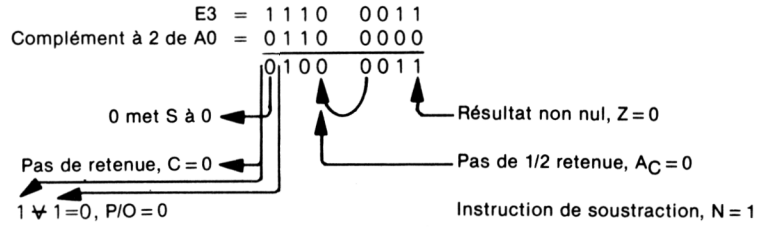
000 pour reg=B
001 pour reg=C
010 pour reg=D
011 pour reg=E
100 pour reg=H
101 pour reg=L
111 pour reg=A

Cette instruction soustrait du contenu de l'accumulateur le contenu du registre spécifié.

Supposons que $xx = E3_{16}$ et que le registre H contient $A0_{16}$. A la fin de l'exécution de l'instruction

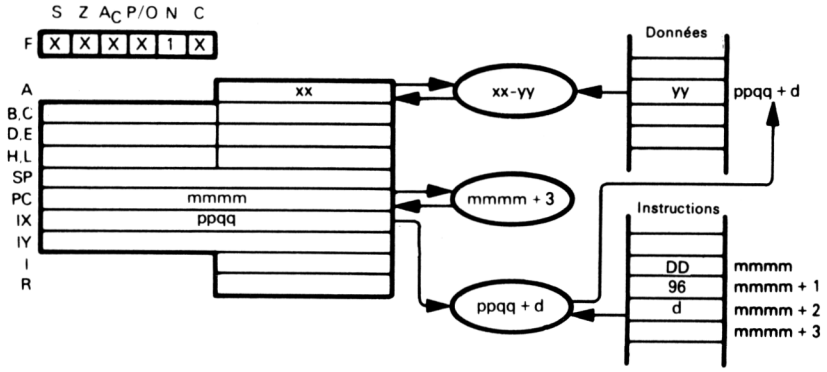
SUB H

l'accumulateur contient 43_{16}.



Remarque : La retenue résultante est complémentée.

SUB (HL) — SOUSTRACTION DU CONTENU D'UN OCTET MÉMOIRE SUB (IX + disp) A PARTIR DE L'ACCUMULATEUR SUB (IY + disp)



Ce schéma illustre l'exécution de l'instruction SUB (IX + d) :

SUB (IX+disp)

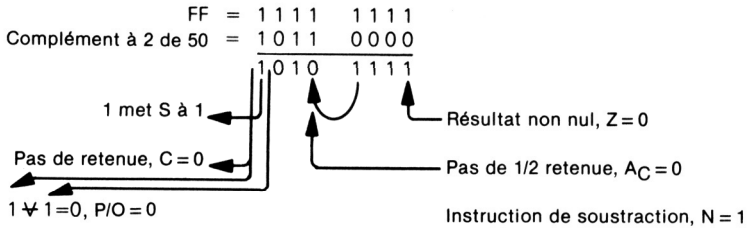
DD 96 d

Cette instruction soustrait du contenu de l'accumulateur le contenu de l'adresse (résultant de la somme du contenu du registre IX et du déplacement d).

Supposons que ppqq = 4000_{16}, xx = FF_{16} et que l'adresse $40FF_{16}$ contient 50_{16}. A la fin de l'exécution de l'instruction

SUB (IX+0FFH)

l'accumulateur contient AF_{16}.



Remarque : La retenue résultante est complémentée.

SUB (IY+disp)

FD 96 d

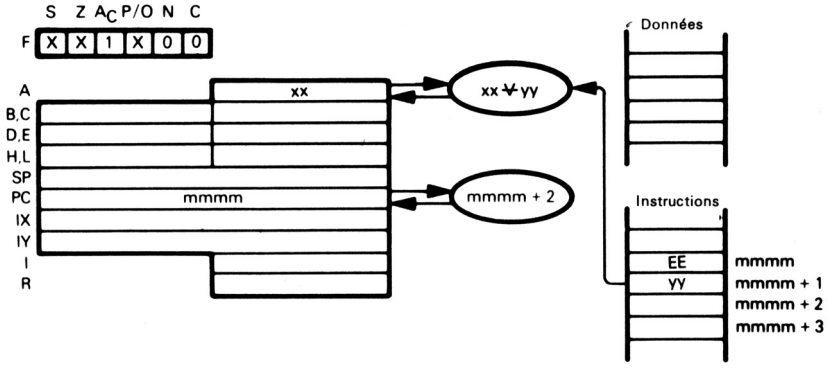
Cette instruction est identique à la précédente, si ce n'est qu'elle utilise le registre IY à la place de IX.

SUB (HL)

96

Cette instruction soustrait du contenu de l'accumulateur le contenu de l'adresse (spécifiée par le contenu du double registre HL).

XOR data — OU EXCLUSIF D'UNE DONNÉE AVEC L'ACCUMULATEUR



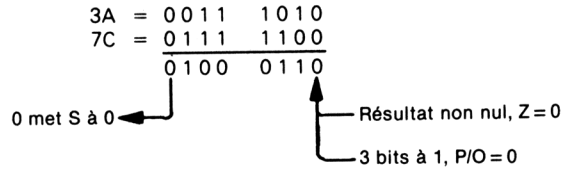
XOR data

EE yy

Cette instruction effectue le OU exclusif entre le contenu de l'accumulateur et celui du second octet du code objet.
Supposons que $xx = 3A_{16}$. A la fin de l'exécution de l'instruction

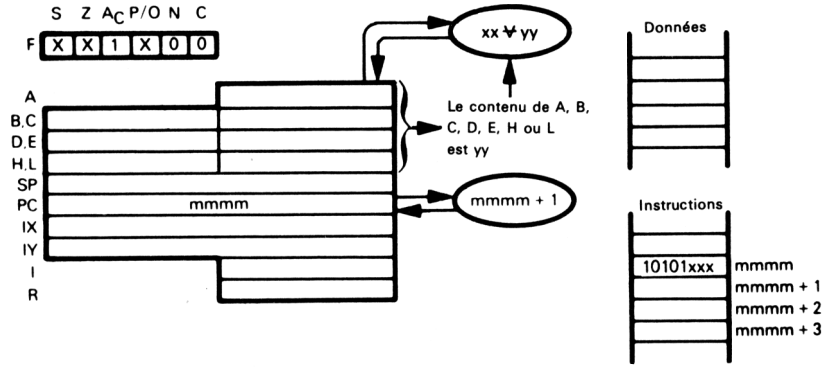
XOR 7CH

l'accumulateur contient 46_{16}.



Cette instruction sert à complémenter des bits spécifiques de l'accumulateur.

XOR reg — OU EXCLUSIF ENTRE UN REGISTRE ET L'ACCUMULATEUR



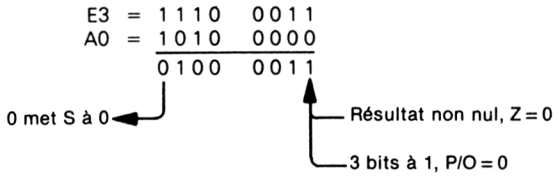
XOR reg
10101 xxx

000 pour reg=B
001 pour reg=C
010 pour reg=D
011 pour reg=E
100 pour reg=H
101 pour reg=L
111 pour reg=A

Cette instruction effectue un OU exclusif entre le contenu du registre spécifié et celui de l'accumulateur.
Supposons que $xx = E3_{16}$, le registre E contient $A0_{16}$. A la fin de l'exécution de l'instruction

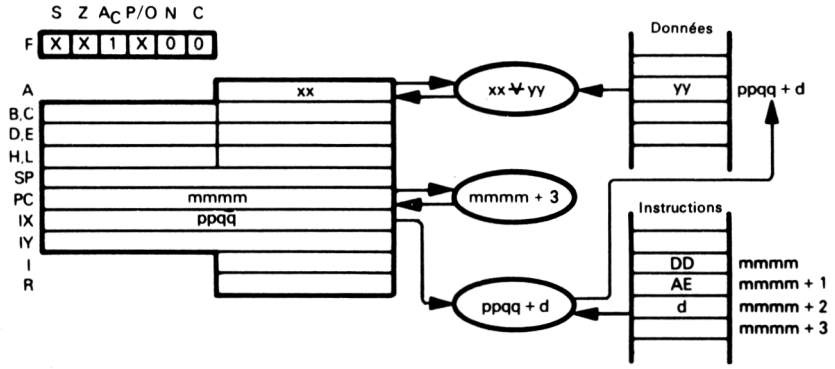
XOR E

l'accumulateur contient 43_{16}.



Cette instruction sert à complémenter des bits spécifiques de l'accumulateur.

XOR (HL) — OU EXCLUSIF ENTRE LA MÉMOIRE ET L'ACCUMULATEUR
XOR (IX + disp)
XOR (IY + disp)



Ce schéma illustre l'exécution de l'instruction XOR (IX + disp) :

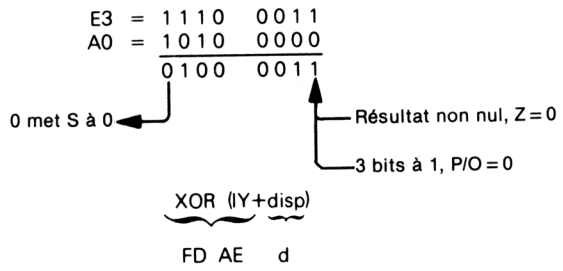
XOR (IX+disp)
DD AE d

Cette instruction effectue un OU exclusif entre le contenu de l'accumulateur et celui de l'adresse (résultant de la somme du contenu du registre IX et du déplacement d).

Supposons que xx = $E3_{16}$, ppqq = 4500_{16} et que l'adresse $45FF_{16}$ contient $A0_{16}$. A la fin de l'exécution de l'instruction

XOR (IX + 0FFH)

l'accumulateur contient 43_{16}.



Cette instruction est identique à XOR (IX + disp), si ce n'est qu'elle utilise le registre IY à la place de IX.

XOR (HL)
AE

Cette instruction effectue un OU exclusif entre le contenu de l'accumulateur et celui de l'adresse (spécifiée par le contenu du double registre HL).

COMPATIBILITÉ 8080A/Z80

CARACTÉRISTIQUES DE COMPATIBILITÉ 8080A/Z80

Bien que les qualités intrinsèques du microprocesseur Z80 soient amplement suffisantes, une de ses plus importantes caractéristiques reste sa compatibilité avec le microproceseur 8080A. Voici comment se présente cette compatibilité.

1) Toutes les instructions du langage machine du 8080A sont aussi des instructions du langage machine du Z80.
2) Tous les registres du 8080A sont aussi des registres du Z80 (voir tableau 3.6).
3) Presque tous les programmes du 8080A tournent sur un Z80, avec quelques différences minimes qui seront vues plus tard.
4) Le Z80 dispose d'instructions, de registres et d'autres éléments absents du 8080A ; donc les programmes du Z80 ne tournent pas en général sur des processeurs 8080A.

CONVERSION NIVEAU ASSEMBLEUR 8080A/Z80

On peut noter que **cette compatibilité ne s'étend pas aux instructions source du langage assembleur** puisque les assembleurs du Z80 et du 8080A utilisent des mnémoniques de code opératoire différents.

Le tableau 3-7 contient une liste des codes mnémoniques du 8080A et des codes correspondants du Z80 ; le tableau 3.8 représente la même liste rangée suivant l'ordre des codes du Z80.

CODES OPÉRATOIRES INUTILISÉS DU 8080A

Il faut remarquer les limitations du codage binaire qu'entraîne cette compatibilité sur les autres caractéristiques du Z80. Le 8080A dispose de codes opératoires inutilisés (voir tableau 3.9) qui servent à quelques instructions supplémentaires du Z80. Mais ces codes inutilisés ne sont pas en nombre suffisant pour couvrir de façon simple le grand nombre de caractèristiques du Z80.

CODES OPÉRATOIRES SUR 2 OCTETS

Ainsi, beaucoup des instructions supplémentaires du Z80 nécessitent un code opératoire sur 2 octets. Le premier octet est CB, DD, ED ou FD.

Voici la signification de ces codes d'après le tableau 3.9 :

CB — une opération relative à un registre ou à un bit
DD — une opération relative au registre IX
ED — une instruction diverse absente du 8080A, et non définie autre part
FD — une opération relative au registre IY.
Le second octet du code opératoire décrit l'opération qui doit être effectivement exécutée.

INSTRUCTIONS S'EXÉCUTANT PLUS OU MOINS RAPIDEMENT

En conclusion, ces instructions sur plusieurs octets s'exécutent plutôt lentement (et prennent plus de mémoire) car un accès mémoire supplémentaire est exigé. On doit être conscient de cette différence dans les temps d'exécution pour essayer d'utiliser si possible des instructions qui s'exécutent plus rapidement. Cet avertissement concerne particulièrement les instructions de décalage (RLC, RRC, RL, RR, SRA, SRL) ainsi que les instructions relatives aux registres d'index IX et IY.

INCOMPATIBILITÉS 8080A/Z80

Il y a quelques incompatibilités minimes entre le 8080A et le Z80. Les voici :

1) Le Z80 utilise l'indicateur P (ou P/O) pour indiquer un dépassement de capacité après les opérations arithmétiques (complément à 2). Le 8080A réserve cet indicateur à la parité.
2) Le Z80 et le 8080A exécutent de façon différente l'instruction DAA. Sur le Z80, cette instruction effectue une correction de la soustraction décimale comme de l'addition décimale. Sur le 8080A, elle ne corrige que l'addition décimale.
3) Les instructions de rotation du Z80 mettent à zéro l'indicateur A_C. Les instructions de rotation du 8080A ne modifient pas l'indicateur A_C.

Registres du Z80	Registres du 8080A
A	A
A'	—
B	B
B'	—
C	C
C'	—
D	D
D'	—
E	E
E'	—
F	Partie basse du PSW
F'	—
H	H
H'	—
I	—
IX	—
IY	—
L	L
L'	—
R	—
PC	PC
SP	SP

Paires de registres du Z80	Paires de registres du 8080A
BC	B
DE	D
HL	H
AF	PSW

Indicateurs du Z80	Indicateurs du 8080A
C (retenue)	C (retenue)
H (1/2 retenue)	AC (retenue auxiliaire)
N (soustraction)	—
P/O (parité/dépassement)	P (parité)
S (signe)	S (signe)
Z (zéro)	Z (zéro)

Tableau 3.6. — Correspondance entre le Z80 et le 8080A pour les registres et les indicateurs

INCOMPATIBILITÉS 8085/Z80

Le Z80 n'est pas compatible avec les caractéristiques supplémentaires des microprocesseurs 8085. Les codes utilisés pour RIM et SIM dans le 8085 servent à des sauts relatifs (NZ et NC) sur le Z80.

INCOMPATIBILITÉS DE TIMING **Les durées des instructions sur le 8080A, le 8085 et le Z80 sont toutes différentes. En conséquence, les programmes qui font intervenir de façon précise le temps d'exécution des instructions ne donnent des résultats satisfaisants que sur le processeur pour lequel ils sont écrits.**

L'indicateur N du Z80 occupe le bit 2 du registre F ; le bit correspondant dans le PSW (registre d'état) du 8080A est toujours sur un '1' logique.

Tableau 3.7. — Correspondance entre les mnémoniques du 8080A et du Z80

Mnémonique 8080A		Mnémonique Z80	
ACI	data	ADC	A,data
ADC	reg ou M	ADC	A,reg ou (HL)
ADD	reg ou M	ADD	A,reg ou (HL)
ADI	data	ADD	A,data
ANA	reg ou M	AND	reg ou (HL)
ANI	data	AND	data
CALL	addr	CALL	addr
CC	addr	CALL	C,addr
CM	addr	CALL	M,addr
CMA		CPL	
CMC		CCF	
CMP	reg ou M	CP	reg ou (HL)
CNC	addr	CALL	NC,addr
CNZ	addr	CALL	NZ,addr
CP	addr	CALL	P,addr
CPE	addr	CALL	PE,addr
CPI	data	CP	data
CPO	addr	CALL	PO,addr
CZ	addr	CALL	Z,addr
DAA		DAA	
DAD	rp	ADD	HL,rp
DCR	reg ou M	DEC	reg ou (HL)
DCX	rp	DEC	rp
DI		DI	
EI		EI	
HLT		HALT	
IN	port	IN	A,(port)
INR	reg ou M	INC	reg ou (HL)
INX	rp	INC	rp
JC	addr	JP	C,addr
JM	addr	JP	M,addr
JMP	addr	JP	addr
JNC	addr	JP	NC,addr
JP	addr	JP	P,addr
JNZ	addr	JP	NZ,addr
JPE	addr	JP	PE,addr
JPO	addr	JP	PO,addr
JZ	addr	JP	Z,addr
LDA	addr	LD	A,(addr)
LDAX	B ou D	LD	A,(BC) ou (DE)

Mnémonique 8080A		Mnémonique Z80	
LHLD	addr	LD	HL,(addr)
LXI	rp,data16	LD	rp,data16
MOV	reg,reg ou M	LD	reg, reg ou (HL)
MOV	reg ou M,reg	LD	reg ou (HL),reg
MVI	reg ou M,data	LD	reg ou (HL),data
NOP		NOP	
ORA	reg ou M	OR	reg ou (HL)
ORI	data	OR	data
OUT	port	OUT	(port),A
PCHL		JP	(HL)
POP	pr	POP	pr
PUSH	pr	PUSH	pr
RAL		RLA	
RAR		RRA	
RC		RET	C
RET		RET	
RLC		RLCA	
RM		RET	M
RNC		RET	NC
RNZ		RET	NZ
RP		RET	P
RPE		RET	PE
RPO		RET	PO
RRC		RRCA	
RST	n	RST	n
RZ		RET	Z
SBB	reg ou M	SBC	A,reg ou (HL)
SBI	data	SBC	A,data
SHLD	addr	LD	(addr),HL
SPHL		LD	SP,HL
STA	addr	LD	(addr),A
STAX	B ou D	LD	(BC) ou (DE),A
STC		SCF	
SUB	reg ou M	SUB	reg ou (HL)
SUI	data	SUB	data
XCHG		EX	DE,HL
XRA	reg ou M	XOR	reg ou (HL)
XRI	data	XOR	data
XTHL		EX	(SP),HL

Tableau 3.8. — Correspondance entre les mnémoniques du Z80 et du 8080A

Mnémonique Z80		Mnémonique 8080A	
ADC	A,data	ACI	data
ADC	A,(HL)	ADC	M
ADC	A,reg	ADC	reg
ADC	A,(xy + disp)	—	
ADC	HL,rp	—	
ADD	A,data	ADI	data
ADD	A,(HL)	ADD	M
ADD	A,reg	ADD	reg
ADD	A,(xy + disp)	—	
ADD	HL,rp	DAD	rp
ADD	IX,pp	—	
ADD	IY,rr	—	
AND	data	ANI	data
AND	(HL)	ANA	M
AND	reg	ANA	reg
AND	(xy + disp)	—	
BIT	b,(HL)	—	
BIT	b,reg	—	
BIT	b,(xy + disp)	—	
CALL	addr	CALL	addr
CALL	C,addr	CC	addr
CALL	M,addr	CM	addr
CALL	NC,addr	CNC	addr
CALL	NZ,addr	CNZ	addr
CALL	P,addr	CP	addr
CALL	PE,addr	CPE	addr
CALL	PO,addr	CPO	addr
CALL	Z,addr	CZ	addr
CCF		CMC	
CP	data	CPI	data
CP	(HL)	CMP	M
CP	reg	CMP	reg
CP	(xy + disp)	—	
CPD		—	
CPDR		—	
CPI		—	
CPIR		—	
CPL		CMA	
DAA		DAA	
DEC	(HL)	DCR	M
DEC	reg	DCR	reg
DEC	rp	DCX	rp
DEC	xy	—	
DEC	(xy + disp)	—	
DI		DI	
DJNZ	disp	—	
EI		EI	
EX	AF,AF'	—	
EX	DE,HL	XCHG	
EX	(SP),HL	XTHL	
EX	(SP),xy	—	
EXX		—	
HALT		HLT	
IM	m	—	
IN	A,(port)	IN	port
IN	reg,(C)	—	
INC	(HL)	INR	M
INC	reg	INR	reg

Mnémonique Z80		Mnémonique 8080A	
INC	rp	INX	rp
INC	xy	—	
INC	(xy + disp)	—	
IND		—	
INDR		—	
INI		—	
INIR		—	
JP	addr	JMP	addr
JP	C,addr	JC	addr
JP	(HL)	PCHL	
JP	M,addr	JM	addr
JP	NC,addr	JNC	addr
JP	NZ,addr	JNZ	addr
JP	P,addr	JP	addr
JP	PE,addr	JPE	addr
JP	PO,addr	JPO	addr
JP	Z,addr	JZ	addr
JP	xy	—	
JR	C,disp	—	
JR	disp	—	
JR	NC,disp	—	
JR	NZ,disp	—	
JR	Z,disp	—	
LD	A,(addr)	LDA	addr
LD	A,(BC) ou (DE)	LDAX	B ou D
LD	A,I	—	
LD	A,R	—	
LD	(addr),A	STA	addr
LD	(addr),BC ou DE	—	
LD	(addr),HL	SHLD	addr
LD	(addr),SP	—	
LD	(addr),xy	—	
LD	(BC) ou (DE),A	STAX	B ou D
LD	BC ou DE,(addr)	—	
LD	HL,(addr)	LHLD	addr
LD	(HL),data	MVI	M,data
LD	(HL),reg	MOV	M,reg
LD	I,A	—	
LD	R,A	—	
LD	reg,data	MVI	reg,data
LD	reg,(HL)	MOV	reg,M
LD	reg,reg	MOV	reg,reg
LD	reg,(xy + disp)	—	
LD	rp,data16	LXI	rp,data16
LD	SP,(addr)	—	
LD	SP,HL	SPHL	
LD	SP,xy	—	
LD	xy,data16	—	
LD	xy,(addr)	—	
LD	(xy + disp),data	—	
LD	(xy + disp),reg	—	
LDD		—	
LDDR		—	
LDI		—	
LDIR		—	
NEG		—	
NOP		NOP	
OR	data	ORI	data

— indique qu'il n'y a pas d'instruction correspondante

Tableau 3.8. — (suite)

Mnémonique Z80		Mnémonique 8080A	
OR	(HL)	ORA	M
OR	reg	ORA	reg
OR	(xy + disp)	—	
OTDR		—	
OTIR		—	
OUT	(C),reg	—	
OUT	(port),A	OUT	port
OUTD		—	
OUTI		—	
POP	pr	POP	pr
POP	xy	—	
PUSH	pr	PUSH	pr
PUSH	xy	—	
RES	b,(HL)	—	
RES	b,reg	—	
RES	b,(xy + disp)	—	
RET		RET	
RET	C	RC	
RET	M	RM	
RET	NC	RNC	
RET	NZ	RNZ	
RET	P	RP	
RET	PE	RPE	
RET	PO	RPO	
RET	Z	RZ	
RETI		—	
RETN		—	
RL	(HL)	—	
RL	reg	—	
RL	(xy + disp)	—	
RLA		RAL	
RLC	(HL)	—	
RLC	reg	—	
RLC	(xy + disp)	—	
RLCA		RLC	
RLD		—	

Mnémonique Z80		Mnémonique 8080A	
RR	(HL)	—	
RR	reg	—	
RR	(xy + disp)	—	
RRA		RAR	
RRC	(HL)	—	
RRC	reg	—	
RRC	(xy + disp)	—	
RRCA		RRC	
RRD		—	
RST	n	RST	n
SBC	A,data	SBI	data
SBC	A,(HL)	SBB	M
SBC	A,reg	SBB	reg
SBC	A,(xy + disp)	—	
SBC	HL,rp	—	
SCF		STC	
SET	b,(HL)	—	
SET	b,reg	—	
SET	b,(xy + disp)	—	
SLA	(HL)	—	
SLA	reg	—	
SLA	(xy + disp)	—	
SRA	(HL)	—	
SRA	reg	—	
SRA	(xy + disp)	—	
SRL	(HL)	—	
SRL	reg	—	
SRL	(xy + disp)	—	
SUB	data	SUI	data
SUB	(HL)	SUB	M
SUB	reg	SUB	reg
SUB	(xy + disp)	—	
XOR	data	XRI	data
XOR	(HL)	XRA	M
XOR	reg	XRA	reg
XOR	(xy + disp)	—	

— indique qu'il n'y a pas d'instruction correspondante

CONVENTIONS DE L'ASSEMBLEUR DU Z80 ZILOG

L'assembleur standard Z80 est disponible chez tous les fabricants du Z80 et sur la plupart des réseaux à temps partagé ; il est aussi inclus dans la plupart des systèmes de développement. La plupart des gros calculateurs et beaucoup de minis disposent de versions de « cross assembleur ».

Tableau 3.9. — Codes opératoires inutilisés du 8080A et leur usage sur Z80

Code opératoire 8080A	Usage Z80
08	EX AF,AF'
10	DJNZ disp
18	JR disp
20 (RIM sur 8085)	JR NZ,disp
28	JR Z,disp
30 (SIM sur 8085)	JR NC,disp
38	JR C,disp
CB	BIT, RES, RL, RLC, RR, RRC, SET, SLA, SRA, SRL
D9	EXX
DD	Toute instruction relative au registre IY.
ED	ADC HL,rp / CPD / CPDR / CPI / CPIR / IM m / IN reg,(C) / IND / INDR / INI / INIR / LD (addr),rp / LD A,I / LD A,R / LD I,A / LD R,A / LD rp,(addr) / LDD / LDDR / LDI / LDIR / NEG / OTDR / OTIR / OUT (C),reg / OUTD / OUTI / RETI / RETN / RLD / RRD / SBC HL,rp
FD	Toute instruction relative au registre IX.

STRUCTURE DE L'ASSEMBLEUR

Les instructions du langage assembleur ont une structure standard (voir tableau 2.1). Les délimiteurs nécessaires sont :

1) deux points (:) après une étiquette, sauf pour les pseudo-opérations EQU, DEFL et MACRO qui nécessitent un espace ;
2) un blanc après le code opératoire ;
3) une virgule entre les opérandes dans la partie opérande ;
4) un point-virgule devant un commentaire ;
5) des parenthèses autour des références mémoire.

Voici un exemple typique d'instructions en assembleur Z80 :

```
DEBUT :   LD      A,(1000)      ; CHARGE LONGUEUR
          ADD     HL,DE
          HALT
```

ÉTIQUETTES

L'assembleur autorise jusqu'à 6 caractères dans les étiquettes ; le premier caractère doit être une lettre, les caractères suivants pouvant être indifféremment des lettres, des chiffres ou les caractères ? et _ (souligné). Nous n'utilisons ici que des majuscules ou des chiffres, bien que d'autres versions de l'assembleur autorisent des minuscules ou d'autres symboles.

NOMS RÉSERVÉS

Quelques noms sont réservés à des mots clés et ne doivent pas être utilisés par le programmeur. Il s'agit des noms des registres (A, B, C, D, E, H, L, I, R), les noms des registres 16 bits (IX, IY, SP), les noms des doubles registres (AF, BC, DE, HL, AF', BC', DE', HL') et les différents états des 4 indicateurs (C, NC, Z, NZ, M, P, PE, PO).

PSEUDO-OPÉRATIONS

L'assembleur dispose des pseudo-opérations élementaires suivantes :

DEFB	— **déclaration d'un octet** (« define byte »)
DEFL	— **déclaration d'une étiquette** (« define label »)
DEFM	— **déclaration d'une chaîne** (« define message »)
DEFS	— **réservation d'espace mémoire** (« define storage »)
DEFW	— **déclaration d'un mot** (« define word »)
END	— **fin** (« end »)
EQU	— **valeur** (« equate »)
ORG	— **origine** (« origin »)

PSEUDO-OPÉRATIONS DEFB, DEFM et DEFW

DEFB, DEFM et DEFW sont les pseudo-opérations qui servent à mettre en ROM des données. DEFB est utilisée pour une donnée 8 bits, DEFW pour une donnée 16 bits et DEFM pour des chaînes ASCII (de 63 caractères au plus). Le seul point particulier à se rappeler est que DEFW range en mémoire les 8 bits les moins significatifs de la donnée dans le premier mot et les 8 bits les plus significatifs dans le second mot. C'est la procédure standard 8080A/8085/Z80 pour le rangement d'adresses en mémoire mais qui est contraire à la pratique normale. Il faut être conscient de cet ordre lorsque l'on range des données 16 bits.

On peut noter que DEFB et DEFW définissent respectivement la valeur d'un seul octet ou d'un seul mot. La mise en place d'une table de valeurs nécessite autant de pseudo-opérations DEFB (DEFW) qu'il y a d'octets (mots) de données.

Exemples :

```
ADR:    DEFW    3165H
```

donne (ADR) = 65, (ADR + 1) = 31 hexadécimal

La pseudo-opération :

```
TCONV:  DEFB    32
```

place le nombre 32 dans l'octet suivant de mémoire et assigne le nom TCONV à l'adresse de cet octet.
La pseudo-opération :

```
ERROR:  DEFM    'ERREUR'
```

place les caractères ASCII 7-bits E, R, R, E, U, R dans les six octets suivants de mémoire et assigne le nom ERROR à l'adresse du premier octet.
La série de pseudo-opérations :

```
OPERS:  DEFW    FADD
        DEFW    FSUB
        DEFW    FMUL
        DEFW    FDIV
```

place les adresses FADD, FSUB, FMUL et FDIV dans les 8 octets suivants de mémoire et assigne le nom OPERS à l'adresse du premier octet.
Remarque : le premier octet contient les bits de poids faible de l'adresse FADD.

PSEUDO-OPÉRATION DEFS DEFS est la pseudo-opération qui sert à assigner des emplacements mémoire en RAM ; elle réserve un nombre spécifique d'octets.

PSEUDO-OPÉRATION EQU EQU est la pseudo-opération qui sert à assigner des valeurs aux noms.

PSEUDO-OPÉRATION DEFL La pseudo-opération DEFL est similaire à EQU, si ce n'est que DEFL permet une redéfinition ultérieure du nom. DEFL ressemble à la directive SET d'autres assembleurs. Cette pseudo-opération ne devrait servir qu'à définir des variables au moment de l'assemblage (c'est-à-dire les variables utilisées en assembleur conditionnel ou dans des macro-instructions).

PSEUDO-OPÉRATION ORG ORG est la pseudo-opération Origine standard. Les programmes du Z80 ont habituellement plusieurs origines ; les origines sont utilisées comme suit :

1) pour indiquer l'adresse de RESET (habituellement 0) ;
2) pour indiquer les points d'entrée d'interruption (habituellement entre 0 et 66_{16} mais ils peuvent être mis n'importe où en mémoire) ;
3) pour indiquer l'adresse de départ du programme principal ;
4) pour indiquer les adresses de départ des sous-programmes ;

5) pour définir des zones pour le rangement en RAM ;
6) pour définir une zone pour la pile RAM ;
7) pour indiquer des adresses destinées aux ports d'entrée/sortie et à des fonctions spéciales.

Exemples :

```
RESET     EQU     0
          ORG     RESET
```

Cette pseudo-instruction permet de placer la séquence d'instruction d'initialisation en début de mémoire à l'adresse 0.

```
INT1      EQU     38H
          ORG     INT1
```

Les instructions qui suivent seront rangées en mémoire à partir de l'adresse 38_{16}.

PSEUDO-OPÉRATION END La pseudo-opération END marque la fin du programme en langage assembleur.

Les pseudo-opérations particulières COND, MACRO, ENDC et ENDM sont décrites ultérieurement dans ce chapitre.

ÉTIQUETTES DES PSEUDO-OPÉRATIONS

Les règles et les recommandations relatives aux étiquettes des pseudo-opérations du Z80 sont définies comme suit :
1) les étiquettes sont obligatoires pour les pseudo-opérations EQU, DEFL et MACRO, puisque la fonction de ces pseudo-opérations est de définir la signification de l'étiquette.
2) Les pseudo-opérations DEFB, DEFM, DEFW et DEFS ont habituellement des étiquettes.
3) Les pseudo-opérations ORG, COND, ENDC, ENDM et END ne doivent pas avoir d'étiquettes puisque la signification de telles étiquettes est peu claire.

ADRESSES

DIFFÉRENTES FORMES DU CHAMP D'ADRESSE L'assembleur ZILOG du Z80 permet d'utiliser les formes suivantes dans le champ d'adresse :

1) décimale (cas par défaut)
   ex. : 1247
2) hexadécimale (doit commencer par un chiffre et se terminer par un H)
   ex. : 142CH, 0E7H
3) octale (doit finir par O ou Q, mais Q est moins ambigu)
   ex. : 1247Q ou 1247O
4) binaire (doit finir par B)
   ex. : 1001001000111B

5) ASCII (entourée par des apostrophes)
ex. : 'ICI'
6) sous forme de déplacement à partir du compteur ordinal ($)
ex. : $ + 237H

OPÉRATIONS ARITHMÉTIQUES ET LOGIQUES DE L'ASSEMBLEUR

Les opérandes des opérations arithmétiques et logiques à l'intérieur du champ d'adresse sont considérés ainsi que les résultats produits comme des données 16 bits. Ces opérations sont autorisées en tant que membres d'expressions dans le champ d'adresse.

Lorsque l'on définit des constantes d'adresse, on devrait utiliser la notation hexadécimale. Les constantes binaires de 16 bits sont une source permanente d'erreurs. Les constantes octales sont peu pratiques du fait que les adresses sont rangées suivant le format octet de bas rang - octet de haut rang. Cette division apparaît au milieu d'un chiffre octal, ce qui oblige à fractionner le chiffre. Par exemple, pour exprimer l'adresse 9D7FH ou 116577Q dans le format bas-haut, on obtient 7F9DH ou 77236Q. Comme on le constate, les chiffres sont simplement transposés dans la notation hexadécimale, ce qui n'est pas le cas pour la notation octale.

OPERATEUR	FONCTION	PRIORITE
+	PLUS UNAIRE	1
-	MOINS UNAIRE	1
.NOT. ou \	NON LOGIQUE	1
.RES.	RESULTAT	1
**	EXPONENTIATION	2
*	MULTIPLICATION	3
/	DIVISION	3
.MOD.	MODULO	3
.SHR.	DECALAGE LOGIQUE A DROITE	3
.SHL.	DECALAGE LOGIQUE A GAUCHE	3
+	ADDITION	4
-	SOUSTRACTION	4
.AND. ou &	ET LOGIQUE	5
.OR. ou !	OU LOGIQUE	6
.XOR.	OU EXCLUSIF	6
.EQ. ou =	EGALE	7
.GT. ou >	SUPERIEUR A	7
.LT. ou <	INFERIEUR A	7
.UGT.	SUPERIEUR A (NON SIGNE)	7
.ULT.	INFERIEUR A (NON SIGNE)	7

Dans les expressions d'adresse qui comportent plus d'un opérateur, l'ordre de calcul est défini par les priorités données par la liste ci-dessus. Les opérateurs ayant les mêmes priorités sont évalués de gauche à droite. Les expressions entre parenthèses sont calculées d'abord. On ne doit pas oublier que mettre entièrement une expression entre parenthèses indique une adresse mémoire.
Remarques :

1) L'opérateur résultat (.RES.) provoque la suppression du dépassement (c'est-à-dire qu'un changement de signe provoqué par le dépassement dans le bit de signe ne génère pas d'erreur à l'assemblage).

2) Les décalages ont la forme :

```
.SHR.   op1,op2
.SHL.   op1,op2
```

avec : op1 : nombre à décaler
op2 : nombre de décalages

Les décalages sont logiques donc les zéros sont générés dans les bits de poids fort ou de poids faible (suivant le sens du décalage).

3) Les opérateurs de comparaison produisent soit un VRAI logique (tous les bits à 1) soit un FAUX logique (0).

4) Les opérandes des opérateurs .GT. et .LT. doivent être des nombres complémentés à 2 et signés alors que ceux des opérateurs .UGT. et .ULT. sont non signés. Ceci signifie que pour .GT. et .LT., les nombres positifs complémentés à 2 sont plus grands que les nombres négatifs complémentés à 2, alors que c'est le cas inverse pour .UGT. et .ULT.

ASSEMBLAGE CONDITIONNEL

PSEUDO-OPÉRATIONS COND et ENDC **L'assembleur Z80 dispose d'une simple capacité d'assemblage conditionnel fondée sur les pseudo-opérations COND et ENDC.**

COND est suivie par une expression, par exemple :

```
COND    BASE – 100H
          ou
COND    BASE – OPER1
```

Si cette expression n'est pas nulle, l'assembleur inclue toutes les instructions jusqu'à la pseudo-opération ENDC du programme ; si l'expression est nulle, l'assembleur ignore toutes les instructions entre COND et ENDC.

Nous n'utiliserons pas d'assemblage conditionnel et n'y ferons pas référence ; mais c'est parfois un moyen pratique d'ajouter ou de supprimer des instructions de recherche d'erreur, ou pour configurer certaines versions d'un programme courant.

PSEUDO-OPÉRATIONS MACRO et ENDM **L'assembleur standard Z80 autorise la création de macro-instructions qui permettent d'assigner des noms à des séquences d'instructions.** On utilise la pseudo-opération MACRO pour commencer la définition et ENDM pour la terminer. Une macro-instruction peut avoir des paramètres ou peut comprendre n'importe quelle instruction du langage assembleur, sauf les définitions d'autres macro-instructions.

La possibilité qu'offre une macro-instruction est souvent un raccourci pratique dans la programmation, mais nous ne l'utiliserons pas.

On peut remarquer que les séquences d'instructions définies sont généralement très courtes ; elles ne doivent pas excéder 10 ou 15 instructions. Des séquences plus longues doivent être transformées en sous-programmes pour sauvegarder de l'espace mémoire.

Chaque pseudo-opération MACRO doit avoir une étiquette ; l'étiquette est le nom qui identifie la macro. A ce sujet, voir le chapitre 2.

NDT. Ajoutons également que ZILOG propose pour ses microproceseurs et pour le Z80 en particulier, des assembleurs de type structuré permettant une programmation en assembleur plus lisible et donc plus efficace (cf. *Z80 PLZ/ASM Programming Language,* documentation ZILOG).

CHAPITRE 4

PROGRAMMES SIMPLES

Le meilleur moyen d'apprendre à programmer en langage assembleur est d'écrire des programmes en assembleur. C'est ce qui sera vu dans les six chapitres suivants qui contiennent des exemples d'applications spécifiques des microprocesseurs. Des exercices en fin de chapitre donnent des variantes des exemples traités en cours de chapitre. L'idéal serait de programmer ces exemples sur un système bâti autour du Z80 et ce pour être sûr d'avoir bien compris l'objet du chapitre.
Dans ce chapitre, on commencera par traiter des programmes très simples.

FORMAT GÉNÉRAL DES EXEMPLES

Chaque exemple de programme contient les parties suivantes :

FORMAT TYPE D'UN EXEMPLE

1) Un titre qui décrit le problème dans son ensemble.
2) Une phrase qui décrit la tâche spécifique du programme, ainsi que les emplacements mémoire qu'il utilise.
3) Un exemple pratique : données entrées et résultats correspondants.
4) Un organigramme si la logique du programme est complexe.
5) Le listing du programme source ou assembleur.
6) Le listing du code objet ou hexadécimal du programme.
7) Des notes explicatives concernant les instructions et les méthodes utilisées dans le programme.

Les exercices de la fin du chapitre sont semblables aux exemples. Ces exercices devront être programmés sur un système bâti autour du Z80 en utilisant les exemples comme guides.
Les programmes source des exemples ont été construits comme suit :

DIRECTIVES POUR LES EXEMPLES

1) La notation standard de l'assembleur Z80 est utilisée (voir chapitre 3).
2) Le format des données et des adresses a été déterminé par un souci de clarté plutôt que d'homogénéité. On utilisera :

— des nombres hexadécimaux pour les adresses mémoire, les codes d'instruction et les données DCB (décimales codées en binaire) ;
— des nombres décimaux pour les constantes numériques ;
— des nombres binaires pour les masques logiques ;
— le code ASCII pour les caractères.

3) Les instructions et les techniques de programmation fréquemment utilisées sont mises en relief.
4) Des exemples illustrent les tâches réalisées par les microprocesseurs dans le domaine des communications, de l'instrumentation, de l'informatique, de l'équipement de bureau, dans les domaines industriel et militaire.
5) Les programmes contiennent des commentaires détaillés.
6) Des structures simples et claires sont mises en relief, mais les programmes sont aussi performants que possible compte tenu de ces directives. Les annotations décrivent souvent des procédures plus performantes.
7) Les programmes utilisent toujours les mêmes emplacements mémoire. Chaque programme commence à l'adresse 0000 (l'adresse de RESET) et se termine par l'instruction HALT. Si votre micro-ordinateur n'a ni moniteur, ni interruptions, il est possible de terminer les programmes par une instruction qui boucle, par exemple :

```
ICI :        JR          ICI
```

Le codage de cette instruction en hexadécimal sera 18 suivi de FE. On peut remplacer l'instruction HALT ou JR ICI par une instruction RESTART ou JP qui redonne le contrôle au moniteur dans certains micro-ordinateurs à base de Z80.
Consultez le manuel d'utilisation de votre microcalculateur pour connaître les emplacements mémoire utilisés ainsi que l'instruction finale de votre système particulier.

DIRECTIVES POUR LES EXERCICES

DIRECTIVES DE PROGRAMMATION

En abordant les exercices de fin de chapitre, essayez de respecter les consignes suivantes :

1) Commenter chaque programme de telle sorte qu'il puisse être compris par d'autres. Les commentaires peuvent être brefs et grammaticalement incorrects. Ils doivent expliquer l'objet d'une section ou d'une instruction du programme. Un commentaire ne doit pas décrire l'action des instructions (cette description se trouve dans les manuels). Il est inutile de commenter chaque phrase ou d'expliquer l'évidence. On peut suivre le format type des exemples, mais sans détailler autant.
2) Rechercher la clarté, la simplicité et une bonne structure des programmes. Faire des programmes relativement performants sans chercher à gagner un octet ou quelques microsecondes.
3) Faire des programmes relativement généraux. Attention à ne pas confondre les paramètres (nombre d'éléments d'un tableau,...) avec des constantes fixes (π, caractère ASCII c,...)
4) Ne pas donner aux paramètres des valeurs fixées initialement, mais pour initialiser les paramètres, utiliser une instruction de chargement.
5) Utiliser la notation standard (définie au chapitre 3) comme dans les exemples.
6) Utiliser la notation hexadécimale pour les adresses. Choisir le format le plus clair possible pour les données.
7) Si c'est possible, commencer tous les programmes à l'adresse 0000 et utiliser les adresses à partir de 0040_{16} pour les données et le stockage temporaire. Sinon, prendre les adresses équivalentes de votre micro-ordinateur et utiliser les mêmes. A nouveau, consulter le manuel d'utilisation.

8) Choisir des noms expressifs pour les étiquettes et les variables (CONTROLE ou SOMME plutôt que X, Y ou Z).
9) Exécuter chaque programme sur le micro-ordinateur. Il n'y a pas d'autre moyen de tester le programme. Un exemple pratique est fourni avec chaque exercice. S'assurer que le programme marche aussi pour les cas particuliers.

Voici maintenant quelques informations utiles à se rappeler en écrivant des programmes :

UTILISATION DE L'ACCUMULATEUR Presque toutes les instructions de calcul (comme ADD, SUB, AND, OR) utilisent l'accumulateur. Le plus souvent, on chargera des données dans l'accumulateur avec LD :

— soit l'instruction LD A,(addr) pour charger une donnée à partir d'une adresse quelconque ;
— soit l'instruction LD A,(HL) pour charger une donnée à partir de l'adresse contenue dans les registres H et L.

Attention ! Les parenthèses indiquent une adresse et non une donnée.

UTILISATION DE LA PAIRE DE REGISTRES HL La meilleure méthode d'adressage mémoire est l'emploi de l'adressage implicite à l'aide des registres H et L — c'est-à-dire l'utilisation de (HL). Ce code provoque un accès mémoire à l'adresse contenue dans les registres H et L. On utilisera l'instruction LD HL,data16 pour charger un nombre fixe dans les registres H et L, et l'instruction LD HL,(addr) pour charger le contenu de 2 adresses successives dans H et L. On utilisera INC HL ou DEC HL pour incrémenter ou décrémenter (de 1) l'adresse dans les registres H et L.
Les opérations logiques et arithmétiques sur 8 bits utilisent toutes la donnée contenue dans l'accumulateur comme un opérande et mettent le résultat dans l'accumulateur.

INSTRUCTIONS SPÉCIALES Certaines des opérations logiques et arithmétiques sur 8 bits ont des rôles spécifiques. Par exemple :

SUB A (ou XOR A) remet l'accumulateur à zéro.
ADD A,A effectue sur l'accumulateur un décalage logique à gauche sur 1 bit. Cette instruction multiplie aussi par 2 le contenu de l'accumulateur.
AND A (ou OR A) remet l'indicateur de retenue (Carry) à zéro sans changer le contenu de l'accumulateur.
Un ET logique permet de masquer certaines parties d'un mot. Le masque doit contenir des '1' aux positions que l'on veut conserver et des '0' à celles qu'on veut mettre à zéro.

EXEMPLES DE PROGRAMME

Complémentation à un

But : effectuer la complémentation logique du contenu de l'adresse 0040 et mettre le résultat à l'adresse 0041.

Exemple pratique :

(0040) = 6A
Résultat : (0041) = 95

Programme source :

```
LD      A,(40H)     ; CHARGER DONNEE
CPL                 ; COMPLEMENTER
LD      (41H),A     ; RANGER LE RESULTAT
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3A	LD	A,(40H)
0001	40		
0002	00		
0003	2F	CPL	
0004	32	LD	(41H),A
0005	41		
0006	00		
0007	76	HALT	

Les instructions LD A,(addr) et LD (addr),A contiennent les adresses qui permettent d'identifier l'origine et la destination de la donnée. Les adresses sont sur 16 bits, les 8 bits de poids le plus faible se trouvant dans l'octet qui suit immédiatement le code d'instruction et les 8 bits de poids fort dans l'octet suivant (l'ordre est ici contraire à l'usage normal des calculateurs).
CPL est une instruction d'un octet qui inverse chaque bit de l'accumulateur. Elle remplace chaque '0' par '1' et vice-versa, comme un ensemble de circuits inverseurs.
L'instruction HALT est employée pour terminer tous les exemples.
On pourrait aussi mettre une adresse dans les registres H et L et ensuite l'utiliser tout au long du programme. C'est ce qui est fait dans le programme suivant.

Programme source :

```
LD      HL,40H      ; POINTE SUR OPERANDE
LD      A,(HL)      ; LECTURE DONNEE
CPL                 ; COMPLEMENTATION
INC     HL          ; POINTE SUR DESTINATION
LD      (HL),A      ; RANGE RESULTAT
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)
0000	21	LD HL,40H
0001	40	
0002	00	
0003	7E	LD A,(HL)
0004	2F	CPL
0005	23	INC HL
0006	77	LD (HL),A
0007	76	HALT

A votre avis, quelle est la meilleure version ? Les deux versions nécessitent le même nombre d'octets mémoire, bien que la seconde version comporte 2 instructions supplémentaires. En effet, la seconde version emploie moins d'adresses explicites.

Addition 8-bits

But : additionner les contenus des adresses 0040 et 0041 et mettre le résultat à l'adresse 0042.

Exemple pratique :

(0040) = 38
(0041) = 2B
Résultat : (0042) = 63

Programme source :

```
LD      A,(40H)     ; CHARGER 1er OPERANDE
LD      B,A         ; SAUVEGARDER 1er OPERANDE
LD      A,(41H)     ; CHARGER 2e OPERANDE
ADD     A,B         ; ADDITIONNER OPERANDE
LD      (42H),A     ; RANGER SOMME
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)
0000	3A	LD A,(40H)
0001	40	
0002	00	
0003	47	LD B,A
0004	3A	LD A,(41H)
0005	41	
0006	40	
0007	80	ADD A,B
0008	32	LD (42H),A
0009	42	
000A	00	
000B	76	HALT

Ici encore, on pourrait aussi bien se servir des registres H et L comme origine de toutes les adresses.

Programme source :

```
LD      HL,40H
LD      A,(HL)      ; CHARGER 1er OPERANDE
INC     HL
ADD     A,(HL)      ; ADDITION 2e OPERANDE
INC     HL
LD      (HL),A      ; RANGER RESULTAT
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	7E	LD	A,(HL)
0004	23	INC	HL
0005	86	ADD	A,(HL)
0006	23	INC	HL
0007	77	LD	(HL),A
0008	76	HALT	

Dans ce cas, le programme qui utilise les registres H et L est plus court que celui qui utilise l'adressage direct. Pourquoi ?
LD HL,40H charge le contenu des 2 octets suivant le code de l'instruction dans la paire de registres HL. Le premier octet va dans le registre L, le second dans le registre H.
Le code (HL) indique que les registres H et L contiennent l'adresse source ou destination de la donnée.
Ainsi, LD A,(HL) charge le contenu de l'adresse donnée par H et L dans l'accumulateur ; LD (HL),A charge le contenu de l'accumulateur dans la mémoire adressée par H et L.
ADD A,(HL) additionne le contenu de l'adresse donnée par H et L avec le contenu de l'accumulateur.
Il faut se rappeler que H et L contiennent une adresse 16 bits, mais que le contenu de l'adresse est un mot de 8 bits. Faire bien la distinction entre ADD A,(HL) et ADD A,H ou ADD A,L.
L'instruction INC HL effectue une incrémentation sur 16 bits pendant un cycle d'instructions. L'unité centrale n'utilise pas l'unité arithmétique 8 bits pour l'incrémentation ; elle se sert du même incrémenteur que pour le compteur ordinal sur 16 bits.
Il est préférable d'utiliser les instructions LD A,(HL) et LD (HL),A plutôt que LD A,(addr) et LD (addr),A chaque fois que l'on se sert plusieurs fois des mêmes adresses ou que l'on utilise des adresses adjacentes. En effet, les instructions LD A,(HL) et LD (HL),A nécessitent à la fois moins de mémoire et de temps. Toutefois, il ne faut pas oublier de charger les registres H et L avant d'utiliser (HL).

Décalage de 1 bit à gauche

But : décaler le contenu de l'adresse 0040 de 1 bit à gauche et ranger le résultat à l'adresse 0041. Remettre à zéro la position du bit vide. Un tel décalage est dit décalage logique. Dans un décalage logique, on remet toujours à zéro les positions des bits vides.

Exemple pratique :

(0040) = 6F
Résultat : (0041) = DE

Programme source :

```
LD      A,(40H)     ; CHARGER DONNEE
ADD     A,A         ; DECALER A GAUCHE
LD      (41H),A     ; RANGER RESULTAT
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)
0000	3A	LD A,(40H)
0001	40	
0002	00	
0003	87	ADD A,A
0004	32	LD (41H),A
0005	41	
0006	00	
0007	76	HALT

L'instruction ADD A,A ajoute simplement le contenu de l'accumulateur à lui-même. Le résultat, bien sûr, est de 2 fois la donnée d'origine ce qui revient exactement à un décalage logique à gauche. Le bit de poids le plus faible du résultat est zéro, puisque $0+0=1+1=0$; $1+1$ produit aussi une retenue pour le bit suivant.
On pourrait remplacer de façon équivalente ADD A,A par SLA A, ce qui paraît un choix plus évident. Cependant, l'instruction SLA A nécessite deux octets de mémoire programme et 8 cycles d'horloge alors que ADD A,A ne nécessite qu'un octet de mémoire programme et 4 cycles d'horloge. La différence provient de ce que l'instruction SLA A a été rajoutée au jeu d'instructions original du 8080A (cf. la comparaison présentée au chapitre 3).

Masquage de bits

But : mettre les 4 bits de poids faible de la mémoire d'adresse 0040 dans les 4 bits de poids faible de la mémoire d'adresse 0041. Mettre à zéro les 4 bits de poids fort de la mémoire d'adresse 0041.

Exemple pratique :

(0040) = 3D
Résultat : (0041) = 0D

Programme source :

```
LD      A,(40H)          ; LIRE DONNEE
AND     00001111B        ; MASQUER 4 BITS POIDS FORT
LD      (41H),A          ; RANGER RESULTAT
HALT
```

Remarque : B signifie binaire dans la notation standard de l'assembleur Z80.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3A	LD	A,(40H)
0001	40		
0002	00		
0003	E6	AND	00001111B
0004	0F		
0005	32	LD	(41H),A
0006	41		
0007	00		
0008	76	HALT	

Le masque (00001111) est écrit en binaire pour rendre sa compréhension plus facile au lecteur. La notation binaire des masques est en général plus claire que la notation hexadécimale, bien que les résultats soient les mêmes. La notation hexadécimale convient pour des masques plus longs que 4 bits et doit être accompagnée de commentaires explicatifs. Lorsque l'argument du champ d'adresse est un nombre, l'instruction AND effectue l'opération logique ET entre le contenu de l'accumulateur et le contenu du mot de la mémoire qui suit immédiatement l'instruction. L'instruction AND peut servir à remettre à zéro les bits qui ne sont pas utilisés.
Les 4 bits de poids faible peuvent constituer une entrée à partir d'un interrupteur ou une sortie pour un affichage numérique.

Remise à zéro d'un emplacement mémoire

But : mettre à zéro la mémoire d'adresse 0040.

Programme source :

```
SUB     A
LD      (40H),A          ; MISE A ZERO ADRESSE 40
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)
0000	97	SUB A
0001	32	LD (40H),A
0002	40	
0003	00	
0004	76	HALT

L'instruction SUB A soustrait le contenu de l'accumulateur à lui-même, pour remettre l'accumulateur à zéro. Les instructions SUB A, XOR A ou LD A,0 peuvent toutes mettre l'accumulateur à zéro. LD A,0 prend à la fois plus de temps et de mémoire mais elle ne modifie pas l'état des indicateurs.

Déassemblage d'un octet

But : Diviser le dontenu de l'adresse 0040 en 2 sections de 4 bits et les ranger dans les mémoires d'adresse 0041 et 0042. Mettre les 4 bits de poids fort de la mémoire d'adresse 0040 dans les positions des 4 bits de poids faible de la mémoire d'adresse 0041 ; mettre les 4 bits de poids faible de la mémoire d'adresse 0040 dans les positions des 4 bits de poids faible de la mémoire d'adresse 0042. Mettre à zéro les positions des 4 bits de poids fort des mémoires d'adresse 0041 et 0042.

Exemple pratique :

```
                (0040) = 3F
Résultat :      (0041) = 03
                (0042) = 0F
```

Programme source :

```
LD      HL,40H
LD      A,(HL)          ; CHARGER DONNEE
LD      B,A
RRA                     ; DECALAGE DROITE 4 FOIS
RRA
RRA
RRA
AND     00001111B       ; RAZ 4 BITS POIDS FAIBLE
INC     HL
LD      (HL),A          ; RANGER 4 BITS POIDS FORT
LD      A,B             ; SAUVEGARDE DONNEE ORIGINE
AND     00001111B       ; RAZ 4 BITS POIDS FORT
INC     HL
LD      (HL),A          ; RANGER 4 BITS POIDS FAIBLE
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	7E	LD	A,(HL)
0004	47	LD	B,A
0005	1F	RRA	
0006	1F	RRA	
0007	1F	RRA	
0008	1F	RRA	
0009	E6	AND	00001111B
000A	0F		
000B	23	INC	HL
000C	77	LD	(HL),A
000D	78	LD	A,B
000E	E6	AND	00001111B
000F	0F		
0010	23	INC	HL
0011	77	LD	(HL),A
0012	76	HALT	

Les instructions qui utilisent l'adresse des registres H et L prennent seulement un octet de mémoire programme. Toutefois, HL doit être chargé avant que l'adresse soit utilisée. Ainsi, l'adressage implicite permet de gagner du temps et de la mémoire si on le compare à l'adressage direct et ceci seulement lorsque le programme utilise la même adresse plusieurs fois ou bien des adresses consécutives.
L'instruction RRA effectue un décalage circulaire à droite d'un bit sur l'accumulateur, le bit le moins significatif allant à la position du bit le plus significatif et dans l'indicateur de retenue (Carry). Le décalage de l'accumulateur à droite 4 fois nécessite 4 instructions RRC. On aurait pu utiliser SRL A qui effectue directement un décalage logique (et l'instruction finale AND serait devenue inutile). Toutefois, l'instruction SRL A demande deux fois plus de temps et de mémoire qu'une instruction RRC. Faites-en l'essai vous-même en substituant une instruction SRL A à une RRC.
Une autre solution consiste à utiliser l'instruction RLD qui effectue à la fois le masquage et le rangement en mémoire. Cependant, cette solution n'est optimale ni pour le rangement en mémoire, ni pour la vitesse d'exécution et ceci du fait que le demi-octet de plus haut rang de chaque résultat doit valoir zéro.
Beaucoup d'instructions du Z80 modifient des paires de registres 8 bits. Les paires sont HL (H et L), DE (D et E) et BC (B et C). Les registres B, D et H constituent les 8 bits les plus significatifs des paires. Les registres C, E et L constituent les 8 bits les moins significatifs. Les instructions usuelles concernant les paires de registres sont :

— LD rp (chargement d'une paire de registres)
— INC rp (incrémentation d'une paire de registres)
— DEC rp (décrémentation d'une paire de registres)
— ADD HL,rp (addition d'une paire de registres à H et L)

Recherche du plus grand de deux nombres

But : mettre le plus grand contenu des 2 adresses 0040 et 0041 à l'adresse 0042. On considère que les adresses 0040 et 0041 contiennent des nombres binaires non signés.

Exemple pratique :

a. (0040) = 3F
(0041) = 2B
Résultat : (0042) = 3F

b. (0040) = 75
(0041) = A8
Résultat : (0042) = A8

Programme source :

```
        LD      HL,40H
        LD      A,(HL)      ; LIRE 1er OPERANDE
        INC     HL
        CP      (HL)        ; 2e OPERANDE + GRAND ?
        JR      C,FINI
        LD      A,(HL)      ; OUI, CHARGER 2e OPERANDE A LA PLACE
FINI :  INC     HL
        LD      (HL),A      ; STOCKER + GRAND OPERANDE
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	7E	LD	A,(HL)
0004	23	INC	HL
0005	BE	CP	(HL)
0006	30	JR	NC,FINI
0007	01		
0008	7E	LD	A,(HL)
0009	23	FINI : INC	HL
000A	77	LD	(HL),A
000B	76	HALT	

L'instruction CP (HL) positionne les indicateurs comme si le contenu de la mémoire adressée par H et L était soustrait du contenu de l'accumulateur. Cependant, le contenu de l'accumulateur reste inchangé pour des comparaisons ultérieures ou d'autres traitements.
Si A est le contenu de l'accumulateur et X le second opérande d'une instruction CP, les indicateurs sont positionnés comme suit :
1) Zéro = 1 si A = X
Zéro = 0 si A ≠ X

2) Carry = 1 si A<X
Carry = 0 si A≥X
(A et X sont des nombres binaires non signés.)
CP met l'indicateur de retenue (Carry) à 1 si une retenue est nécessaire pour effectuer la soustraction, c'est-à-dire si le nombre soustrait du contenu de l'accumulateur est plus grand que ce contenu. Ainsi, la séquence CP, JR NC, FINI provoque un saut à FINI si le contenu de l'accumulateur est plus grand ou égal à l'autre nombre.
L'instruction JR NC,FINI provoque un saut à l'adresse FINI si l'indicateur de retenue = 0. Sinon (si Carry = 1) le calculateur continue en séquence après l'instruction JR.
FINI est une étiquette, un nom donné à un emplacement mémoire pour se le rappeler facilement. Remarque : les étiquettes sont suivies du caractère ':' sur la ligne où elles sont définies.
Une étiquette rend plus claire la destination d'un branchement, surtout lorsque l'adressage relatif est utilisé. L'assembleur calcule le déplacement qui en découle (attention : quelques assembleurs Z80 ne le font pas !). Il est préférable d'utiliser une étiquette plutôt que de se contenter de spécifier le déplacement (comme JR NC,$ + 3) pusique les instructions Z80 varient en longueur ; aussi une erreur dans le calcul du déplacement est-elle très courante.
Si les conditions du branchement ne sont pas satisfaites, le processeur traite en séquence l'adresse suivante (c'est-à-dire, il exécute l'instruction LD A,(HL).
Les assembleurs Z80 autorisent généralement 6 caractères pour les étiquettes. Le premier caractère doit être une lettre, les autres sont indifféremment des lettres ou des chiffres (quelques caractères spéciaux sont tolérés mais nous ne les utiliserons pas).
L'instruction JR utilise un adressage relatif dans lequel le second octet de l'instruction est un nombre 8 bits complémenté à deux ; l'UCT ajoute ce nombre à l'adresse de l'instruction suivante pour trouver l'adresse finale. Dans l'exemple, le déplacement relatif est 0009 (adresse finale) moins 0008 (adresse qui suit immédiatement le branchement) soit 01.
On peut noter que certains assembleurs Z80 ne calculent pas le déplacement de cette façon. Ces assembleurs utilisent un déplacement dans le champ d'adresse et non l'étiquette de l'instruction finale. Si vous disposez d'un tel assembleur, utilisez la forme JR NC,FINI – $. Rappelez-vous que $ signifie « l'adresse de l'instruction en cours ».
Le Z80 dispose de deux jeux d'instructions de saut, JP (saut) et JR (saut relatif). Les instructions JP nécessitent une adresse mémoire complète ; elles occupent trois octets de mémoire et s'exécutent en dix cycles d'horloge. Les instructions JR utilisent seulement un déplacement de un octet ; elles occupent deux octets de mémoire et s'exécutent soit en 12 cycles, si un saut s'effectue réellement, soit en 7 sinon. Les instructions JR utilisent donc moins de place mémoire que les instructions JP mais peuvent prendre un petit peu plus de temps si un saut est effectué (ce temps supplémentaire sert à exécuter l'addition 16 bits entre le compteur ordinal et le déplacement).

Addition sur 16 bits

But : additionner un nombre 16 bits contenu dans les mémoires d'adresses 0040 et 0041 à un nombre 16 bits contenu dans les mémoires à adresses 0042 et 0043. Les huit bits de poids fort se trouvent dans les mémoires d'adresses 0041 et 0043. Ranger le résultat dans les mémoires d'adresses 0044 et 0045 avec les huit bits de poids fort dans la mémoire 0045.

Exemple pratique :

(0040) = 2A
(0041) = 67
(0042) = F8
(0043) = 14

Résultat : 672A + 14F8 = 7C22
(0044) = 22
(0045) = 7C

Programme source :

```
LD      HL,(40H)     ; CHARGER 1er NOMBRE 16 BITS
LD      DE,(42H)     ; CHARGER 2e NOMBRE 16 BITS
ADD     HL,DE        ; ADDITION 16 BITS
LD      (44H),HL     ; RANGER RESULTAT 16 BITS
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	2A	LD	HL,(40H)
0001	40		
0002	00		
0003	ED	LD	DE,(42H)
0004	5B		
0005	42		
0006	00		
0007	19	ADD	HL,DE
0008	22	LD	(44H),HL
0009	44		
000A	00		
000B	76	HALT	

L'instruction LD HL,(addr) charge les registres H et L à partir de deux adresses, la première spécifiée dans l'instruction, la seconde suivant immédiatement la première. Le contenu de la première adresse va dans le registre L. Le contenu de la seconde adresse va dans le registre H. Ainsi l'instruction LD HL,(40H) signifie L = (40), H = (41).
Le transfert effectif traite un octet à la fois et prend 16 cycles d'horloge. L'avantage que présente l'instruction de chargement 16 bits sur l'instruction de chargement 8 bits est de permettre à l'UCT de ne chercher en mémoire qu'une seule instruction.
Ne pas confondre l'instruction LD HL,(addr) qui charge le contenu de deux mémoires localisées en addr et addr + 1 dans les registres H et L avec l'instruction LD HL,data16, qui charge dans H et L le contenu des deux octets sur lesquels pointe le compteur d'instruction. Puisque ces deux octets suivent immédiatement le code opératoire, des chargements de ce type sont appelés chargements immédiats.
L'instruction LD DE,(addr) est semblable à LD HL,(addr) mais elle prend un mot de mémoire supplémentaire ainsi que 4 cycles d'horloge de plus. C'est l'une des instructions qui figure dans le jeu du Z80 mais non dans ceux du 8080/8085. Voici une autre approche :

```
EX      DE,HL        ; SAUVEGARDE 1er NOMBRE 16 BITS EN DE
LD      HL,(42H)     ; CHARGEMENT 2e NOMBRE 16 BITS
```

EX DE,HL échange le contenu du registre D et E avec celui des registres H et L. Aucun nombre n'est modifié ou détruit. L'avantage de l'instruction EX DE,HL devient évident lorsqu'on essaye de le remplacer par une série d'instructions LD.
L'instruction ADD HL,DE additionne le nombre 16 bits des registres D et E au nombre 16 bits des registres H et L. Le résultat est mis dans les registres H et L. L'instruction ADD HL,DE effectue l'addition d'un octet à la fois ; elle s'exécute en 11 cycles d'horloge.
L'instruction LD (addr),HL range le contenu des registres H et L dans deux adresses, l'une spécifiée dans l'instruction et l'autre la suivant immédiatement. Le contenu du registre L va à l'adresse spécifiée et celui de H à l'adresse suivante. Ainsi l'instruction LD (44H),HL signifie (44) = L, (45) = H. Comme pour LD HL,(addr) le transfert effectif traite un octet à la fois et prend 16 cycles d'horloge.
Bien que le Z80 soit un processeur 8 bits, il comporte des instructions qui traitent des nombres 16 bits. Ces instructions sont surtout destinées à gérer des adresses mais il est aussi possible de les utiliser pour des données 16 bits. Voici la liste principale de ces instructions ainsi que de leurs utilisations :

1) ADD HL,rp — Addition 16 bits
utilisée pour l'accès aux tables et l'addition de données 16 bits.
2) DEC rp — Décrémentation 16 bits
utilisée pour retrancher 1 au contenu d'une paire de registres.
3) INC rp — Incrémentation 16 bits
utilisée pour ajouter 1 au contenu d'une paire de registres.
4) LD rp,data16 — Chargement immédiat 16 bits
utilisée pour initialiser une paire de registres à une valeur fixe, par exemple l'adresse de début d'un tableau ou d'une table.
5) LD HL,addr — Chargement 16 bits du registre HL
utilisée pour mettre des adresses variables dans le registre principal d'adresse (H et L).
6) LD addr,HL — Rangement 16 bits à partir du registre HL
utilisée pour ranger des adresses en mémoire à partir du registre principal d'adresse (H et L).

Table des carrés

But : calculer le carré du contenu de l'adresse 0041 à partir d'une table et mettre le résultat à l'adresse 0042. Prendre un nombre compris entre 0 et 7 inclusivement comme contenu de l'adresse 0041 ($0 \leq (0041) \leq 7$).
La table occupe les adresses 0050 à 0057.

Adresse mémoire (Hexa)	Entrée	
	(hexa)	(décimal)
0050	00	0 (0^2)
0051	01	1 (1^2)
0052	04	4 (2^2)
0053	09	9 (3^2)
0054	10	16 (4^2)
0055	19	25 (5^2)
0056	24	36 (6^2)
0057	31	49 (7^2)

Exemple pratique :

a. (0041) = 03
Résultat : (0042) = 09
b. (0041) = 06
Résultat : (0042) = 24

Programme source :

```
        LD      A,(40H)     ; CHARGER DONNEE
        LD      L,A         ; LA TRANSFORMER EN INDEX 16 BITS
        LD      H,0
        LD      DE,CARRE    ; CHARGER ADRESSE DEBUT TABLE
        ADD     HL,DE       ; INDEXER TABLE PAR DONNEE
        LD      A,(HL)      ; CHARGER CARRE DE LA DONNEE
        LD      (41H),A
        HALT
        ORG     50H         ; TABLE DES CARRES
CARRE : DEFB    0
        DEFB    1
        DEFB    4
        DEFB    9
        DEFB    16
        DEFB    25
        DEFB    36
        DEFB    49
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3A	LD	A,(40H)
0001	40		
0002	00		
0003	6F	LD	L,A
0004	26	LD	H,0
0005	00		
0006	11	LD	DE, CARRE
0007	50		
0008	00		
0009	19	ADD	HL,DE
000A	7E	LD	A,(HL)
000B	32	LD	(41H),A
000C	41		
000D	00		
000E	76		
0050	00	CARRE : DEFB	0
0051	01	DEFB	1
0052	04	DEFB	4
0053	09	DEFB	9
0054	10	DEFB	16
0055	19	DEFB	25
0056	24	DEFB	36
0057	31	DEFB	49

Il est aussi possible de mettre en mémoire la table des carrés (à l'aide de la pseudo-opération DEFB de l'assembleur). La table des carrés est composée de constantes et non de paramètres variables ; c'est pourquoi l'on peut initialiser la table en utilisant la pseudo-opération DEFB plutôt qu'en exécutant des instructions qui chargent des valeurs dans la table. Il ne faut pas oublier que la table est une partie de la mémoire programme (ROM dans la plupart des systèmes).

LD L,A transfère la donnée de l'accumulateur vers le registre L. Cette donnée constitue les 8 bits de poids faible de l'index. Il n'est pas toujours possible de considérer que la donnée du programme appartient au bon intervalle. Une bonne habitude consiste à éliminer au préalable les valeurs critiques. Pour cela, il suffit de tester une valeur pour être sûr qu'elle est comprise entre les limites inférieure et supérieure. N'importe quel octet peut avoir une valeur comprise entre 0 et 255. Si la valeur rangée dans l'octet à l'adresse 0040H est plus grande que 7, le programme se reporte à un octet indéfini, en dehors de la table des carrés et donne ainsi des résultats erronés. La vérification de l'intervalle évite de telles erreurs.

L'instruction LD H,0 remet à zéro le registre H pour éviter une perturbation avec l'addition 16 bits de l'adresse de départ et de l'index. Il ne faut jamais considérer qu'un registre est à zéro au début d'un programme.

L'instruction LD DE,CARRE charge l'adresse de départ de la table dans les registres D et E. On utilise D et E comme adresse de départ puisque l'instruction ADD HL ne modifie pas D et E. Ainsi, on trouvera encore l'adresse de départ de la table en D et E après l'addition dans le cas où l'on veut un autre élément de la table.

L'instruction ADD HL,DE additionne adresse de départ et index. Le résultat qui se trouve dans H et L constitue ainsi l'adresse de l'entrée correcte. L'instruction LD A,(HL) transfère ensuite cette entrée dans l'accumulateur.

Il n'est pas toujours possible de résoudre un problème arithmétique en quelques instructions alors que souvent l'usage de tables de correspondance le permet très bien. Ces tables contiennent toutes les solutions possibles du problème ; elles sont organisées de façon à ce que la solution soit trouvée facilement. Le problème d'arithmétique devient alors un problème d'accès. Comment trouver la solution correcte à partir de la table ? Il faut connaître deux choses : le rang de la solution dans la table (appelé index) et l'adresse de base (ou de départ) de la table. L'adresse de la solution est alors l'adresse de base plus l'index.

L'adresse de base est bien sûr un nombre fixe pour une table particulière. Comment peut-on déterminer l'index ? Dans des cas simples où un seul élément de donnée rentre en jeu, on peut arranger la table de façon à ce que la donnée soit l'index. Dans la table des carrés, l'entrée 0 de la table contient 0 au carré, la première entrée 1 au carré, etc.

Dans des cas plus complexes comme pour une grande quantité de valeurs d'entrée ou lorsque plusieurs éléments de donnée interviennent (par exemple racines d'un carré, nombre de permutations), il faut utiliser des méthodes plus compliquées pour déterminer les index.

Le compromis de base lorsqu'on utilise une table est un compromis temps/espace mémoire. Le traitement par les tables est plus rapide, puisqu'il n'y a pas de calcul et plus simple puisque l'on évite d'utiliser des méthodes mathématiques. Cependant, les tables peuvent occuper beaucoup de mémoire s'il y a une grande quantité de données en entrée. Il est souvent possible de réduire la taille de la table en limitant la précision des résultats, en réduisant les données en entrée ou en organisant astucieusement la table. Des tables sont souvent utilisées pour calculer des fonctions transcendantales et trigonométriques, linéariser des entrées, convertir des codes et réaliser d'autres applications mathématiques.

Complémentation à un sur 16 bits

But : mettre dans les mémoires d'adresse 0042 et 0043 le complément à un d'un nombre 16 bits rangé dans les mémoires d'adresse 0040 et 0041. Les bits de poids fort se trouvent dans les mémoires d'adresses 0041 et 0043.

Exemple pratique :

(0040) = 67
(0041) = E2

Résultat : (0042) = 98
(0043) = 1D

La complémentation à un inverse chaque bit du nombre original ; la somme du nombre et de son comportement à un ne doit donner que des bits à 1.

Programme source :

```
LD      HL,(40H)     ; LIRE DONNEE
LD      A,L          ; COMPLEMENT 8 BITS POIDS FAIBLE
CPL
LD      L,A
LD      A,H          ; COMPLEMENT 8 BITS POIDS FORT
CPL
LD      H,A
LD      (40H),HL     ; RANGER COMPLEMENT A UN
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	2A	LD	HL,(40H)
0001	40		
0002	00		
0003	7D	LD	A,L
0004	2F	CPL	
0005	6F	LD	L,A
0006	7C	LD	A,H
0007	2F	CPL	
0008	67	LD	H,A
0009	22	LD	(42H),HL
000A	42		
000B	00		
000C	76	HALT	

Bien que le Z80 comporte des instructions 16 bits, il est préférable d'utiliser des instructions 8 bits pour réaliser la plupart des opérations arithmétiques et logiques. Les instructions 16 bits peuvent toutefois servir à charger et ranger des données et parfois effectuer quelques opérations arithmétiques 16 bits telles que l'addition, la soustraction, l'incrémentation et la décrémentation. Vous apprendrez vite que la panoplie des instructions 16 bits du Z80 est loin d'être complète et que l'utilisation de ces instructions pour traiter des données 16 bits peut vous entraîner dans des problèmes délicats.

EXERCICES

1) Complémentation à deux

But : mettre le complément à 2 du contenu de l'adresse 0040 dans la mémoire d'adresse 0041. Le complément à 2 est égal au complément à un plus un.

Exemple pratique :

(0040) = 3E
Résultat : (0041) = C2

La somme du nombre original et de son complément à deux est nulle. (Essayez le cas de l'exemple.)

2) Soustraction 8 bits

But : soustraire le contenu de l'adresse 0041 au contenu de l'adresse 0040. Mettre le résultat à l'adresse 0042.

Exemple pratique :

(0040) = 77
(0041) = 39
Résultat : (0042) = 3E

3) Décalage à gauche de deux bits

But : décaler à gauche de 2 bits le contenu de l'adresse 0040 et mettre le résultat à l'adresse 0041. Remettre à zéro les 2 positions des bits de poids faible de la mémoire d'adresse 0041.

Exemple pratique :

(0040) = 5D
Résultat : (0041) = 74

4) Masquage des quatre bits de poids faible

But : mettre les 4 bits de poids fort de la mémoire d'adresse 0040 à l'adresse 0041. Mettre à zéro les positions des 4 bits de poids faible de la mémoire d'adresse 0041.

Exemple pratique :

(0040) = C4
Résultat : (0041) = C0

5) Mise à un d'un emplacement mémoire

But : mettre à 1 toutes les positions des bits de la mémoire d'adresse 0040 (FF en hexa).

6) Assemblage d'un octet

But : combiner les 4 bits de poids faible des mémoires d'adresses 0040 et 0041 dans un octet et le ranger à la mémoire d'adresse 0042. Mettre les 4 bits de poids faible de la mémoire d'adresse 0040 aux positions des 4 bits de poids fort de la mémoire d'adresse 0042 ; mettre les 4 bits de poids faible de la mémoire d'adresse 0041 aux positions des 4 bits de poids faible de la mémoire d'adresse 0042.

Exemple pratique :

(0040) = 6A
(0041) = B3
Résultat : (0042) = A3

7) Recherche du plus petit de deux nombres

But : mettre le plus petit contenu de 2 adresses 0040 et 0041 à l'adresse 0042. Prendre des nombres binaires non signés comme contenus des adresses 0040 et 0041.

Exemple pratique :

a. (0040) = 3F
(0041) = 2B
Résultat : (0042) = 2B

b. (0040) = 75
(0041) = A8
Résultat : (0042) = 75

8) Addition sur 24 bits

But : additionner un nombre 24 bits contenu aux adresses 0040, 0041 et 0042 à un nombre 24 bits contenu aux adresses 0043, 0044 et 0045. Les 8 bits de poids fort se trouvent aux mémoires d'adresses 0042 et 0045 ; les 8 bits de poids faible aux mémoires d'adresses 0040 et 0043. Ranger le résultat en mémoire aux adresses 0046, 0047 et 0048 avec les 8 bits de poids fort en 0048 et les 8 bits de poids faible en 0046.

Exemple pratique :

(0040) = 2A
(0041) = 67
(0042) = 35
(0043) = F8
(0044) = A4
(0045) = 51

Résultat : (0046) = 22
(0047) = 0C
(0048) = 87

c'est-à-dire : 35672A
+51A4F8
870C22

9) Somme de carrés

But : calculer les carrés des contenus des adresses 0040 et 0041 et les additionner ensemble. Mettre le résultat en 0042. Prendre comme contenus des adresses 0040 et 0041 des nombres compris inclusivement entre 0 et 7 ($0 \leq (0040) \leq 7$) et ($0 \leq (0041) \leq 7$). Utiliser la table des carrés donnée dans l'exemple : table des carrés.

Exemple pratique :

(0040) = 03
(0041) = 06

Résultat : (0042) = 2D

c'est-à-dire : $3^2 + 6^2 = 9 + 36 = 45$ (décimal)
= 2D (hexa)

10) Complémentation à deux sur 16 bits

But : mettre aux adresses 0042 et 0043 (bits de poids fort en 0043) le complément à deux d'un nombre 16 bits contenu aux adresses 0040 et 0041 (bits de poids fort en 0041).

Exemple pratique :

a.
(0040) = 00
(0041) = 58

Résultat : (0042) = 00
(0043) = A8

b.
(0040) = 72
(0041) = 00

Résultat : (0042) = 8E
(0043) = FF

CHAPITRE 5

LES BOUCLES SIMPLES

Une boucle est la structure fondamentale qui force l'unité centrale à répéter une séquence d'instructions. Une boucle est composée de 4 sections :

1) L'INITIALISATION : qui fixe les valeurs initiales des compteurs, des registres d'adresse et d'autres variables.
2) L'EXECUTION : où s'effectue la manipulation des données. C'est cette section qui fait le travail.
3) LE CONTROLE : qui met à jour les compteurs et les pointeurs pour l'itération suivante.
4) LA CONCLUSION : qui analyse et stocke les résultats.

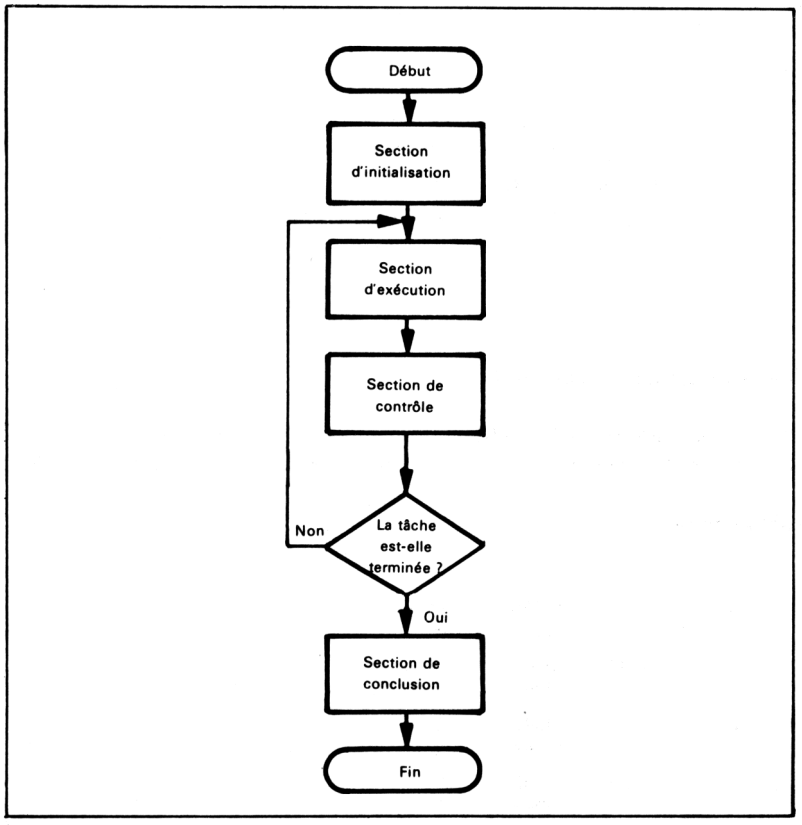


Fig. 5.1. — Organigramme d'une boucle.

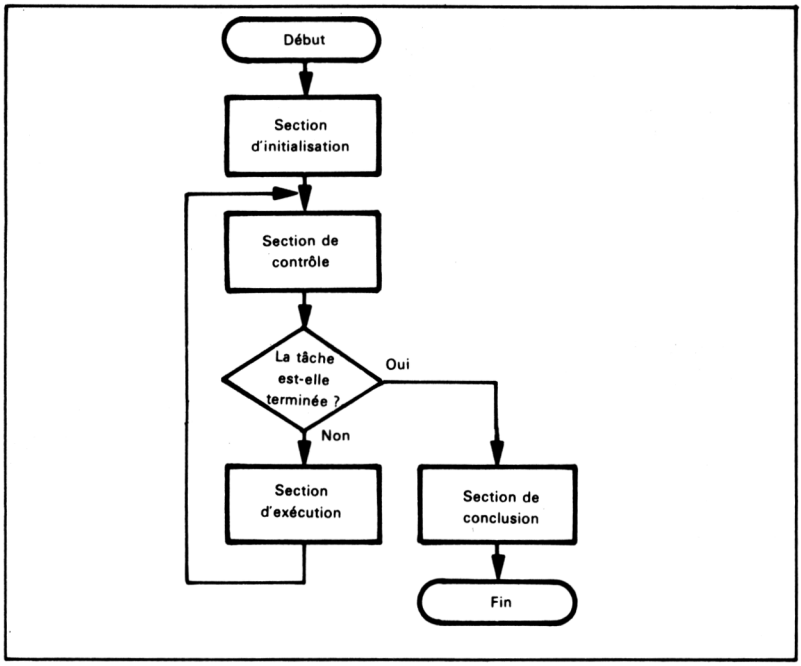


Fig. 5.2. — Boucle autorisant un nombre d'itérations nul.

Les sections 1 et 4 ne sont exécutées qu'une seule fois alors que les sections 2 et 3 peuvent être exécutées plusieurs fois. Le temps d'exécution d'une boucle dépend donc principalement du temps d'exécution des sections 2 et 3. En conséquence, il faut s'efforcer de réduire au maximum le temps d'exécution des sections 2 et 3 tandis que l'on peut négliger celui des sections 1 et 4.

L'organigramme d'une boucle classique est représenté en figure 5.1 comme en figure 5.2 où l'ordre des sections d'exécution et de contrôle est inversé.

En figure 5.1, la section d'exécution est toujours exécutée au moins une fois tandis qu'en figure 5.2, cette section peut ne pas être exécutée du tout. La figure 5.1 semble plus naturelle mais la figure 5.2 est souvent plus performante et évite le problème d'absence de données (hantise des calculateurs et cause fréquente de situations incohérentes comme l'édition d'un chèque de 0,00 F).

La structure de boucle peut être utilisée pour traiter des blocs entiers de données. Pour cela, le programme doit incrémenter un registre d'adresse (généralement le double registre HL) après chaque itération de façon à ce que le registre d'adresse pointe sur l'élément suivant du bloc de données. L'itération suivante exécutera alors les mêmes opération sur la donnée qui suit en mémoire.

L'ordinateur peut manipuler des blocs de n'importe quelle longueur avec le même ensemble d'instructions.

L'adressage implicite à travers des paires de registres (en particulier HL) est le moyen clé pour traiter un bloc de données sur le Z80, car il autorise le changement des adresses mémoires réelles par la modification du contenu des registres.
L'adressage indexé, quoique plus long et plus lent que l'adressage implicite sur le Z80 peut être pratique pour traiter plus d'un bloc de données. Il est à noter que pour les modes d'adressage direct et immédiat, les adresses utilisées sont entièrement déterminées par l'instruction en cours. (Elles sont donc fixes si l'on accède à la mémoire programme en lecture uniquement.)

Exemples

Somme de données

SOMMATION SUR 8 BITS

But : calculer la somme d'une série de nombres. La longueur de la série est à l'emplacement mémoire 0041 et la série commence à l'emplacement mémoire 0042. Ranger la somme à l'emplacement mémoire 0040. Considérer que la somme est un nombre de 8 bits de façon à ignorer les retenues.

Exemple pratique :

(0041) = 03
(0042) = 28
(0043) = 55
(0044) = 26

Résultat : (0040) = (0042) + (0043) + (0044)
= 28+55+26
= A3

Il y a 3 nombres dans la somme, puisque (0041) = 03.

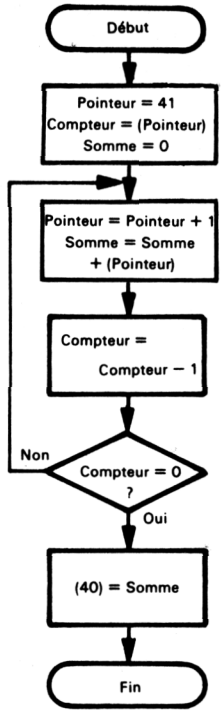
Organigramme : (voir page 235)

Programme source :

```
        LD      HL,41H
        LD      B,(HL)      ; COMPTEUR = LONGUEUR D'UNE SERIE
        SUB     A           ; SOMME = ZERO
SOMD:   INC     HL
        ADD     A,(HL)      ; SOMME = SOMME + DONNEE
        DEC     B
        JR      NZ,SOMD
        LD      (40H),A     ; RANGER SOMME
        HALT
```

La section d'initialisation de ce programme correspond aux trois premières instructions qui initialisent la somme, le compteur et le pointeur de donnée à leur valeur de départ. On peut utiliser LD pour transférer des données entre la mémoire et n'importe quel registre primaire d'utilisation générale (c'est-à-dire A, B, C, D, E, H, L) en utilisant l'adresse dans les registres H et L. Cependant, en adressage direct, seuls les transferts en provenance ou à destination de l'accumulateur sont autorisés. (C'est-à-dire LD A,(adresse) et LD (adresse),A — il n'y a pas d'instruction LD E,(adresse), par exemple.)

Organigramme



Remarque : (Pointeur) représente le contenu de l'emplacement mémoire adressé par pointeur. Sur le Z80, Pointeur est une adresse de 16 bits alors que (Pointeur) est un octet de 8 bits de donnée.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,41H
0001	41		
0002	00		
0003	46	LD	B,(HL)
0004	97	SUB	A
0005	23	SOMD : INC	HL
0006	86	ADD	A,(HL)
0007	05	DEC	B
0008	20	JR	NZ,SOMD
0009	FB		
000A	32	LD	(40H),A
000B	40		
000C	00		
000D	76	HALT	

La section d'exécution de ce programme se résume à la seule instruction ADD A,(HL) qui ajoute le contenu de la mémoire adressée par les registres H et L au contenu de l'accumulateur et range le résultat dans l'accumulateur. Cette instruction effectue le travail principal de ce programme.
La section de contrôle du programme comprend les instructions INC HL et DEC B.

INC HL met à jour le pointeur pour que l'itération suivante ajoute le nombre suivant à la somme.

DEC B décrémente le compteur qui contient le nombre d'itérations à effectuer.
L'instruction JR NZ déclenche un branchement si l'indicateur de zéro vaut zéro. Le déplacement est représenté par un nombre complémenté à 2 et son calcul commence à l'adresse qui suit immédiatement l'instruction JR. Dans ce cas, le saut nécessaire va de l'adresse 000A à l'adresse 0005.
Le déplacement est donc de :

$$\begin{array}{rcr} 0005 & = & 05 \\ -000A & & +F6 \\ \hline & & FB \end{array}$$

Si l'indicateur de zéro vaut 1, l'UC exécute l'instruction suivante (c'est-à-dire LD (40H),A).
Puisque DECB est la dernière instruction avant JR à modifier l'indicateur de zéro, JR NZ,SOMD déclenche un saut à SOMD si DECB n'a pas zéro pour résultat, c'est-à-dire :

$$PC = \begin{cases} \text{SOMD si } B \neq 0 \\ PC+2 \text{ si } B = 0 \end{cases}$$

(2 car JR est une instruction 2 octets.)

La séquence de contrôle DEC suivie par JR NZ revient tellement souvent que le Z80 a une instruction spéciale qui à la fois décrémente le compteur et réalise le saut.
Cette instruction est DJNZ : décrémentation et saut si différent de zéro, ce qui décrémente le registre B et effectue ensuite le saut de la valeur du déplacement relatif spécifié si le résultat dans le registre B est non nul.
La fin de l'exemple précédent peut donc être remplacée par :

```
DJNZ    SOMD
LD      (40H),A
HALT
```

ce qui donne sous la forme objet :

```
07      10      DJNZ    SOMD
08      FC
09      32      LD      (40H),A
0A      40
0B      00
0C      76      HALT
```

Cette modification fait gagner un octet de mémoire et 3 cycles d'horloge. Cependant, cela oblige à utiliser le registre B comme compteur puisque c'est le registre que l'instruction DJNZ décrémente.
Etant donné qu'un déplacement pour les sauts relatifs sur Z80 ne peut être codifé que sur un octet, de tels sauts ne peuvent dépasser 127 adresses en avant ou 128 en arrière (en fait, 129 en avant ou 126 en arrière, puisque le calcul commence à la fin de l'instruction qui fait 2 octets de long). Les sauts plus longs devront utiliser les instructions JP.
Les boucles de la plupart des calculateurs comptent en décrémentant plutôt qu'en incrémentant afin que l'indicateur de zéro puisse servir de condition de sortie. Rappelons que l'indicateur de zéro est positionné à 1 si le résultat vaut zéro et il est positionné à zéro si le résultat est différent de zéro. Essayer de ré-écrire le programme en incrémentant au lieu de décrémenter le compteur. Quelle méthode est la plus performante ?
L'ordre des instructions est souvent très important. L'instruction DECB doit immédiatement précéder JRNZ SOMD car sinon le résultat zéro donné par DECB pourrait être modifié par une autre instruction. L'instruction INC HL doit précéder ADD A,(HL) sinon le premier nombre ajouté à la somme serait le contenu de l'adresse 0041 au lieu du contenu de l'adresse 0042.

Somme de données sur 16 bits

But : calculer la somme d'une série de nombres. La longueur de la série est contenue à l'adresse 0042 et la série elle-même commence à l'adresse 0043. Ranger la somme dans les adresses 0040 et 0041 (les 8 bits de poids le plus faible en 0040).

Exemple pratique :

(0042) = 03
(0043) = C8
(0044) = FA
(0045) = 96

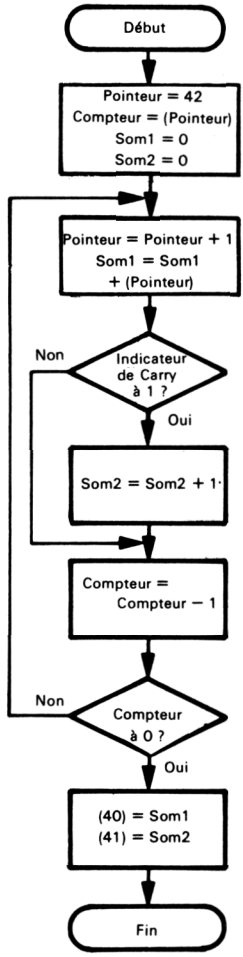
Résultat : C8 + FA + 96 = 0258
(0040) = 58
(0041) = 02

Organigramme : (voir fig. A page 238)

Programme source : (voir fig. B page 238)

La structure de ce programme est la même que celle du programme précédent. Les bits les plus significatifs de la somme doivent maintenant être initialisés et rangés en mémoire. La section d'exécution consiste en trois instructions : (ADD A,(HL) ; JR NC,TEST ; et INC C), comprenant un saut conditionnel.
L'instruction JR NC,TEST provoque un saut à l'adresse TEST si la retenue est nulle (Carry = Ø).

Organigramme :



Programme source :

```
        LD      HL,42H
        LD      B,(HL)      ; COMPTEUR = LONGUEUR DE LA SERIE
        SUB     A           ; BITS POIDS FAIBLE DE SOMME = 0
        LD      C,A         ; BITS POIDS FORT DE SOMME = 0
SOMD :  INC     HL
        ADD     A,(HL)      ; SOMME = SOMME + DONNEE
        JR      NC,TEST
        INC     C           ; AJOUTER RETENUE AUX BITS POIDS FORT DE SOMME
TEST :  DJNZ    SOMD
        LD      HL,40H
        LD      (HL),A      ; RANGER BITS POIDS FAIBLE DE SOMME
        INC     HL
        LD      (HL),C      ; RANGER BITS POIDS FORT DE SOMME
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,42H
0001	42		
0002	00		
0003	46	LD	B,(HL)
0004	97	SUB	A
0005	4F	LD	C,A
0006	23	SOMD : INC	HL
0007	86	ADD	A,(HL)
0008	30	JR	NC,TEST
0009	01		
000A	0C	INC	C
000B	10	TEST : DJNZ	SOMD
000C	F9		
000D	21	LD	HL,40H
000E	40		
000F	00		
0010	77	LD	(HL),A
0011	23	INC	HL
0012	71	LD	(HL),C
0013	76	HALT	

Donc, s'il n'y a pas de retenue lors de l'addition 8 bits, le programme saute l'instruction qui incrémente les bits les plus significatifs de la somme. Le déplacement relatif est :

$$\begin{array}{r} 000B \\ -000A \\ \hline 01 \end{array}$$

Le déplacement relatif pour l'instruction DJNZ,SOMD est :

$$\begin{array}{rcr} 0006 & = & 06 \\ -000D & & +F3 \\ & & \hline & & F9 \end{array}$$

L'instruction INC C ajoute 1 au contenu du registre C.
Remarque : l'instruction INC BC est une incrémentation sur 16 bits qui ajoute 1 au registre C et ajoute la retenue qui en résulte au registre B. L'instruction INC C est une incrémentation sur 8 bits qui ne tient pas compte de la retenue.

Nombre d'éléments négatifs

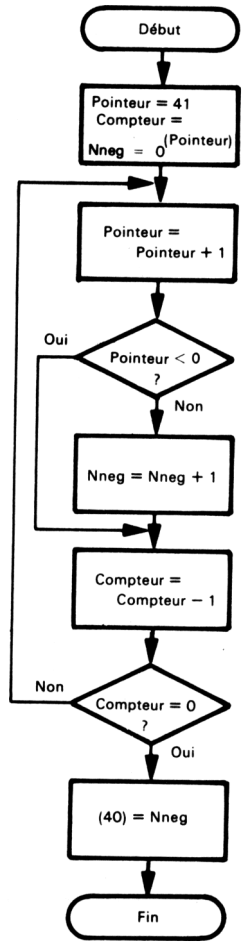
But : déterminer le nombre d'éléments négatifs (bit le plus significatif à 1) dans un bloc. La longueur du bloc est contenue dans la mémoire d'adresse 0041 et le bloc lui-même commence à l'adresse 0042. Mettre à l'adresse 0040 le nombre d'éléments négatifs.

Exemple pratique :

(0041) = 06
(0042) = 68
(0043) = F2
(0044) = 87
(0045) = 30
(0046) = 59
(0047) = 2A

Résultat : (0040) = 02, puisque 0042 et 0043 contiennent des nombres dont le bit le plus significatif est à 1.

Organigramme :



Programme source :

```
        LD      HL,41H
        LD      B,(HL)      ; COMPTEUR = NOMBRE D'ELEMENTS
        LD      C,0         ; NOMBRE DE NEGATIFS = 0
BOUCLE: INC     HL
        LD      A,(HL)      ; LIRE ELEMENT SUIVANT
        AND     A           ; BIT DE POIDS FORT = 0 ?
        JP      P,TEST
        INC     C           ; NON, AJOUTER 1 AU NOMBRE DE NEGATIFS
TEST:   DJNZ    BOUCLE
        LD      A,C         ; RANGER NOMBRE DE NEGATIFS
        LD      (40H),A
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,41H
0001	41		
0002	00		
0003	46	LD	B,(HL)
0004	0E	LD	C,0
0005	00		
0006	23	BOUCLE : INC	HL
0007	7E	LD	A,(HL)
0008	A7	AND	A
0009	F2	JP	P,TEST
000A	0D		
000B	00		
000C	0C	INC	C
000D	10	TEST : DJNZ	BOUCLE
000E	F7		
000F	79	LD	A,C
0010	32	LD	(40H),A
0011	40		
0012	00		
0013	76	HALT	

AND A positionne simplement les indicateurs en fonction du contenu de l'accumulateur, sans affecter ce contenu.
OR A a le même effet. Ces instructions sont nécessaires car en chargeant simplement l'accumulateur, les indicateurs ne sont pas affectés.
JP P,TEST nécessite une adresse complète sur 16 bits. Il n'y a pas de saut relatif conditionné par l'indicateur de signe (S) comme pour les indicateurs zéro et de retenue.
Remarquez que notre but est de tester la valeur du bit 7 de la location mémoire adressée par les registres H et L. Le Z80 a une instruction spéciale pour tester les bits, spécialement conçue pour cela, cette instruction est BIT.
BIT positionne l'indicateur Z au complément du bit spécifié dans le registre ou la location mémoire indiqués. Par exemple, BIT 5,D positionnera Z à 1 si le bit 5 du registre D est égal à zéro, et à zéro si le bit 5 du registre D est égal à 1. L'exécution de ce cas sera traitée comme suit :

Programme source :

```
        LD      HL,41H
        LD      B,(HL)      ; COMPTE NOMBRE D'ELEMENTS
        LD      C,0         ; NOMBRE DE NEGATIFS = 0
BOUCLE: INC     HL
        BIT     7,(HL)      ; ELEMENT SUIVANT NEGATIF ?
        JR      Z,TEST
        INC     C           ; OUI, + 1 AU NOMBRE DE NEGATIFS
TEST:   DJNZ    BOUCLE
        LD      A,C         ; RANGER NOMBRE DE NEGATIFS
        LD      (40H),A
        HALT
```

Programme objet :

Adresse Mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	46		LD	B,(HL)
0004	0E		LD	C,0
0005	00			
0006	23	BOUCLE :	INC	HL
0007	CB		BIT	7,(HL)
0008	7E			
0009	28		JR	Z,TEST
000A	01			
000B	0C		INC	C
000C	10	TEST :	DJNZ	BOUCLE
000D	F8			
000E	79		LD	A,C
000F	32		LD	(40H),A
0010	40			
0011	00			
0012	76		HALT	

BIT 7,(HL) positionne l'indicateur Z à 1 si le bit 7 de la location mémoire adressée par les registres H et L est à zéro et initialise l'indicateur Z à zéro si le bit 7 de cette mémoire est à 1. BIT n'affecte aucun registre ni emplacement mémoire.
Ce programme utilise JR Z,TEST car aucune incrémentation n'est nécessaire si le bit adressé est nul.
On pourrait aussi utiliser l'instruction RLC (HL) pour décaler le bit de signe de la donnée en mémoire sur la retenue. Le branchement nécessaire serait alors JR NC,TEST.
Cependant, cette méthode prend plus de temps (RLC (HL) nécessite 15 cycles alors que BIT 7,(HL) n'en nécessite que 12). De plus, elle modifie la donnée en mémoire alors que celle-ci peut être nécessaire pour d'autres traitements.

Notons que ces inconvénients sont liés ; le temps supplémentaire sert à ranger le résultat dans la mémoire.

Recherche du maximum

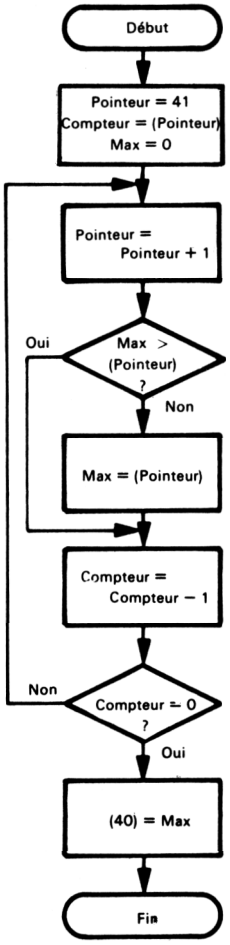
But : trouver le plus grand élément dans un bloc de données. La longueur du bloc est à l'adresse 0041 et le bloc lui-même commence à l'adresse 0042. Ranger le maximum à l'adresse 0040. Considérer que les nombres du bloc sont tous des nombres binaires non signés sur 8 bits.

Exemple pratique :

(0041) = 05
(0042) = 67
(0043) = 79
(0044) = 15
(0045) = E3
(0046) = 72

Résultat : (0040) = E3, puisque c'est le plus grand des 5 nombres non signés

Organigramme :



Programme source :

```
        LD      HL,41H      ; POINTER SUR COMPTEUR
        LD      B,(HL)      ; COMPTEUR = NOMBRE D'ELEMENTS
        SUB     A           ; MAXIMUM = MINIMUM POSSIBLE (ZERO)
SUITE : INC     HL
        CP      (HL)        ; ELEMENT SUIVANT > MAXIMUM ?
        JR      NC,DECPT
        LD      A,(HL)      ; OUI, METTRE ELEMENT DANS MAXIMUM
DECPT : DJNZ    SUITE
        LD      (40H),A     ; RANGER MAXIMUM
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	46		LD	B,(HL)
0004	97		SUB	A
0005	23	SUITE :	INC	HL
0006	BE		CP	(HL)
0007	30		JR	NC,DECPT
0008	01			
0009	7E		LD	A,(HL)
000A	10	DECPT :	DJNZ	SUITE
000B	F9			
000C	32		LD	(40H),A
000D	40			
000E	00			
000F	76		HALT	

Le déplacement relatif pour JR NC,DECPT est :

$$\begin{array}{r} 000A \\ -0009 \\ \hline 01 \end{array}$$

Le déplacement relatif pour DJNZ SUITE est :

$$\begin{array}{rcr} 0005 & = & 05 \\ -000C & & +F4 \\ & & \hline & & F9 \end{array}$$

Les trois premières instructions de ce programme constituent la section d'initialisation. Ce programme utilise le fait que zéro est le plus petit nombre binaire non signé, tenant sur 8 bits. Si on initialise le registre contenant la valeur maximum — dans ce cas, l'accumulateur — à la plus petite valeur possible avant d'entrer dans la boucle, alors le programme initialisera l'accumulateur à une plus grande valeur, à moins que tous les éléments pris en compte soient zéro.

Le programme marche correctement s'il y a deux éléments, mais pas s'il n'y en a qu'un ou pas du tout. Pourquoi ? Comment pourriez-vous résoudre ce problème ?
L'instruction CP (HL) initialise l'indicateur de retenue comme suit (ELEMENT est le contenu de l'adresse située dans les registres H et L et MAX est le contenu de l'accumulateur) :

$$\text{RETENUE} = 1 \text{ si ELEMENT} > \text{MAX}$$
$$\text{RETENUE} = 0 \text{ si ELEMENT} \leq \text{MAX}$$

Si RETENUE = zéro, le programme exécute directement DECPT et ne modifie pas le maximum.
Si RETENUE = 1, remplace l'ancien maximum par l'élément courant en exécutant l'instruction LD A,(HL).
Le programme ne marche pas avec des nombres signés car les nombres négatifs seraient considérés plus grands que les nombres positifs. Le problème est quelque peu délicat car un dépassement de capacité pourrait faire croire que le résultat a un mauvais signe.
Rappelons qu'un dépassement de capacité se produit lorsque l'ampleur d'un résultat affecte son bit de signe. Le Z80 a un indicateur pour parité/dépassement de capacité (P/O) qui indique quand un dépassement complémenté à 2 survient. Les opérations arithmétiques qui provoquent un dépassement de capacité initialisent cet indicateur. On peut alors tester sa valeur à l'aide des instructions JP PE,ADDR (branchement en cas de parité paire ou de dépassement de capacité) ou JP PO,ADDR (branchement en cas de parité impaire ou de non dépassement de capacité).
Il serait bon de vérifier si cette utilisation sur Z80 n'est pas incompatible avec les microprocesseurs 8080A ou 8085, qui eux utilisent toujours l'indicateur P pour indiquer la parité. Les microprocesseurs 8080A et 8085 n'ont pas d'indicateur de dépassement.

Cadrage binaire

But : décaler à gauche le contenu de l'adresse 0040 jusqu'à ce que le bit le plus significatif du nombre soit 1. Ranger le résultat à l'adresse 0041 et le nombre de décalage effectués à l'adresse 0042. Si le contenu de l'adresse 0040 est nul, remettre à zéro les adresses 0041 et 0042.

Remarque : le principe est le même que celui de la conversion d'un nombre en notation scientifique ; par exemple :

$$0.0057 = 5.7 \times 10^{-3}$$

Exemple pratique :

a.		(0040)	= 22
	Résultat :	(0041)	= 88
		(0042)	= 02
b.		(0040)	= 01
	Résultat :	(0041)	= 80
		(0042)	= 07

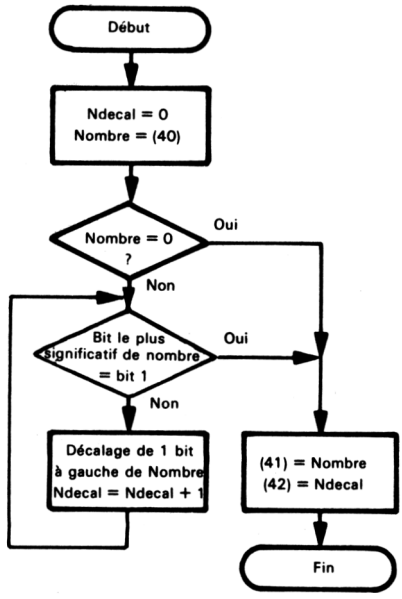
c. (0040) = CB

Résultat : (0041) = CB
(0042) = 00

d. (0040) = 00

Résultat : (0041) = 00
(0042) = 00

Organigramme :



Programme source :

```
        LD      B,0         ; NOMBRE DE DECALAGES = 0
        LD      HL,40H
        LD      A,(HL)      ; CHARGER DONNEE
        AND     A           ; DONNEE A ZERO
        JR      Z,FINI      ; OUI, FINI
TEST :  JP      M,FINI      ; FINI, SI BIT DE SIGNE = 1
        INC     B           ; + 1 AU NOMBRE DE DECALAGES
        ADD     A,A         ; DECALAGE 1 BIT A GAUCHE
        JP      TEST
FINI :  INC     HL
        LD      (HL),A      ; RANGER DONNEE CADREE
        INC     HL
        LD      (HL),B      ; RANGER NOMBRE DECALAGES
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	06	LD	B,0
0001	00		
0002	21	LD	HL,40H
0003	40		
0004	00		
0005	7E	LD	A,(HL)
0006	A7	AND	A
0007	28	JR	Z,FINI
0008	08		
0009	FA	TEST : JP	M,FINI
000A	11		
000B	00		
000C	04	INC	B
000D	87	ADD	A,A
000E	C3	JP	TEST
000F	09		
0010	00		
0011	23	FINI : INC	HL
0012	77	LD	(HL),A
0013	23	INC	HL
0014	70	LD	(HL),B
0015	76	HALT	

L'instruction JP M,FINI a pour effet un branchement à l'adresse FINI si le bit de signe est 1. Cette condition peut signifier que le dernier résultat est un nombre négatif ou bien que le bit le plus significatif de ce résultat est 1. Le calculateur fournit uniquement les résultats ; le programmeur doit fournir l'interprétation.
L'instruction ADD A,A additionne le nombre contenu dans l'accumulateur à lui-même. Le programme utilise cette instruction plutôt que RLA ou RLCA, parce que ADD A positionne l'indicateur de signe contrairement à RLA et RLCA.
Il est possible de réorganiser le programme de façon à éliminer les sauts inutiles et à utiliser des sauts relatifs plutôt que des sauts absolus. En voici une autre version :

```
        LD      B,0         ; NOMBRE DE DECALAGES = 0
        LD      HL,40H
        LD      A,(HL)      ; CHARGER DONNEE
        AND     A           ; DONNEE = ZERO
        JR      Z, FINI     ; OUI, FINI
        DEC     B           ; INITIALISER NOMBRE DECALAGES
TEST :  INC     B           ; + 1 AU NOMBRE DECALAGES
        RLA                 ; DECALAGE 1 BIT A GAUCHE
        JR      NC, TEST    ; CONTINUER SI BIT DE POIDS FORT = 0
        RRA                 ; RECADRAGE DONNEE
FINI :  INC     HL
        LD      (HL),A      ; RANGER DONNEE CADREE
        INC     HL
        LD      (HL),D      ; RANGER NOMBRE DECALAGES
        HALT
```

Essayez de faire marcher cette version.
Quels en sont les avantages et les inconvénients par rapport au précédent programme ?
Essayez de trouver une version plus performante (en temps et nombre d'octets).

EXERCICES

1) Octet de contrôle

But : calculer l'octet de contrôle d'une suite de nombres. La longueur de la suite est contenue à l'adresse 0041 et la suite elle-même commence à l'adresse 0042. Ranger le résultat à l'adresse 0040. L'octet de contrôle est calculé par l'opération logique OU exclusif de tous les nombres de la suite.

Remarque : les octets de contrôle sont souvent utilisés dans les systèmes à cassette et à bande perforée pour vérifier la lecture correcte de la donnée. L'octet de contrôle calculé est comparé à celui qui est stocké avec la donnée. Si les deux octets ne correspondent pas, le système indique généralement une erreur à l'opérateur ou relit automatiquement la donnée.

Exemple pratique :

(0041) = 03
(0042) = 28
(0043) = 55
(0044) = 26

Résultat : (0040) = (0042) ⊕ (0043) ⊕ (0044)
= 28 ⊕ 55 ⊕ 26
= 0 0 1 0 1 0 0 0
⊕ 0 1 0 1 0 1 0 1
0 1 1 1 1 1 0 1
⊕ 0 0 1 0 0 1 1 0
0 1 0 1 1 0 1 1
= 5B

2) Somme de données sur 16 bits

But : calculer la somme d'une suite de nombres sur 16 bits. La longueur de la suite est contenue à l'adresse 0042 et la suite elle-même commence à l'adresse 0043. Ranger la somme aux adresses 0040 et 0041 (les 8 bits les plus significatifs en 0041). Chaque nombre sur 16 bits occupe deux emplacements mémoire, avec les 8 bits les plus significatifs à l'adresse la plus haute.
On suppose que la somme est contenue sur 16 bits.

Exemple pratique :

(0042) = 03
(0043) = F1
(0044) = 28
(0045) = 1A
(0046) = 30
(0047) = 89
(0048) = 4B

Résultat : 28F1 + 301A + 4B89 =A494
(0040) = 94
(0041) = A4

3) Nombre d'éléments nuls, positifs et négatifs

But : Déterminer dans un bloc le nombre d'éléments nuls, d'éléments positifs (bit le plus significatif à zéro sans que le nombre entier soit à zéro) et d'éléments négatifs (bit le plus significatif à 1). La longueur du bloc se trouve à l'adresse 0043 et le bloc lui-même commence à l'adresse 0044. Ranger le nombre d'éléments négatifs à l'adresse 0040, le nombre d'éléments nuls à l'adresse 0041 et le nombre d'éléments positifs à l'adresse 0042.

Exemple pratique :

(0043) = 06
(0044) = 68
(0045) = F2
(0046) = 87
(0047) = 00
(0048) = 59
(0049) = 2A

Résultat : 2 négatifs, 1 nul, et 3 positifs, soit
(0040) = 02
(0041) = 01
(0042) = 03

4) Recherche du minimum

But : trouver le plus petit élément d'un bloc d'une donnée. La longueur du bloc se trouve à l'adresse 0041 et le bloc lui-même commence à l'adresse 0042. Ranger le minimum à l'adresse 0040. On suppose que les nombres du bloc sont des nombres binaires sur 8 bits non signés.

Exemple pratique :

(0041) = 05
(0042) = 67
(0043) = 79
(0044) = 15
(0045) = E3
(0046) = 72

Résultat : (0040) = 15, puisque c'est le plus petit des 5 nombres non signés.

5) Comptage des bits à 1

But : déterminer le nombre de bits à 1 de la mémoire d'adresse 0040 et ranger le résultat à l'adresse 0041.

Exemple pratique :

(0040) = 3B = 00111011

Résultat : (0041) = 05

CHAPITRE 6

TRAITEMENT DES CARACTÈRES

Les microprocesseurs ont souvent à traiter des données représentant des caractères. Non seulement les claviers, télétypes, appareils de communication, écrans et terminaux reçoivent ou fournissent des caractères, mais aussi beaucoup d'instruments, de systèmes de test et de contrôleurs nécessitent des données de ce type. Ces caractères sont naturellement codés sous forme binaire de façon à pouvoir être traités par des calculateurs. Le code le plus usité est l'ASCII (American Standard Code for Interchange of Information), les codes Baudot et EBCDIC (IBM) étant utilisés moins fréquemment. Nous n'utiliserons à partir de maintenant que des caractères codés sous forme ASCII (7 bits par octet, le bit de poids fort étant nul : voir tableau 6.1).

TRAITEMENT DES DONNÉES EN ASCII

Voici quelques principes à se rappeler lorsque l'on utilise des données en code ASCII :

1) Les codes pour les nombres et pour les lettres constituent des sous-ensembles ordonnés. Les codes pour les nombres décimaux vont de 30 à 39 hexa, de façon à pouvoir passer du décimal à de l'ASCII avec un simple facteur additif. Les codes pour les lettres en majuscules vont de 41 à 5A hexa, de façon à retrouver l'ordre alphabétique en triant les données par ordre numérique croissant.

2) Le calculateur ne fait aucune distinction entre les caractères imprimables et non imprimables. Cette distinction est faite au niveau des organes périphériques d'entrée-sortie.

3) Un périphérique ASCII ne traite que des données en ASCII. Pour imprimer un 7 sur une imprimante ASCII, le microprocesseur doit envoyer 37 hexa à l'imprimante. Le caractère hexa 07 est le caractère 'bell' (sonnette ou bip électronique). De façon similaire, le microprocesseur reçoit le caractère 9 d'un clavier ASCII comme le caractère hexa 39 ; le code hexa 09 correspond au caractère 'tab' (tabulation).

4) Quelques périphériques ASCII n'utilisent pas le jeu complet des caractères. Par exemple, des caractères de contrôle ou des lettres minuscules peuvent être ignorés ou imprimés sous forme de blancs ou de points d'interrogation.

5) Voici quelques caractères ASCII utilisés très couramment :

$0A_{16}$ — nouvelle ligne (LF)
$0D_{16}$ — retour chariot (CR)
20_{16} — blanc (ou espace) (SP)
$3F_{16}$ — ? (point d'interrogation)
$7F_{16}$ — effacement caractère

6) Chaque caractère ASCII occupe 7 bits. Ceci permet un large ensemble de caractères mais gâche beaucoup de place lorsque les données se réduisent à un petit sous-ensemble tel que les nombres décimaux. Un octet 8 bits, par exemple, ne peut contenir qu'un chiffre décimal codé en ASCII alors qu'il en contient deux en code DCB (décimal codé binaire).

Tableau 6.1. — Tableau Hexa-ASCII

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

EXEMPLES

Longueur d'une chaîne de caractères

But : déterminer la longueur d'une chaîne de caractères ASCII (7 bits avec bit zéro de poids fort). La chaîne commence à l'adresse 0041 ; la fin de la chaîne est déterminée par le caractère retour chariot ('CR', hexa OD). Mettre la longueur de la chaîne (retour chariot non compris) à l'adresse 0040.

Exemple pratique :

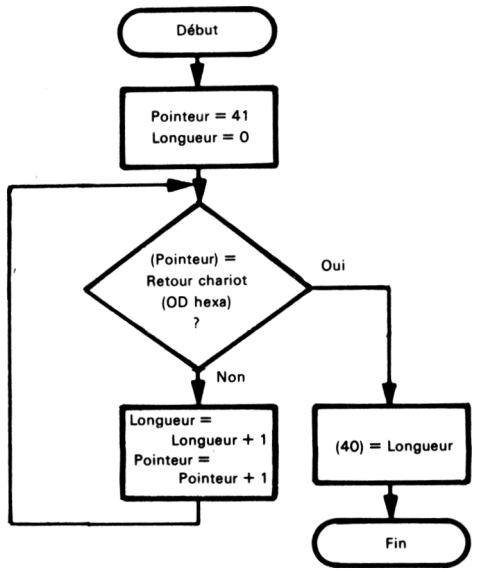
a. (0041) = 0D

Résultat : (0040) = 00 puisque le premier caractère est le retour chariot

b. (0041) = 52 'R'
(0042) = 41 'A'
(0043) = 54 'T'
(0044) = 48 'H'
(0045) = 45 'E'
(0046) = 52 'R'
(0047) = 0D CR

Résultat : (0040) = 06

Organigramme :



Programme source :

```
        LD      HL,41H      ; POINTEUR = DEBUT CHAINE
        LD      B,0         ; LONGUEUR CHAINE = 0
        LD      A,0DH       ; LIRE RETOUR CHARIOT POUR COMPARAISON
TESTCR: CP      (HL)        ; CARACTERE = RETOUR CHARIOT ?
        JR      Z,FINI      ; OUI, SAUT A FINI
        INC     B           ; NON, + 1 SUR LONGUEUR CHAINE
        INC     HL
        JR      TESTCR      ; ESSAI CARACTERE SUIVANT
FINI:   LD      A,B         ; SAUVEGARDE LONGUEUR CHAINE
        LD      (40H),A
        HALT
```

Programme objet :

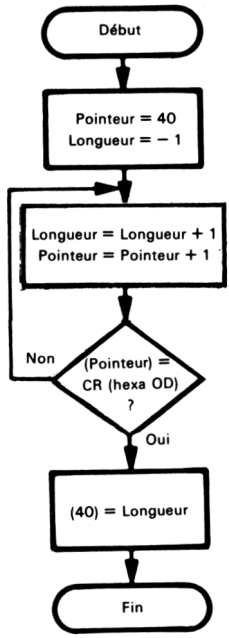
Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	06		LD	B,0
0004	00			
0005	3E		LD	A,0DH
0006	0D			
0007	BE	TESTCR	CP	(HL)
0008	28		JR	Z,FINI
0009	04			
000A	04		INC	B
000B	23		INC	HL
000C	18		JR	TESTCR
000D	F9			
000E	78	FINI :	LD	A,B
000F	32		LD	(40H),A
0010	40			
0011	00			
0012	76		HALT	

Le retour chariot (CR) est un caractère ASCII comme un autre (hexa 0D) pour le calculateur. Le fait que l'unité de sortie traite le retour chariot comme un caractère de contrôle plutôt que comme un caractère d'impression ne change rien pour le calculateur.
L'instruction de comparaison, CP, positionne les indicateurs comme dans le cas d'une soustraction mais laisse le caractère de retour chariot dans l'accumulateur pour des comparaisons ultérieures. L'indicateur de zéro (Z) est modifié comme suit :

Z = 1 si le caractère de la chaîne est un retour chariot
Z = 0 sinon.

L'instruction INC B ajoute 1 au compteur de longueur de chaîne dans le registre B. L'instruction LD B,0 remet ce compteur à zéro avant que la boucle ne commence. Ne pas oublier d'initialiser des variables avant de les utiliser dans une boucle.
Cette boucle ne s'arrête pas lorsque le compteur atteint zéro. Le calculateur continue alors simplement à examiner des caractères jusqu'à ce qu'il trouve un retour chariot.
Pour éviter le problème des chaînes erronées qui ne contiennent pas de retour chariot, il faut faire intervenir une borne maximum dans une boucle comme celle-là. Qu'arriverait-il si le programme ci-dessus utilisait une telle chaîne ?
Remarquez qu'en modifiant la logique et les conditions initiales du programme, on peut le raccourcir et réduire son temps d'exécution. Si l'on ajuste l'organigramme de façon à incrémenter les compteur et pointeur avant la comparaison avec le retour chariot, seule une instruction de saut est nécessaire au lieu de deux. D'où les nouveaux organigramme et programme :

Organigramme :



Programme source :

```
        LD      HL,40H      ; POINTEUR = OCTET AVANT CHAINE
        LD      B,0FFH      ; LONGUEUR = – 1
        LD      A,0DH       ; LIRE RETOUR CHARIOT ASCII POUR COMPARAISON
TESTCR: INC     HL
        INC     B           ; + 1 SUR LONGUEUR CHAINE
        CP      (HL)        ; CARACTERE = RETOUR CHARIOT ?
        JR      NZ,TESTCR   ; NON, COMPARAISON CARACTERE SUIVANT
        LD      A,B         ; OUI, SAUVEGARDE LONGUEUR CHAINE
        LD      (40H),A
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)		Instruction (Mnémonique)	
0000	21		LD	HL,40H
0001	40			
0002	00			
0003	06		LD	B,0FFH
0004	FF			
0005	3E		LD	A,0DH
0006	0D			
0007	23	TESTCR:	INC	HL
0008	04		INC	B
0009	BE		CP	(HL)
000A	20		JR	NZ,TESTCR
000B	FB			
000C	78		LD	A,B
000D	32		LD	(40H),A
000E	40			
000F	00			
0010	76		HALT	

La recherche d'une valeur particulière dans une liste, table ou chaîne, est une tâche courante. Le microprocesseur Z80 dispose en fait d'instructions spéciales qui simplifient cette tâche.

INSTRUCTIONS DE RECHERCHE EN BLOC

Ces instructions spéciales s'appellent : instructions de recherche en bloc. Voici en quoi elles consistent :
CPI compare le contenu de l'emplacement mémoire adressé par HL avec le contenu de l'accumulateur (tout comme CP(HL)). Puis elle incrémente HL et décrémente le compteur d'octet (double registre BC). L'indicateur P/O est positionné à zéro si le compteur d'octet est décrémenté jusqu'à zéro, et positionné à un sinon.
CPD est la même instruction si ce n'est qu'elle décrémente HL au lieu de l'incrémenter.
CPIR et CPDR constituent les formes répétitives des instructions de recherche en bloc. Ces instructions répètent l'instruction de base de recherche jusqu'à ce que BC soit décrémenté jusqu'à zéro ou qu'un test de comparaison soit vrai (c'est-à-dire A = (HL)).
Il ne faut pas oublier que la décrémentation de BC jusqu'à zéro remet à zéro l'indicateur P/O, alors que dans le cas d'égalité dans le test de comparaison, l'indicateur Z est positionné à 1.
Il faut remarquer que BC contient un compteur 16 bits. Ainsi, les instructions de recherche en bloc peuvent traiter des chaînes de n'importe quelle longueur ($<$64 K).
Voici une version du programme précédent utilisant l'instruction CPI :

Programme source :

```
        LD      HL,41H      ; POINTEUR = DEBUT DE CHAINE
        LD      BC,0        ; COMPTEUR D'OCTET = ZERO
        LD      A,0DH       ; LIRE RETOUR CHARIOT ASCII POUR COMPARAISON
TESTCR: CPI                 ; CARACTERE = RETOUR CHARIOT ?
        JR      NZ,TESTCR   ; NON, TEST CARACTERE SUIVANT
        LD      A,0FFH      ; OUI, CALCUL LONGUEUR CHAINE
        SUB     C
        LD      (40H),A     ; SAUVEGARDE LONGUEUR CHAINE
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,41H
0001	41		
0002	00		
0003	01	LD	BC,0
0004	00		
0005	00		
0006	3E	LD	A,0DH
0007	0D		
0008	ED	TESTCR : CPI	
0009	A1		
000A	20	JR	NZ,TESTCR
000B	FC		
000C	3E	LD	A,0FFH
000D	FF		
000E	91	SUB	C
000F	32	LD	(40H),A
0010	40		
0011	00		
0012	76	HALT	

Une petite manipulation est nécessaire pour calculer la longueur de la chaîne puisque l'instruction CPI décrémente le compteur d'octets (BC) au lieu de l'incrémenter comme dans le programme précédent avec l'instruction INC B. De plus, le compteur d'octets est décrémenté une fois de plus lorsque le caractère de retour chariot est trouvé. Comment feriez-vous varier les conditions initiales pour traiter ce problème ?

En fait, il est possible d'améliorer encore le programme en utilisant l'instruction CPIR pour éviter le saut relatif JR. L'instruction CPIR a le même effet que CPI, mais répète automatiquement le test de comparaison jusqu'à ce que A = (HL) ou BC soit décrémenté jusqu'à zéro.

Le programme qui utilise CPIR est donné ci-dessous :

Programme source :

```
        LD      HL,41H      ; POINTEUR = DEBUT DE CHAINE
        LD      BC,0        ; COMPTEUR OCTET = ZERO
        LD      A,0DH       ; LIRE RETOUR CHARIOT ASCII POUR COMPARAISON
        CPIR                ; RECHERCHE D'UN RETOUR CHARIOT ?
        LD      A,0FFH      ; CALCUL LONGUEUR CHAINE A PARTIR DU COMPTEUR
        SUB     C
        LD      (40H),A     ; SAUVEGARDE LONGUEUR CHAINE
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,41H
0001	41		
0002	00		
0003	01	LD	BC,0
0004	00		
0005	00		
0006	3E	LD	A,0DH
0007	0D		
0008	ED	CPIR	
0009	B1		
000A	3E	LD	A,0FFH
000B	FF		
000C	91	SUB	C
000D	32	LD	(40H),A
000E	40		
000F	00		
0010	76	HALT	

Des instructions multiples telles que CPI et CPIR ont le même effet que les séquences qu'elles remplacent. Elles permettent des gains en temps d'exécution et en mémoire car le processeur utilise moins d'instructions à chaque passage dans la boucle. D'où les gains réels dans l'exécution de la boucle.
Tous ces programmes supposent une chaîne de moins de 256 octets. Comment pourriez-vous les modifier pour traiter des chaînes plus longues ?

Recherche du premier caractère non blanc

But : chercher dans une chaîne de caractères ASCII (7 bits avec le bit zéro comme bit de poids fort) un caractère non blanc. La chaîne commence à l'adresse 0042. Mettre l'adresse du premier caractère non blanc aux adresses 0040 et 0041 (bits de poids fort en 0041). Un caractère blanc est en ASCII 20 hexa.

Exemple pratique :

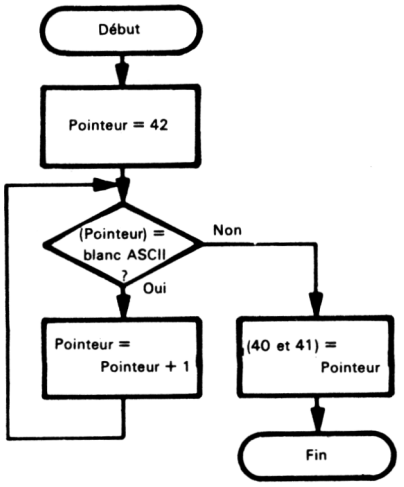
a. (0042) = 37 '7'

Résultat : (0040) = 42, puisque l'adresse 0042 contient un caractère non blanc
(0041) = 00

b. (0042) = 20 SP
(0043) = 20 SP
(0044) = 20 SP
(0045) = 46 F
(0046) = 20 SP

Résultat : (0040) = 45, puisque les 3 adresses précédentes contiennent des blancs
(0041) = 00

Organigramme :



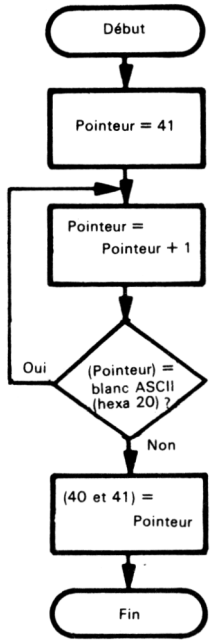
Programme source :

```
        LD      HL,42H      ; POINTEUR = DEBUT DE CHAINE
        LD      A,20H       ; LIRE BLANC ASCII POUR COMPARAISON
TESTBL: CP      (HL)        ; CARACTERE = BLANC ASCII ?
        JR      NZ,FINI     ; NON, FINI
        INC     HL
        JR      TESTBL      ; OUI, TEST CARACTERE SUIVANT
FINI:   LD      (40H),HL    ; NON, SAUVEGARDE ADRESSE PREMIER
                            ;   CARACTERE NON BLANC
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,42H
0001	42		
0002	00		
0003	3E	LD	A,20H
0004	20		
0005	BE	TESTBL : CP	(HL)
0006	20	JR	NZ,FINI
0007	03		
0008	23	INC	HL
0009	18	JR	TESTBL
000A	FA		
000B	22	FINI : LD	(40H),HL
000C	40		
000D	00		
000E	76	HALT	

La recherche de blancs dans des chaînes est une tâche fréquente. Les blancs sont souvent éliminés des chaînes quand ils ne servent qu'à augmenter la lisibilité ou à convenir à des formats particuliers. De toute évidence, il est inutile de ranger en mémoire ou de transmettre des blancs en début, fin ou milieu de chaîne, surtout si cela est coûteux en capacité de communication et en place mémoire. Cependant, les données et les entrées du programme sont beaucoup plus simples si les espaces sont tolérés. Les microcalculateurs utilisent souvent des programmes de conversion de données entre des formes pratiques pour l'homme et des formes optimisées pour les calculateurs et les lignes de transmission.
L'instruction LD (addr),HL est pratique pour ranger en mémoire des adresses dans le format Z80 (octet moins significatif d'abord). LD (40H),HL range en mémoire le contenu du registre L à l'adresse 0040 et celui du registre H à l'adresse 0041.
A nouveau, si l'on modifie les conditions initiales de façon à faire précéder la section d'exécution par la procédure de boucle, on peut réduire le nombre d'octets du programme ainsi que le temps d'exécution de la boucle.
Voici l'organigramme modifié :



Programme source :

```
        LD      HL,41H      ; POINTEUR D'OCTET POSITIONNE AVANT CHAINE
        LD      A,20H       ; LIRE BLANC ASCII POUR COMPARAISON
TESTBL: INC     HL
        CP      (HL)        ; CARACTERE = BLANC ASCII ?
        JR      Z,TESTBL    ; OUI, TEST CARACTERE
        LD      (40H),HL    ; NON, SAUVEGARDE ADRESSE 1er CARACTERE
                            ;   NON BLANC
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,41H
0001	41		
0002	00		
0003	3E	LD	A,20H
0004	20		
0005	23	TESTBL : INC	HL
0006	BE	CP	(HL)
0007	28	JR	Z,TESTBL
0008	FC		
0009	22	LD	(40H),HL
000A	40		
000B	00		
000C	76	HALT	

Comme dans l'exemple précédent, on pourrait remplacer la séquence INC HL,CP(HL) par l'instruction unique CPI. Mais comme l'on n'utilise pas le compteur d'octets dans ce programme, CPI prend autant de place mémoire (2 octets) et plus de temps (16 cycles d'horloge au lieu de 13) que l'instruction qu'elle remplace. On ne peut pas utiliser ici CPIR puisque l'on veut que le programme se termine quand les caractères sont **différents.**

Remplacement des zéros de début de chaîne par des blancs

But : éditer une chaîne de caractères décimaux ASCII en remplaçant les zéros du début de chaîne par des blancs. La chaîne commence à l'adresse 0041 ; on suppose qu'elle se compose uniquement de chiffres décimaux codés en ASCII. La longueur de la chaîne se trouve en 0040.

Exemple pratique :

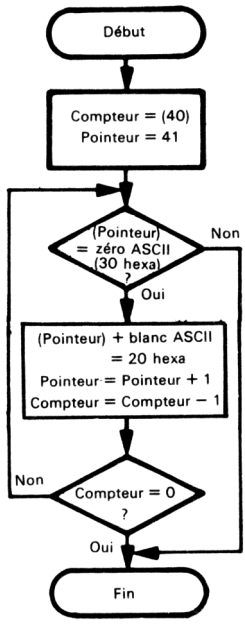
a.
(0040) = 02
(0041) = 36 '6'

Le programme ne change rien à la chaîne puisque le chiffre de départ n'est pas zéro.

b.
(0040) = 08
(0041) = 30 '0'
(0042) = 30 '0'
(0043) = 38 '8'

Résultat : (0041) = 20 SP espace
(0042) = 20 SP espace

Organigramme :



Programme source :

```
        LD      HL,40H
        LD      B,(HL)      ; COMPTEUR = LONGUEUR CHAINE
        LD      A,'0'       ; LIRE ZERO ASCII POUR COMPARAISON
TESTZ : INC     HL
        CP      (HL)        ; CHIFFRE DE DEBUT A ZERO ?
        JR      NZ,FINI     ; NON, FINI
        LD      (HL),20H    ; OUI, REMPLACEMENT PAR UN BLANC
        DJNZ    TESTZ       ; TEST CHIFFRE SUIVANT, S'IL Y EN A
FINI :  HALT
```

Les apostrophes qui entourent les caractères indiquent de l'ASCII.

Programme objet : (voir page 263)

Il est courant que l'on veuille éditer des chaînes décimales avant qu'elles ne soient imprimées ou affichées, et ceci dans un souci de clarté. Les tâches habituelles d'édition comprennent l'élimination des zéros de tête, le cadrage des nombres, le rajout de signes ou de caractères spéciaux ainsi que le calcul d'arrondi. Il est certain que des nombres imprimés ainsi : 0006, 27,34832 FF peuvent prêter à confusion et gêner.
Dans ce programme, la boucle présente 2 sorties — l'une pour le cas où le processeur trouve un chiffre différent de zéro et l'autre si la chaîne a été entièrement testée.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	46	LD	B,(HL)
0004	3E	LD	A,'0'
0005	30		
0006	23	TESTZ : INC	HL
0007	BE	CP	(HL)
0008	20	JR	ZN,FINI
0009	04		
000A	36	LD	(HL),20H
000B	20		
000C	10	DJNZ	TESTZ
000D	F8		
000E	76	FINI : HALT	

L'instruction LD (HL),20H met 20 hexa à l'adresse contenue par H et L. On peut aussi initialiser à 20 hexa le registre C (c'est-à-dire LD C,20H) et utiliser l'instruction LD (HL),C pour remplacer le zéro de tête par un blanc. Voyons les compromis qu'implique cet exemple.
L'instruction LD (HL),C s'exécute plus vite que LD (HL),20H et réduirait ainsi le temps d'exécution de la boucle. Elle doit cependant être précédée de l'instruction LD C,20H dans la section d'initialisation du programme. Si cet exemple servait à une application de caisse enregistreuse, quelle séquence choisiriez-vous et pourquoi ?
On considère que tous les chiffres de la chaîne sont en ASCII ; c'est-à-dire de 30 à 39 hexa et non en décimal ordinaire 0 à 9. La conversion décimal-ASCII s'obtient simplement en rajoutant 30 hexa au chiffre décimal.
Il faut faire attention en remplaçant les zéros de tête par des blancs à conserver un zéro dans le cas où tous les chiffres sont à zéros. Comment vous y prendriez-vous ?
Il convient de noter que chaque chiffre ASCII nécessite 8 bits contre 4 pour un chiffre DCB (décimal code binaire). C'est pourquoi l'ASCII est un format coûteux pour le rangement en mémoire ou la transmission de données numériques.

Addition d'une parité paire aux caractères ASCII

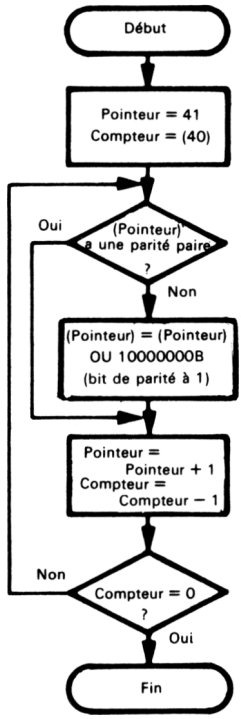
But : ajouter une parité paire à une chaîne de caractères ASCII 7 bits. La longueur de la chaîne est à l'adresse 0040 et la chaîne elle-même commence à l'adresse 0041. Donner une parité paire au bit le plus significatif de chaque caractère en mettant le bit le plus significatif à 1 si cela rend pair le nombre total de bits à 1 du mot.

Exemple pratique :

(0040) = 06
(0041) = 31
(0042) = 32
(0043) = 33
(0044) = 34
(0045) = 35
(0046) = 36

Résultat : (0041) = B1
(0042) = B2
(0043) = 33
(0044) = B4
(0045) = 35
(0046) = 36

Organigramme :



Programme source :

```
        LD      HL,40H
        LD      B,(HL)          ; LECTURE LONGUEUR CHAINE
        LD      C,10000000B     ; LECTURE BIT DE PARITE 1
CALPAR: INC     HL
        LD      A,(HL)          ; LECTURE CARACTERE
        OR      C               ; MISE A 1 DU BIT DE PARITE ET TEST PARITE
        JP      PO,BOUCLE       ; PARITE PAIRE ?
        LD      (HL),A          ; OUI, SAUVEGARDE CARACTERE PARITE PAIRE
BOUCLE: DJNZ    CALPAR
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	46	LD	B,(HL)
0004	0E	LD	C,10000000H
0005	80		
0006	23	CALPAR: INC	HL
0007	7E	LD	A,(HL)
0008	B1	OR	C
0009	E2	JP	PO,BOUCLE
000A	0D		
000B	00		
000C	77	LD	(HL),A
000D	10	BOUCLE: DJNZ	CALPAR
000E	F7		
000F	76	HALT	

On ajoute souvent la parité aux caractères ASCII avant qu'ils ne soient transmis sur des lignes de communication parasitée, pour permettre une plus simple vérification d'erreur. La parité détecte toutes les erreurs bit par bit mais ne permet pas de corriger l'erreur (c'est-à-dire, on sait qu'il y a erreur en cas de parité fausse mais on ne peut savoir quel bit elle concerne).

L'instruction LD C,10000000B charge un bit de parité de 1 dans le registre C. (Il faut remarquer l'usage du masque binaire : l'objet de ce masque apparaît plus clairement ainsi représenté que par 80H ou 128 décimal.)

L'instruction OR C met le bit de parité (bit le plus significatif) à 1 sans modifier les autres bits, tout en positionnant l'indicateur de parité du Z80 à 1.

La procédure qui suit sert à déterminer si la parité de l'octet en mémoire est paire ou impaire. Un OU logique est effectué entre le bit de parité et l'octet chargé à partir de la mémoire, suivi d'un test pour voir si la parité est impaire. Si c'est le cas, l'octet en mémoire est de parité paire et l'on saute à BOUCLE pour décrémenter le compteur d'octets restants. Si la parité est paire, on sait alors que l'octet en mémoire possède une parité impaire et c'est pourquoi on range l'octet contenu dans l'accumulateur à cette adresse.

Les sauts conditionnels JP PO (saut sur parité impaire) et JP PE (saut sur parité paire) sont rarement utilisés si ce n'est pour la génération et le contrôle de la parité. Il faut

remarquer qu'il n'y a pas de sauts *relatifs* qui dépendent de la valeur du bit de parité, comme il n'y en a pas qui dépendent de la valeur du bit de signe.
Il ne faut pas confondre le bit de parité compris dans chaque caractère et l'indicateur de parité du Z80, qui est positionné à 1 si le dernier résultat arithmétique ou Booléen possède une parité paire.
Une autre solution consiste à utiliser l'instruction SET du Z80. Cette version prend un peu plus de temps mais ne nécessite pas de registre temporaire pour le bit de parité.

Programme source :

```
        LD      HL,40H
        LD      B,(HL)      ; LECTURE LONGUEUR CHAINE
CALPAR: INC     HL
        LD      A,(HL)      ; LECTURE CARACTERE
        OR      A           ; CARACTERE A PARITE PAIRE ?
        JP      PE,CHCNT
        SET     7,(HL)      ; NON, BIT PARITE POSITIONNE A 1
BOUCLE: DJNZ    SETPR
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	46	LD	B,(HL)
0004	23	CALPAR : INC	HL
0005	7E	LD	A,(HL)
0006	B7	OR	A
0007	EA	JP	PE,BOUCLE
0008	0C		
0009	00		
000A	CB	SET	7,(HL)
000B	FE		
000C	10	BOUCLE : DJNZ	CALPAR
000D	F6		
000E	76	HALT	

Egalité de séquences

But : comparer deux chaînes de caractères ASCII pour voir si elles sont les mêmes. La longueur des chaînes se trouvent à l'adresse 0041, la première commence à l'adresse 0042 et l'autre à l'adresse 0052. Si les 2 chaînes sont égales, mettre à zéro l'adresse 0040 ; sinon mettre cette adresse à FF hexa (tous les bits à 1).

Exemples pratiques :

a.

(0041) = 03

(0042) = 43 'C'
(0043) = 41 'A'
(0044) = 54 'T'

(0052) = 43 'C'
(0053) = 41 'A'
(0054) = 54 'T'

Résultat : (0040) = 00, puisque les 2 chaînes sont les mêmes

b.

(0041) = 03

(0042) = 52 'R'
(0043) = 41 'A'
(0044) = 54 'T'

(0052) = 43 'C'
(0053) = 41 'A'
(0054) = 54 'T'

Résultat : (0040) = FF, puisque les premiers caractères des chaînes sont différents

Remarque : le processus de test cesse aussitôt que l'unité centrale trouve une différence. Le reste de la chaîne n'est pas contrôlé.

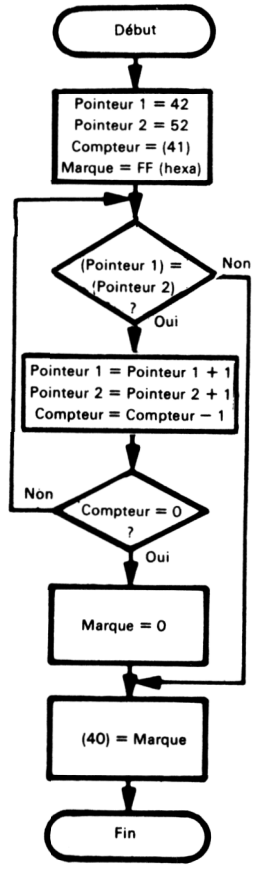
Organigramme : (voir page 268)

Programme source :

```
        LD      HL,41H
        LD      B,(HL)          ; COMPTEUR = LONGUEUR CHAINES
        INC     HL              ; POINTEUR 1 = DEBUT CHAINE 1
        LD      DE,52H          ; POINTEUR 2 = DEBUT CHAINE 2
        LD      C,0FFH          ; MARQUE = FF (HEXA)
TEST :  LD      A,(DE)          ; LECTURE CARACTERE CHAINE 2
        CP      (HL)            ; EGALITE ?
        JR      NZ,FINI         ; NON, FINI
        INC     DE
        INC     HL
        DJNZ    TEST            ; TEST PAIRE SUIVANTE SI ENCORE
        LD      C,0             ; MARQUE = 0 SI EGALITE COMPLETE
FINI :  LD      A,C
        LD      (40H),A         ; SAUVEGARDE MARQUE
        HALT
```

Programme objet : (voir page 268)

L'égalité de chaînes de caractères ASCII est une partie essentielle de la recherche des commandes, des noms à reconnaître, des variables ou des codes opératoires à identifier dans les assembleurs et compilateurs, des fichiers et de bien d'autres tâches.
Le programme utilise deux pointeurs, l'un dans la paire de registres HL et l'autre dans la paire de registres DE. Les seules instructions qui utilisent l'adresse contenue dans DE



Organigramme

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,41H
0001	41		
0002	00		
0003	46	LD	B,(HL)
0004	23	INC	HL
0005	11	LD	DE,52H
0006	52		
0007	00		
0008	0E	LD	C,0FFH
0009	FF		
000A	1A	TEST : LD	A,(DE)
000B	BE	CP	(HL)
000C	20	JR	NZ,FINI
000D	06		
000E	13	INC	DE
000F	23	INC	HL
0010	10	DJNZ	TEST
0011	F8		
0012	0E	LD	C,0
0013	00		
0014	79	FINI : LD	A,C
0015	32	LD	(40H),A
0016	40		
0017	00		
0018	76	HALT	

sont LD A,(DE) (chargement de l'accumulateur à partir du contenu de l'adresse située en DE) et LD (DE),A (rangement du contenu de l'accumulateur à l'emplacement mémoire adressé par DE). Les opérations arithmétiques et logiques avec la mémoire ainsi que les transferts de registres (comme ADD A,(HL) ; AND (HL) ; LB B,(HL) ; LD (HL),E) se servent tous de l'adresse contenue dans le double registre HL ou utilisent un registre d'index.

L'ordre des opérations est très important en raison du petit nombre d'instructions qui utilisent le double registre DE. Il faut transférer dans l'accumulateur un caractère de la chaîne sur laquelle pointe DE et le comparer à un caractère de la chaîne sur laquelle pointe HL.

Les opérations doivent être effectuées dans cet ordre car le Z80 ne dispose pas d'instruction permettant une comparaison avec un caractère de la chaîne sur laquelle pointe DE. Par exemple, si l'on remplaçait LD A,(DE) par LD A,(HL), quelle serait l'instruction suivante ? Cette asymétrie est particulière au Z80 et peut engendrer une programmation inextricable.

Il convient de noter que chaque itération met à jour les deux pointeurs.

Ce programme pourrait profiter du fait qu'un registre contient systématiquement zéro après l'exécution d'un saut conditionnel particulier. Lorsque l'instruction DJNZ TEST est exécutée, si le branchement n'est pas effectué, on sait alors que le registre B contient zéro. En conséquence, on peut transférer le registre B au registre C, notre registre d'état, pour signaler qu'une égalité a été trouvée.

Il est aussi possible d'utiliser les instructions du Z80 SET et RESET pour positionner l'indicateur s'il est utile de conserver les bits pour d'autres fins.

EXERCICES

1) Longueur d'un message d'une téléimprimante

But : déterminer la longueur d'un message ASCII. Tous les caractères sont de l'ASCII sur 7 bits avec MSB = 0 (bit le plus significatif à zéro). La chaîne de caractères qui contient le message commence à l'adresse 0041. Le message lui-même commence par le caractère ASCII STX (02 hexa) et se termine par ETX (03 hexa). Mettre la longueur du message à l'adresse 0040 (nombre de caractères entre STX et ETX, STX et ETX exclus).

Exemple pratique :

(0041) = 40
(0042) = 02 STX
(0043) = 47 'G'
(0044) = 4F 'O'
(0045) = 03 ETX

Résultat : (0040) = 02, puisqu'il y a 2 caractères entre STX à l'adresse 0042 et ETX à l'adresse 0045

2) Recherche du dernier caractère non blanc

But : chercher dans une chaîne de caractères ASCII le dernier caractère non blanc. La chaîne commence à l'adresse 0042 et se termine par un caractère de retour chariot (0D hexa). Mettre l'adresse du dernier caractère non blanc aux adresses 0040 et 0041 (bits les plus significatifs en 0041).

Exemple pratique :

a.
(0042) = 37 '7'
(0043) = 0D CR

Résultat : (0040) = 42, 42 puisque le dernier (et unique caractère non blanc est à l'adresse 0042
(0041) = 00

b.
(0042) = 41 'A'
(0043) = 20 SP
(0044) = 48 'H'
(0045) = 41 'A'
(0046) = 54 'T'
(0047) = 20 SP
(0048) = 20 SP
(0049) = 0D CR

Résultat : (0040) = 46
(0041) = 00

3) Troncature d'une chaîne décimale sous forme entière

But : éditer une chaîne de caractères ASCII décimaux en remplaçant tous les chiffres à droite du point décimal par des blancs ASCII (20 hexa). La chaîne commence à l'adresse 0041 et l'on suppose qu'elle est composée uniquement de chiffres décimaux codés en

ASCII et éventuellement du point décimal (2E hexa). La longueur de la chaîne se trouve à l'adresse 0040. Si le point décimal n'apparaît pas dans la chaîne, considérer qu'il se trouve implicitement à l'extrême droite.

Exemples pratiques :

a.
(0040) = 04
(0041) = 37 '7'
(0042) = 2E '.'
(0043) = 38 '8'
(0044) = 31 '1'

Résultat :
(0041) = 37 '7'
(0042) = 2E '.'
(0043) = 20 SP
(0044) = 20 SP

b.
(0040) = 03
(0041) = 26 '6'
(0042) = 37 '7'
(0043) = 31 '1'

Résultat : Inchangé, car on considère que le nombre est : 671

4) Contrôle de la parité paire dans les caractères ASCII

But : contrôler la parité paire dans une chaîne de caractères ASCII. La longueur de la chaîne se trouve à l'adresse 0041, et la chaîne elle-même commence à l'adresse 0042. Si la parité de tous les caractères de la chaîne est correcte, mettre à zéro le contenu de l'adresse 0040 ; sinon, le mettre à FF hexa (tous bits à 1).

Exemples pratiques :

a.
(0041) = 03
(0042) = B1
(0043) = B2
(0044) = 33

Résultat : (0040) = 00, puisque tous les caractères ont une parité paire

b.
(0041) = 03
(0042) = B1
(0043) = B6
(0044) = 33

Résultat : (0040) = FF puisque le caractère à l'adresse 0042 n'a pas une parité paire

5) Comparaison de chaînes

But : comparer 2 chaînes de caractères ASCII pour déterminer laquelle est la plus grande (c'est-à-dire laquelle suit l'autre dans l'ordre « alphabétique »).
La longueur des chaînes se trouve à l'adresse 0041 ; une chaîne commence à l'adresse 0042 et l'autre à l'adresse 0052. Si la chaîne qui commence à l'adresse 0042 est plus grande ou égale à l'autre chaîne, mettre à zéro le contenu de l'adresse 0040 ; sinon, mettre à FF hexa (tous bits à 1) le contenu de l'adresse 0040.

Exemples pratiques :

a.

(0041)	= 03		
(0042)	= 43	'C'	
(0043)	= 41	'A'	
(0044)	= 54	'T'	
(0052)	= 42	'B'	
(0053)	= 41	'A'	
(0054)	= 54	'T'	
Résultat : (0040)	= 00,	puisque CAT est 'supérieur' à BAT	

b.

(0041)	= 03		
(0042)	= 43	'C'	
(0043)	= 41	'A'	
(0044)	= 54	'T'	
(0052)	= 43	'C'	
(0053)	= 41	'A'	
(0054)	= 54	'T'	
Résultat : (0040)	= 00,	lorsque les 2 chaînes sont égales	

c.

(0041)	= 03		
(0042)	= 43	'C'	
(0043)	= 41	'A'	
(0044)	= 54	'T'	
(0052)	= 43	'C'	
(0053)	= 55	'U'	
(0054)	= 54	'T'	
Résultat : (0040)	= FF,	puisque CUT est 'supérieur' à CAT	

CHAPITRE 7

CONVERSION DE CODE

La conversion de code est un problème continuel dans la plupart des applications de microcalculateurs. Les périphériques d'entrée fournissent les données en ASCII, DCB (décimal codé binaire), ou d'autres codes variés. Le système doit convertir les données en une forme standard pour l'exécution. Les périphériques de sortie traitent les données en ASCII, DCB, affichage 7-segments ou d'autres codes. C'est pourquoi le système doit convertir les résultats sous la forme adéquate après la fin de l'exécution.
Il y a plusieurs moyens d'aborder la conversion de code :

1) Certaines conversions peuvent être facilement traitées par des algorithmes impliquant des fonctions arithmétiques ou logiques. Cependant, il se peut que le programme ait à traiter séparément quelques cas particuliers.
2) Des conversions plus complexes peuvent être traitées à l'aide de tables de correspondance. Cette méthode nécessite peu de programmation et est facile à employer. Cependant, cette table peut occuper beaucoup de mémoire programme s'il y a un grand nombre de valeurs d'entrée.
3) Le matériel effectue certaines tâches de conversion. Des exemples typiques sont donnés par les décodeurs DCB-affichage 7 segments ainsi que les contrôleurs de communication asynchrones (UART, etc.) pour la conversion entre les formats parallèle (ASCII) et série (terminaux éloignés, par exemple).

Dans la plupart des applications, le programme devrait faire autant de conversion de code que possible. Ceci permet un gain de place ainsi qu'une meilleure fiabilité. En plus, la plupart des conversions de code sont faciles à programmer et n'utilisent que peu de temps d'exécution.

EXEMPLES

Hexa-ASCII

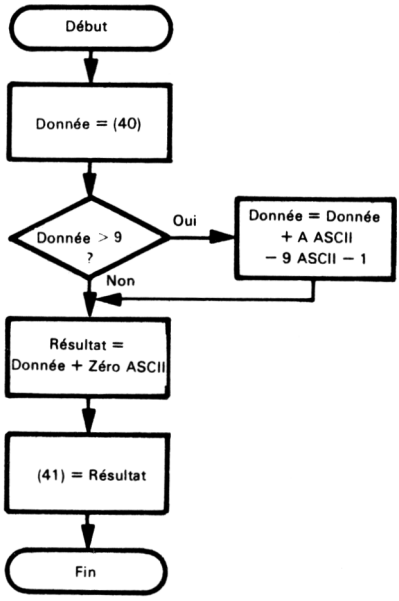
But : convertir le contenu dè l'adresse 0040 en un caractère ASCII. L'adresse 0040 contient un seul chiffre hexadécimal (les 4 bits les plus significatifs sont à zéro). Ranger le caractère ASCII à l'adresse 0041.

Exemples pratiques :

a. (0040) = 0C
Résultat : (0041) = 43 'C'

b. (0040) = 06
Résultat : (0041) = 36 '6'

Organigramme :



Programme source :

```
        LD      A,(40H)     ; LECTURE DONNEE
        CP      10          ; DONNEE = 10 OU PLUS
        JR      C,ASCZ
        ADD     A,'A'-'9'-1 ; OUI, AJOUTER DECALAGE POUR LETTRES
ASCZ:   ADD     A,'0'       ; AJOUTER DECALAGE POUR ASCII
        LD      (41H),A     ; RANGER RESULTAT ASCII
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3A	LD	A,(40H)
0001	40		
0002	00		
0003	FE	CP	10
0004	0A		
0005	38	JR	C,ASCZ
0006	02		
0007	C6	ADD	A,'A'-'9'-1
0008	07		
0009	C6	ASCZ: ADD	A,'0'
000A	30		
000B	32	LD	(41H),A
000C	41		
000D	00		
000E	76	HALT	

Dans ce programme, l'idée de base est d'ajouter le caractère ASCII 0 à tous les chiffres hexadécimaux. Cette addition effectue le conversion correcte des chiffres décimaux ; il faut cependant tenir compte de la coupure entre les caractères ASCII 9 (39 hexa) et A (41 hexa). Ce décalage doit être ajouté aux chiffres non décimaux A, B, C, D, E et F. Ceci est réalisé par l'instruction ADD A qui ajoute le décalage 'A' – '9' – 1 au contenu de l'accumulateur. Pouvez-vous expliquer pourquoi le décalage est 'A' – '9' – 1 ?
Il faut remarquer que les arguments de l'addition figurent sous forme ASCII dans le programme en langage assembleur (des apostrophes entourent le caractère ou la chaîne de caractères ASCII). Le décalage est conservé sous la forme d'une expression arithmétique pour apparaître aussi clairement que possible sur le listing du programme assembleur. La perte de temps résultant est minime compte tenu du gain énorme de clarté du programme.
Cette procédure pourrait servir à de nombreux programmes ; par exemple, à des programmes de moniteur qui effectuent la conversion de chiffres hexadécimaux en ASCII pour afficher le contenu hexadécimal d'emplacements mémoire sur un écran vidéo ou une imprimante ASCII.
Une autre méthode de conversion (plus rapide) qui ne nécessite aucun saut conditionnel, fait l'objet du programme suivant, décrit par Allison dans *Computer* magazine.

```
LD      A,(40H)     ; LECTURE CHIFFRE HEXA
ADD     A,90H       ; RETENUE SI >9
DAA
ADC     A,40H       ; AJOUTER 1 ET DECALER SI LETTRE
DAA
LD      (41H),A     ; RANGER CHIFFRE ASCII
HALT
```

Essayez ce programme avec quelques chiffres. Pouvez-vous expliquer pourquoi il marche ?

Décimal-affichage 7 segments

But : convertir le contenu de l'adresse 0040 en code affichage 7-segments à l'adresse 0042. Si l'adresse 0040 ne contient pas un chiffre décimal, mettre à zéro l'adresse 0042. Table 7-segments : la table suivante peut servir à convertir en code 7-segments les nombres décimaux. Le code 7-segments est organisé ainsi : le bit le plus significatif est toujours à 0 suivi du code (1 : segment allumé, 0 : segment éteint) pour les segments g, f, e, d, c, b et a (voir figure 7.1).

Digit	Code
0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7D
7	07
8	7F
9	6F

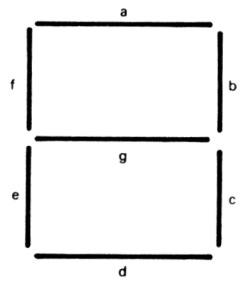


Figure 7.1. — Convention de l'affichage 7-segments.

On peut noter que la table utilise 7D pour 6 plutôt que 7C (segment supérieur éteint) pour éviter une confusion avec le segment inférieur b, et 6F pour 9 plutôt que 67 (segment inférieur éteint) sans raison particulière.

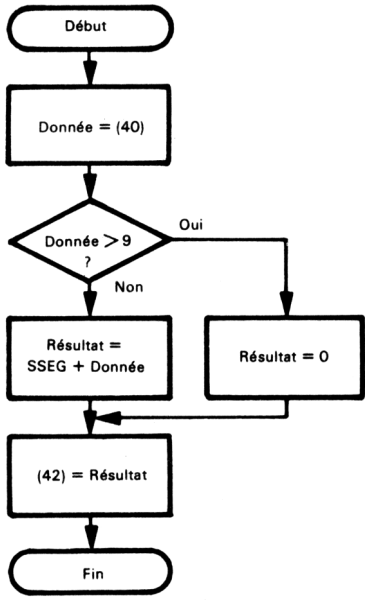
Exemples pratiques :

a. (0040) = 03
Résultat : (0042) = 4F
b. (0040) = 28
Résultat : (0042) = 00

Organigramme : (voir page 277)

Note : l'addition de l'adresse de base SSEG et de l'index (DONNEE) donne l'adresse qui contient la solution.

Organigramme :



Programme source :

```
        LD      B,0         ; LECTURE CODE ERREUR POUR AFFICHAGE BLANC
        LD      A,(40H)     ; LECTURE DONNEE
        CP      10          ; DONNEE CHIFFRE DECIMAL ?
        JR      NC,FINI     ; NON, ERREUR CODE
        LD      L,A         ; OUI, DONNEE DANS INDEX 16 BITS
        LD      H,0
        LD      DE,SSEG     ; LECTURE ADRESSE BASE DE TABLE 7 SEGMENTS
        ADD     HL,DE       ; INDEXATION ELEMENT
        LD      B,(HL)      ; LECTURE CODE 7 SEGMENTS DANS TABLE
FINI :  LD      A,B         ; SAUVEGARDE CODE 7 SEGMENTS OU ERREUR
        LD      (42H),A
        HALT
        ORG     20H         ; TABLE CODES 7 SEGMENTS
SSEG:   DEFB    3FH
        DEFB    06H
        DEFB    5BH
        DEFB    4FH
        DEFB    66H
        DEFB    6DH
        DEFB    7DH
        DEFB    07H
        DEFB    7FH
        DEFB    6FH
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	06	LD	B,0
0001	00		
0002	3A	LD	A,(40H)
0003	40		
0004	00		
0005	FE	CP	10
0006	0A		
0007	30	JR	NC,FINI
0008	08		
0009	6F	LD	L,A
000A	26	LD	H,0
000B	00		
000C	11	LD	DE,SSEG
000D	20		
000E	00		
000F	19	ADD	HL,DE
0010	46	LD	B,(HL)
0011	78	FINI : LD	A,B
0012	32	LD	(42H),A
0013	42		
0014	00		
0015	76	HALT	
0020	3F	SSEG: DEFB	3FH
0021	06	DEFB	06H
0022	5B	DEFB	5BH
0023	4F	DEFB	4FH
0024	66	DEFB	66H
0025	6D	DEFB	6DH
0026	7D	DEFB	7DH
0027	07	DEFB	07H
0028	7F	DEFB	7FH
0029	6F	DEFB	6FH

Ce programme calcule l'adresse mémoire du code voulu en ajoutant l'index (c'est-à-dire le chiffre à effectuer) à l'adresse de base de la table des codes 7-segments. Il s'agit d'une procédure de consultation de table.

La pseudo-opération du langage assembleur DEFB (définition d'octet) met une donnée constante dans la mémoire programme. Cette donnée peut représenter des tables, des en-têtes, des messages d'erreur, des messages principaux, des caractères formattés, des niveaux, etc. L'étiquette de la pseudo-opération DEFB a la valeur de l'adresse où se trouve l'octet de la donnée.
On se sert souvent de tables pour des conversions de codes plus complexes que celle de l'exemple précédent. Ces tables sont organisées de façon à contenir tous les résultats en fonction des données en entrée avec la convention suivante : la première entrée est le code correspondant au nombre zéro.
L'affichage 7-segments est un moyen de reconnaître les chiffres décimaux, peu de lettres et quelques autres caractères.
L'affichage 7 segments sur calculateur est peu coûteux, facile à faire et n'utilise que peu de puissance. Cependant, les chiffres codés 7-segments sont un peu difficiles à lire.
L'assembleur met simplement les données de la table en mémoire. On peut remarquer qu'une pseudo-opération DEFB remplit un octet de mémoire. Il reste de la place mémoire entre le programme et la table pour permettre des ajouts ou des corrections ultérieurs.

UTILISATION DES REGISTRES D'INDEX DU Z80

Une autre solution consiste à utiliser les registres d'index du Z80, par exemple IX. Le programmeur doit bien connaître les caractéristiques suivantes des registres d'index du Z80.

1) Le déplacement fixe en mémoire programme est de 8 bits seulement et ne peut en conséquence contenir une adresse mémoire complète. Il peut servir soit à un déplacement court, soit à contenir les 8 bits de poids faible d'une adresse mémoire.

2) Les registres d'index font 16 bits. On peut charger indifféremment IX ou IY à partir de la mémoire de la même façon qu'un double registre — à partir de 2 adresses consécutives, les 8 bits de poids faible à l'adresse la plus basse.

3) Toutes les opérations concernant les registres d'index prennent un peu plus de temps et de mémoire car un octet du code opératoire sert uniquement à indiquer l'emploi des registres d'index.
Le programme suivant utilise le registre IX pour consulter la table :

Programme source :

```
        LD      B,0          ; LECTURE CODE ERREUR POUR AFFICHAGE BLANC
        LD      A,(40H)      ; LECTURE DONNEE
        CP      10           ; DONNEE CHIFFRE DECIMAL ?
        JR      NC,FINI      ; NON, CODE ERREUR
        LD      HL,41H       ; OUI, SAUVEGARDE NUMERO PAGE DE TABLE
        LD      (HL),0
        LD      IX,(40H)     ; LECTURE DEPLACEMENT TABLE
        LD      B,(IX+SSEG)  ; LECTURE CODE 7 SEGMENTS EN TABLE
FINI :  LD      A,B          ; SAUVEGARDE CODE 7 SEGMENTS OU ERREUR
        LD      (42H),A
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	06	LD	B,0
0001	00		
0002	3A	LD	A,(40H)
0003	40		
0004	00		
0005	FE	CP	10
0006	0A		
0007	30	JR	NC,FINI
0008	0C		
0009	21	LD	HL,41H
000A	41		
000B	00		
000C	36	LD	(HL),0
000D	00		
000E	DD	LD	IX,(40H)
000F	2A		
0010	40		
0011	00		
0012	DD	LD	B,(IX+SSEG)
0013	46		
0014	20		
0015	78	FINI : LD	A,B
0016	32	LD	(42H),A
0017	42		
0018	00		
0019	76	HALT	

L'instruction de chargement indexée LD B,(IX + SSEG) ajoute l'index (c'est-à-dire le chiffre à afficher) à la base de la table 7-segments pour obtenir l'adresse du code voulu. Il convient de remarquer l'organisation spéciale du registre d'index 16 bits : les 8 bits de poids faible représentent la donnée et les 8 bits de poids fort représentent les bits de poids de l'adresse de début de table. Un tel arrangement est rendu nécessaire du fait du déplacement inhérent à l'instruction indexée : ce déplacement est seulement de 8 bits et ne peut en conséquence contenir que les 8 bits de poids faible du début de l'adresse de la table. Un programme plus général permettrait que la table soit mise n'importe où en mémoire. Si l'adresse de début de table est PSSEG (8 bits les plus significatifs) et MSSEG (8 bits les moins significatifs), l'instruction LD (HL),0 doit être remplacée par LD (HL),PSSEG. Pourquoi ce changement est-il nécessaire ?
Il faut noter que toutes les opérations concernant le registre d'index IX comportent un code opératoire sur 2 octets et que le premier octet est DD.

TRANSFERT DE DONNEES A L'INTÉRIEUR D'UN BLOC

Cet usage n'est pas vraiment adapté aux registres d'index. De tels registres deviennent vraiment indispensables lorsque l'on doit accéder à plusieurs données d'un bloc. Le bloc peut contenir les caractéristiques d'un message, les paramètres d'une équation, l'état en cours d'un processus ou d'une machine ou les données pour un écran vidéo.

On peut, par exemple, prendre le contenu de la douzième adresse du bloc et le transférer à la vingtième adresse en considérant que l'adresse de début de bloc est rangée en PTR et PTR + 1. C'est ce qui est fait dans chacun des programmes suivants :

1) en utilisant DE et HL

```
LD      DE,(PTR)     ; LECTURE ADRESSE DEBUT
LD      HL,12        ; CALCUL ADRESSE SOURCE
ADD     HL,DE
LD      A,(HL)       ; LECTURE DONNEE A PARTIR DE SOURCE
LD      HL,20        ; CALCUL ADRESSE DESTINATION
ADD     HL,DE
LD      (HL),A       ; TRANSFERT DONNEE A DESTINATION
```

2) en utilisant IX

```
LD      IX,(PTR)     ; LECTURE ADRESSE DEBUT
LD      A,(IX+12)    ; LECTURE DONNEE A PARTIR DE SOURCE
LD      (IX+20),A    ; TRANSFERT DONNEE A DESTINATION
```

Le programme qui utilise les registres d'index est de loin le plus court et le plus clair. Sa seule faiblesse tient à la taille des déplacements qui ne doit pas excéder 8 bits.

ASCII-Décimal

But : effectuer la conversion d'un caractère ASCII contenu à l'adresse 0040 en un chiffre décimal et ranger le résultat à l'adresse 0041. Si le contenu de l'adresse 0040 n'est pas la représentation ASCII d'un nombre décimal, mettre FF (hexa) en 0041.

Exemples pratiques :

a. (0040) = 37 '7'
Résultat : (0041) = 07

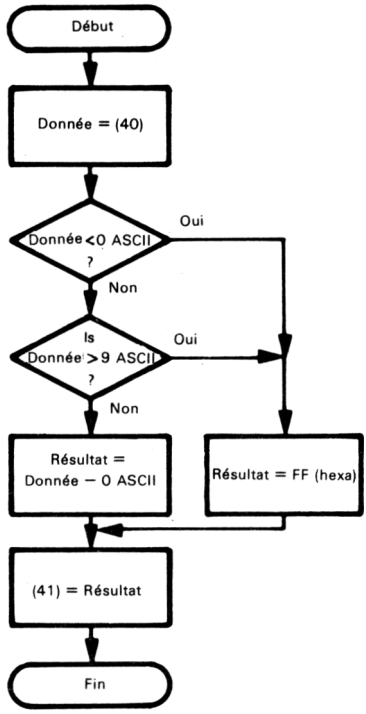
b. (0040) = 55
Résultat : (0041) = FF

Organigramme : (voir page 282)

Programme source :

```
        LD      B,0FFH      ; LECTURE MARQUE ERREUR
        LD      A,(40H)     ; LECTURE DONNEE
        SUB     '0'         ; DONNEE<0 ASCII ?
        JR      C,FINI      ; OUI, ERREUR
        CP      '9'+1       ; DONNEE>9 ASCII ?
        JR      NC,FINI     ; OUI, ERREUR
        LD      B,A         ; SAUVEGARDE CHIFFRE CORRECT
FINI :  LD      A,B         ; SAUVEGARDE CHIFFRE OU ERREUR
        LD      (41H),A
        HALT
```

Organigramme



Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	06	LD	B,OFFH
0001	FF		
0002	3A	LD	A,(40H)
0003	40		
0004	00		
0005	D6	SUB	'0'
0006	30		
0007	38	JR	C,FINI
0008	05		
0009	FE	CP	'9'+1
000A	3A		
000B	30	JR	NC,FINI
000C	01		
000D	47	LD	B,A
000E	78	FINI : LD	A,B
000F	32	LD	(41H),A
0010	41		
0011	00		
0012	76	HALT	

Ce programme traite les caractères codés en ASCII de la même façon que des nombres ordinaires. On peut remarquer que les chiffres décimaux et les lettres constituent des groupes de codes consécutifs. Des chaînes de lettres (comme les noms) peuvent être rangées dans l'ordre alphabétique si l'on met leurs représentations ASCII en ordre numérique croissant (B ASCII = A ASCII + 1, par exemple).
La soustraction de n'importe quel chiffre décimal ASCII par zéro ASCII (30 hexa) donne la représentation DCB (décimal codé binaire) de ce chiffre.
La conversion ASCII-décimal est nécessaire lorsque les nombres décimaux sont rentrés à partir d'un périphérique ASCII comme un télétype ou un clavier-vidéo.
L'idée fondamentale du programme est de savoir si le caractère est compris inclusivement entre 0 et 9 ASCII. Si oui, le caractère est un chiffre décimal ASCII, puisque les chiffres constituent une suite, Il peut alors être converti en décimal par la simple soustraction de 30 hexa (ASCII 0). (Ex. : 7 ASCII − 0 ASCII = 37 − 30 = 7.)

Remarque : une comparaison est effectuée par la soustraction effective (SUB '0') puisque la soustraction est nécessaire pour la conversion d'ASCII en décimal. L'autre comparaison est effectuée par la soustraction implicite (CP '9' + 1) puisque le résultat final est contenu dans l'accumulateur dans le cas d'un nombre correct.

DCB (décimal codé binaire) - Binaire

But : convertir 2 chiffres DCB rangés aux adresses 0040 et 0041 en un nombre décimal à l'adresse 0042. Le chiffre DCB le plus significatif se trouve à l'adresse 0040.

Exemples pratiques :

a.
(0040) = 02
(0041) = 09
Résultat : (0042) = 1D (hex) = 29 (décimal)

b.
(0040) = 07
(0041) = 01
Résultat : (0042) = 47 (hex) = 71 (décimal)

Remarque : il n'y a pas d'organigramme ici puisque le programme multiplie par 10 le chiffre le plus significatif en utilisant simplement la formule 10x = 8x + 2x. La multiplication par 2 s'obtient à l'aide d'un décalage arithmétique à gauche et celle par 8 à l'aide de 3 décalages semblables.

Programme source :

```
LD      HL,40H      ; LECTURE DU CHIFFRE LE PLUS SIGNIFICATIF (CPS)
LD      A,(HL)
ADD     A,A         ; CPS x 2
LD      B,A         ; SAUVEGARDE CPS x 2
ADD     A,A         ; CPS x 4
ADD     A,A         ; CPS x 8
ADD     A,B         ; CPS x 10
INC     HL          ; POINTE SUR CHIFFRE LE MOINS SIGNIFICATIF (CMS)
ADD     A,(HL)      ; AJOUT POUR EQUIVALENT BINAIRE
INC     HL
LD      (HL),A      ; RANGE EQUIVALENT BINAIRE
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	7E	LD	A,(HL)
0004	87	ADD	A,A
0005	47	LD	B,A
0006	87	ADD	A,A
0007	87	ADD	A,A
0008	80	ADD	A,B
0009	23	INC	HL
000A	86	ADD	A,(HL)
000B	23	INC	HL
000C	77	LD	(HL),A
000D	76	HALT	

Les entrées DCB sont converties en binaire pour optimiser le rangement en mémoire et simplifier les calculs. Cependant la conversion peut réduire quelques avantages du stockage binaire et du calcul arithmétique.
Ce programme multiplie par 10 le chiffre DCB à l'adresse 0040 en répétant des multiplications[2]. Notez que l'instruction ADD A,A multiplie le contenu de l'accumulateur par 2. Ceci permet de multiplier par de petits nombres décimaux le contenu de l'accumulateur en peu d'instructions. Comment utiliseriez-vous cette procédure pour effectuer une multiplication par 16 ? par 12 ? par 7 ?
Les nombres DCB nécessitent environ 20 % de plus de rangement mémoire que les nombres binaires. La représentation DCB de 0 à 99 nécessite 12 bits alors qu'en binaire seulement 10 bits sont nécessaires (puisque $2^{10} = 1024 \cong 1000$).

Conversion d'un nombre binaire en une chaîne ASCII

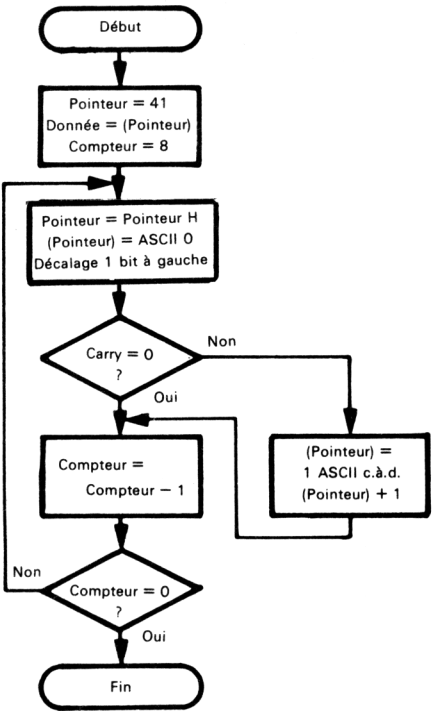
But : convertir le nombre binaire 8 bits rangé à l'adresse 0041 en 8 caractères ASCII (soit 0, soit 1 ASCII) aux adresses 0042, 0043,... 0049 (le bit le plus significatif en 0042).

Exemple pratique :

(0041) = D2 = 11010010

Résultat :
(0042) = 31 '1'
(0043) = 31 '1'
(0044) = 30 '0'
(0045) = 31 '1'
(0046) = 30 '0'
(0047) = 30 '0'
(0048) = 31 '1'
(0049) = 30 '0'

Organigramme :



Programme source :

```
        LD      HL,41H
        LD      A,(HL)          ; LECTURE DONNEE
        LD      B,8             ; COMPTEUR = NOMBRE BITS DU MOT
        LD      C,'0'           ; LECTURE 0 ASCII POUR METTRE DANS CHAINE
CONV:   INC     HL
        LD      (HL),C          ; METTRE 0 ASCII EN CHAINE
        RLA                     ; BIT SUIVANT DE DONNEE A 1 ?
        JR      NC,COMPT        ; OUI, METTRE 1 ASCII EN CHAINE
        INC     (HL)
COMPT : DJNZ    CONV
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	7E		LD	A,(HL)
0004	06		LD	B,8
0005	08			
0006	0E		LD	C,'0'
0007	30			
0008	23	CONV:	INC	HL
0009	71		LD	(HL),C
000A	17		RLA	
000B	30		JR	NC,COMPT
000C	01			
000D	34		INC	(HL)
000E	10	COMPT:	DJNZ	CONV
000F	F8			
0010	76		HALT	

Les chiffres ASCII constituent une suite telle que ASCII 1 = ASCII 0 + 1. N'oubliez pas que les registres du Z80 ont des utilisations spéciales. Il est préférable de mettre un compteur de boucle dans le registre B pour utiliser l'instruction DJNZ.
Il faut bien faire la différence entre l'instruction INC HL, qui ajoute 1 au contenu 16 bits du double registre HL et INC (HL), qui ajoute 1 au contenu 8 bits de l'emplacement mémoire adressé par le double registre HL.
La conversion du binaire en ASCII est nécessaire lorsque les nombres sont imprimés en binaire sur un périphérique ASCII.
La conversion en ASCII consiste simplement à additionner le caractère 0 ASCII (30 hexa).

EXERCICES

1) Conversion d'ASCII en hexa

But : faire la conversion du contenu de l'adresse 0040 en un chiffre hexadécimal ; ranger le résultat à l'adresse 0041. Considérer que l'adresse 0040 contient la représentation ASCII d'un chiffre hexadécimal (7 bits avec le bit 0 comme bit le plus significatif).

Exemples pratiques :

a. (0040) = 43 'C'
Résultat : (0041) = 0C

b. (0040) = 36 '6'
Résultat : (0041) = 06

2) Conversion du code 7-segments en décimal

But : faire la conversion du contenu de l'adresse 0040 à partir d'un code 7-segments en un nombre décimal à l'adresse 0041. Si l'adresse 0040 ne contient pas de code correct 7-segments, mettre à FF (hexa) la mémoire d'adresse 0041. Utiliser la table 7-segments donnée à l'exemple décimal-affichage 7-segments et essayer des codes appropriés.

Exemples pratiques :

a. (0040) = 4F
Résultat : (0041) = 03

b. (0040) = 28
Résultat : (0041) = FF

3) Conversion de décimal en ASCII

But : faire la conversion d'un chiffre décimal rangé à l'adresse 0040 en un caractère ASCII et mettre le résultat à l'adresse 0041. Si le nombre rangé en 0040 n'est pas un chiffre décimal, mettre à blanc ASCII (20 hexa) le contenu de l'adresse 0041.

Exemples pratiques :

a. (0040) = 07
Résultat : (0041) = 37 '7'

b. (0040) = 55
Résultat : (0041) = 20 SP

4) Conversion de binaire en DCB (décimal codé binaire)

But : faire la conversion du contenu de l'adresse 0040 en 2 chiffres DCB aux adresses 0041 et 0042 (chiffre le plus significatif en 0041). Le nombre rangé à l'adresse 0040 est non signé et inférieur à 100.

Exemples pratiques :

a. (0040) = 1D (29 décimal)
Résultat : (0041) = 02
(0042) = 09

b. (0040) = 47 (71 décimal)
Résultat : (0041) = 07
(0042) = 01

5) Conversion d'une chaîne ASCII en un nombre binaire

But : faire la conversion de 8 caractères ASCII rangés de l'adresse 0042 à 0049 en un nombre binaire 8 bits à l'adresse 0041 (bit le plus significatif en 0042). Mettre à zéro la mémoire d'adresse 0040 si tous les caractères ASCII sont soit 1 ASCII, soit 0 ASCII et la mettre à FF sinon.

Exemples pratiques :

a.

	(0042)	=	31	'1'
	(0043)	=	31	'1'
	(0044)	=	30	'0'
	(0045)	=	31	'1'
	(0046)	=	30	'0'
	(0047)	=	30	'0'
	(0048)	=	31	'1'
	(0049)	=	30	'0'
Résultat :	(0041)	=	D2	
	(0040)	=	00	

b. identique à 'a' sauf

	(0045)	=	37	'7'
Résultat :	(0040)	=	FF	

RÉFÉRENCES

1. Allison, D.R., « A Design Philosophy for Microcomputer Architectures », *Computer.* February 1977, pp. 35-41. Voici un excellent article que nous recommandons vivement.
2. D'autres méthodes de conversion de DCB en binaire sont proposées dans J.A. Tabb et M.L. Roginsky, « Microprocessor Algorithms Make BCD-Binary Conversions Super-fast », *EDN,* January 5, 1977, pp. 46-50 et dans J.B. Peatman, *Microcomputer-based Design,* McGraw-Hill, New York, 1977, pp. 400-406.

CHAPITRE 8

PROBLÈMES ARITHMÉTIQUES

L'arithmétique dans les applications sur microprocesseurs se ramène principalement à des manipulations décimales ou binaires sur plusieurs octets. Une correction décimale (« decimal adjust ») ou autre opération de ce genre est fréquemment la seule instruction arithmétique fournie en dehors des opérations élémentaires : l'addition et la soustraction. Les autres opérations arithmétiques doivent être composées à partir de combinaisons de plusieurs instructions.
L'arithmétique binaire portant sur plusieurs mots (« multiple-précision ») nécessite de simples répétitions des instructions élémentaires sur un octet. Le bit de retenue (Carry) transfère l'information entre les octets. L'addition et la soustraction avec retenue utilisent l'information résultant des précédentes opérations arithmétiques. Il ne faut pas oublier de remettre à zéro le bit de retenue avant d'exécuter des opérations sur les premiers octets (il est bien évident qu'il n'y a pas de retenue sur les octets les moins significatifs).
L'arithmétique décimale est une tâche suffisamment courante sur les microprocesseurs pour que la plupart d'entre eux disposent d'instructions spéciales à cet effet. Ces instructions peuvent soit exécuter directement des opérations décimales, soit transposer les résultats d'opérations binaires sous la forme décimale adéquate. L'arithmétique décimale est fondamentale dans des applications telles que les caisses enregistreuses, les ordinateurs, les distributeurs de billets, les terminaux bancaires, etc.
La multiplication (respectivement la division) peut être réalisée par une série successive d'additions (respectivement de soustractions), de la même façon qu'elle est faite à la main. Des opérations sur deux octets sont nécessaires puisqu'une multiplication donne un résultat deux fois plus long que les opérandes, tandis qu'une division réduit la longueur du résultat de façon similaire.
Les multiplications et les divisions effectuées par logiciel prennent beaucoup de temps machine du fait de la répétition des opérations arithmétiques et de décalage nécessaires. Bien sûr, la multiplication ou la division par une puissance de 2 est simple car on peut les réaliser avec un nombre approprié de décalages à gauche ou à droite.

EXEMPLES

Addition en multiple précision

But : additionner 2 nombres binaires sur 2 octets. La longueur des nombres (en octets) se trouve à l'adresse 0040. Les nombres eux-mêmes commencent (les bits les moins significatifs d'abord) aux adresses 0041 et 0051, respectivement, et la somme prend la place du nombre qui commence en 0041.

Exemples pratiques :

(0040) = 04

(0041) = C3
(0042) = A7
(0043) = 5B
(0044) = 2F

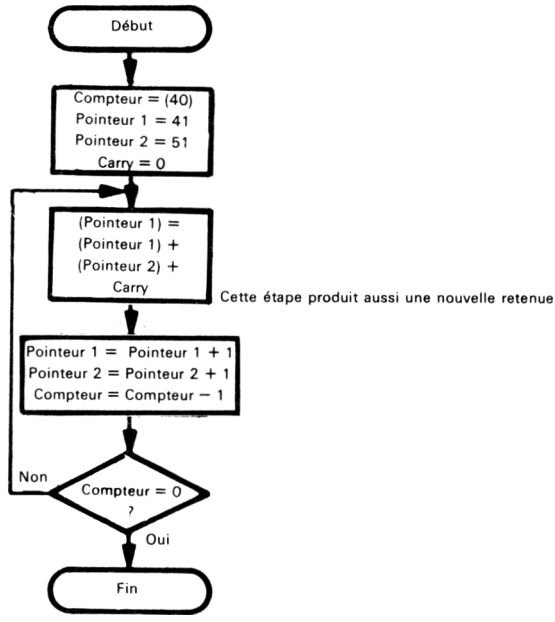
(0051) = B8
(0052) = 35
(0053) = DF
(0054) = 14

Résultat : (0041) = 7B
(0042) = DD
(0043) = 3A
(0044) = 44

Ce qui donne,

```
   2F5BA7C3
+  14DF35B8
-----------
   443ADD7B
```

Organigramme :



Programme source :

```
        LD      HL,40H   ; COMPTEUR = NOMBRE D'OCTETS
        LD      B,(HL)
        INC     HL       ; POINTEUR 1 = 1er MOT CHAINE 1
        LD      DE,51H   ; POINTEUR 2 = 1er MOT CHAINE 2
        AND     A        ; RAZ RETENUE AVANT DE COMMENCER
SOM :   LD      A,(DE)   ; CHARGER OCTET, CHAINE 2
        ADC     A,(HL)   ; AJOUTER OCTET CHAINE 1
        LD      (HL),A   ; RANGEMENT RESULTAT DANS CHAINE 1
        INC     DE
        INC     HL
        DJNZ    SOM
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	46	LD	B,(HL)
0004	23	INC	HL
0005	11	LD	DE,51H
0006	51		
0007	00		
0008	A7	AND	A
0009	1A	SOM : LD	A,(DE)
000A	8E	ADC	A,(HL)
000B	77	LD	(HL),A
000C	13	INC	DE
000D	23	INC	HL
000E	10	DJNZ	SOM
000F	F9		
0010	76	HALT	

L'adresse relative pour l'instruction DJNZ SOM est :

$$\begin{array}{r} 09 \\ -10 \end{array} = \begin{array}{r} 09 \\ +F0 \\ \hline F9 \end{array}$$

L'instruction AND A sert à remettre à zéro le bit de retenue. N'importe qu'elle autre opération logique aurait le même effet. L'indicateur de retenue doit être remis à zéro puisque l'addition des octets de poids faible n'utilise pas de retenue.
L'instruction ADC, (« Add with Carry »), englobe dans l'addition la retenue des octets précédents. ADC est la seule instruction de la boucle qui modifie l'indicateur de retenue. Ce n'est le cas, ni pour l'instruction INC, ni pour l'instruction DJNZ.
Les deux pointeurs, celui contenu dans le double registre DE et celui dans HL doivent être incrémentés à chaque itération.

PRÉCISION DÉCIMALE EN BINAIRE

Cette procédure peut additionner des nombres binaires jusqu'à 256 octets de long. On peut noter que dix bits binaires correspondent à 3 chiffres décimaux, puisque $2^{10} = 1024 \cong 1000$. Ainsi, on peut calculer le nombre de bits nécessaires pour donner une certaine précision aux nombres décimaux. Par exemple, un nombre à 10 chiffres significatifs nécessite :

$$(10\,\text{chiffres}) \times \left(\frac{10\,\text{bits}}{3\,\text{chiffres}}\right) = 33\,\text{bits}$$

INSTRUCTIONS DE TRANSFERT EN BLOC

Si l'on se contentait de transférer une donnée d'une place mémoire à une autre, sans la traiter, on pourrait utiliser la puissante instruction du Z80 : LDIR, instruction de transfert en bloc. Cette seule instruction transfère un octet de donnée depuis l'adresse contenue dans HL vers l'adresse contenue en DE, incrémente les pointeurs dans HL et DE et décrémente le compteur d'octets dans BC. Elle répète l'opération de transfert jusqu'à ce que BC atteigne zéro. L'instruction LDI a le même effet sans le facteur répétitif ; les instructions LDD et LDDR représentent des transferts respectivement non répétitifs et répétitifs, qui décrémentent les pointeurs au lieu de les incrémenter.
Le programme qui suit transfère un nombre fixé d'octets (LONG) d'une place mémoire (pointée par PTR1) à une autre place mémoire (pointée par PTR2).

Transfert de blocs

But : transférer un bloc de données de longueur BC depuis l'adresse pointée par HL vers l'adresse pointée par DE.

Exemple pratique :

```
                (HL) = 40
                (DE) = 50
                (BC) =  3

              (0040) = 31
              (0041) = 32
              (0042) = 33

              (0050) =  0
              (0051) =  0
              (0052) =  0

Résultat :    (0050) = 31
              (0051) = 32
              (0052) = 33
```

Programme source :

```
LD      BC,LONG     ; COMPTEUR = NOMBRE D'OCTETS DE TRANSFERT
LD      HL,PTR1     ; POINTEUR 1 = DEBUT ZONE DONNEE SOURCE
LD      DE,PTR2     ; POINTEUR 2 = DEBUT ZONE DONNEE
                    ;  DESTINATION
LDIR
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	01	LD	BC,LONG
0001			
0002	LONG		
0003	21	LD	HL,PTR1
0004			
0005	PTR1		
0006	11	LD	DE,PTR2
0007			
0008	PTR2		
0009	ED	LDIR	
000A	B0		
000B	76	HALT	

Essayez de réaliser le même programme sans utiliser l'instruction LDIR. Calculez le nombre d'octets mémoire et de cycles d'horloge que chacun d'eux utilise.

Addition décimale

But : additionner 2 nombres décimaux (DCB) sur plusieurs octets. La longueur des nombres se trouve à l'adresse 0040, les nombres eux-mêmes commencent (bits de poids faible d'abord) aux adresses 0041 et 0051, respectivement, et la somme remplace le nombre commençant en 0041.

Exemple pratique :

(0040) = 04

(0041) = 85
(0042) = 19
(0043) = 70
(0044) = 36

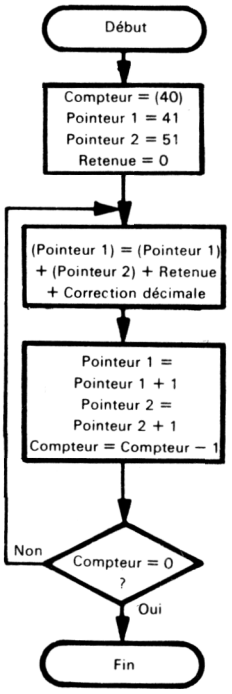
(0051) = 59
(0052) = 34
(0053) = 66
(0054) = 12

Résultat : (0041) = 44
(0042) = 54
(0043) = 36
(0044) = 49

ce qui donne :

```
  36701985
+ 12663459
----------
  49365444
```

Organigramme :



(Cette étape produit aussi une nouvelle retenue)

Programme source :

```
        LD      HL,40H
        LD      B,(HL)      ; COMPTEUR = LONGUEUR CHAINES (OCTETS)
        INC     HL          ; POINTEUR 1 = 1er OCTET CHAINE 1
        LD      DE,51H      ; POINTEUR 2 = 1er OCTET CHAINE 2
        AND     A           ; RAZ RETENUE AVANT DE COMMENCER
DECAD:  LD      A,(DE)      ; LECTURE 2 CHIFFRES DECIMAUX CHAINE 2
        ADC     A,(HL)      ; SOMME 2 CHIFFRES CHAINE 1
        DAA                 ; CORRECTION DECIMALE ("DECIMAL ADJUST")
        LD      (HL),A      ; RANGER RESULTAT DANS CHAINE 1
        INC     DE
        INC     HL
        DJNZ    DECAD
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	21		LD	HL,40H
0001	40			
0002	00			
0003	46		LD	B,(HL)
0004	23		INC	HL
0005	11		LD	DE,51H
0006	51			
0007	00			
0008	A7		AND	A
0009	1A	DECAD:	LD	A,(DE)
000A	8E		ADC	A,(HL)
000B	27		DAA	
000C	77		LD	(HL),A
000D	13		INC	DE
000E	23		INC	HL
000F	10		DJNZ	DECAD
0010	F8			
0011	76		HALT	

AJUSTEMENT DÉCIMAL L'instruction d'ajustement décimal (DAA) utilise les indicateurs de retenue C et de demi-retenue H pour traiter les situations suivantes :

1) La somme de 2 chiffres est comprise entre 10 et 15, bornes incluses. Dans ce cas, on doit ajouter six à la somme pour obtenir le bon résultat, c'est-à-dire :

```
   0101  (5)
 + 1000  (8)
 ------
   1101  (D)
 + 0110
 ------
0001 0011  (13 DCB, ce qui est correct)
```

2) La somme de 2 chiffres donne 16 ou plus. Dans ce cas, le résultat est effectivement un nombre DCB mais inférieur de six au résultat attendu, c'est-à-dire :

```
   1000  (8)
 + 1001  (9)
 ------
0001 0001  (BCD 11)
 + 0110
 ------
0001 0111  (17 DCB, ce qui est correct)
```

Six doit être ajouté dans les 2 cas. Cependant, on peut distinguer le cas 1 par le fait que la somme n'est pas un nombre DCB, il est compris entre 10 et 15 (ou A et F hexadécimal). On distingue le cas 2 uniquement par le fait que la retenue (chiffre le plus significatif) ou la 1/2 retenue (chiffre le moins significatif) est égale à 1, puisque le résultat est un nombre DCB correct. DAA est la seule instruction qui utilise la 1/2 retenue. On peut noter que l'instruction DAA opère uniquement sur l'accumulateur.

INDICATEUR DE SOUSTRACTION (N)

Le microprocesseur Z80 dispose également d'un indicateur qui permet de distinguer les instructions d'addition (ADD, ADC) des instructions de soustraction (SUB, SBC). Cet indicateur, appelé indicateur de soustraction ou indicateur N est remis à zéro par toutes les instructions d'addition et positionné à un par toutes les instructions de soustraction. La seule utilité de cet indicateur est de permettre à l'instruction DAA de passer correctement de l'addition binaire à l'addition DCB et de la soustraction binaire à la soustraction DCB. Les microprocesseurs 8080 et 8085 ne possèdent pas d'indicateur N et de ce fait leurs instructions DAA n'opèrent correctement qu'après addition.

On ne peut utiliser l'instruction DAA qu'après les instructions qui rangent leur résultat dans l'accumulateur et tiennent compte de la retenue, la 1/2 retenue, et de l'indicateur N. Ainsi, on ne peut pas utiliser DAA après les instructions INC (puisque INC ne tient pas compte de la retenue), DEC, ou n'importe quelle instruction double-mot qui range ses résultats dans les registres d'index ou le double registre HL.

PRÉCISION EN BINAIRE ET EN DCB

Cette procédure peut additionner des nombres décimaux (DCB) de n'importe quelle longueur. Ici, quatre bits binaires par chiffre décimal sont nécessaires, d'où la précision pour 10 chiffres :

$$10 \times 4 = 40 \text{ bits}$$

au lieu de 33 bits dans le cas binaire ; ce qui donne 5 octets au lieu de 4. Cette procédure décimale prend également un peu plus de temps par octet à cause de l'instruction supplémentaire DAA.

Multiplication binaire 8 bits

But : multiplier le nombre non signé 8 bits d'adresse 0040 par le nombre non signé 8 bits d'adresse 0041. Mettre les 8 bits de poids faible du résultat dans la mémoire d'adresse 0042 et les 8 bits de poids fort dans la mémoire d'adresse 0043.

Exemples pratiques :

a.
(0040) = 03
(0041) = 05
Résultat : (0042) = 0F
(0043) = 00
ou en décimal 3 x 5 = 15

b.
(0040) = 6F
(0041) = 61
Résultat : (0042) = 0F
(0043) = 2A
ou 111 x 97 = 10.767

On peut effectuer une multiplication sur calculateur de la même façon que l'on effectue une longue multiplication à la main. Puisque les nombres sont binaires, on n'aura à traiter que des multiplications par zéro ou par 1 ; la multiplication par zéro donne évidem-

ment zéro pour résultat, alors que la multiplication par 1 donne pour résultat le premier nombre de l'opération (multiplicande). De cette façon, chaque étape de la multiplication binaire peut être réduite à l'opération suivante :

ALGORITHME DE MULTIPLICATION

Si pour le multiplicateur, le bit en cours est 1, ajouter le multiplicande au produit partiel.
Le seul problème qui reste est de s'assurer que chaque résultat partiel est bien reporté correctement à chaque fois — c'est ce que réalisent les opérations suivantes :

1) Décaler d'un bit à gauche le multiplicateur pour que le bit à examiner soit chargé dans l'indicateur de retenue.
2) Décaler le produit d'un bit à gauche pour que l'addition suivante soit alignée correctement.
Voici le processus complet de la multiplication binaire[1] :

Etape 1 — Initialisation
Produit = 0
Compteur = 8
Etape 2 — Décaler le produit pour un alignement correct
Produit = 2 × Produit (bit le moins significatif = 0)
Etape 3 — Décaler le multiplicateur pour que le bit aille dans l'indicateur de retenue
Multiplicateur = 2 × Multiplicateur
Etape 4 — Ajouter le multiplicande au produit si l'indicateur de retenue est à 1.
Si retenue = 1, Produit = Produit + Multiplicande
Etape 5 — Décrémenter le compteur et tester s'il vaut zéro
Compteur = Compteur − 1
Si Compteur ≠ 0 aller à l'étape 2.

Dans le cas de l'exemple pratique b, où le multiplicateur vaut 61 (hexa) et le multiplicande vaut 6F (hexa) le processus donne :

Initialisation :

Produit	0000
Multiplicateur	61
Multiplicande	6F
Compteur	08

Après la première itération des étapes 2 à 5 :

Produit	0000
Multiplicateur	C2
Multiplicande	6F
Compteur	07
Retenue	0

Après la deuxième itération :

Produit	006F
Multiplicateur	84
Multiplicande	6F
Compteur	06
Retenue	1

Après la troisième itération :

Produit	014D
Multiplicateur	08
Multiplicande	6F
Compteur	05
Retenue	1

Après la quatrième itération :

Produit	029A
Multiplicateur	10
Multiplicande	6F
Compteur	04
Retenue	0

Après la cinquième itération :

Produit	0534
Multiplicateur	20
Multiplicande	6F
Compteur	03
Retenue	0

Après la sixième itération :

Produit	0A68
Multiplicateur	40
Multiplicande	6F
Compteur	02
Retenue	0

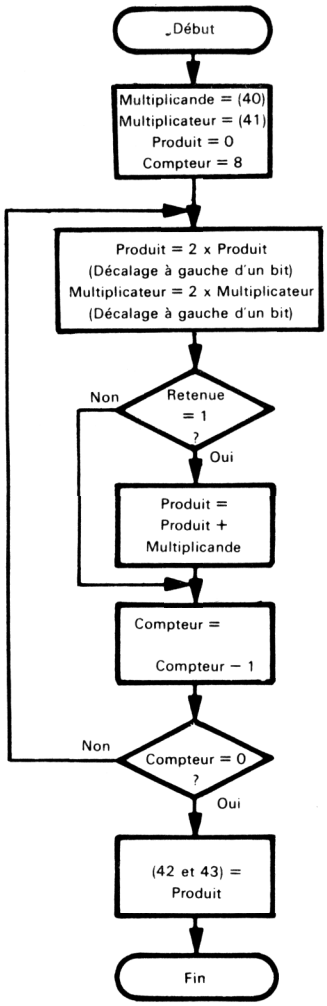
Après la septième itération :

Produit	14D0
Multiplicateur	80
Multiplicande	6F
Compteur	01
Retenue	0

Après la huitième itération :

Produit	2A0F
Multiplicateur	00
Multiplicande	6F
Compteur	00
Retenue	1

Organigramme :



Programme source :

```
        LD      HL,40H
        LD      E,(HL)      ; LIRE MULTIPLICANDE
        LD      D,0         ; EXTENSION SUR 16 BITS
        INC     HL
        LD      A,(HL)      ; LIRE MULTIPLICATEUR
        LD      HL,0        ; PRODUIT = ZERO (INITIALISATION)
        LD      B,8         ; COMPTEUR = LONGUEUR BITS MULTIPLICATEUR
MULT:   ADD     HL,HL       ; DECALAGE PRODUIT (1 BIT A GAUCHE)
        RLA                 ; DECALAGE MULTIPLICATEUR (1 BIT A GAUCHE)
        JR      NC,CHCNT    ; RETENUE = 1 ?
        ADD     HL,DE       ; OUI, AJOUT MULTICANDE AU PRODUIT
CHCNT:  DJNZ    MULT
        LD      (42H),HL    ; SAUVEGARDE PRODUIT
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	5E	LD	E,(HL)
0004	16	LD	D,0
0005	00		
0006	23	INC	HL
0007	7E	LD	A,(HL)
0008	21	LD	HL,0
0009	00		
000A	00		
000B	06	LD	B,8
000C	08		
000D	29	MULT: ADD	HL,HL
000E	17	RLA	
000F	30	JR	NC,TESTC
0010	01		
0011	19	ADD	HL,DE
0012	10	TESTC : DJNZ	MULT
0013	F9		
0014	22	LD	(42H),HL
0015	42		
0016	00		
0017	76	HALT	

Notons que le multiplicande doit être étendu à 16 bits, en mettant à zéro le registre D, afin qu'il puisse être ajouté au produit en utilisant l'instruction ADD HL,DE. L'instruction ADD HL,HL a le même effet qu'un décalage logique à gauche 16 bits. Dans ce programme, les instructions 16 bits du Z80 traitent des données plutôt que des adresses. L'instruction LD HL,0 sert à initialiser le produit ; l'instruction ADD HL,HL à effectuer un décalage logique à gauche 16 bits ; l'instruction ADD HL,DE à ajouter le multiplicande au produit partiel ; et l'instruction LD (42H),HL à ranger le résultat en mémoire. Il est important d'étendre sur 16 bits les quantités 8 bits à traiter (comme le multiplicande dans cet exemple). On ne peut pas utiliser les avantages du 16 bits à la

fois pour l'adressage et le traitement des données. Cependant, si l'on ne se sert pas des registres auxiliaires, on peut y sauvegarder le contenu des registres usuels et les retransférer ensuite à l'aide de l'instruction EXX. Cette instruction échange les contenus des doubles registres BC, DE et HL avec les contenus de leurs homologues auxiliaires (BC', DE', HL') et ceci en 4 cycles d'horloge seulement.
En dehors de son intérêt primordial pour les calculateurs et les caisses enregistreuses, la multiplication est une partie clé de la plupart des algorithmes de contrôle et de traitement de signal. La vitesse à laquelle les multiplications sont exécutées détermine l'utilité d'un microprocesseur dans le contrôle de processus, la détection et l'analyse du signal.
L'algorithme de multiplication prend entre 390 et 440 cycles d'horloge sur un microprocesseur Z80. Le temps précis dépend du nombre de bits à un du multiplicateur. D'autres algorithmes peuvent réduire quelque peu le temps moyen d'exécution, mais on estime à 400 cycles d'horloge le temps moyen d'exécution d'une multiplication effectuée par logiciel.

Division binaire sur 8 bits

But : diviser un nombre non signé 16 bits rangé aux adresses 0040 et 0041 (bits de poids fort en 0041) par le nombre 8 bits non signé à l'adresse 0042. Les nombres respectent les normes suivantes :
1) les bits les plus significatifs à la fois du dividende et du diviseur valent zéro ;
2) le nombre à l'adresse 0042 est plus grand que le nombre à l'adresse 0041, c'est-à-dire que le quotient sera un nombre 8 bits. Ranger le quotient à l'adresse 0043 et le reste à l'adresse 0044.

Exemples pratiques :

a.
(0040) = 40 (64 décimal)
(0041) = 00
(0042) = 08

Résultat : = (0043) = 08
(0044) = 00
c'est-à-dire, 64/8 = 8

b.
(0040) = 6D (12909 décimal)
(0041) = 32
(0042) = 47 (71 décimal)

Résultat : = (0043) = B5 (181 décimal)
(0044) = 3A (58 décimal)
c'est-à-dire, 12,909/71 = 181 avec un reste de 58

ALGORITHME DE DIVISION

On peut effectuer une division sur calculateur de la même façon qu'avec un stylo et du papier, c'est-à-dire en utilisant des soustractions d'essai. Puisque les nombres sont binaires, la seule question est de savoir si le bit du quotient est 0 ou 1, c'est-à-dire si l'on doit ou non soustraire le diviseur de ce qui reste du dividende. Il est possible de ramener chaque étape de la division binaire à l'opération suivante :

Si le diviseur peut être soustrait des 8 bits les plus significatifs du dividende sans retenue, le bit correspondant du quotient est 1, sinon il vaut 0.

Le seule problème qui reste est d'aligner correctement le dividende et le quotient. Ceci est effectué par le décalage logique d'un bit à gauche du dividende et du quotient avant chaque soustraction d'essai.
Le dividende et le quotient peuvent se partager un registre 16 bits, puisque la procédure met à zéro un bit du dividende chaque fois qu'elle calcule un bit du quotient.

Le processus complet de la division binaire est :

Etape 1 — Initialisation
Quotient = 0
Compteur = 8
Etape 2 — Décalage du dividende et du quotient pour un alignement correct
Dividende = 2 × Dividende
Quotient = 2 × Quotient
Etape 3 — Soustraction. S'il n'y a pas de retenue ajout de 1 au quotient :
Si les 8 BPS (Bits les Plus Significatifs) du dividende $\geq$ Diviseur alors
BPS du Dividende = BPS du Dividende – Diviseur
Quotient = Quotient + 1
Etape 4 — Décrémentation du compteur et test du compteur à zéro :
Compteur = Compteur – 1
Si Compteur $\neq$ 0, aller à l'étape 2
Reste = 8 BPS du Dividende

Dans le cas de l'exemple pratique b, où le dividende vaut 326D (hexa) et le diviseur 47 (hexa), le processus fonctionne ainsi :

Initialisation :

Dividence	326D
Diviseur	47
Quotient	00
Compteur	08

Après la première itération des étapes 2 à 4
(à noter : le dividende est décalé avant la soustraction)

Dividende	1DDA
Diviseur	47
Quotient	01
Compteur	07

Après la deuxième itération des étapes 2 à 4 :

Dividende	3BB4
Diviseur	47
Quotient	02
Compteur	06

Après la troisième itération :

Dividende	3068
Diviseur	47
Quotient	05
Compteur	05

Après la quatrième itération :

Dividende	19D0
Diviseur	47
Quotient	0B
Compteur	04

Après la cinquième itération :

Dividende	33A0
Diviseur	47
Quotient	16
Compteur	03

Après la sixième itération :

Dividende	2040
Diviseur	47
Quotient	2D
Compteur	02

Après la septième itération :

Dividende	4080
Diviseur	47
Quotient	5A
Compteur	01

Après la huitième itération :

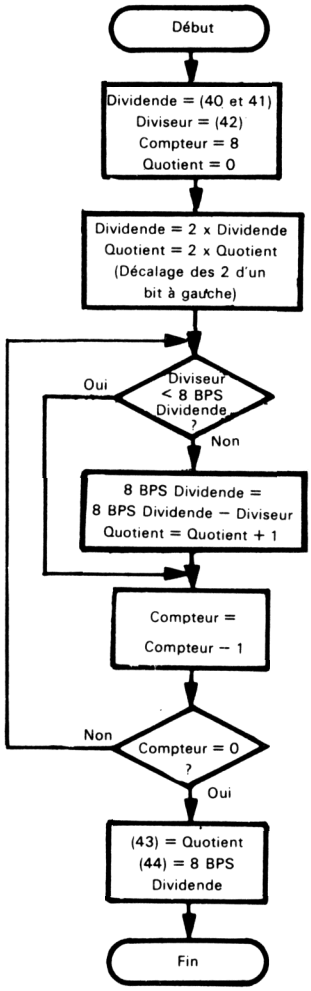
Dividende	3A00
Diviseur	47
Quotient	B5
Compteur	00

Le quotient est donc B5 et le reste 3A.
On considère que les bits les plus significatifs du dividende et du diviseur sont à zéro pour simplifier les calculs (sinon le décalage qui précède la soustraction mettrait les bits les plus significatifs du dividende dans la retenue). On peut simplifier les divisions qui ne se présentent pas sous cette forme en isolant la partie du quotient qui dépasse un octet. Par exemple :

$$\frac{1024}{3} = \frac{400\ (\text{Hexa})}{3} = 100 + \frac{100\ (\text{Hexa})}{3}$$

Cet exemple est maintenant sous la forme adéquate. Une division supplémentaire peut être nécessaire.

Organigramme :



Programme source :

```
        LD      HL,(40H)    ; LECTURE DIVIDENDE
        LD      A,(42H)     ; LECTURE DIVISEUR
        LD      C,A
        LD      B,8         ; COMPTEUR = NOMBRE BITS DIVISEUR
DIV:    ADD     HL,HL       ; DECALAGE DIVIDENDE ET QUOTIENT (1 BIT GAUCHE)
        LD      A,H         ; SOUSTRACTION DIVISEUR POSSIBLE ?
        SUB     C
        JR      C,CPT       ; NON, ETAPE SUIVANTE
        LD      H,A         ; OUI, SOUSTRACTION DIVISEUR DU DIVIDENDE
        INC     L           ; QUOTIENT + 1
CPT :   DJNZ    DIV         ; ETAPE SUIVANTE
        LD      (43H),HL    ; SAUVEGARDE QUOTIENT, RESTE EN MEMOIRE
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	2A	LD	HL,(40H)
0001	40		
0002	00		
0003	3A	LD	A,(42H)
0004	42		
0005	00		
0006	4F	LD	C,A
0007	06	LD	B,8
0008	08		
0009	29	DIV: ADD	HL,HL
000A	7C	LD	A,H
000B	91	SUB	C
000C	38	JR	C,CPT
000D	02		
000E	67	LD	H,A
000F	2C	INC	L
0010	10	CPT : DJNZ	DIV
0011	F7		
0012	22	LD	(43H),HL
0013	43		
0014	00		
0015	76	HALT	

Le double registre HL contient à la fois le dividende et le quotient. Le quotient remplace simplement le dividende dans le registre L lorsque le dividende est décalé logiquement à gauche.
Pour des divisions plus longues, on peut utiliser l'instruction SBC HL, qui soustrait le contenu d'un double registre et l'indicateur de retenue du contenu du double registre HL.
L'instruction INC L met le bit le moins significatif du quotient à 1, puisque l'instruction ADD HL,HL a préalablement remis ce bit à zéro.
La division est nécessaire pour les calculateurs, les terminaux, la détection d'erreur de communication, les algorithmes de contrôle et bien d'autres applications.
Cet algorithme prend entre 400 et 430 cycles d'horloge sur un microprocesseur Z80. Le temps précis dépend du nombre de bits à un du quotient. D'autres algorithmes peuvent légèrement réduire le temps moyen d'exécution mais l'on estime à 400 cycles d'horloge le temps nécessaire d'une division effectuée par logiciel. Quelques ouvrages dont les références sont données à la fin du chapitre développent des méthodes plus rapides pour la mise en œuvre de la division.

Chiffres de contrôle

Méthode de doublage Modulo 10

But : calculer le chiffre de contrôle d'une chaîne de chiffres DCB. La longueur de la chaîne (nombre d'octets) se trouve à l'adresse 0041 ; la chaîne des chiffres (2 chiffres DCB par octet) commence à l'adresse 0042. Calculer le chiffre de contrôle à l'aide de la méthode[3] de doublage Modulo 10 et le ranger à l'adresse 0040.

CHIFFRES DE CONTRÔLE

La méthode de doublage Modulo 10 consiste à :

1) initialiser à zéro la somme de contrôle
2) multiplier le chiffre de tête par 2 et ajouter le résultat à la somme de contrôle
3) ajouter le chiffre suivant à la somme de contrôle
4) continuer 2) et 3) jusqu'au dernier chiffre (inclus) de la chaîne
5) le chiffre de contrôle est le chiffre le moins significatif de la somme calculée.

Les chiffres de contrôle sont ajoutés de façon courante aux numéros d'identification des cartes de crédit, des étiquettes d'inventaire, des bagages, colis, etc., lorsque le traitement est effectué par ordinateur. Ils servent aussi au routage de messages, à l'identification de fichiers et à d'autres applications. Ces chiffres ont été conçus pour minimiser les erreurs d'entrée comme la transposition de chiffres (69 au lieu de 96), le décalage de chiffres (7260 au lieu de 3726), la transformation légère de nombres (65 au lieu de 64), etc. Le chiffre de contrôle peut être automatiquement vérifié à l'entrée et beaucoup d'erreurs peuvent être ainsi immédiatement éliminées.
L'analyse des méthodes de contrôle est assez complexe. Par exemple, une somme de contrôle élémentaire ne détecte pas les erreurs de transposition ($4+9=9+4$). L'algorithme du doublage permet de trouver les erreurs simples de transposition ($2\times4+9=17\neq 2\times9+4$) ; par contre, il se révèle inutile dans le cas de transpositions de chiffres séparés par un nombre impairs de chiffres (367 au lieu de 763). Cependant, cette méthode détecte un grand nombre d'erreurs courantes. L'efficacité d'une méthode dépend du type d'erreurs qu'elle détecte et de la probabilité d'erreurs particulières non détectées dans une application.

Par exemple, considérons la chaîne de chiffres 549321. Le résultat de la méthode donne :

somme de contrôle = $5\times2+4+9\times2+3+2\times2+1=40$
chiffre de contrôle = 0 (chiffre le moins significatif de la somme)

Notons qu'une entrée erronnée comme 543921 donne un chiffre de contrôle différent (4) mais d'autres entrées erronnées comme 049321 ou 945321 ne sont pas détectées par cette méthode.

Exemples pratiques :

a.
(0041) = 03
(0042) = 36
(0043) = 68
(0044) = 51

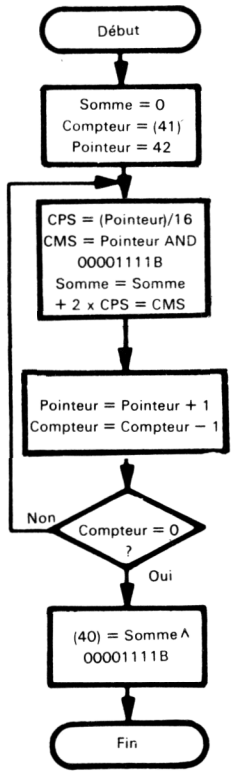
Résultat : Somme = 3 x 2 + 6 + 6 x 2 + 8 + 5 x 2 + 1 = 43
(0040) = 03

b.
(0041) = 04
(0042) = 50
(0043) = 29
(0044) = 16
(0045) = 83

Résultat : Somme = 5 x 2 + 0 + 2 x 2 + 9 + 1 x 2 + 6 + 8 x 2 + 3 = 50
(0040) = 00

Organigramme :

Avec les notations : CPS : Chiffre le Plus Significatif
CMS : Chiffre le Moins Significatif



Programme source :

```
        LD      A,(41H)     ; COMPTEUR = LONGUEUR CHAINE EN OCTETS
        LD      B,A
        LD      C,0         ; SOMME = 0
        LD      HL,42H      ; POINTEUR SUR DEBUT CHAINE
SUIT :  LD      A,(HL)      ; LECTURE 2 CHIFFRES DCB DE CHAINE
        LD      D,A         ; SAUVEGARDE CHIFFRES
        RRA                 ; OBTENTION CPS PAR DECALAGE
        RRA                 ; ET MASQUAGE
        RRA
        RRA
        AND     00001111B
        ADD     A,A         ; CPS x 2
        DAA                 ; EN DECIMAL
        ADD     A,C         ; SOMME + CPS x 2
        DAA                 ; EN DECIMAL
        LD      C,A
        LD      A,D         ; LECTURE CMS
        AND     00001111B   ; (PAR MASQUAGE DU CPS)
        ADD     A,C         ; AJOUT CPS A LA SOMME
        DAA                 ; EN DECIMAL
        LD      C,A
        INC     HL
        DJNZ    SUIT
        AND     00001111B   ; OBTENTION CHIFFRE CONTROLE PAR MASQUAGE
        LD      (40H),A     ; SAUVEGARDE CHIFFRE CONTROLE
        HALT
```

Programme objet : (voir page 309)

Les chiffres sont traités à l'aide du décalage et du masquage. Quatre décalages à droite sont nécessaires pour isoler le chiffre le plus significatif.
Un ajustement décimal (DAA) doit suivre chaque addition pour obtenir le résultat décimal correct. Une seule instruction DAA après une série d'additions ne donnera pas le bon résultat. (Essayez !) Il ne faut pas oublier que DAA n'opère que sur l'accumulateur. Il n'y a pas de problème de retenue dans la somme décimale, puisque de toute façon, la procédure n'utilise que le chiffre le moins significatif de la somme.

INSTRUCTIONS DE DÉCALAGE DÉCIMAL

Une autre solution (meilleure aussi) consiste à utiliser l'instruction de décalage décimal RLD du Z80. Cette instruction représente un décalage 4 bits qui transfère le contenu des 4 bits de poids faible de la mémoire adressée par HL dans les 4 bits de poids fort de cette même mémoire, le précédent contenu des 4 bits de poids fort de cette adresse dans les 4 bits de poids faible de l'accumulateur et le précédent contenu des 4 bits de poids faible de l'accumulateur dans les 4 bits de poids faible de la mémoire. Ainsi, l'instruction RLD ne se contente pas simplement de transférer un seul chiffre dans l'accumulateur, mais elle décale le chiffre suivant pour qu'il soit transféré dans l'accumulateur au cours de l'instruction RLD suivante. La figure 8.1 présente un exemple du fonctionnement de l'instruction RLD ; l'instruction RRD est la même instruction avec un décalage à droite au lieu du décalage à gauche.
Le programme p. 310 montre la mise en œuvre de l'algorithme de doublage Modulo 10 à l'aide de l'instruction RLD :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3A	LD	A,(41H)
0001	41		
0002	00		
0003	47	LD	B,A
0004	0E	LD	C,0
0005	00		
0006	21	LD	HL,42H
0007	42		
0008	00		
0009	7E	SUIT : LD	A,(HL)
000A	57	LD	D,A
000B	1F	RRA	
000C	1F	RRA	
000D	1F	RRA	
000E	1F	RRA	
000F	E6	AND	00001111B
0010	0F		
0011	87	ADD	A,A
0012	27	DAA	
0013	81	ADD	A,C
0014	27	DAA	
0015	4F	LD	C,A
0016	7A	LD	A,D
0017	E6	AND	00001111B
0018	0F		
0019	81	ADD	A,C
001A	27	DAA	
001B	4F	LD	C,A
001C	23	INC	HL
001D	10	DJNZ	SUIT
001E	EA		
001F	E6	AND	00001111B
0020	0F		
0021	32	LD	(40H),A
0022	40		
0023	00		
0024	76	HALT	

Programme objet

Programme source :

```
        LD      A,(41H)     ; COMPTEUR = LONGUEUR CHAINE (OCTETS)
        LD      B,A
        LD      C,0         ; SOMME = 0
        LD      HL,42H      ; POINTE SUR DEBUT CHAINE
SUIT :  SUB     A           ; CPS A ZERO
        RLD                 ; OBTENTION PSC CHAINE
        ADD     A,A         ; CPS x 2
        DAA                 ; EN DECIMAL
        ADD     A,C         ; SOMME + CPS x 2
        DAA                 ; EN DECIMAL
        LD      C,A
        SUB     A           ; PSC A ZERO
        RLD                 ; OBTENTION CMS CHAINE
        ADD     A,C         ; SOMME + CMS
        DAA                 ; EN DECIMAL
        LD      C,A
        INC     HL
        DJNZ    SUIT
        AND     00001111B   ; OBTENTION CHIFFRE CONTROLE PAR MASQUAGE
        LD      (40H),A     ; SAUVEGARDE CHIFFRE CONTROLE
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3A	LD	A,(41H)
0001	41		
0002	00		
0003	47	LD	B,A
0004	0E	LD	C,0
0005	00		
0006	21	LD	HL,42H
0007	42		
0008	00		
0009	97	SUIT : SUB	A
000A	ED	RLD	
000B	6F		
000C	87	ADD	A,A
000D	27	DAA	
000E	81	ADD	A,C
000F	27	DAA	
0010	4F	LD	C,A
0011	97	SUB	A
0012	ED	RLD	
0013	6F		
0014	81	ADD	A,C
0015	27	DAA	
0016	4F	LD	C,A
0017	23	INC	HL
0018	10	DJNZ	SUIT
0019	EF		
001A	E6	AND	00001111B
001B	0F		
001C	32	LD	(40H),A
001D	40		
001E	00		
001F	76	HALT	

Ce programme peut encore être amélioré (cette version est déjà plus courte que la précédente). En effet, puisque le chiffre le moins significatif est perdu à la fin, il n'y a pas de raison de le mettre à zéro chaque fois avec une instruction SUB A.

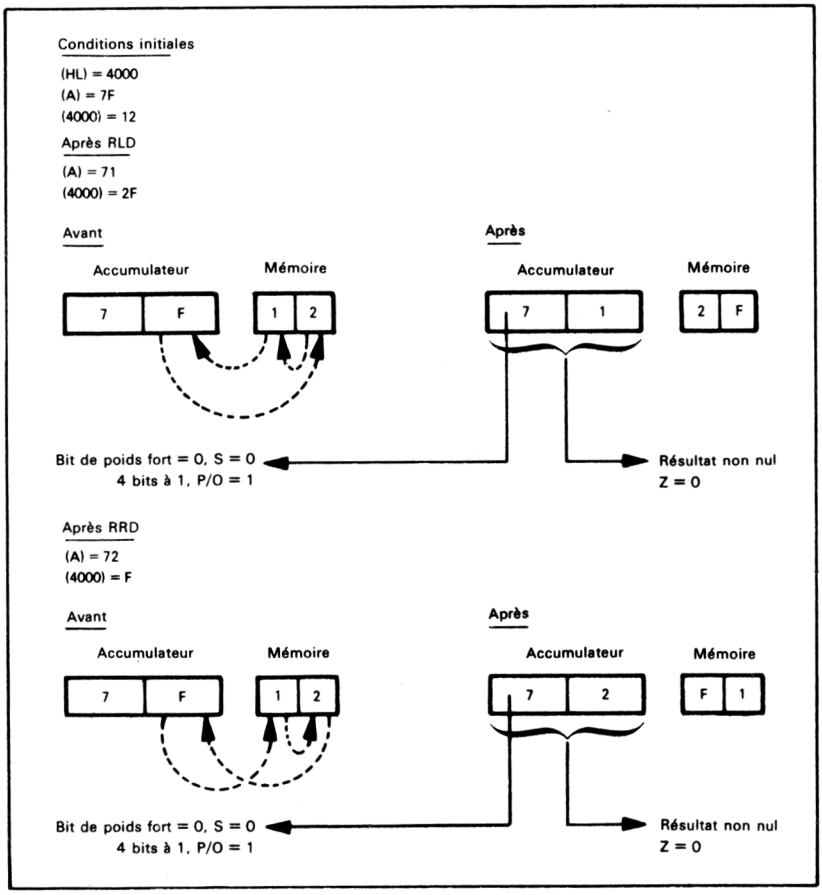


Fig. 8.1. — Exemples de décalage de chiffres du Z80.

MULTIPLICATION PAR 2 ET DIVISION PAR 2 DE NOMBRES BINAIRES

Il est possible de doubler un nombre (dans l'accumulateur) en l'additionnant à lui-même puis en effectuant une correction décimale, c'est-à-dire :

```
ADD     A       ; NOMBRE x 2
DAA             ; CORRECTION DECIMALE
```

Ne pas oublier que l'accumulateur ne peut contenir que des chiffres décimaux valides compris entre 0 et 99.
Il n'est pas possible d'utiliser l'instruction SLA A (décalage arithmétique à gauche de A) car cette instruction remet toujours à zéro la demi-retenue (seules l'addition et la soustraction positionnent H correctement).
On peut diviser un nombre décimal par 2 simplement en le décalant logiquement à droite puis en enlevant 3 à chaque chiffre supérieur ou égal à 8 (puisque 10 DCB = 16 binaire).
Le programme qui suit divise par 2 le nombre rangé à l'adresse 0040 et met le résultat à l'adresse 0041.

```
        LD      A,(40H)   ; LECTURE NOMBRE DECIMAL
        SRL     A         ; DIVISION PAR 2 EN BINAIRE
        BIT     3,A       ; CMS≥8 ?
        JR      Z,FINI
        SUB     3         ; OUI, -3 POUR CORRECTION DECIMALE
FINI :  LD      (41H),A   ; RANGER RESULTAT
        HALT
```

Essayez le programme et la méthode sur les nombres décimaux 28, 30 et 37.

ARRONDI BINAIRE

Le calcul de l'arrondi est simple si les nombres sont binaires ou décimaux. Un nombre binaire peut être arrondi de la façon suivante :

> Si le bit le plus significatif devant être tronqué est 1, ajouter 1 aux bits qui restent. Sinon, ne rien changer.

Cette règle se justifie du fait que 1 est à mi-chemin entre 0 et 10 en binaire, comme 5 en décimal (c'est-à-dire 0,5 décimal = 0,1 binaire).
En appliquant cette règle, le programme suivant arrondit le nombre 16 bits contenu aux adresses 0040 et 0041 (bits de poids fort en 0041) en un nombre 8 bits à l'adresse 0041.

```
        LD      HL,40H
        BIT     7,(HL)    ; MSB OCTET A TRONQUER = 1 ?
        JR      Z,FINI
        INC     HL        ; OUI, ARRONDIR A LA VALEUR SUPERIEURE
        INC     (HL)
FINI :  HALT
```

Si le nombre fait plus de 16 bits, l'arrondi doit être répercuté sur tous les octets concernés.

ARRONDI DÉCIMAL

L'arrondi décimal est un peu plus difficile car le point critique est maintenant 50 DCB et l'arrondi doit donner un résultat décimal. Voici la règle :

> Si le chiffre le plus significatif devant être tronqué vaut 5 ou plus, ajouter 1 aux chiffres qui restent.

Le programme suivant arrondit un nombre de 4 chiffres DCB rangé aux adresses 0040 et 0041 (bits de poids fort en 0041) en un nombre de 2 chiffres DCB à l'adresse 0041.

```
        LD      HL,40H
        LD      A,(HL)      ; OCTET A PERDRE≥50 ?
        CP      50H
        JR      C,DONE
        INC     HL          ; OUI, ARRONDIR
        LD      A,(HL)
        ADD     A,1
        DAA                 ; CORRECTION DECIMALE
        LD      (HL),A
DONE:   HALT
```

Il ne faut pas oublier que l'instruction DAA opère uniquement sur les nombres de l'accumulateur. Dans ce cas, on pourrait effectuer l'arrondi avec l'instruction INC A, puisque nous savons que l'indicateur de retenue vaut zéro (pourquoi ? — rappelez-vous l'instruction JR). Habituellement, la séquence ADD A,1 suivie de DAA est nécessaire puisque l'instruction INC A ne modifie pas l'indicateur de retenue.

PROPAGATION DU SIGNE

Très souvent, en effectuant de l'arithmétique multi-octet, complémentée à 2 et signée, il est nécessaire de propager le bit de signe tout au long des octets de poids fort. Cette opération peut être exécutée de façon directe si, comme c'est habituellement le cas, le signe se trouve dans l'indicateur de retenue. L'instruction SBC A,A transmet l'état de la retenue à travers un octet. Puisque A-A vaut toujours 0, l'instruction SBC A,A revient à soustraire la retenue de la valeur 0, ce qui donne soit 0 soit FF (hexa).

EXERCICES

1) Soustraction en multiple précision

But : soustraire un nombre sur plusieurs octets d'un autre. La longueur des nombres se trouve à l'adresse 0040, les nombres eux-mêmes commencent (par les bits de poids faible) aux adresses 0041 et 0051, respectivement ; la différence remplace le nombre qui commence à l'adresse 0041. Soustraire le nombre qui commence en 0051 de celui qui commence en 0041.

Exemple pratique :

```
(0040) = 04
(0041) = C3
(0042) = A7
(0043) = 5B
(0044) = 2F

(0051) = B8
(0052) = 35
(0053) = DF
(0054) = 14

Résultat : (0041) = 0B
           (0042) = 72
           (0043) = 7C
           (0044) = 1A

soit :        2F5BA7C3
            - 14DF35B8
              1A7C720B
```

2) Soustraction décimale

But : soustraire un nombre décimal (DCB) sur plusieurs octets d'un autre. La longueur des nombres se trouve à l'adresse 0040, les nombres eux-mêmes commencent (bits de poids faible d'abord) aux adresses 0041 et 0051, respectivement ; la différence remplace le nombre qui commence à l'adresse 0041.
Soustraire le nombre qui commence en 0051 de celui qui commence en 0041.

Exemple pratique :

(0040) = 04

(0041) = 85
(0042) = 19
(0043) = 70
(0044) = 36

(0051) = 59
(0052) = 34
(0053) = 66
(0054) = 12

Résultat : (0041) = 26
(0042) = 85
(0043) = 03
(0044) = 24

soit :

```
  36701985
  12663459
  --------
  24038526
```

3) Multiplication binaire 8 bits par 16 bits

But : multiplier un nombre 16 bits non signé contenu aux adresses 0040 et 0041 (bits de poids fort en 0041) par le nombre 8 bits non signé à l'adresse 0042. Ranger le résultat aux adresses 0043, 0044, 0045 avec les bits de poids fort à l'adresse 0045.

Exemples pratiques :

a.

(0040) = 03
(0041) = 00
(0042) = 05

Résultat : (0043) = 0F
(0044) = 00
(0045) = 00

soit : 3 x 5 = 15

b.

(0040) = 6F
(0041) = 72 (29 295 décimal)
(0042) = 61 (97 décimal)

Résultat : (0043) = 0F
(0044) = 5C
(0045) = 2B

soit : 29 295 x 97 = 2 841 615

4) Division binaire signée

But : diviser un nombre 16 bits rangé aux adresses 0040 et 0041 (bits de poids fort en 0041) par un nombre 8 bits signé à l'adresse 0042.
Ranger le quotient (signé) à l'adresse 0043 et le reste (toujours positif) à l'adresse 0044.

Exemples pratiques :

a.
(0040) = C0
(0041) = FF (-64)
(0042) = 08

Résultat :
(0043) = F8 (-8) quotient
(0044) = 00 (0) reste

b.
(0040) = 93
(0041) = ED (-4717)
(0042) = 47 (71 décimal)

Résultat :
(0043) = BD (-67 décimal)
(0044) = 28 (+ 40 décimal)

Conseil : déterminer le signe du résultat, effectuer une division non signée, puis la correction décimale du quotient et du reste.

Chiffres de contrôle — Méthode de l'alignement 1, 3, 7 Modulo 10

But : calculer le chiffre de contrôle d'une chaîne de nombres DCB. La longueur de la chaîne (nombre d'octets) se trouve à l'adresse 0041 ; la chaîne de chiffres (2 chiffres DCB par octet) commence à l'adresse 0042. Calculer le chiffre de contrôle par la méthode de l'alignement 1, 3, 7 Modulo 10 et ranger ce chiffre à l'adresse 0040.
Le principe de la méthode de l'alignement 1, 3, 7 Modulo 10 est le suivant :

1) Initialiser la somme de contrôle.
2) Ajouter le chiffre de tête à la somme.
3) Multiplier le chiffre suivant par 3 et ajouter le résultat à la somme.
4) Multiplier le chiffre suivant par 7 et ajouter le résultat à la somme.
5) Reprendre les étapes 2) à 4) jusqu'à ce que la chaîne soit terminée.
6) Le chiffre de contrôle est le chiffre le moins significatif de la somme. Par exemple, si la chaîne est 549321, le résultat donne :

Somme de contrôle = $5 + 3 \times 4 + 7 \times 9 + 3 + 3 \times 2 + 7 \times 1 = 96$
Chiffre de contrôle = 6

Exemples pratiques :

a.
(0041) = 03
(0042) = 36
(0043) = 68
(0044) = 51

Résultat : Somme = 3 + 3 x 6 + 7 x 6 + 8 + 3 x 5 + 7 x 1 = 93
(0040) = 03

b.
(0041) = 04
(0042) = 50
(0043) = 29
(0044) = 16
(0045) = 83

Résultat : Somme = 5 + 3 x 0 + 7 x 2 + 9 + 3 x 1 + 7 x 6 + 8
+ 3 x 3 = 90
(0040) = 00

Conseil : $7 = 2 \times 3 + 1$ et $3 = 2 \times 1 + 1$. On peut utiliser la formule $M_i = 2 \times M_i + 1$ pour calculer le facteur multiplicatif suivant.

RÉFÉRENCES

1. **Plusieurs algorithmes de multiplication sont décrits dans : T. Dolhoff, « Microprocessor Software : How to Optimize Timing and Memory Usage. Part Four. Techniques for the Zilog Z80 »,** *Digital Design,* Février 1977, pp. 44-51.
2. Quelques microprocesseurs (comme le 9900, 8086 et le Z8000) disposent d'instructions de multiplication qui sont un peu plus rapides, mais la vitesse maximum nécessite l'apport d'un support matériel
   D'autres méthodes pour mettre en œuvre la multiplication, la division et d'autres applications arithmétiques sont traitées dans :
   Geist, D.J., « MOS Processor Picks up Speed with Bipolar Multipliers », *Electronics,* 7 juillet 1977, pp. 113-115.
   Kolodzinski, A. et D. Wainland, « Multiplying with a Microcomputer », *Electronic Design,* 18 janvier 1978, pp. 78-83.
   Mick, J.R. et J. Springer, « Single-chip Multiplier Expands Digital Role in Signal Processing », *Electronics,* 13 mai 1976, pp. 103-108.
   Parasuraman, B., « Hardware Multiplication Techniques for Microprocessor Systems », *Computer Design,* avril 1977, pp. 75-82.
   Tao, T.F. et al., « Applications of Microprocessors in Control Problems », 1977 Joint Automatic Control Conference Proceedings, San Francisco, CA., 22-24 juin 1977.
   Waser, S., « State-of-the-art in High-Speed Arithmetic Integrated Circuits », *Computer Design,* juillet 1978, pp. 67-75.
   Weissberger, A.J. et T. Toal, « Tough Mathematical Tasks Are Child's Play for Number Cruncher », *Electronics,* 17 février 1977, pp. 102-107.
3. Consulter l'article de J.R. Herr, « Self-Checking Number Systems », *Computer Design,* juin 1974, pp. 85-91.

CHAPITRE 9

TABLES ET LISTES

Les tables et les listes constituent deux des structures de donnée fondamentales utilisées par tous les calculateurs. Nous avons déjà vu l'usage de tables pour des conversions de code et des opérations arithmétiques. Les tables peuvent aussi servir à identifier des commandes et des instructions, à linéariser des données, à permettre l'accès aux fichiers ou aux enregistrements, à définir la signification des clés ou des interrupteurs et à faire un choix entre différents programmes.
Les listes sont habituellement moins structurées que les tables. Les listes peuvent contenir les tâches à exécuter par le processeur, des messages ou des données que le processeur doit enregistrer ou des conditions qui ont été modifiées ou qui doivent être traitées. Les tables sont un moyen simple de prendre des décisions ou de résoudre des problèmes lorsqu'aucun calcul ni fonction logique n'est nécessaire. La tâche se réduit alors à organiser la table de façon à trouver facilement l'entrée correcte. Les listes permettent l'exécution de combinaisons de tâches, la préparation d'ensembles de résultats, et la construction de fichiers de données interconnectés (ou bases de données). L'insertion et le retrait d'éléments d'une liste sont traités dans les exercices du chapitre.

EXEMPLES

Ajout d'un élément à une liste

But : ajouter le contenu de l'adresse 0040 à une liste si celui-ci ne figure pas déjà sur la liste. La longueur de la liste se trouve à l'adresse 0041 et la liste elle-même commence à l'adresse 0042.

Exemples pratiques :

a.

(0040) = 6B
(0041) = 04
(0042) = 37
(0043) = 61
(0044) = 38
(0045) = 1D

Result: (0041) = 05
(0046) = 6B

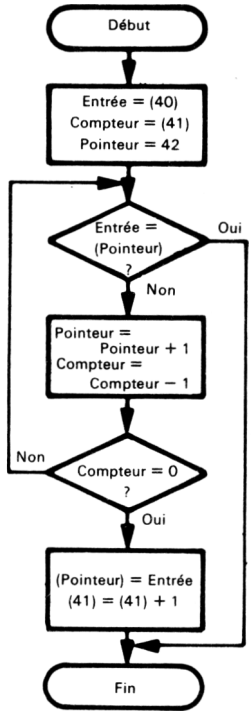
La donnée est ajoutée à la liste, puisque celle-ci n'y figure pas. La longueur de la liste est incrémentée de 1.

b.

(0040) = 6B
(0041) = 04
(0042) = 37
(0043) = 6B
(0044) = 38
(0045) = 1D

Résultat : Pas de changement, puisque la donnée figure sur la liste.

Organigramme :



Programme source :

```
        LD      HL,40H      ; POINTE SUR ENTREE
        LD      A,(HL)      ; LECTURE ENTREE
        INC     HL          ; POINTE SUR COMPTEUR
        LD      B,(HL)      ; COMPTEUR = LONGUEUR LISTE
        INC     HL          ; POINTE SUR DEBUT DE LISTE
LIST :  CP      (HL)        ; ENTREE = ELEMENT LISTE ?
        JR      Z,FINI      ; OUI, FINI
        INC     HL          ; NON, ELEMENT SUIVANT
        DJNZ    LIST
        LD      (HL),A      ; AJOUT ENTREE A LISTE
        LD      HL,41H      ; LONGUEUR LISTE + 1
        INC     (HL)
FINI :  HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	21		LD	HL,40H
0001	40			
0002	00			
0003	7E		LD	A,(HL)
0004	23		INC	HL
0005	46		LD	B,(HL)
0006	23		INC	HL
0007	BE	LIST :	CP	(HL)
0008	28		JR	Z,FINI
0009	08			
000A	23		INC	HL
000B	10		DJNZ	LIST
000C	FA			
000D	77	ADELM:	LD	(HL),A
000E	21		LD	HL,41H
000F	41			
0010	00			
0011	34		INC	(HL)
0012	76	FINI :	HALT	

On peut aussi utiliser dans cet exemple l'instruction de recherche en bloc CPIR, ce qui donne :

Programme source :

```
        LD      HL,40H      ; POINTE SUR ENTREE
        LD      A,(HL)      ; LECTURE ENTREE
        INC     HL          ; POINTEUR SUR COMPTEUR
        LD      B,0         ; COMPTEUR = LONGUEUR LISTE (16 BITS)
        LD      C,(HL)
        INC     HL          ; POINTE SUR DEBUT LISTE
        CPIR                ; RECHERCHE ENTREE DANS LISTE
        JR      Z,FINI      ; FINI, SI ENTREE TROUVEE
        LD      (HL),A      ; SINON, AJOUT ENTREE A LISTE
        LD      HL,41H      ; LONGUEUR LISTE + 1
FINI :  HALT
```

Programme objet : (voir page 320)

Rappelons que l'instruction CPIR répète automatiquement l'instruction de recherche en bloc jusqu'à ce que soit BC atteigne 0, soit qu'il y ait égalité dans la comparaison (c'est-à-dire A = (HL)).

Attention aux légères différences qui vont suivre par rapport à la version précédente :

1) BC est un compteur 16 bits. Ainsi l'instruction CPIR peut traiter des chaînes de plus de 256 octets.
2) Le bit de Parité/Overflow (P/O) est remis à zéro si BC est décrémenté jusqu'à 0 et P/O est positionné à 1 sinon.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	7E	LD	A,(HL)
0004	23	INC	HL
0005	06	LD	B,0
0006	00		
0007	4E	LD	C,(HL)
0008	23	INC	HL
0009	ED	CPIR	
000A	B1		
000B	28	JR	Z,FINI
000C	05		
000D	77	LD	(HL),A
000E	21	LD	HL,41H
000F	41		
0010	00		
0011	34	INC	(HL)
0012	76	FINI : HALT	

HASHING En fait, cette méthode d'ajout d'éléments n'est pas du tout efficace, si la liste est longue. Il est possible d'améliorer la procédure en limitant la recherche à une partie de la liste ou en ordonnant la liste. La recherche peut être limitée si l'on utilise la donnée à ajouter pour obtenir le point d'entrée de la liste. Cette méthode appelée « hashing » est tout à fait semblable à la recherche de la page de départ d'un dictionnaire ou d'un annuaire d'après la première lettre de l'élément à trouver. On peut ordonner la liste par valeur numérique. La recherche s'arrête alors lorsque la liste des valeurs dépasse l'entrée (plus grande ou plus petite, cela dépend de la relation d'ordre choisie). Si un autre élément doit être inséré correctement, tous les autres doivent être déplacés en conséquence dans la liste.
Le programme peut être remanié de façon à utiliser deux tables. L'une d'elle fournit un point d'entrée dans l'autre : par exemple, le point de recherche peut être le chiffre 4-bits le plus significatif ou le moins significatif de l'entrée.
Le programme ne marche pas si la longueur de la liste est à zéro (qu'arrive-t-il ?). On peut éviter ce problème en vérifiant la longueur au préalable. La procédure d'initialisation du premier programme devient alors :

```
        LD      HL,40H      ; POINTE SUR ENTREE
        LD      A,(HL)      ; LECTURE ENTREE
        INC     HL          ; POINTE SUR LONGUEUR
        LD      B,(HL)      ; COMPTEUR = LONGUEUR LISTE
        INC     HL          ; POINTE SUR DEBUT LISTE
        INC     B           ; COMPTEUR A ZERO ?
        DEC     B
        JR      Z,ADELM     ; OUI, AJOUT ENTREE A LISTE
        .
        .
        .
ADELM:  LD      (HL),A      ; AJOUT ENTREE A LISTE
```

Notons que la séquence INC,DEC est un moyen facile de tester une valeur à zéro dans un registre sans utiliser l'accumulateur ou modifier la valeur d'un registre.
La procédure :

```
LD      HL,ADDR
INC     (HL)
```

est un moyen rapide d'ajouter 1 à un compteur à l'adresse ADDR sans utiliser l'accumulateur. Il est possible d'utiliser l'instruction DEC (HL) d'une façon similaire pour soustraire 1 du compteur.
L'instruction LD (HL),CONST permet de placer une valeur départ (0 par exemple) dans le compteur. Les adresses peuvent être, bien sûr, utilisées comme compteurs lorsqu'aucun registre accessible en lecture n'est disponible.
Si chaque entrée dépasse un octet, un programme de comparaison de formats devient nécessaire. Le programme passe à l'entrée suivante s'il n'y a pas égalité, c'est-à-dire qu'il arrête la comparaison de l'entrée en cours dès qu'il trouve une différence.

Vérification d'une liste ordonnée

But : vérifier le contenu d'adresse 0041 pour voir s'il figure dans une liste ordonnée. La longueur de la liste se trouve à l'adresse 0042 ; la liste elle-même commence à l'adresse 0043 et est constituée de nombres binaires non signés par ordre croissant. Si le contenu d'adresse 0041 se trouve dans la liste, mettre à zéro le contenu d'adresse 0040 ; sinon le mettre à FF (hexa).

Exemples pratiques :

a.
(0041) = 6B
(0042) = 04
(0043) = 37
(0044) = 55
(0045) = 7D
(0046) = A1

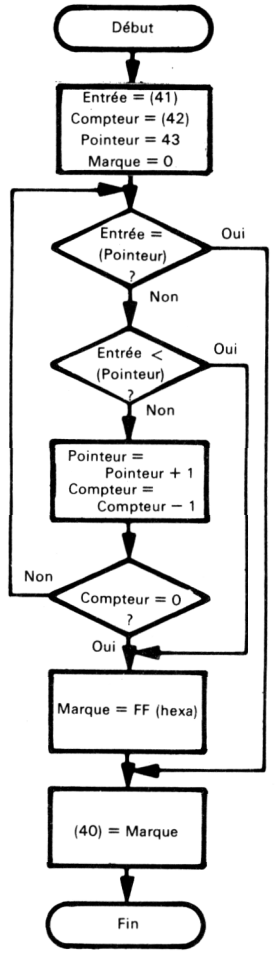
Résultat : (0040) = FF, puisque 6B ne figure pas sur la liste

b.
(0041) = 6B
(0042) = 04
(0043) = 37
(0044) = 55
(0045) = 6B
(0046) = A1

Résultat : (0040) = 00, puisque 6B figure sur la liste

Organigramme : (voir page 322)

Le processus de recherche est légèrement différent ici puisque les éléments sont ordonnés. Une fois qu'un élément supérieur à l'entrée est trouvé, la recherche est terminée puisque les éléments suivants seront encore plus grands.
Essayez un exemple pour vous convaincre que la procédure marche.



Organigramme

MÉTHODES DE RECHERCHE

Comme dans le problème précédent, une table ou une autre méthode utilisant un bon point d'entrée permettrait d'accélérer la recherche. Une méthode consiste à commencer au milieu et à déterminer dans quelle moitié de la liste l'entrée se trouve, puis à diviser la moitié en moitiés, etc. Cette méthode est appelée recherche *dichotomique* (« binary search »), puisqu'à chaque fois elle divise en 2 la partie de la liste qui reste[1].

Programme source :

```
        LD      HL,41H      ; POINTE SUR ENTREE
        LD      A,(HL)      ; LECTURE ENTREE
        INC     HL          ; POINTE SUR LONGUEUR
        LD      B,(HL)      ; COMPTEUR = LONGUEUR LISTE
        LD      C,0         ; MARQUE = 0 (ENTREE DANS LISTE)
        INC     HL          ; POINTE SUR DEBUT LISTE
LIST :  CP      (HL)        ; ENTREE = ELEMENT LISTE
        JR      Z,FINI      ; OUI, RECHERCHE TERMINEE
        JR      C,ABS       ; SI ENTREE<ELEMENT, EN DEHORS
        INC     HL
        DJNZ    LIST
ABS :   LD      C,OFFH      ; MARQUE = FF, ENTREE EN DEHORS LISTE
FINI :  LD      A,C         ; SAUVEGARDE MARQUE
        LD      (40H),A
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	21		LD	HL,41H
0001	41			
0002	00			
0003	7E		LD	A,(HL)
0004	23		INC	HL
0005	46		LD	B,(HL)
0006	0E		LD	C,0
0007	00			
0008	23		INC	HL
0009	BE	LIST :	CP	(HL)
000A	28		JR	Z,FINI
000B	07			
000C	38		JR	C,ABS
000D	03			
000E	23		INC	HL
000F	10		DJNZ	SRLST
0010	F8			
0011	0E	ABS :	LD	C,OFFH
0012	FF			
0013	79	FINI :	LD	A,C
0014	32		LD	(40H),A
0015	40			
0016	00			
0017	76		HALT	

Les instructions de recherche en bloc du Z80 ne sont pas aussi utiles ici que dans l'exemple précédent car ce programme n'effectue pas une simple recherche. Il vérifie aussi que la partie intéressante de la liste a bien été contrôlée (c'est-à-dire la partie où les éléments sont inférieurs ou égaux à l'entrée). Essayez de ré-écrire le programme en utilisant l'instruction CPI. N'oubliez pas que vous devez utiliser l'indicateur (P/O) pour déterminer si le compteur d'octets a été décrémenté jusqu'à zéro.

Retrait d'un élément d'une file

But : les adresses 0042 et 0043 contiennent l'adresse de la tête de file (bits de poids fort en 0043). Mettre l'adresse du premier élément (tête) de la file aux adresses 0040 et 0041 (bits de poids fort en 0041) et mettre la file à jour en retirant l'élément. Chaque élément de la file fait 2 octets de long et contient l'adresse de l'élément suivant. Le dernier élément contient zéro pour indiquer la fin de la file.

Les files servent à ranger des données dans l'ordre où elles seront utilisées ou des tâches dans l'ordre où elles seront exécutées. La file est structurée en FIFO « first-in, first-out », c'est-à-dire que les éléments sont enlevés de la file dans l'ordre dans lequel ils y sont entrés. Les systèmes d'exploitation mettent les tâches dans des files pour qu'elles soient exécutées dans le bon ordre. Les « drivers » d'entrée/sortie transfèrent des données à partir de files ou vers elles pour qu'elles soient traitées dans le bon ordre. De même des « buffers » peuvent être mis dans des files de façon à ce que le premier disponible soit trouvé facilement et que les « buffers » libérés soient facilement ajoutés à la mémoire disponible. Les files peuvent aussi servir à traiter des besoins en mémoire, temps, ou entrée/sortie de façon à ce que ces besoins soient satisfaits dans le bon ordre.

Dans des opérations réelles, chaque élément de la file contient en général une grande quantité d'informations ou d'espace mémoire en plus de l'adresse qui le relie à l'élément suivant.

Exemples pratiques :

a.

(0042) = 46
(0043) = 00 } adresse du 1er élément de la file

(0046) = 4D
(0047) = 00 } adresse du 2e élément de la file

(004D) = 00
(004E) = 00 } fin de la file

Résultat :

(0040) = 46
(0041) = 00 } adresse de l'élément retiré de la file

(0042) = 4D
(0043) = 00 } adresse du nouveau 1er élément

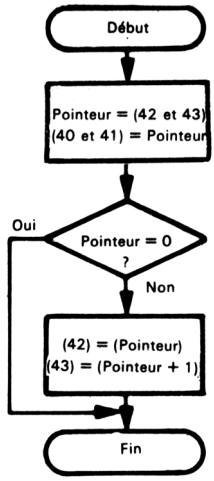
b.

(0042) = 00
(0043) = 00 } file vide

Résultat :

(0040) = 00
(0041) = 00 } pas d'élément de la file

Organigramme :



Programme source :

```
        LD      HL,(42H)    ; LECTURE ADRESSE DE TETE DE FILE
        LD      (40H),HL    ; RETRAIT TETE DE FILE
        LD      A,H         ; FILE VIDE ?
        OR      L
        JR      Z,FINI      ; OUI, FINI
        LD      E,(HL)      ; NON, LECTURE ELEMENT SUIVANT
        INC     HL
        LD      D,(HL)
        LD      (42H),DE    ; TRANSFERT ELEMENT SUIVANT EN TETE DE FILE
FINI :  HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	2A	LD	HL,(42H)
0001	42		
0002	00		
0003	22	LD	(40H),HL
0004	40		
0005	00		
0006	7C	LD	A,H
0007	B5	OR	L
0008	28	JR	Z,FINI
0009	07		
000A	5E	LD	E,(HL)
000B	23	INC	HL
000C	56	LD	D,(HL)
000D	ED	LD	(42H),DE
000E	53		
000F	42		
0010	00		
0011	76	FINI : HALT	

Les files peuvent traiter des listes qui ne se trouvent pas à des adresses consécutives. Chaque élément doit contenir l'adresse de l'élément suivant. De telles listes permettent de traiter des données ou des tâches dans l'ordre correct, de changer de variables, ou d'ajouter des définitions dans un programme. De la mémoire supplémentaire est nécessaire mais les éléments peuvent facilement être ajoutés à la file ou en être retirés.
Notons l'emploi de la séquence :

```
LD      A,H
OR      L
```

pour déterminer si le contenu d'un double registre 16 bits est nul. Rappelons que les instructions INC et DEC ne modifient pas l'état des indicateurs quand elles concernent un double registre. Essayez de trouver d'autres séquences pour résoudre ce problème — qui se posera de manière évidente chaque fois que l'on utilisera un compteur 16 bits au lieu du compteur 8 bits employé dans la plupart des exemples précédents.
Un problème se pose : il n'existe pas d'instruction qui charge un double registre en utilisant une mémoire adressée par un double registre. Une séquence d'instructions est nécessaire chaque fois qu'un double registre doit être chargé directement.
Il peut être utile de conserver des pointeurs à chaque extrémité de la file, plutôt qu'en tête uniquement. La structure de donnée peut alors être soit utilisée en mode FIFO (« first-in, first-out », c'est-à-dire : premier entré, premier sorti) soit en mode LIFO (« last-in, first-out, c'est-à-dire dernier entré, premier sorti), selon l'extrémité où les nouveaux éléments sont ajoutés. Comment modifieriez-vous le programme pour que les adresses 0044 et 0045 contiennent l'adresse du dernier élément (fin) de la file ?
Si la file ne contient pas d'élément, le programme met à zéro les adresses 0040 et 0041. Un programme qui demande un élément de la file devra contrôler ces adresses pour voir si sa demande peut être satisfaite. Avez-vous une autre idée pour obtenir cette information ?

Tri sur 8 bits

But : trier dans l'ordre décroissant un tableau de nombres binaires non signés. La longueur du tableau se trouve à l'adresse 0040 et le tableau lui-même commence à l'adresse 0041.

Exemples pratiques :

```
          (0040) = 06
          (0041) = 2A
          (0042) = B5
          (0043) = 60
          (0044) = 3F
          (0045) = D1
          (0046) = 19
Résultat: (0041) = D1
          (0042) = B5
          (0043) = 60
          (0044) = 3F
          (0045) = 2A
          (0046) = 19
```

ALGORITHME SIMPLE DE TRI

Voici une méthode simple de tri :

Etape 1) Mettre à zéro l'indicateur INTER.

Etape 2) Contrôler chaque paire de nombres consécutifs du tableau. Si au moins une paire est dans le désordre, faire l'échange et positionner INTER à 1.

Etape 3) Si INTER = 1 à la fin du contrôle du tableau complet, revenir à l'étape 1).

INTER est positionné à 1 dès qu'une paire de nombres consécutifs est en désordre. Ainsi, quand INTER est nul après un examen complet du tableau, celui-ci est bien rangé.

Cette méthode de tri est appelée « bubble sort ». C'est un algorithme facile à mettre en place. Toutefois, il faut faire appel à d'autres techniques de tri lorsqu'il s'agit de trier rapidement de longues listes[2].

Voici comment cette méthode est appliquée dans un cas simple. Considérons que l'on veut trier un tableau dans l'ordre décroissant ; le tableau a 4 éléments : 12, 03, 15, 08.

Première itération :

Etape 1) INTER = 0

Etape 2) Ordre final du tableau : 12, 15, 08, 03 puisque la seconde paire (03, 15) est inversée de même que la troisième (03, 08)
INTER = 1

Deuxième itération :

Etape 1) INTER = 0

Etape 2) Ordre final du tableau : 15, 12, 08, 03 puisqu'on procède à l'échange de la première paire
INTER = 1

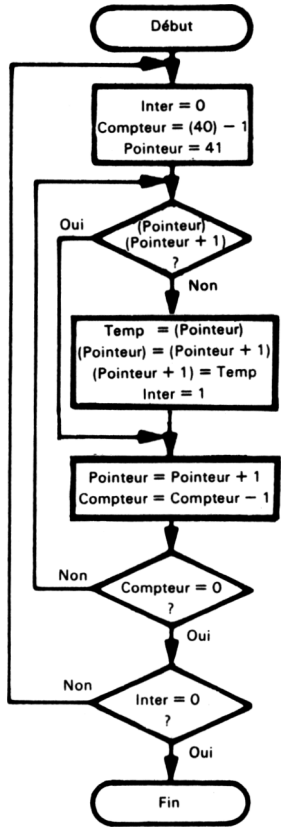
Troisième itération :

Etape 1) INTER = 0
Les éléments sont déjà en ordre, il n'y a pas besoin d'autre échange, INTER reste nul

Organigramme : (voir page 328)

Programme source :

```
TRI :   LD      C,0         ; INTER = 0
        LD      HL,40H      ; COMPTEUR = LONGUEUR TABLEAU
        LD      B,(HL)
        DEC     B           ; NOMBRE PAIRES = COMPTEUR - 1
        INC     HL          ; POINTEUR SUR DEBUT TABLEAU
PASS1:  LD      A,(HL)      ; LECTURE ELEMENT DANS TABLEAU
        INC     HL
        CP      (HL)        ; <ELEMENT SUIVANT ?
        JR      NC,CNT      ; NON, ECHANGE INUTILE
        LD      D,(HL)      ; OUI, ECHANGE ELEMENTS
        LD      (HL),A
        DEC     HL
        LD      (HL),D
        INC     HL
        LD      C,1         ; INTER = 1
CNT:    DJNZ    PASS1
        DEC     C           ; INTER ETAIT A 1 ?
        JR      Z,TRI       ; OUI, AUTRE PASSAGE
        HALT
```



Organigramme

Le cas où 2 éléments du tableau sont égaux est très important ici. Le programme ne doit pas procéder à l'échange dans ce cas, puisque cet échange aurait lieu à chaque passage suivant. L'indicateur serait systématiquement positionné et on aboutirait à une boucle infinie.

Le programme doit réduire de 1 le compteur, puisque le nombre de paires consécutives est de 1 inférieur au nombre d'éléments (le dernier élément n'a pas de successeur). Avant de commencer chaque passage dans la boucle de tir, il est important de ré-initialiser le compteur, le pointeur et l'indicateur d'échange.

Il existe de nombreuses variantes de ce programme. Par exemple, on pourrait utiliser les instructions RES 0,C et SET 0,C pour positionner l'indicateur d'échange à 0 ou 1 au lieu de LD C,0 et LD C,1. On pourrait aussi utiliser la séquence :

```
LD      B,C
DJNZ    TRI        pour tester l'indicateur d'échange.
```

Il est préférable d'utiliser le registre B comme compteur pour la boucle intérieure, puisque ce compteur est décrémenté plus fréquemment. Ceci nous permet d'exploiter l'instruction DJNZ au maximum.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	0E	TRI :	LD	C,0
0001	00			
0002	21		LD	HL,40H
0003	40			
0004	00			
0005	46		LD	B,(HL)
0006	05		DEC	B
0007	23		INC	HL
0008	7E	PASS1:	LD	A,(HL)
0009	23		INC	HL
000A	BE		CP	(HL)
000B	30		JR	NC,CNT
000C	07			
000D	56		LD	D,(HL)
000E	77		LD	(HL),A
000F	2B		DEC	HL
0010	72		LD	(HL),D
0011	23		INC	HL
0012	0E		LD	C,1
0013	01			
0014	10	CNT:	DJNZ	PASS1
0015	F2			
0016	0D		DEC	C
0017	28		JR	Z,TRI
0018	E7			
0019	76		HALT	

L'indexage serait une façon pratique de réaliser l'échange si les registres d'index du Z80 étaient plus accessibles. Essayez de ré-écrire le programme en utilisant un des registres d'index et comparez les temps d'exécution et les utilisations de mémoire des deux programmes.

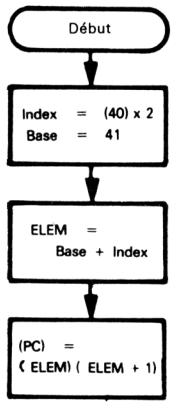
Emploi d'une table de saut ordonnée

But : utiliser le contenu d'adresse 0040 comme l'index d'une table de saut commençant à l'adresse 0041. Chaque élément de cette table contient une adresse 16 bits avec les bits de poids faible dans le premier octet. Le programme doit donner le contrôle à l'adresse avec l'index approprié ; donc, si l'index vaut 6, le programme saute à l'entrée d'adresse #6 dans la table. Considérer que la table possède moins de 128 entrées.

Exemple pratique : (0040) = 02
(0041) = 48
(0042) = 00
(0043) = 4C
(0044) = 00
(0045) = 50
(0046) = 00
(0047) = 54
(0048) = 00

Résultat : (PC) = 0050, puisque c'est l'entrée n° 2 (en partant de zéro) dans la table de saut.

Organigramme :



Le dernier pavé indique le transfert à l'adresse obtenue à partir de la table.

Programme source :

```
LD      HL,40H      ; POINTE SUR INDEX
LD      A,(HL)      ; LECTURE INDEX
ADD     A,A         ; INDEX X 2 POUR OCTETS TABLE
LD      E,A
LD      D,O         ; EXTENSION INDEX SUR 16 BITS
INC     HL          ; ADRESSE DE BASE DE LA TABLE DE SAUT
ADD     HL,DE       ; INDEX DANS TABLE DE SAUT
LD      E,(HL)      ; CHARGER BMS ADRESSE DESTINATION
INC     HL
LD      D,(HL)      ; CHARGER BPS ADRESSE DESTINATION
EX      DE,HL
JP      (HL)        ; TRANSFERT CONTROLE A DESTINATION
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	21	LD	HL,40H
0001	40		
0002	00		
0003	7E	LD	A,(HL)
0004	87	ADD	A,A
0005	5F	LD	E,A
0006	16	LD	D,0
0007	00		
0008	23	INC	HL
0009	19	ADD	HL,DE
000A	5E	LD	E,(HL)
000B	23	INC	HL
000C	56	LD	D,(HL)
000D	EB	EX	DE,HL
000E	E9	JP	(HL)

Les tables de saut sont très utiles dans le cas où un programme doit être choisi parmi plusieurs. De tels cas arrivent lors du décodage de commandes, de la sélection de programmes de test, du choix d'alternatives, ou de la sélection d'une configuration d'entrée/sortie.
La table de saut remplace toute une série d'opérations de sauts conditionnels. Le programme qui utilise la table de saut peut utiliser plusieurs autres tables en changeant simplement l'adresse de départ[3].
L'index correct est obtenu en multipliant par 2 la donnée puisque chaque entrée de la table est une adresse sur 2 octets.

SAUTS INDIRECTS L'instruction JP (HL) qui transfère le contenu du double registre HL au compteur ordinal (PC) constitue un saut indirect qui est très pratique dans les tables de saut et les programmes de contrôle. Notons que l'instruction JP (HL) est une instruction de saut, puisqu'elle met une nouvelle valeur dans le compteur ordinal (PC) ; elle permet cependant de mettre directement une adresse variable dans le compteur ordinal. Toutes les instructions de saut conditionnel (et les instructions d'appel « call ») utilisent des adresses fixes.
Les seules instructions de saut ayant une même souplesse sont les instructions sur 2 octets JP (IX) et JP (IY).
Aucune opération finale n'est nécessaire, puisque JP (HL) transfère le contrôle à l'adresse obtenue à partir de la table de saut.

EXERCICES

1) Retrait d'un élément d'une liste

But : retirer le contenu d'adresse 0040 d'une liste si l'élément y figure. La longueur de la liste se trouve à l'adresse 0041 et la liste elle-même commence à l'adresse 0042. Remettre à jour la liste à partir de l'élément retiré et réduire de 1 la longueur de la liste.

Exemples pratiques :

a.
(0040) = 6B
(0041) = 04
(0042) = 37
(0043) = 61
(0044) = 28
(0045) = 1D

Résultat : pas de changement puisque l'élément ne figure pas sur la liste

b.
(0040) = 6B
(0041) = 04
(0042) = 37
(0043) = 6B
(0044) = 28
(0045) = 1D

Résultat :
(0041) = 03
(0042) = 37
(0043) = 28
(0044) = 1D

L'élément 6B est retiré de la liste et ceux qui suivent sont remontés d'une position, la longueur de la liste est diminuée de 1.

2) Ajout d'un élément à une liste ordonnée

But : mettre le contenu d'adresse 0040 dans une liste ordonnée s'il n'y figure pas déjà. La longueur de la liste se trouve à l'adresse 0041 et la liste elle-même commence à l'adresse 0042 ; la liste est constituée de nombres binaires non signés rangés par ordre croissant. Mettre le nouvel élément à la bonne place dans la liste, repositionner les éléments qui suivent et augmenter de 1 la longueur de la liste.

Exemples pratiques :

a.

(0040) = 6B
(0041) = 04
(0042) = 37
(0043) = 55
(0044) = 7D
(0045) = A1

Résultat :

(0041) = 05
(0044) = 6B
(0045) = 7D
(0046) = A1

b.

(0040) = 6B
(0041) = 04
(0042) = 37
(0043) = 55
(0044) = 6B
(0045) = A1

Résultat : pas de changement, puisque l'entrée figure déjà sur la liste

3) Ajout d'un élément à une file

But : mettre l'adresse contenue en 0040 et 0041 (bits de poids fort en 0041) dans une file. L'adresse du premier élément de la file se trouve en 0042 et 0043 (bits de poids fort en 0043). Chaque élément de la file contient soit l'adresse de l'élément suivant soit zéro s'il est le dernier ; toutes les adresses font 16 bits de long avec les bits de poids fort dans le second octet de l'élément. Le nouvel élément va à la fin de la file : il contient zéro pour marquer la fin de la file et son adresse est contenue par l'élément précédent.

Exemple pratique :

(0040) = 4D
(0041) = 00 } nouvel élément à ajouter à la file
(0042) = 46
(0043) = 00 } pointe sur la tête de file
(0046) = 00
(0047) = 00 } queue de file

Résultat :

(0046) = 4D
(0047) = 00 } ex queue de file pointant vers la nouvelle queue de file
(004D) = 00
(004E) = 00 } nouvelle queue de file

Comment insèreriez-vous un élément à la file si les adresses 0044 et 0045 contenaient la queue de file (dernier élément) ?

4) Tri sur 16 bits

But : trier un tableau de nombres binaires non signés 16 bits suivant l'ordre décroissant. La longueur du tableau se trouve à l'adresse 0040 et le tableau lui-même commence à l'adresse 0041. Chaque nombre 16 bits est rangé avec les bits de poids faible dans le premier octet.

Exemple pratique :

(0040) = 03
(0041) = D1
(0042) = 19
(0043) = 60
(0044) = 3F
(0045) = 2A
(0046) = B5

Résultat : (0041) = 2A
(0042) = B5
(0043) = 60
(0044) = 3F
(0045) = D1
(0046) = 19

Les nombres sont B52A, 3F60 et 19D1.

5) Emploi d'une table de saut avec clé

But : utiliser le contenu d'adresse 0040 comme clé d'une table de saut commençant à l'adresse 0041. Chaque entrée de la table contient une valeur de clé sur 8 bits suivie d'une adresse 16 bits (bits de poids fort dans le second octet) ; c'est à cette adresse que le programme doit donner le contrôle si la clé est bien égale à la valeur de clé.

Exemple pratique :

(0040) = 38
(0041) = 32
(0042) = 4B
(0043) = 00
(0044) = 35
(0045) = 4D
(0046) = 00
(0047) = 38
(0048) = 4F
(0049) = 00

Résultat : (PC) = 004F, puisque cette adresse correspond à la clé de valeur 38.

Essayez d'écrire le programme avec et sans l'instruction CPIR. Avez-vous une idée pour simplifier la version qui utilise l'instruction CPIR ? Conseil : mettre tous les octets correspondants dans des tables séparées de façon à ce que le programme n'ait qu'à ajouter 1 au pointeur de la table pour passer d'une valeur de clé à l'autre[4].

RÉFÉRENCES

1. Knuth décrit d'autres méthodes de recherche dans son livre *The Art of Computer Programming, Volume III : Sorting and Searching.* Addison-Wesley, Reading, Mass., 1978.
   Knuth a aussi traité de façon plus élémentaire les méthodes de recherche et « hashing » dans un article intitulé « Algorithms » (consulter *Scientific American* édition d'avril 1977).
2. Il existe de nombreux algorithmes de tri dont l'efficacité varie très largement. Knuth en décrit quelques-uns dans son livre mentionné ci-dessus : *The Art of Computer Programming, Volume III : Sorting and Searching.*
   Des algorithmes de tri et de recherche sont également traités dans K.A. Schember et J.R. Rumsey, « Minimal Storage Sorting and Searching Techniques for RAM Applications, a Tutorial », *Computer,* juin 1977, pp. 92-100.
3. Il existe d'autres exemples de l'emploi des tables de saut dans L.A. Leventhal « Cut Your Processor's Computation Time », *Electronic Design,* 16 août 1977, pp. 82-89, et dans le chapitre 7 de J.B. Peatman, *Microcomputer-Based Design,* McGraw-Hill, New York, 1977.
4. Cette méthode est traitée par T. Dolhoff dans « Microprocessor Software : How to Optimize Timing and Memory Usage ; Part Four : Techniques for the Zilog Z80 », *Digital Design,* février 1977, pp. 48-49.

CHAPITRE 10

SOUS-PROGRAMMES

Aucun des exemples donnés jusqu'à présent ne constitue un programme en lui-même. La plupart des programmes réels effectuent une série de tâches qui peuvent revenir plusieurs fois dans le même programme ou dans des programmes différents. Il est nécessaire de formaliser ces tâches une fois pour toutes et de rendre cette forme disponible et accessible dans les différentes parties du programme en cours mais aussi pour d'autres programmes.

BIBLIOTHÈQUE DE SOUS-PROGRAMMES La méthode standard consiste à écrire des sous-programmes qui réalisent des tâches particulières. Les séquences d'instructions qui en résultent peuvent être écrites et testées une fois pour toutes, puis réutilisées indéfiniment. On peut constituer une bibliothèque de sous-programmes qui fournit des solutions documentées à des problèmes courants.

INSTRUCTIONS DE SOUS-PROGRAMMES La plupart des microprocesseurs disposent d'instructions spéciales pour transférer le contrôle aux sous-programmes puis le redonner au programme principal. On fait souvent référence à l'instruction spéciale qui donne la main au sous-programme telle que « Call », « Jump-to-Subroutine », « Jump and Mark Place » ou « Jump and Link ».

L'instruction spéciale qui redonne la main au programme principal est couramment appelée « Return ».
Sur le microprocesseur Z80, l'instruction d'appel (CALL) sauvegarde l'ancienne valeur du compteur ordinal (PC) dans la pile RAM avant de mettre l'adresse de départ du sous-programme dans le compteur ordinal ; l'instruction de retour « Return » (RET) va chercher l'ancienne valeur dans la pile et la remet dans le compteur ordinal. Il s'agit de transférer le contrôle du programme d'abord au sous-programme, puis au programme principal. En fait, le sous-programme peut lui-même transférer le contrôle à un sous-programme, et ainsi de suite.
De façon à être vraiment utile, un sous-programme doit être général. Un programme capable de réaliser uniquement une tâche très spécifique, telle que la recherche d'une lettre particulière dans une chaîne d'entrée de longueur fixe, n'est pas vraiment utile. Si, au contraire, ce sous-programme peut rechercher n'importe quelle lettre dans des chaînes de n'importe quelle longueur, il présente bien plus d'intérêt. On appelle paramètres les données ou les adresses susceptibles de varier pour le sous-programme. Une partie importante de l'écriture des sous-programmes consiste à déterminer les paramètres qui seront utilisés.

PASSAGE DE PARAMÈTRES

Un des problèmes est de transférer les paramètres au sous-programme ; c'est ce qu'on appelle le passage des paramètres. La méthode la plus simple consiste pour le programme principal à mettre les paramètres dans des registres. Il suffit ensuite que le sous-programme considère que les paramètres y sont effectivement placés. Bien sûr, cette technique est limitée par le nombre de registres disponibles. Les paramètres peuvent cependant être aussi bien des adresses que des données. Par exemple, un sous-programme de tri pourrait commencer avec l'adresse de départ d'un tableau dans le double registre HL.

D'autres méthodes sont nécessaire lorsque les paramètres sont plus nombreux. Une des possibilités consiste à utiliser la pile. Le programme principal met les paramètres dans la pile et le sous-programme les en retire. Les avantages de cette méthode résident dans la taille illimitée de la pile et dans le fait que les données de la pile ne sont pas perdues même si la pile est réutilisée.

Les inconvénients tiennent aux instructions du Z80, qui utilisent rarement la pile et à l'instruction d'appel (CALL) qui met l'adresse de retour dans la pile. Une autre méthode peut être d'affecter une zone mémoire aux paramètres. Le programme principal met l'adresse de la zone dans le double registre HL ou dans l'un des registres d'index et le sous-programme peut en retirer les données nécessaires. Toutefois, cette procédure est peu commode si les paramètres eux-mêmes représentent des adresses.

SOUS-PROGRAMME RELOGEABLE

Un sous-programme doit parfois posséder des caractéristiques spéciales. Un sous-programme est relogeable s'il peut être mis n'importe où en mémoire. Il est possible d'utiliser facilement un tel programme sans tenir compte de l'emplacement d'autres programmes ou de l'organisation de la mémoire. Un programme vraiment relogeable ne doit pas utiliser d'adresses absolues ; toutes les adresses doivent être relatives au début du programme. Un chargeur translateur (cf. chapitre 2) est nécessaire pour implanter correctement le programme en mémoire ; le chargeur place le programme derrière d'autres programmes et additionne l'adresse de départ ou constante de translation à toutes les adresses du programme.

SOUS-PROGRAMME RÉ-ENTRANT

Un sous-programme est ré-entrant s'il peut être interrompu et appelé par le programme d'interruption et s'il peut néanmoins donner les résultats corrects pour à la fois les programmes interrompu et d'interruption. La ré-entrance est importante pour les sous-programmes standards d'un système qui utilise les interruptions. Sans cela, les programmes d'interruption n'utilisent pas les sous-programmes standards sans causer d'erreurs. Les sous-programmes sur microprocesseurs sont faciles à rendre ré-entrants, puisque l'instruction d'appel (CALL) utilise la pile et que cette procédure est automatiquement ré-entrante. Il reste une seule contrainte : le sous-programme doit utiliser les registres et la pile et non les adresses fixes pour le stockage temporaire*. C'est peu commode mais peut habituellement se faire en cas de nécessité.

Un sous-programme est récursif s'il s'appelle lui-même. Un tel sous-programme doit évidemment être ré-entrant. Cependant, les sous-programmes récursifs sont peu fréquents dans les applications sur microprocesseurs.

* NDT : le programmeur doit cependant sauvegarder (dans la pile) les registres utilisés (sans oublier le registre d'état) sous peine d'avoir de désagréables surprises lors du retour au sous-programme interrompu...

La plupart des programmes sont constitués par un programme principal et plusieurs sous-programmes. Ceci présente l'avantage de pouvoir utiliser des programmes « qui tournent » et de tester et corriger les autres sous-programmes séparément. Il faut, toutefois, faire attention à utiliser correctement les sous-programmes et à se rappeler leur action précise sur les registres et la mémoire.

DOCUMENTATION DES SOUS-PROGRAMMES

DOCUMENTATION DES SOUS-PROGRAMMES

Les listings d'un sous-programme doivent fournir assez d'informations pour que les utilisateurs n'aient pas besoin d'aller examiner sa structure interne.

Voici certaines des spécifications nécessaires :

- une description de l'objet du sous-programme
- une liste des paramètres d'entrée et de sortie
- les registres et la mémoire utilisée
- un exemple pratique.

Si ces conseils sont suivis, le sous-programme sera facile à utiliser.

EXEMPLES

Il est important de noter que les exemples suivants réservent tous une zone de mémoire pour la pile RAM. Si le moniteur de votre microcalculateur dispose d'une zone similaire, elle peut être utilisée à cet effet. Si l'on souhaite essayer de faire sa propre zone de pile, il faut se rappeler de sauvegarder puis de restaurer le pointeur de pile du moniteur pour effectuer un retour correct à la fin du programme principal.
Pour sauvegarder le pointeur de pile du moniteur, il faut utiliser l'instruction LD (addr),SP et pour le restaurer LD SP,(addr). Ces deux instructions nécessitent un code opératoire sur 2 octets (ED 7B pour le chargement du pointeur de pile, ED 73 pour son rangement) en plus des 2 octets d'adresse.
L'adresse 0080 (hexa) a été utilisée comme point de départ de la pile. Il se peut que vous ayez à changer cette adresse par une autre plus adaptée à votre configuration. Pour cela, consultez le manuel de votre microcalculateur.

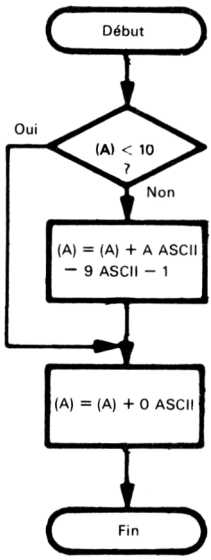
Conversion hexa-ASCII

But : convertir le contenu de l'accumulateur en un caractère ASCII. Mettre le résultat dans l'accumulateur. Considérer que l'accumulateur contient un seul chiffre hexadécimal.

Exemples pratiques :

a. (A) = 0C
Résultat : (A) = 43 'C'
b. (A) = 06
Résultat : (A) = 36 '6'

Organigramme :



Programme source :

Le programme appelant utilise l'adresse 0080 comme début de pile, va chercher la donnée à l'adresse 0040, appelle le sous-programme de conversion et va ranger le résultat à l'adresse 0041.

```
        ORG     0
        LD      SP,80H      ; DEBUT PILE ADRESSE 0080
        LD      A,(40H)     ; LIRE DONNEE
        CALL    ASDEC       ; CONVERSION
        LD      (41H),A     ; RANGER RESULTAT
        HALT
```

Le sous-programme convertit un chiffre hexadécimal en ASCII.

```
        ORG     20H
ASDEC:  CP      10          ; DONNEE : CHIFFRE DECIMAL ?
        JR      C,ASCZ
        ADD     A,'A'-'9'-1 ; NON, AJOUTER DECALAGE LETTRES
ASCZ:   ADD     A,'0'       ; CONVERSION DONNEE EN ASCII
        RET
```

Documentation du sous-programme :

```
;
; BUT : ASDEC CONVERTIT UN CHIFFRE
;       HEXADECIMAL DE L'ACCUMULATEUR
;       EN UN CHIFFRE ASCII DANS L'ACCUMULATEUR
;
; CONDITIONS INITIALES : CHIFFRE HEXA EN A
;
; CONDITIONS FINALES : CARACTERE ASCII EN A
;
; REGISTRES UTILISES : A
;
; EXEMPLE PRATIQUE
;       CONDITIONS INITIALES : 6 DANS L'ACCUMULATEUR
;       CONDITIONS FINALES : 6 ASCII (36 HEXA) DANS
;         L'ACCUMULATEUR
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
1) Programme appelant				
0000	31		LD	SP,80H
0001	80			
0002	00			
0003	3A		LD	A,(40H)
0004	40			
0005	00			
0006	CD		CALL	ASDEC
0007	20			
0008	00			
0009	32		LD	(41H),A
000A	41			
000B	00			
000C	76		HALT	
2) Sous-programme				
0020	FE	ASDEC:	CP	10
0021	0A			
0022	38		JR	C,ASCZ
0023	02			
0024	C6		ADD	A,'A'-'9'-1
0025	07			
0026	C6	ASCZ:	ADD	A,'0'
0027	30			
0028	C9		RET	

L'instruction LD SP,80H place le sommet de pile à l'adresse 0080. Il faut se rappeler que la pile évolue vers le bas (vers des adresses plus basses). On met couramment la pile à l'extrémité supérieure de la RAM (c'est-à-dire la plus haute adresse) de façon à ne pas interférer avec un autre rangement temporaire de données.
L'instruction d'appel (« CALL ») met l'adresse de départ du sous-programme (0020 hexa) dans le compteur ordinal et sauvegarde l'ancienne valeur du compteur ordinal dans la pile.

Voici la procédure :

Etape 1) Décrémentation du pointeur de pile, sauvegarde des bits de poids fort de l'ancienne valeur du compteur ordinal dans la pile.
Etape 2) Décrémentation du pointeur de pile, sauvegarde des bits de poids faible de l'ancienne valeur du compteur ordinal dans la pile.

Notons que le pointeur de pile du Z80 contient en permanence l'adresse du dernier emplacement occupé dans la pile.
Le résultat dans ce cas est :

(007F) = 00
(007E) = 09
(SP) = 007E

La valeur sauvegardée est celle du compteur programme lorsque le processeur a terminé de chercher en mémoire l'instruction d'appel complète. Notons que l'adresse est rangée comme les autres adresses Z80, les bits de poids faible dans la plus petite adresse.
L'instruction de retour (« RET ») charge le compteur ordinal avec le contenu des 2 adresses les plus basses de la pile.
Voici la procédure :

Etape 1) Chargement des 8 bits de la pile dans les 8 bits de poids faible du compteur ordinal. Incrémentation du pointeur de pile.
Etape 2) Chargement des 8 bits de la pile dans les 8 bits de poids fort du compteur ordinal. Incrémentation du pointeur de pile.

Le résultat dans ce cas est :

(PC) = (007F) et (007E)
= 0009
(SP) = 0080

Ce sous-programme n'a qu'un seul paramètre d'entrée et ne produit qu'un seul résultat. L'accumulateur est l'endroit idéal pour les y ranger tous deux.
Le programme appelant comprend 3 étapes : mettre la donnée dans l'accumulateur, appeler le sous-programme, et ranger le résultat. L'initialisation générale doit aussi mettre la pile dans la zone mémoire appropriée.
Le sous-programme est relogeable puisque l'adresse ASCZ est relative.
Remarquons que l'instruction CALL se traduit par l'exécution de 4 ou 5 instructions prenant 36 ou 38 cycles d'horloge. Un appel à un sous-programme peut prendre beaucoup de temps même s'il n'apparaît que comme une simple instruction du programme.
Si l'on pense utiliser la pile pour les paramètres, on doit se rappeler que l'instruction CALL met l'adresse de retour au sommet de la pile. On peut incrémenter le pointeur de pile 2 fois (INC SP) pour obtenir cette adresse de retour, mais on doit alors remettre à jour le pointeur de pile avant de revenir au programme principal. On peut aussi transférer le pointeur de pile dans les registres H et L à l'aide de la séquence d'instructions :

```
LD      HL,0
ADD     HL,SP       ; TRANSFERT POINTEUR PILE VERS REGISTRE
```

On peut alors utiliser l'adressage mémoire implicite avec H et L pour obtenir la donnée contenue dans la pile. Une autre solution consiste à transférer le pointeur de pile dans le registre d'index (IX) avec la séquence :

```
LD      IX,0
ADD     IX,SP        ; TRANSFERT POINTEUR PILE VERS REGISTRE INDEX
```

L'avantage de cette solution est que l'on peut alors obtenir les données et les adresses de la pile avec des déplacements d'index. De plus, le double registre HL peut immédiatement servir dans le sous-programme. Notons que les instructions LD SP,HL ou LD SP,IX peuvent servir à remettre à jour la valeur du pointeur de pile.

Longueur d'une chaîne de caractères

But : déterminer la longueur d'une chaîne de caractères ASCII. L'adresse de début de chaîne se trouve dans le double registre HL. La fin de la chaîne est signalée par un caractère de retour chariot (CR, 0D hexa). Mettre la longueur de la chaîne (hormis le retour chariot) dans l'accumulateur.

Exemples pratiques :

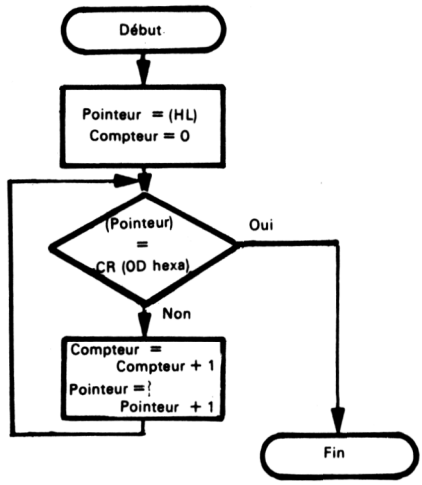
a.
(HL) = 0043
(0043) = 0D

Résultat : (A) = 00

b.
(HL) = 0043
(0043) = 52 'R'
(0044) = 41 'A'
(0045) = 54 'T'
(0046) = 48 'H'
(0047) = 45 'E'
(0048) = 52 'R'
(0049) = 0D CR

Résultat : (A) = 06

Organigramme :



Programme source :

Le programme appelant utilise l'adresse 0080 comme sommet de pile, va chercher l'adresse de départ de la chaîne aux adresses 0040 et 0041, appelle le sous-programme de longueur de chaîne et range le résultat à l'adresse 0042.

```
        LD      SP,80H      ; DEBUT PILE ADRESSE 0080
        LD      HL,(40H)    ; LIRE ADRESSE DEBUT CHAINE
        CALL    STLEN       ; CALCUL LONGUEUR CHAINE
        LD      (42H),A     ; RANGER LONGUEUR CHAINE
        HALT
```

Le sous-programme calcule la longueur d'une chaîne de caractères ASCII et la met dans l'accumulateur.

```
        ORG     20H
CHLON : LD      B,0         ; LONGUEUR CHAINE = 0
        LD      A,0DH       ; LIRE RETOUR CHARIOT ASCII
CNTCR : CP      (HL)        ; CARACTERE = RETOUR CHARIOT ?
        JR      Z,FINI      ; OUI, FIN DE CHAINE
        INC     B           ; NON, + 1 SUR LONGUEUR CHAINE
        INC     HL
        JR      CNTCR
FINI :  LD      A,B
        RET
```

Documentation du sous-programme :

```
;
; SOUS-PROGRAMME CHLON
;
; BUT : CHLON CALCULE LA LONGUEUR D'UNE
;     CHAINE (NOMBRE DE CARACTERES AVANT
;     UN RETOUR CHARIOT)
;
; CONDITIONS INITIALES : ADRESSE DE DEBUT DE
;     CHAINE DANS LE DOUBLE REGISTRE HL
;
; REGISTRES UTILISES : A, B, H, L
;
; EXEMPLE PRATIQUE :
;
; CONDITIONS INITIALES : (HL) = 0043
;     (0043) = 35, (0044) = 46, (0045) = 0D
; CONDITIONS FINALES : (A) = 02
;
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
1) Programme appelant			
0000	31	LD	SP,80H
0001	80		
0002	00		
0003	2A	LD	HL,(40H)
0004	40		
0005	00		
0006	CD	CALL	CHLON
0007	20		
0008	00		
0009	32	LD	(42H),A
000A	42		
000B	00		
000C	76	HALT	
2) Sous-programme			
0020	06	CHLON : LD	B,0
0021	00		
0022	3E	LD	A,0DH
0023	0D		
0024	BE	CNTCR : CP	(HL)
0025	28	JR	Z,FINI
0026	04		
0027	04	INC	B
0028	23	INC	HL
0029	18	JR	CNTCR
002A	F9		
002B	78	FINI : LD	A,B
002C	C9	RET	

Le programme comprend 4 parties : l'initialisation du pointeur de pile, le chargement de l'adresse de début de chaîne dans le double registre HL, l'appel au sous-programme et le rangement en mémoire du résultat.

Le sous-programme est relogeable puisque toutes les instructions de saut utilisent des adresses relatives.

Le sous-programme modifie le registre B, l'adresse dans le double registre HL ainsi que l'accumulateur. Le programmeur doit être conscient que la donnée précédemment rangée dans le registre B ainsi que l'adresse précédemment chargée dans HL seront perdues ; la documentation du sous-programme doit préciser quels registres sont utilisés.

Le moyen de prévenir la perte du contenu du registre est de le sauvegarder dans la pile puis de le recharger avant de revenir au programme appelant. Cette solution simplifie le programme appelant, mais est plus coûteuse en temps et mémoire (dans le programme et dans la pile).

Ce sous-programme n'a qu'un seul paramètre en entrée : une adresse. Le meilleur moyen de passer ce paramètre est d'utiliser un double registre et, comme le double registre HL est de loin le plus souple pour les options d'adressage, le meilleur choix est HL.

Le sous-programme contient une instruction de saut inconditionnel : JR CNTCR. Est-il possible d'éviter ce saut en modifiant les conditions initiales avant d'entrer dans la boucle du sous-programme ?

Si le caractère de fin de chaîne n'était pas systématiquement le retour chariot, on pourrait paramétrer ce caractère. Le programme appelant devrait alors mettre le caractère de fin de chaîne dans l'accumulateur et l'adresse de début de chaîne dans le double registre HL avant d'appeler le sous-programme.
Un moyen de passer les paramètres indépendamment des données variables consiste à mettre les valeurs en mémoire immédiatement après l'instruction d'appel. On peut utiliser l'ancienne valeur du compteur ordinal (sauvegardée au sommet de la pile), pour accéder à la donnée mais il est nécessaire de remettre correctement cette valeur à jour avant de redonner le contrôle au programme principal. Par exemple, si l'on passe la valeur du caractère de fin de chaîne de cette façon, le programme principal et le sous-programme deviennent :

Programme appelant :

```
        ORG     0
        LD      SP,80H      ; DEBUT PILE ADRESSE 0080
        LD      (HL),40H    ; LIRE ADRESSE DEBUT CHAINE
        CALL    CHLON       ; CALCUL LONGUEUR CHAINE
        DEFB    '.'         ; CARACTERE FINAL = POINT ASCII
        LD      (42H),A     ; RANGER LONGUEUR CHAINE
        HALT
```

Sous-programme :

```
        ORG     20H
CHLON : POP     DE          ; DEBUT LISTE PARAMETRES
        LD      A,(DE)      ; LIRE CARACTERE FINAL
        INC     DE          ; MAJ ADRESSE RETOUR
        PUSH    DE
        LD      B,0         ; LONGUEUR CHAINE = 0
CNTCR : CP      (HL)        ; CARACTERE = CARACTERE FINAL ?
        JR      Z,FINI      ; OUI, FIN DE CHAINE
        INC     B           ; NON, + 1 SUR LONGUEUR CHAINE
        INC     HL
        JR      CNTCR
FINI :  LD      A,B
        RET
```

Ce sous-programme est plus long et utilise le double registre DE, mais le programme appelant n'a pas besoin de charger dans un registre le caractère final. L'instruction INC DE est indispensable pour permettre le retour à l'instruction suivante et non à la liste de paramètres[1].
Les instructions PUSH et POP transfèrent le contenu des doubles registres ou des registres d'index dans et à partir de la pile RAM. Les 8 bits de poids faible sont enlevés les premiers et rangés les derniers pour conserver la logique de la méthode de rangement des adresses 16 bits du Z80. Il est important de se rappeler que la pile RAM évolue de haut en bas (vers les adresses les plus basses).

Calcul de parité sur les caractères ASCII

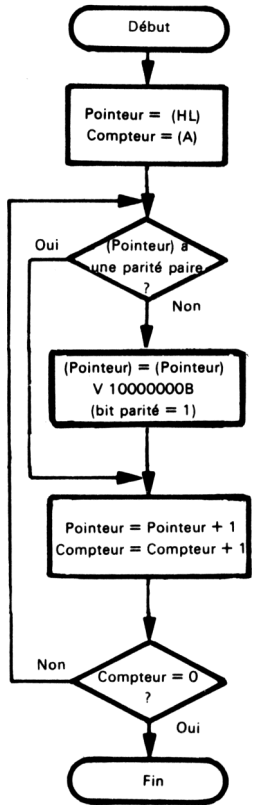
But : assigner une parité paire à une chaîne de caractères ASCII 7 bits. La longueur de la chaîne se trouve dans l'accumulateur et l'adresse de début de chaîne dans le double regis-

tre HL. Mettre la parité paire dans le bit le plus significatif de chaque caractère, c'est-à-dire mettre à 1 le bit le plus significatif si cela rend pair le nombre total de bits à 1 de l'octet.

Exemple pratique :

	(A)	= 06
	(HL)	= 0041
	(0041)	= 31
	(0042)	= 32
	(0043)	= 33
	(0044)	= 34
	(0045)	= 35
	(0046)	= 36
Résultat :	(0041)	= B1
	(0042)	= B2
	(0043)	= 33
	(0044)	= B4
	(0045)	= 35
	(0046)	= 36

Organigramme :



Programme source :

Le programme appelant utilise l'adresse 0080 comme sommet de pile, met l'adresse de début de chaîne en 0041, va chercher la longueur de la chaîne à l'adresse 0030 et appelle le sous-programme.

```
        ORG     0
        LD      SP,80H      ; DEBUT PILE ADRESSE 0080
        LD      HL,41H      ; LIRE ADRESSE DEBUT CHAINE
        LD      A,(30H)     ; LIRE LONGUEUR CHAINE
        CALL    PARP
        HALT
```

Le sous-programme ajoute la parité paire à une chaîne de caractères ASCII.

```
        ORG     20H
PARP :  LD      B,A
        LD      C,10000000B ; CHARGER 1 DANS BIT PARITE
BPAR1 : LD      A,(HL)      ; LIRE CARACTERE
        OR      C           ; METTRE A 1 BIT PARITE
        JP      PO,CNTL0    ; PARITE PAIRE ?
        LD      (HL),A      ; OUI, SAUVEGARDE CARACTERE
CNTL0 : INC     HL
        DJNZ    BPAR1
        HALT
```

Documentation du sous-programme :

```
;
; SOUS-PROGRAMME PARP
; BUT : PARP ASSIGNE UNE PARITE PAIRE
;     A UNE CHAINE DE CARACTERES
;     ASCII 7 BITS
;
; CONDITIONS INITIALES : ADRESSE DE
;     DEBUT DE CHAINE EN HL, LONGUEUR
;     DE LA CHAINE EN A
;
; CONDITIONS FINALES : PARITE PAIRE DANS
;     LE BIT DE POIDS FORT DE CHAQUE
;     CARACTERE
;
; REGISTRES UTILISES : A, B, C, H, L
;
; EXEMPLE PRATIQUE :
;     CONDITIONS INITIALES : (HL) = 0041
;     (A) = 2, (0041) = 32, (0042) = 33
;     CONDITIONS FINALES (0041) = B2
;                        (0042) = 33
```

Ce sous-programme possède 2 paramètres, une adresse et un nombre. Le double registre HL sert à passer l'adresse et l'accumulateur, le nombre. Il n'y a pas de résultats explicites en retour, puisque le sous-programme modifie uniquement chaque bit de poids fort de la chaîne.

Le programme appelant doit mettre l'adresse de départ dans le double registre HL et la longueur de la chaîne dans l'accumulateur avant de transmettre le contrôle au sous-programme.
Le sous-programme modifie les valeurs des registres A, H et L et utilise les registres B et C pour un stockage provisoire.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)		Instruction (Mnémonique)	
1) Programme appelant				
0000	31		LD	SP,80H
0001	80			
0002	00			
0003	21		LD	HL,41H
0004	41			
0005	00			
0006	3A		LD	A,(30H)
0007	30			
0008	00			
0009	CD		CALL	PARP
000A	20			
000B	00			
000C	76		HALT	
2) Sous-programme				
0020	47	PARP :	LD	B,A
0021	0E		LD	C,10000000B
0022	80			
0023	7E	BPAR1 :	LD	A,(HL)
0024	B1		OR	C
0025	E2		JP	PO,CNTL0
0026	29			
0027	00			
0028	77		LD	(HL),A
0029	23	CNTL0 :	INC	HL
002A	10		DJNZ	BPAR1
002B	F7			
002C	C9		RET	

Egalité de chaînes

But : comparer 2 chaînes de caractères ASCII pour voir si elles sont égales. La longueur des chaînes se trouve dans l'accumulateur. Le double registre HL contient l'adresse de départ d'une des chaînes, le double registre DE contient l'autre. Si les 2 chaînes sont égales, mettre à zéro l'accumulateur, sinon le mettre à FF (hexa).

Exemples pratiques :

a.

(A) = 03
(DE) = 50
(HL) = 60

(0050) = 43 'C'
(0051) = 41 'A'
(0052) = 54 'T'

(0060) = 43 'C'
(0061) = 41 'A'
(0062) = 54 'T'

Résultat : (A) = 0, puisque les chaînes sont les mêmes

b.

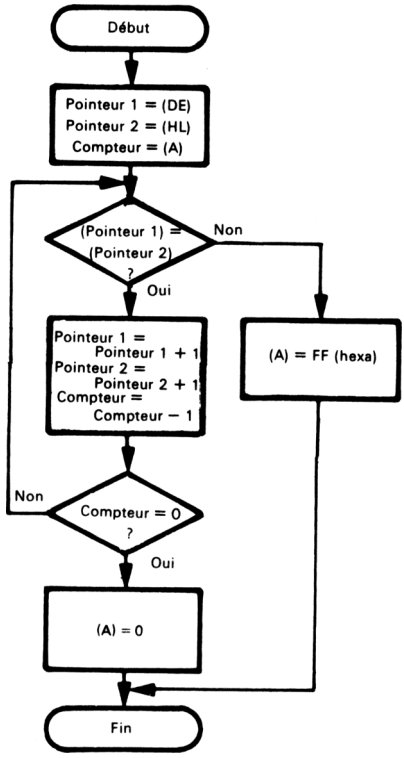
(A) = 03
(DE) = 50
(HL) = 60

(0050) = 52 'R'
(0051) = 41 'A'
(0052) = 54 'T'

(0060) = 43 'C'
(0061) = 41 'A'
(0062) = 54 'T'

Résultat : (A) = FF (hexa), puisque les premiers caractères sont différents.

Organigramme :



Programme source :

Le programme source appelant utilise l'adresse 0080 comme sommet de pile, met les adresses de début de chaîne respectivement aux adresses 0050 et 0060, va chercher la longueur de la chaîne à l'adresse 0040, appelle le sous-programme d'égalité de chaînes et met le résultat à l'adresse 0041.

```
        ORG     0
        LD      SP,80H      ; ADRESSE DEBUT PILE = 0080
        LD      DE,60H      ; LIRE ADRESSE DEPART CHAINE 1
        LD      HL,50H      ; LIRE ADRESSE DEPART CHAINE 2
        LD      A,(40H)     ; LIRE LONGUEUR CHAINE
        CALL    EGACH       ; RECHERCHE EGALITE DES CHAINES
        LD      (41H),A     ; SAUVEGARDE INDICATEUR EGALITE
        HALT
```

Le sous-programme détermine si les 2 chaînes sont égales.

```
        ORG     20H
EGACH : LD      B,A         ; COMPTEUR = LONGUEUR CHAINE
        LD      C,0FFH      ; MARQUE = FF (HEXA) SI CHAINES DIFFERENTES
CHCAR:  LD      A,(DE)      ; LECTURE CARACTERE CHAINE 1
        CP      (HL)        ; EGALITE AVEC CHAINE 2 ?
        JR      NZ,FINI     ; NON, FINI - CHAINES DIFFERENTES
        INC     DE
        INC     HL
        DJNZ    CHCAR
        LD      C,0         ; MARQUE = 0, MEMES CHAINES
FINI :  LD      A,C
        RET
```

Documentation du sous-programme :

```
;
; SOUS-PROGRAMME EGACH
;
; BUT : EGACH DETERMINE SI 2 CHAINES
;     SONT EGALES
;
; CONDITIONS INITIALES : ADRESSES DE DEBUT
;     DE CHAINE DANS DE ET HL
;     LONGUEUR DES CHAINES DANS
;     L'ACCUMULATEUR
;
; CONDITIONS FINALES : 0 DANS A EN CAS
;     D'EGALITE, SINON FF DANS A
;
; REGISTRES UTILISES : A, B, D, E, H, L
;
; EXEMPLE PRATIQUE :
;     CONDITIONS INITIALES : (HL) = 0050
;     (DE) = 0060, (A) = 2, (0050) = 36
;     (0051) = 39, (0060) = 36, (0061) = 39
;     CONDITIONS FINALES : (A) = 0 EGALITE DES CHAINES
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
1) Programme appelant			
0000	31	LD	SP,80H
0001	80		
0002	00		
0003	11	LD	DE,60H
0004	60		
0005	00		
0006	21	LD	HL,50H
0007	50		
0008	00		
0009	3A	LD	A,(40H)
000A	40		
000B	00		
000C	CD	CALL	EGACH
000D	20		
000E	00		
000F	32	LD	(41H),A
0010	41		
0011	00		
0012	76	HALT	
2) Sous-programme			
0020	47	EGACH : LD	B,A
0021	0E	LD	C,0FFH
0022	FF		
0023	1A	CHCAR: LD	A,(DE)
0024	BE	CP	(HL)
0025	20	JR	NZ,FINI
0026	06		
0027	13	INC	DE
0028	23	INC	HL
0029	10	DJNZ	CHCAR
002A	F8		
002B	0E	LD	C,0
002C	00		
002D	79	FINI : LD	A,C
002E	C9	RET	

Ce sous-programme, comme les précédents, modifie tous les indicateurs. Il faut généralement considérer qu'un sous-programme peut modifier les indicateurs, à moins que le contraire ne soit précisé de façon explicite. Si le programme principal a besoin des anciennes valeurs des indicateurs (pour un contrôle ultérieur), il doit les sauvegarder dans la pile avant d'appeler le sous-programme. Ceci peut être fait avec l'instruction PUSH AF.
Ce sous-programme possède 3 paramètres — les 2 adresses de début de chaîne et la longueur de la chaîne. Ces paramètres utilisent 5 registres à usage général.

Addition à multiple précision

But : ajouter 2 nombres binaires de plusieurs octets. La longueur des nombres en octets se trouve dans l'accumulateur. Les adresses de début de ces nombres se trouvent dans les

registres DE et HL. L'adresse de début du résultat se trouve dans le registre d'index IX. Tous les nombres commencent par leurs bits de poids faibles.

Exemple pratique :

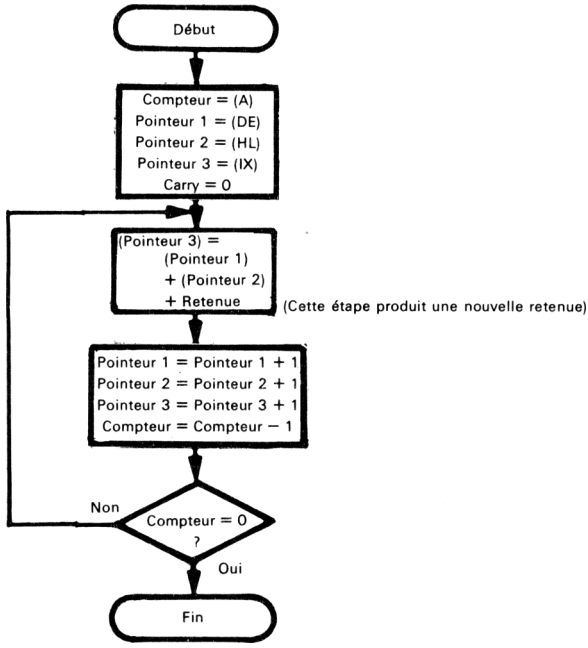
(A) = 04
(DE) = 51
(HL) = 61
(IX) = 71

(0051) = C3
(0052) = A7
(0053) = 5B
(0054) = 2F
(0061) = B8
(0062) = 35
(0063) = DF
(0064) = 14

Résultat : = (0071) = 7B
(0072) = DD
(0073) = 3A
(0074) = 44

soit : 2F5BA7C3
+ 14DF35B8
443ADD7B

Organigramme :



Programme source :

Le programme appelant utilise l'adresse 0080 comme début de pile, met les adresses de départ des différents nombres en 0050, 0060 et 0070 respectivement, va chercher la longueur des nombres à l'adresse 0040 et appelle le sous-programme d'addition à multiple précision.

```
        ORG     0
        LD      SP,80H      ; ADRESSE DEBUT PILE : 0080
        LD      HL,50H      ; LIRE ADRESSE DEBUT NOMBRE 1
        LD      DE,60H      ; LIRE ADRESSE DEBUT NOMBRE 2
        LD      IX,70H      ; LIRE ADRESSE DEBUT RESULTAT
        LD      A,(40H)     ; LIRE LONGUEUR NOMBRES (OCTETS)
        CALL    MPADD       ; ADDITION MULTIPLE PRECISION
        HALT
```

Le sous-programme effectue une addition binaire sur plusieurs octets.

```
        ORG     20H
MPADD:  LD      B,A         ; COMPTEUR = NOMBRE D'OCTETS DES NOMBRES
        AND     A           ; RAZ RETENUE AVANT DE COMMENCER
ADDM :  LD      A,(DE)      ; LIRE MOT 1er NOMBRE
        ADC     A,(HL)      ; SOMME MOT 2e NOMBRE
        LD      (IX),A      ; RANGER 1 MOT DU RESULTAT
        INC     DE
        INC     HL
        INC     IX
        DJNZ    ADDM
        RET
```

Documentation du sous-programme :

```
;
; SOUS-PROGRAMME MPADD
;
; BUT : MPADD ADDITIONNE 2 NOMBRES BINAIRES
;     DE PLUSIEURS OCTETS
;
; CONDITIONS INITIALES : ADRESSES DE DEPART DES
;     NOMBRES DANS D ET E, H ET L, ADRESSE
;     DE DEPART DU RESULTAT DANS IX,
;     LONGUEUR DES NOMBRES DANS A
;
; REGISTRES UTILISES A, B, D, E, H, H, IX
;
; EXEMPLE PRATIQUE :
;     CONDITIONS INITIALES : (HL) = 0050,
;     (DE) = (0060), (IX) = (0070), (A) = 2,
;     (0050) = C3, (0051) = A7, (0060) = B8, (0061) = 35
; CONDITIONS FINALES : (0070) = 7B, (0071) = DD
;
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
1) Programme appelant			
0000	31	LD	SP,80H
0001	80		
0002	00		
0003	21	LD	HL,50H
0004	50		
0005	00		
0006	11	LD	DE,60H
0007	60		
0008	00		
0009	DD	LD	IX,70H
000A	21		
000B	70		
000C	00		
000D	3A	LD	A,(40H)
000E	40		
000F	00		
0010	CD	CALL	MPADD
0011	20		
0012	00		
0013	76	HALT	
2) Sous-programme			
0020	47	LD	B,A
0021	A7	AND	A
0022	1A	ADDM : LD	A,(DE)
0023	8E	ADC	A,(HL)
0024	DD	LD	(IX),A
0025	77		
0026	00		
0027	13	INC	DE
0028	23	INC	HL
0029	DD	INC	IX
002A	23		
002B	10	DJNZ	ADDM
002C	F5		
002D	C9	RET	

Le registre d'index IX est utilisé pour contenir l'adresse du résultat. Essayez de modifier le programme en utilisant le registre BC à cet effet. Que se passe-t-il pour le compteur ? On pourrait aussi mettre l'adresse du résultat au sommet de la pile. L'instruction EX (SP),HL échange le sommet de la pile avec le double registre HL. Modifiez le programme en utilisant cette instruction mais n'oubliez pas d'incrémenter les 3 pointeurs après chaque itération.
Ce sous-programme possède 4 paramètres — 3 adresses et la longueur des nombres. Six registres 8 bits et le registre d'index 16 bits IX servent pour le passage des paramètres.

EXERCICES

Ces exercices consistent à écrire à la fois un programme appelant pour l'exemple pratique et un sous-programme correctement documenté.

1) Conversion ASCII-Hexa

But : convertir la représentation ASCII d'un chiffre hexadécimal contenue dans l'accumulateur en la représentation binaire 4 bits de ce chiffre. Mettre le résultat dans l'accumulateur.

Exemples pratiques :

a. (A) = 43 'C'
Résultat : (A) = 0C

b. (A) = 36 '6'
Résultat : (A) = 06

2) Longueur d'un message ASCII

But : calculer la longueur d'un message codé en ASCII. L'adresse de départ de la chaîne de caractères dans laquelle est rangé le message se trouve dans le double registre HL. Le message lui-même commence par le caractère ASCII STX (02 hexa) et se termine par le caractère ASCII ETX (03 hexa). Mettre la longueur du message (le nombre de caractères entre STX et ETX) dans l'accumulateur.

Exemple pratique :

(HL) = 0041
(0041) = 49
(0042) = 02 STX
(0043) = 47 'G'
(0044) = 4F 'O'
(0045) = 03 ETX
Résultat : (A) = 02

3) Contrôle de la parité paire dans des caractères ASCII

But : vérifier la parité paire d'une chaîne de caractères ASCII. La longueur de la chaîne se trouve dans l'accumulateur et l'adresse de début de chaîne dans le double registre HL. Si la parité de tous les caractères de la chaîne est correcte, mettre l'accumulateur à zéro ; sinon, le mettre à FF (hexa).

Exemples pratiques :

a. (A) = 03
(HL) = 0042
(0042) = B1
(0043) = B2
(0044) = 33
Résultat : (A) = 00, puisque tous les caractères ont une parité paire

b.

(A) = 03
(HL) = 0042
(0042) = B1
(0043) = B6
(0044) = 33

Résultat : (A) = FF, puisque le caractère de l'adresse 0043 n'a pas une parité paire

4) Comparaison de chaînes

But : comparer 2 chaînes de caractères ASCII pour voir laquelle est la plus grande (c'est-à-dire laquelle suit l'autre dans l'ordre alphabétique).
La longueur des chaînes se trouve dans l'accumulateur ; l'adresse de début de la chaîne 1 se trouve dans le double registre HL et celle de la chaîne 2 dans le double registre DE. Si la chaîne 1 est supérieure ou égale à la chaîne 2, mettre à zéro l'accumulateur ; sinon, mettre à FF (hexa) l'accumulateur.

Exemples pratiques :

a.

(A) = 03
(DE) = 0060
(HL) = 0050
(0050) = 43 'C'
(0051) = 41 'A'
(0052) = 54 'T'

(0062) = 42 'B'
(0063) = 41 'A'
(0064) = 54 'T'

Résultat : = (A) = 00, puisque CAT>BAT

b.

(A) = 03
(DE) = 0060
(HL) = 0050
(0050) = 44 'D'
(0051) = 4F 'O'
(0052) = 47 'G'

(0060) = 44 'D'
(0061) = 4F 'O'
(0062) = 47 'G'

Résultat : = (A) = 00, puisque les 2 chaînes sont égales

c.

(A) = 03
(DE) = 0060
(HL) = 0050
(0050) = 43 'C'
(0051) = 41 'A'
(0052) = 54 'T'

(0060) = 43 'C'
(0061) = 55 'U'
(0062) = 54 'T'

Résultat : = (A) = FF (hexa), puisque CUT>CAT

5) Soustraction décimale

But : soustraire un nombre décimal sur plusieurs chiffres DCB d'un autre nombre. La longueur des nombres (en octets) se trouve dans l'accumulateur et les adresses de départ des nombres sont contenues dans les doubles registres DE et HL. Soustraire le nombre dont l'adresse de départ est en HL de celui dont l'adresse de départ est en DE. L'adresse de départ du résultat se trouve dans le registre d'index IX. Tous les nombres commencent par leurs chiffres les moins significatifs. Le signe du résultat est donné dans l'accumulateur — zéro, si le résultat est positif, FF (hexa) s'il est négatif.

Exemple pratique :

	(A)	=	04
	(DE)	=	0050
	(HL)	=	0060
	(IX)	=	0070
	(0050)	=	85
	(0051)	=	19
	(0052)	=	70
	(0053)	=	36
	(0060)	=	59
	(0061)	=	34
	(0062)	=	66
	(0063)	=	12
Résultat :	(A)	=	00 (positif)
	(0070)	=	26
	(0071)	=	85
	(0072)	=	03
	(0073)	=	24

soit :

```
  36701985
- 12663459
----------
+ 24038526
```

RÉFÉRENCES

1. D'autres exemples de cette méthode (pour le microprocesseur 8080) se trouvent dans : S. Mazor et C. Pitchford, « Develop Cooperative Microprocessor Subroutines », *Electronic Design,* 7 juin 1978, pp. 116-118.

CHAPITRE 11

ENTRÉES/SORTIES

Il y a deux problèmes dans la conception des sections d'entrées/sorties : l'un est l'interface entre les périphériques et le calculateur ainsi que la transmission des données et des signaux de contrôle, l'autre est l'adressage des organes d'E/S de façon à ce que l'UC puisse sélectionner un organe d'E/S particulier pour transmettre les données. En fait, le premier problème est à la fois plus complexe et plus intéressant. Nous allons donc nous pencher maintenant sur l'interfaçage des périphériques et laisser l'adressage pour un livre plus orienté sur le matériel.

E/S ET MÉMOIRE

En théorie, le transfert des données vers ou à partir d'un organe d'E/S est similaire au transfert de données vers ou à partir de la mémoire. En fait, on peut considérer la mémoire comme un autre organe d'E/S. Toutefois, c'est un organe particulier pour les raisons suivantes :

1) La mémoire opère à une vitesse proche de celle du processeur.
2) Elle utilise le même type de signaux que le CPU. Les seuls circuits généralement utilisés pour relier la mémoire au CPU sont les amplificateurs et les récepteurs.
3) Elle ne nécessite aucun format spécial, ni signaux de contrôle en dehors d'un signal de lecture/écriture.
4) Elle conserve automatiquement la donnée qui lui est envoyée.
5) Sa longueur de mot est généralement la même que celle du calculateur.

La plupart des organes d'E/S n'ont pas des caractéristiques aussi pratiques. Ils opèrent généralement à des vitesses très inférieures à celle du processeur ; par exemple, un téléimprimeur ne peut transférer que 10 caractères à la seconde alors qu'un processeur lent transfère environ 10 000 caractères par seconde. La gamme des vitesses est aussi très étendue — un **capteur** fournit une lecture par minute alors qu'un écran vidéo ou une disquette peut transférer 250 000 bits par seconde. De plus, certains organes d'E/S ont besoin de signaux continus (moteurs ou thermomètres), de courants plutôt que de tensions (télé-imprimeurs), ou des tensions à des niveaux très différents des signaux utilisés par le processeur (afficheurs à décharge de gaz). Certains organes d'E/S ont aussi besoin de formats, de protocoles ou de signaux de contrôle spéciaux. Leurs longueurs de mot peuvent être bien plus courtes ou plus longues que celles du calculateur. Ces variations rendent difficiles la conception des sections d'E/S et particulièrement le problème d'interfaçage de chaque périphérique.

CATÉGORIES D'E/S

On peut, cependant, donner une description générale des organes d'E/S et des méthodes d'interfaçage. On peut grossièrement séparer les organes en 3 catégories, à partir de leur vitesse de traitement des données :

1) Les organes lents qui ne changent d'état pas plus d'une fois par seconde. Un changement d'état prend généralement quelques millisecondes ou plus. On compte parmi cette catégorie d'organes d'E/S les écrans lumineux, les interrupteurs, les relais et de nombreux capteurs mécaniques et d'activateurs.
2) Les organes à vitesse moyenne qui transmettent des données à des vitesses de 1 à 10 000 bits par seconde. On compte parmi eux les claviers, les imprimantes, les lecteurs de cartes, les lecteurs et perforeurs de rubans, les lecteurs de cassettes, les lignes de communications ordinaires et de nombreux systèmes d'acquisition de données analogiques.
3) Les organes très rapides qui transmettent les données à des vitesses supérieures à 10 000 bits par seconde. Ce sont les bandes magnétiques, les disques magnétiques, les imprimantes à grande vitesse, les lignes de communications rapides et les écrans vidéo.

INTERFAÇAGE DES ORGANES LENTS

L'interfaçage des organes lents est simple. Peu de signaux de contrôle sont nécessaires à moins que les appareils ne soient multiplexés, c'est-à-dire lorsque plusieurs partagent le même port d'E/S, comme on peut le voir sur les schémas 11.1 à 11.4. Les données en entrée à partir d'organes d'E/S lents n'ont pas besoin d'être mémorisées, puisque leur état est stable pendant un grand intervalle de temps. Les données en sortie doivent bien sûr être mémorisées. Les seuls problèmes de l'entrée sont les transitions qui apparaissent quand le calculateur lit les données. Des monostables, des latches rétrocouplées, ou des temporisations logicielles peuvent traiter ces problèmes.
Un seul port d'E/S peut gérer plusieurs organes lents. La figure 11.1 montre un démultiplexeur qui dirige automatiquement la donnée en sortie vers l'organe adéquat en comp-

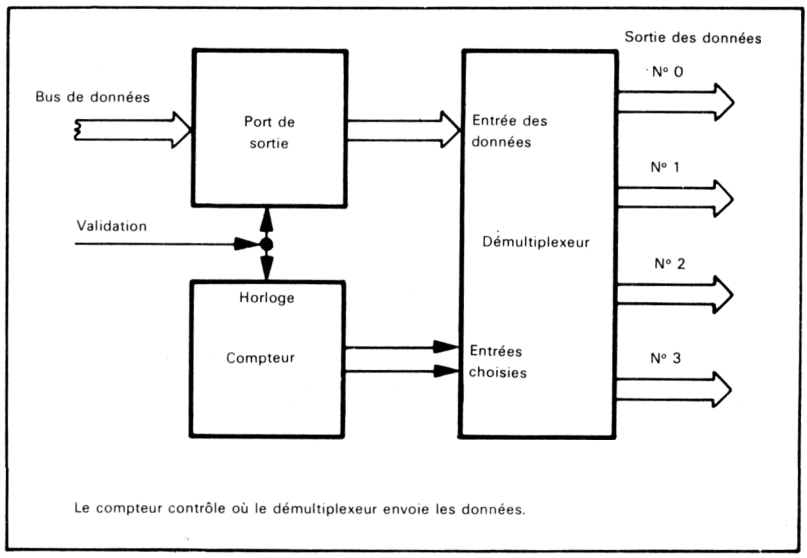


Fig. 11.1. — Un démultiplexeur en sortie contrôlé par un compteur.

tant les opérations de sortie. La figure 11.2 montre un port de contrôle qui fournit au démultiplexeur les entrées sélectionnées. Les sorties des données viennent ici dans n'importe quel ordre, mais une instruction supplémentaire de sortie est nécessaire pour changer l'état du port de contrôle. Les démultiplexeurs en sortie servent à piloter plusieurs afficheurs à partir du même port de sortie. Les figures 11.3 et 11.4 montrent les mêmes possibilités pour un multiplexeur en entrée.

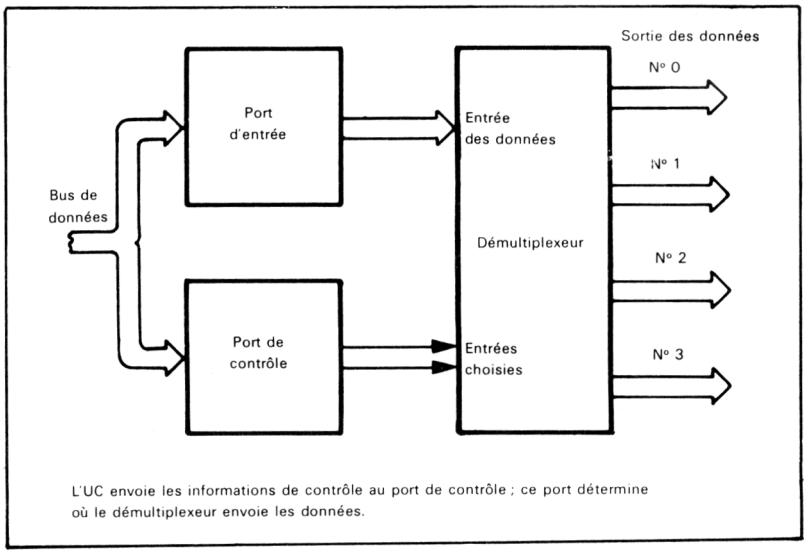


Fig. 11.2. — Un démultiplexeur en sortie contrôlé par un port.

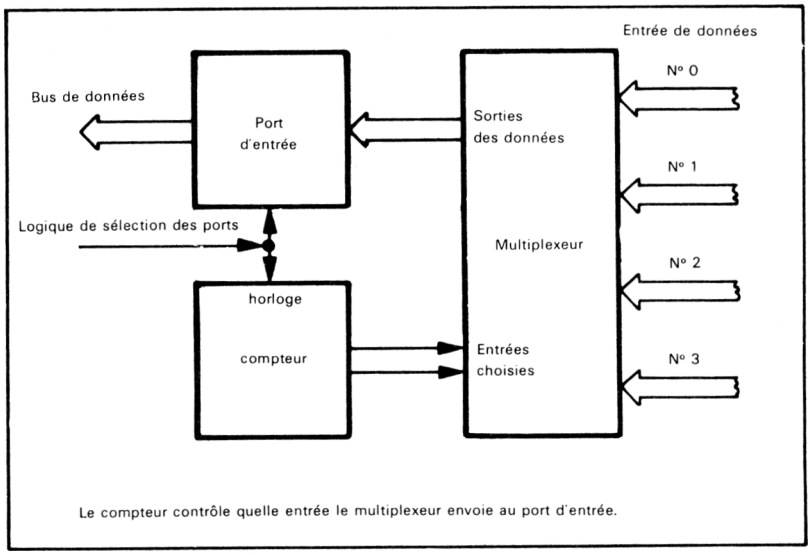


Fig. 11.3. — Un multiplexeur d'entrée contrôlé par un compteur.

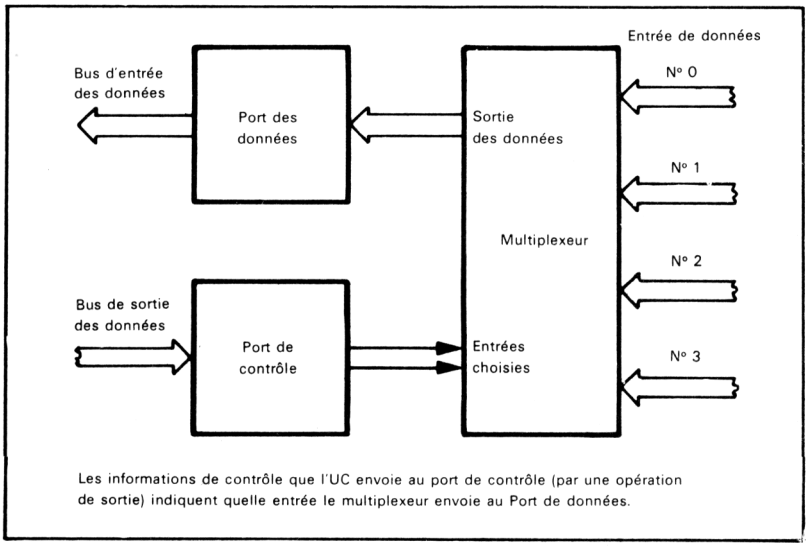


Fig. 11.4. — Un multiplexeur d'entrée contrôlé par un port.

On peut remarquer les différences entre l'entrée et la sortie pour les organes lents :

1) Les données en entrée n'ont pas besoin d'être mémorisées puisque l'organe d'entrée maintient les données pendant un temps très grand comparé aux vitesses de traitement du calculateur. Les données en sortie doivent être mémorisées puisque l'organe de sortie ne prendra pas en compte une donnée présente pendant seulement quelques cycles d'horloge d'UC.
2) Les transitions en entrée provoquent des problèmes en raison de leur durée ; les brèves transitions en sortie ne provoquent aucun problème car les organes de sortie (ou les observateurs) réagissent lentement.
3) Les contraintes majeures de l'entrée sont le temps de réaction et la faculté de réponse, les contraintes majeures de la sortie sont le temps de réponse et la faculté d'observation.

INTERFAÇAGE DES ORGANES A VITESSE MOYENNE

Les organes à vitesse moyenne doivent être synchronisés d'une façon ou d'une autre avec l'horloge du processeur. L'UC ne peut pas traiter ces organes comme s'ils contenaient indéfiniment les données ou pouvaient les recevoir à n'importe quel moment. Au contraire, l'UC doit être capable de déterminer quand un organe dispose de nouvelles données en entrée ou quand il est prêt à recevoir les données en sortie. Il doit aussi pouvoir dire à un organe que la nouvelle donnée en sortie est disponible ou que la précédente donnée en entrée a été acceptée. Notons que le périphérique peut être un autre processeur ou en contenir.

SYSTÈME CONVERSATIONNEL HANDSHAKE

La procédure standard, non synchrone avec l'horloge, est appelée « handshake » : littéralement poignée de mains. Dans ce cas, l'émetteur indique la disponibilité de la donnée au récepteur et transfère cette donnée.

Le récepteur acquiert la donnée et en accuse réception. Le récepteur peut contrôler le processus en demandant une donnée ou en indiquant qu'il est prêt à recevoir une donnée, l'émetteur envoie sa donnée et indique au récepteur qu'elle est présente. Dans les deux cas, l'émetteur sait que le transfert a eu lieu et le récepteur sait quand une nouvelle donnée lui est présentée.

Les figures 11.5 et 11.6 montrent les opérations d'entrées et de sorties utilisant la méthode du « handshake ». La procédure utilisée par l'unité centrale lors d'une opération de sortie de donnée est appelée scrutation (« polling »). Cette procédure peut prendre beaucoup de temps si le système est équipé de nombreux périphériques. Il y a plusieurs moyens pour traiter les signaux de contrôle, parmi ceux-ci, nous avons :

— Des lignes spécialement dédiées aux entrées ou aux sorties. Le processeur peut les traiter comme des ports supplémentaires d'entrées/sorties ou au moyen de lignes spéciales ou à l'aide d'interruptions. Le processeur Z80 ne possède pas de ligne d'entrée/sortie sérielle mais le périphérique d'entrée/sortie parallèle du Z80 (PIO) en possède.
— Une séquence spéciale sur les lignes d'E/S. Cela peut être un simple bit de départ (START bit) et des bits de fin (STOP bit) ou un caractère entier ou un groupe de caractères. Ces signaux doivent être aisément discernables du bruit de fond ou de l'état de repos de la ligne.

VALIDATIONS « STROBE »

On appelle souvent une ligne séparée des entrées/sorties, qui indique la présence d'une donnée ou annonce un transfert, une validation (« strobe »). Une validation peut être par exemple, un signal d'horloge sur un circuit mémoire (« latch ») ou un signal de lecture d'un circuit tampon (« buffer »).
Beaucoup de périphériques transfèrent des données à intervalles réguliers, c'est-à-dire de manière synchrone. Ici le seul problème est de lancer le processus pour acquérir la première entrée ou positionner la première sortie. Dans quelques cas, le périphérique a une sortie d'horloge à partir de laquelle le processeur peut obtenir des informations de mesure du temps (synchronisation).

RÉDUCTION DES ERREURS DE TRANSMISSION

Un problème existe avec les périphériques de vitesse moyenne, il concerne les erreurs dans la transmission. Plusieurs méthodes peuvent réduire la fréquence de telles erreurs :

— Echantillonner la donnée d'entrée au centre de la cellule de transmission pour éviter les transitions parasites, c'est-à-dire s'éloigner des fronts de transitions.
— Echantillonner la donnée plusieurs fois et utiliser une logique majoritaire telle que 3 parmi 5.[1]
— Générer et vérifier la parité : un bit supplémentaire est utilisé de manière à ce que le nombre de 1 dans la donnée soit pair ou impair.
— Utiliser d'autres méthodes de détection et de correction d'erreurs telles que les sommes-test (« checksum ») (LRC Longitudinal Redundancy Checksum), les contrôles cycliques (CRC Cyclic Redundancy Checksum).[2]

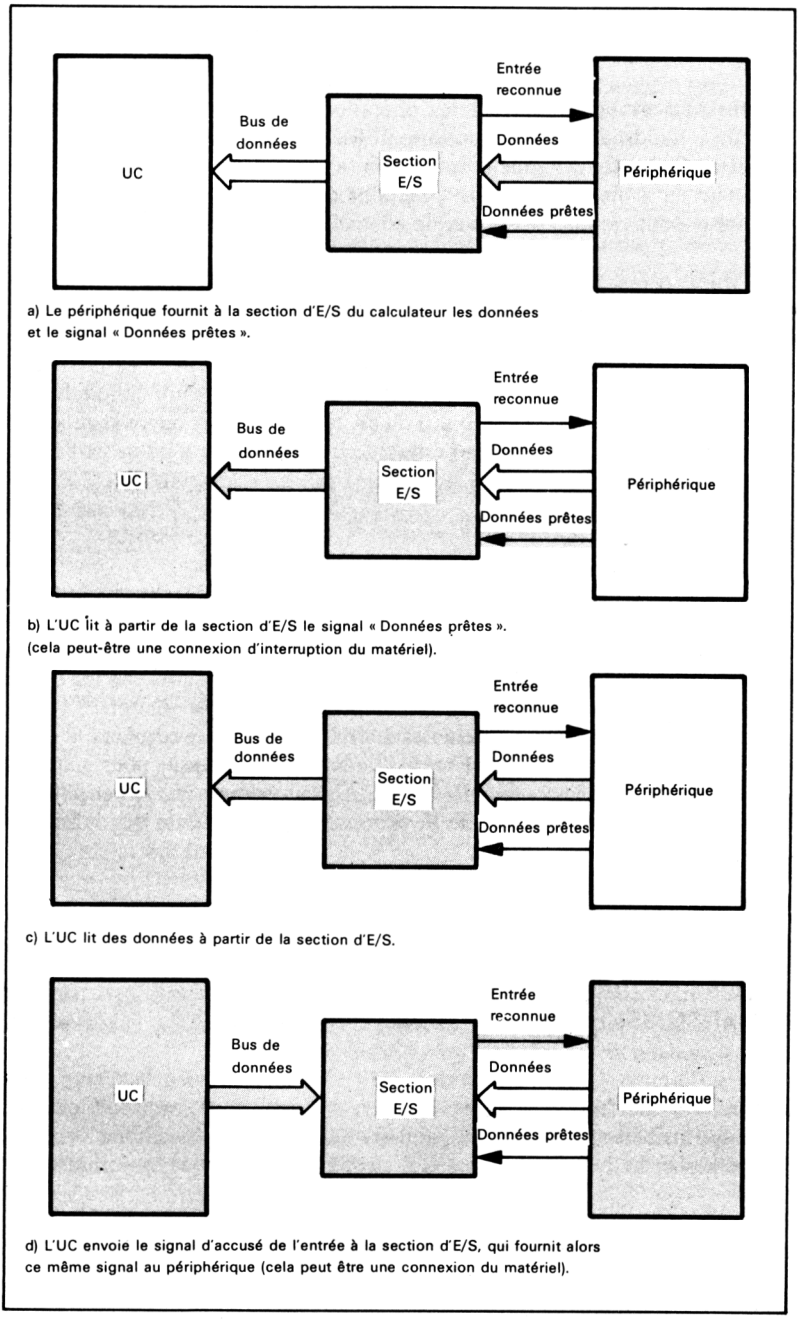


Fig. 11.5. — Une entrée conversationnelle (« input handshake »).

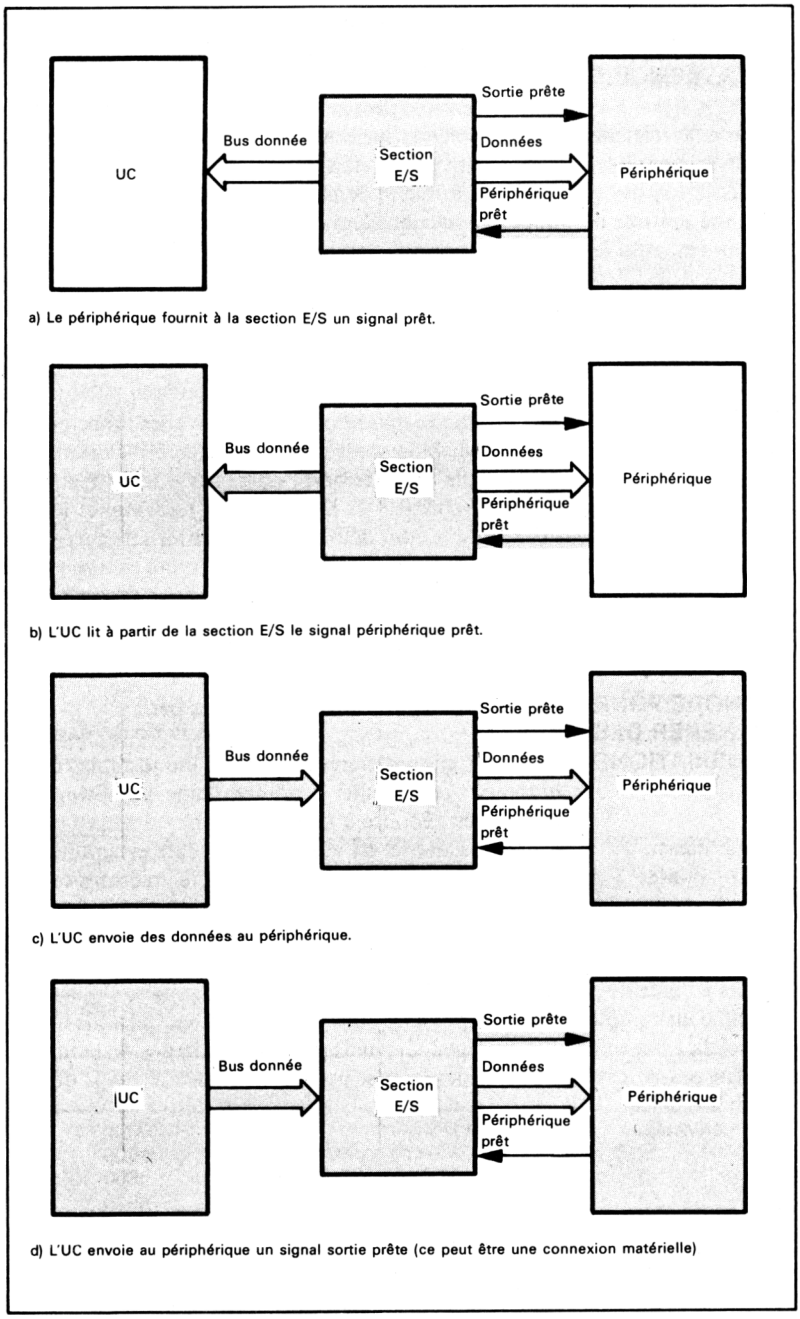


Fig. 11.6. — Une sortie conventionnelle (« output handshake »).

INTERFAÇAGE DES PÉRIPHÉRIQUES A GRANDE VITESSE

Les périphériques à grande vitesse qui transfèrent plus de 10 000 bits par seconde, exigent des techniques spéciales. La technique courante consiste en l'utilisation d'un contrôleur qui assure un transfert direct de la mémoire au périphérique et vice-versa. Un tel processus est appelé accès direct à la mémoire (« DMA »). Le contrôleur de DMA déconnectera l'unité centrale du système (état haute impédance), fournira les adresses et les signaux de contrôle de la mémoire et transférera les données.

ACCÈS DIRECT À LA MÉMOIRE

Alors qu'un tel contrôleur serait relativement complexe et nécessiterait 50 à 100 circuits intégrés classiques, des circuits intégrés haute densité (« LSI ») sont désormais disponibles [3]. L'unité centrale doit initaliser ce contrôleur en chargeant les adresses et les compteurs de données, ainsi le contrôleur saura où commencer et combien il doit transférer de données.

TEMPORISATIONS

UTILISATION DES TEMPORISATIONS

Un problème que nous allons voir apparaître au cours de notre discussion sur les entrées/sorties est la création de temporisations de différentes durées. De telles temporisations sont nécessaires pour supprimer les rebonds des interrupteurs mécaniques (c'est-à-dire éliminer les irrégularités de la transition), pour délivrer des impulsions de durées et de fréquences spécifiques pour des afficheurs, et pour synchroniser l'unité centrale et les périphériques qui reçoivent ou émettent des données régulièrement (exemple : un téléimprimeur qui émet ou reçoit un bit toutes les 9,1 ms).

MÉTHODE POUR GÉNÉRER DES TEMPORISATIONS

On peut générer des temporisations par :

1) L'utilisation de circuits intégrés monostables ou multivibrateurs. Les circuits produisent une impulsion de durée calibrée en réponse à une excitation.

2) Une combinaison de logiciel et de matériel à l'aide de compteurs programmables tels que le Z80 Counter Timer Circuit ou CTC, utilisable avec un microprocesseur Z80 comme décrit dans : « Une introduction aux micro-ordinateurs volume 2 ». Le CTC peut générer des intervalles de temps d'une longueur variable tenant compte de multiples conditions d'arrêt et de départ.

3) Le logiciel à l'aide de programmes de temporisation. Ces programmes utilisent le processeur comme un compteur. Ceci est possible puisque le calculateur possède une horloge stable mais cela sous-emploie le système. Cependant les programmes de temporisations ne nécessitent pas de câblage supplémentaire et utilisent parfois un temps qui de toute façon aurait été perdu.

CHOIX DE LA MÉTHODE DE TEMPORISATION

Le choix parmi ces trois méthodes dépend de votre application. La méthode logicielle n'est pas coûteuse mais surcharge le processeur. Les compteurs programmables coûtent relativement cher mais sont aisés à connecter et peuvent assurer des tâches assez complexes de temporisation.

Programmes de temporisations

Un programme de temporisation simple se déroule comme suit :

Etape 1 : chargement dans un registre d'une valeur donnée.
Etape 2 : décrémentation du registre.
Etape 3 : si le résultat de l'étape 2 n'est pas zéro, répéter l'étape 2.
Ce programme n'est utilisé que pour la temporisation. La durée dépend du temps d'exécution de chaque instruction et de la valeur mise dans le registre. La durée maximale est fixée par la taille du registre, toutefois le programme peut être écrit à l'intérieur d'un programme semblable qui utilise d'autres registres, etc.

PROGRAMME DE TEMPORISATION TRANSPARENT

L'exemple suivant utilise les registres C et A pour assurer une temporisation de 255 ms. Le choix des registres est arbitraire. On peut trouver l'utilisation d'un registre pair (BC) plus avantageuse. Une instruction PUSH BC au début du programme de temporisation et une instruction POP BC à la fin feront que le programme ne modifiera aucun registre. Un tel programme est dit transparent au programme appelant. Remarque : les instructions PUSH et POP devront être comptabilisées dans le temps d'exécution du programme.

EXEMPLE

Programme de temporisation utilisant l'accumulateur.

But : le programme assure une temporisation de n × 1 ms n'étant le contenu de l'accumulateur.

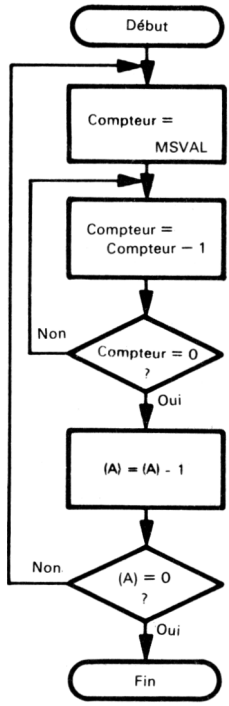
Organigramme : (voir page 366)

Programme source :

```
TEMP0 : LD    C,MSVAL   ; CHARGER COMPTEUR POUR TEMPORISATION DE 1 MS
TEM1:   DEC   C         ; COMPTEUR = COMPTEUR – 1
        JR    NZ,TEM1   ; CONTINUE TANT QUE COMPTEUR ≠ 0
        DEC   A         ; DECREMENTE LE NOMBRE RESTANT DE MS
        JR    NZ,TEMP0  ; CONTINUE TANT QUE LE NOMBRE DE MS ≠ 0
        RET
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0030	0E	TEMPO	LD	C,MSVAL
0031	MSCNT			
0032	0D	TEM1 :	DEC	C
0033	20		JR	NZ,TEM1
0034	FD			
0035	3D		DEC	A
0036	20		JR	NZ,TEMP0
0037	F8			
0038	C9		RET	



La valeur MSVAL dépend de la vitesse de l'UC et du cycle mémoire.

Evaluation du temps :

Instruction		Nombre de fois exécutée
LD	C,MSVAL	(A)
DEC	C	(A) x MSVAL
JR	NZ,TEM1	(A) x MSVAL
DEC	A	(A)
JR	NZ,TEMP0	(A)
RET		

Le temps total doit être (A) × 1 ms. Si la mémoire opère à pleine vitesse, les instructions nécessitent le nombre de cycles d'horloge suivant :

LD	C,MSVAL	7
DEC	C ou A	4
JR	NZ	7 ou 12
RET		10

Pour une instruction JR on a deux possibilités :
— la condition est rencontrée 12 cycles
— la condition n'est pas rencontrée 7 cycles.
En ignorant les instructions CALL et RET (qui n'arrivent qu'une fois), le programme prend :

$$(A) \times (7 + 16 \times MSVAL - 5 + 16) - 5 \text{ cycles}$$

Le retrait de 5 cycles est causé par le fait que l'instruction JR prend moins de temps lors du test final.
Ainsi pour faire une temporisation de 1 ms

$$13 + 16 \times MSVAL = Nc$$

CONSTANTE DE TEMPORISATION POUR LE Z80

où Nc est le nombre de cycle d'horloge par milliseconde, à une fréquence de 4 MHz de l'horloge, Nc = 4000 donc :

$$16 \times MSVAL = 3987$$

d'où MSVAL = 249 (en hexa : F9) à une fréquence de 4 MHz pour le Z80.

CIRCUITS D'ENTRÉES/SORTIES SIMPLES

Le circuit d'entrée/sortie parallèle du Z80 (PIO)

L'élément clé dans la plupart des sections entrées/sorties du Z80 est le circuit d'entrée/sortie parallèle Z80 ou PIO. Ce circuit comprend des mémoires, des amplificateurs, des bascules et d'autres circuits logiques nécessaires pour des techniques d'interfaçage simple ou conversationnel (« handshake »). Le PIO comprend de nombreuses connexions logiques, certaines d'entre elles peuvent être sélectionnées suivant le contenu de registres programmables. Ainsi le concepteur a la possibilité de configurer ce circuit avec beaucoup de souplesse. La phase d'initialisation du programme met les valeurs adéquates dans des registres pour sélectionner les connexions logiques. Une section d'entrée/sortie basée sur des PIO peut gérer de nombreuses applications ; des modifications et corrections peuvent ainsi être réalisées par logiciel plutôt que par matériel.
La figure 11.7 est le schéma fonctionnel d'un PIO. Le circuit comprend deux ports 8 bits sensiblement identiques ; A est utilisé habituellement en port d'entrée et B est généralement employé en sortie. Chaque port contient (cf. figure 11.8) :

LIGNES DE CONTRÔLE ET REGISTRE D'UN PIO

— Un registre de sortie de donnée - 8 bits.
— Un registre d'entrée de donnée - 8 bits.
— Un registre de contrôle de mode 2 bits, qui indique si le port est en entrée, en sortie ou bidirectionnel ou mode contrôle.
— Un registre de contrôle d'entrée/sortie 8 bits qui déterminent la direction des bits de donnée correspondants (entrée 1, sortie 0) dans le mode contrôle.
— Deux lignes de contrôle (STB et RDY) qui sont configurées par le registre de contrôle de mode. Ces lignes peuvent être utilisées pour des signaux de contrôle en conversationnel (« handshake ») (fig. 11.5 et 11.6).
— Un registre de contrôle de masque 2 bits, utilisés seulement dans le mode contrôle qui détermine la polarité active des entrées et si elles génèrent une interruption lorsque une seule entrée est active ou lorsqu'elles sont toutes actives.

— Un registre de masque de 8 bits (utilisé seulement en mode contrôle) qui détermine quelles lignes du port sont prises en compte pour générer le signal d'interruption.
— Un registre d'adresse de vecteur (8 bits) utilisé avec le système d'interruption.

Dans un premier temps, nous nous intéresserons seulement aux registre de contrôle de mode, registre de contrôle des entrées/sorties et des lignes de contrôles ; nous discuterons dans le chapitre 12 des dispositifs relatifs aux interruptions du PIO. Les significations des bits dans les divers registres de contrôles et de masques sont relatives au matériel et sont

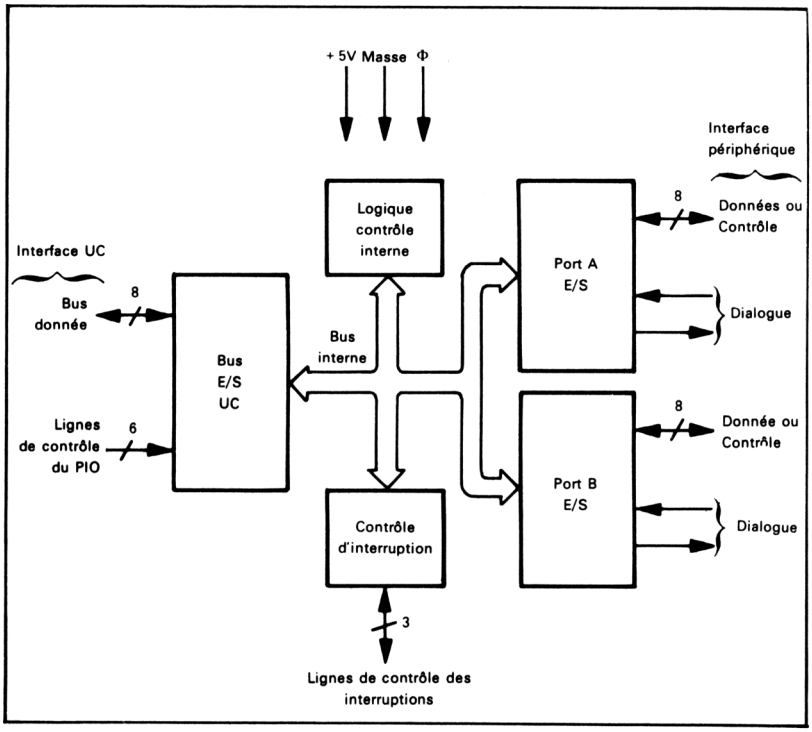


Fig. 11.7. — Diagramme fonctionnel d'un PIO (doc. Zilog).

complètement arbitraires pour autant que le programmeur en assembleur soit concerné.

ADRESSAGE DES PIO Chaque PIO occupe quatre adresses de port d'entrée et quatre adresses de port de sortie. Les lignes B/A sel (port B ou A sélectionné) et C/D sel (contrôle ou donnée sélectionné) choisissent l'un des quatre ports comme décrit dans le tableau 11.1. Le plus fréquemment, les concepteurs relient le bit d'adresse A_0 à la ligne B/A sel et le bit d'adresse A_1 à la ligne C/D sel. Ainsi les ports du PIO occupent quatre adresses consécutives comme cela est décrit dans le tableau 11.1.
En fait, il y a plus de registres internes de contrôle que d'adresses de port. En effet, tous les registres de contrôle pour chaque port occupent une adresse suivant l'état de l'entrée C/D sel. Ainsi plusieurs bits de données envoyés dans un registre de contrôle sont utilisés

pour des fonctions d'adressage. Examinons les situations suivantes (tableau 11.2) : $D_0 = 0$ signifie que les autres bits restants sont chargés dans le registre du vecteur d'interruption.

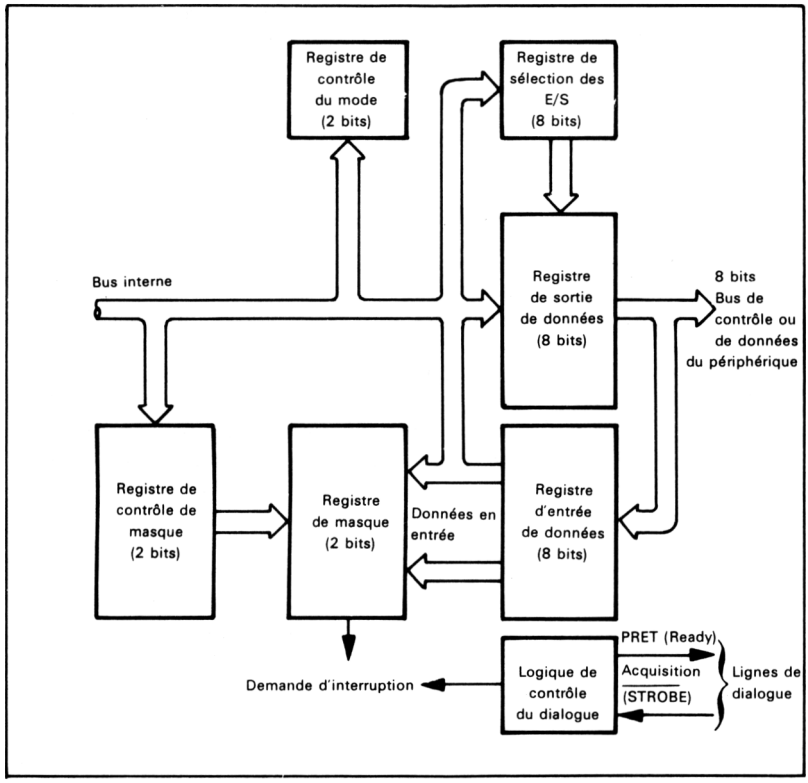


Fig. 11.8. — Diagramme fonctionnel d'un port de PIO (doc. Zilog).

Tableau 11.1. — Adresses du PIO

Sélection des données ou contrôle	Sélection du port A ou B	Registre adressé	Adresse des ports (début en PIOADD)
0	0	Registre donnée A	PIOADD
0	1	Registre donnée B	PIOADD + 1
1	0	Registre contrôle A	PIOADD + 2
1	1	Registre contrôle B	PIOADD + 3
L'adresse des ports suppose que C/D sel est relié à A_1 et B/A sel à A_0.			

Tableau 11.2. — Adressage des registres de contrôle du PIO

Registre	Adressage
Contrôle de mode Contrôle entrées/sorties	$D_3 = D_2 = D_1 = D_0 = 1$ Octet suivant l'octet de sélection du mode 3
Registre contrôle du masque Registre du masque d'interruption	$D_3 = 0$, $D_2 = D_1 = D_0 = 1$ Valeur du masque si D_4 octet précédent = 1
Autorisation des interruptions Vecteur d'interruption	$D_3 = D_2 = 0$, $D_1 = D_0 = 1$ $D_0 = 1$

— $D_3 = 0$, $D_2 = D_1 = D_0 = 1$ signifie que les autres bits sont chargés dans le registre de contrôle du masque. Si $D_4 = 1$, l'octet de contrôle suivant est chargé dans le registre du masque d'interruption. Les interruptions peuvent être autorisées ou inhibées avec $P_3 = D_2 = 0$, $D_1 = D_0 = 1$.
— $D_3 = D_2 = D_1 = D_0 = 1$ signifie que les autres bits sont chargés dans le registre de contrôle de mode. Si $D_7 = D_6 = 1$ (mode contrôle), l'octet suivant est chargé dans le registre de contrôle d'entrées/sorties.
Ce partage des adresses externes implique que :

1. Le programmeur doit faire attention à l'ordre des opérations. La signification d'une instruction de sortie dépend de la séquence précédent cette instruction.
2. Le programme doit documenter la configuration du PIO dans les moindres détails. Ce circuit est assez complexe et le lecteur n'est pas toujours capable d'appréhender les séquences qui configurent ce circuit.

Notons que l'on configure habituellement les registres de contrôle du PIO une seule fois lors de la phase d'initialisation du programme. Le reste du programme utilise seulement les registres de données du PIO.

Contrôle du mode de fonctionnement du PIO*

MODES DU PIO Le fonctionnement d'un PIO est établi en écrivant un octet de contrôle dans le PIO comme il est décrit dans le tableau 11.1

Le tableau 11.3 décrit les significations des différents modes et les octets de contrôle qui les définissent. On remarque que D_5 et D_4 sont initialisés. Lors d'une mise sous tension, le PIO est dans le mode 1 (entrée).
On peut résumer les différents modes comme suit :
(NDT : remarquons qu'il existe un moyen mnémotechnique pour se souvenir du n° d'un mode : 0 = Output, 1 = Input, 2 = BI directionnel.)

* NDT : il ne faut pas confondre le contrôle de mode, qui permet de sélectionner l'un des 4 modes de fonctionnement du PIO, et le mode contrôle (mode 3) qui constitue l'un de ces modes (décrits ci-après).

MODE SORTIE DU PIO

1. Mode 0 - Sortie

En envoyant des données dans le registre de sortie, les données sont mémorisées et apparaissent sur le bus de donnée du port. Le signal prêt (RDY) devient actif (niveau haut) indiquant ainsi que les données sont prêtes. Ce signal reste haut jusqu'à ce que le périphérique génère un front « montant » sur la ligne acquisition (STB) indiquant que les données sont acceptées ou que le périphérique est prêt. Le front montant de STB provoque une interruption si celles-ci sont autorisées.

MODE ENTRÉE DU PIO

2. Mode 1 - Entrée

Le périphérique mémorise les données dans le registre d'entrée du port, en utilisant le signal validation (STB). Le front montant de (STB) provoque une interruption (si elle est autorisée) et désactive le signal (RDY). Quand l'UC lit la donnée, RDY devient actif (haut) et indique ainsi que la donnée est acceptée ou que le registre d'entrée est vide. On remarque que le PIO peut acquérir des données dans son registre sans regarder l'état de RDY. Le programmeur doit ainsi gérer le problème de recouvrement, c'est-à-dire une nouvelle donnée est mise dans le registre avant que l'ancienne ne soit lue.

MODE BIDIRECTIONNEL DU PIO

3. Mode 2 - Bidirectionnel

Ce mode utilise les quatre lignes de dialogue (« handshake »), ce qui signifie que seul le port A peut être utilisé. Les signaux du port A, RDY et STB sont utilisés pour le contrôle des sorties et les signaux du port B, RDY et STB, sont utilisés pour le contrôle des entrées. La seule différence entre ce mode et une combinaison des modes 0 et 1, est que les données du registre de sortie sont transférées vers le port de sortie seulement quand STB est actif. Cela permet au bus du port A d'être utilisé dans les deux directions sous le contrôle du signal ASTB (demande de sortie de donnée) et BSTB (donnée d'entrée disponible). Notons que dans ce mode, les signaux de contrôle de la partie B sont gérés par le registre d'entrée A.

MODE CONTRÔLE DU PIO

4. Mode 3 - Contrôle

Ce mode n'utilise pas les signaux RDY ni STB. Il est conçu pour des applications de contrôle et de test dans lesquelles chaque bit a une signification particulière. Quand le mode 3 est choisi, l'octet de contrôle suivant est envoyé au PIO et définit les directions de chaque bit du port (figure 11.9).

DIRECTIONS DU PIO DANS LE MODE CONTRÔLE

Un « 1 » dans une position de bit provoque la mise en entrée de la ligne correspondante, alors qu'un « 0 » la met en sortie.

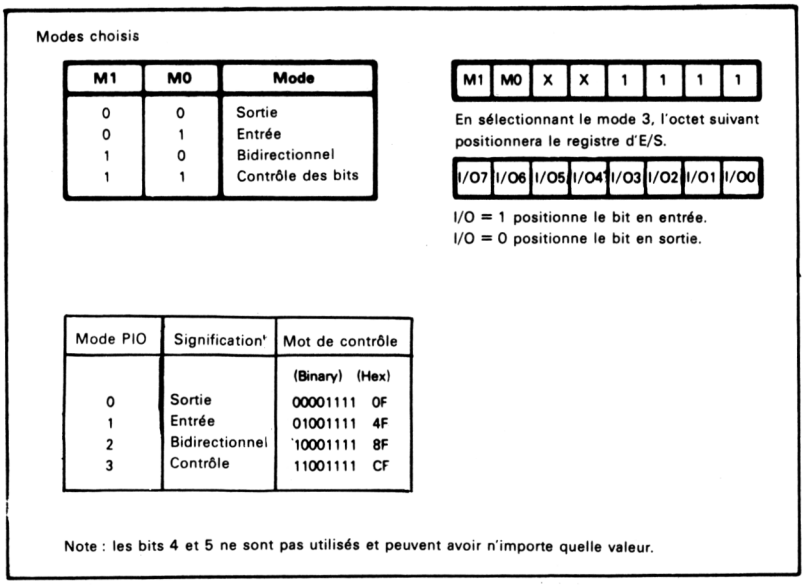


Fig. 11.9. — Contrôle du mode du PIO du Z80.

CARACTÉRISTIQUES DU PIO Remarquons les possibilités suivantes des modes de fonctionnement du PIO :

1) Dans les modes 0, 1 et 2, le périphérique indique données prêtes, périphérique prêt ou données acceptées avec un front montant du signal STB. Ce front provoque aussi une interruption si celle-ci est autorisée.
2) Dans les modes 0, 1 et 2, le PIO indique données prêtes, registre d'entrée vide, ou données acceptées en envoyant un RDY actif (haut). Ce signal reste haut jusqu'au prochain front montant de STB.
3) Seul le port A peut être utilisé en mode bidirectionnel. Si le port A est dans le mode 2 (bidirectionnel), le port B peut être seulement utilisé en mode contrôle, de plus aucune ligne de dialogue n'est disponible (« handshake »).
4) Le mode contrôle (3) est le seul mode dans lequel le registre de contrôle des entrées/sorties est utilisé. Autrement, le port entier est utilisé soit pour des entrées soit pour des sorties.
5) Le processeur ne peut pas déterminer si une impulsion a activé le signal STB si les interruptions ne sont pas utilisées. Le PIO est conçu pour être utilisé dans un système interruptible plutôt que dans un système de scrutation (cf. chapitre 12). STB doit être reliée au zéro si elle ne sert pas.
6) Le processeur ne peut pas contrôler directement les lignes RDY. La ligne RDY sur un port devient active (haute) quand une donnée est transférée de ou vers un port et elle devient basse sur le front montant de STB.
7) Le contenu du registre de sortie de données peut être lu si le port est en sortie ou si le port est bidirectionnel. Si le port est dans le mode contrôle, les données correspondantes à des sorties peuvent être lues. Les contenus des registres de contrôle ne peuvent pas être lus.

8) Si le signal de sortie RDY est connecté au signal d'entrée STB sur un port, en mode sortie, RDY deviendra actif (haut) pendant une période d'horloge après chaque opération de sortie. Cette brêve impulsion peut être utilisée pour le multiplexage d'afficheurs (fig. 11.1).

CONFIGURATION D'UN PIO

Le programme doit sélectionner les connexions logiques dans le PIO avant de transférer des données vers lui ou de lire des données. Cette sélection (configuration) est généralement une partie du programme d'initialisation. Notons qu'à la mise sous tension, le PIO est dans le mode entrée avec toutes les interruptions inhibées et inactives ainsi que les signaux de contrôle. Cependant, le PIO ne possède pas de signal de ré-initialisation (RESET) et donc n'est pas nécessairement ré-initialisé lorsque l'UC l'est.

ÉTAPES POUR LA CONFIGURATION D'UN PIO

Les étapes de configuration sont :

1) Etablir le mode d'opérations en envoyant les octets de contrôle appropriés dans le registre de contrôle de mode. Il peut être nécessaire d'envoyer l'information contrôlant les interruptions aussi bien que le mode d'entrée sortie utilisé.
2) Lorsque l'on est en mode 3, établir les directions des broches d'entrées/sorties en envoyant un octet de contrôle dans le registre d'entrée/sortie. Cet octet doit suivre l'octet de contrôle qui sélectionne le mode 3.

Examinons quelques exemples de configuration d'un PIO, sans interruption :

1) Port en sortie

```
LD      A,00001111B     ; MISE DU PORT B EN SORTIE
OUT     (PIOCRB),A
```

2) Port en entrée

```
LD      A,01001111B     ; MISE DU PORT A EN ENTREE
OUT     (PIOCRA),A
```

3) Port bidirectionnel

```
LD      A,10001111B     ; MISE DU PORT A EN BIDIRECTIONNEL
OUT     (PIOCRA),A
```

Rappel : seul le port A peut être bidirectionnel et le port B doit alors être un port de contrôle.

4) Contrôle du port, tout entrée

```
LD      A,11001111B     ; CONTROLE DU PORT A
OUT     (PIOCRA),A
LD      A,0FFH          ; TOUS LES BITS EN ENTREE
OUT     (PIOCRA),A
```

5) Contrôle du port, tout en sortie

```
LD      A,11001111B     ; MISE EN CONTROLE DU PORT B
OUT     (PIOCRB),A
SUB     A               ; TOUS LES BITS EN SORTIE
OUT     (PIOCRB),A
```

6) Contrôle du port, lignes 1, 5, 6 en entrées ; lignes 0, 2, 3, 4, 7 en sortie.

```
LD      A,11001111B     ; MISE EN CONTROLE DU PORT A
OUT     (PIOCRA),A
LD      A,01100010B     ; LIGNE 1, 5, 6 ENTREE :1, 0, 2, 3, 4, 7 SORTIE : 0
OUT     (PIOCRA),A
```

Instructions d'entrées/sorties du Z80

INSTRUCTION D'ENTRÉE/SORTIE DU Z80

Le microprocesseur Z80 possède un large ensemble d'instructions d'entrées/sorties. Toutes les instructions d'entrée/sortie utilisent un adressage sur 8 bits*, cela permet jusqu'à 256 ports d'entrée et 256 ports de sortie. Cependant il faut se souvenir qu'un PIO occupe 4 adresses de port de sortie et 4 adresses de port d'entrée. Les instructions d'entrées/sorties du Z80 peuvent être groupées comme suit :

1) Instructions utilisant un adressage absolu. IN A,(port) et OUT (port),A transfèrent 8 bits de données entre l'accumulateur et le port adressé par le second octet de l'instruction.
2) Instructions d'un seul octet qui utilise l'adressage indirect IN reg,(C) et OUT (C),reg transfèrent 8 bits de données entre le registre spécifié et le port adressé par le contenu du registre C.
3) Instructions d'entrées-sorties de bloc. Les instructions INI et OUTI transfèrent 8 bits de données entre une mémoire adressée par la paire de registre, HL et le port adresé par le registre C. Ces deux instructions incrémentent ensuite le double registre HL et décrémentent le compteur d'octet dans le registre B. L'indicateur Z est mis à 1 si B est égal à zéro, il est mis à 0 sinon. Les instructions OUTD et IND sont identiques sauf qu'elles décrémentent le double registre HL au lieu de l'incrémenter.
4) Instructions répétitives d'E/S de bloc. Les instructions INIR et OTIR répètent respectivement les effets de INI et OUTI tant que B est non nul. Les instructions INDR et OTDR ont des effets similaires respectivement pour les instructions IND et OUTD.

On notera les possiblités suivantes de chaque groupe d'instructions :

INSTRUCTIONS E/S AVEC ADRESSAGE ABSOLU

1. Instructions avec adressage absolu
— Les données sont toujours transférées à partir de ou vers l'accumulateur.
— les indicateurs (flags) ne sont pas affectés.
— L'adresse du port fait partie du programme et ne peut être changé si la mémoire est à lecture seule.

INSTRUCTIONS E/S AVEC ADRESSAGE INDIRECT

2. Instruction d'un seul octet avec adressage indirect d'un registre
— Les données peuvent être transférées à partir de ou vers n'importe lequel des registres 8 bits 1, B, C, D, E, H ou L.

---

* NDT : on peut cependant utiliser des adresses 16 bits, en se servant de A ou B et C qui sont présents sur le bus d'adresses pendant les instructions d'E/S. On peut même accéder à des ports d'adresses (16 bits) successives en une seule instruction (INI, INIR, etc.) en utilisant $A_8 - A_{15}$ pour adresser la partie BASSE de l'espace d'E/S, et $A_0 - A_7$ pour adresser la partie HAUTE (cf. chapitre 3, les instructions d'E/S).

Cependant, il faut se souvenir que le registre C contient l'adresse du port.
— L'instruction IN reg,(C) positionne les indicateurs de signe (S), de zéro (Z) et de parité (P/O) suivant la valeur de la donnée entrée.
— L'indicateur de retenue (c) n'est pas modifié, tandis que ceux de la demi retenue (H) et négatifs (N) sont mis à zéro.

PROGRAMME D'E/S

— OUT (C), reg n'affecte aucun indicateur.
— L'adresse du port est toujours contenue dans le registre C. Cette adresse n'est pas une partie du programme et peut être un paramètre pour un sous-programme d'entrée/sortie.

Un programme d'entrée/sortie peut être utilisé dans de multiples applications différentes ou avec plusieurs circuits d'entrée/sortie dans une même application.

INSTRUCTIONS E/S DE BLOCS

3. Instructions d'E/S de blocs
— Les données sont toujours transférées vers ou à partir d'une mémoire adressée par le registre double HL.
— L'indicateur Z (zéro) est mis à 1 si le registre B est égal à zéro et remis à zéro sinon. Les indicateurs S (signe), P/O (parité) et H (demi retenue) sont affectés mais leurs valeurs finales sont aléatoires.
— L'adresse du port est toujours dans le registre C. Ici encore, cette adresse peut être un paramètre pour un coupleur d'entrée/sortie.
— Le registre B est un compteur de 8 bits. Ainsi, les instructions E/S de bloc peuvent transférer un maximum de 256 octets. Elles diffèrent des instructions de transfert de bloc et de comparaison de blocs lesquelles utilisent un registre double BC comme compteur de 16 bits et peuvent traiter 64 K octets.

EXEMPLES D'INSTRUCTIONS E/S

Quelques exemples des instructions E/S :

1) Chargement de l'accumulateur à partir du port d'entrée 2
a) Utilisant un adressage absolu

```
IN      A,(2)
```

b) Utilisant un adressage indirect.

```
LD      C,2
IN      A,(C)
```

2) Stockage du contenu de l'accumulateur dans le port de sortie (5).
a) Utilisant un adressage absolu

```
OUT     (5),A
```

b) Utilisant l'adressage indirect

```
LD      C,5
OUT     (C),A
```

3) Chargement de la mémoire 0040 à partir du port d'entrée (2)
a) Utilisant l'adressage absolu

```
IN      A,(2)     ; CHARGEMENT DE LA DONNEE (PORT)
LD      (40H),A   ; CHARGEMENT DE LA DONNEE DANS LA MEMOIRE
```

b) Utilisant l'adressage indirect

```
LD      C,2       ; N° DU PORT
IN      A,(C)     ; ACQUISITION DE LA DONNEE
LD      (40H),A   ; CHARGEMENT DE LA DONNEE ET STOCKAGE
```

c) Utilisant l'E/S par bloc

```
        LD      C,2         ; N° DU PORT
        LD      HL,40H      ; INITIALISATION DE LA MEMOIRE DE DESTINATION
        INI                 ; ACQUISITION DE LA DONNEE
```

4) Stockage du contenu de la mémoire 0040 dans le port de sortie 5

a) Utilisation adressage absolu

```
        LD      A,(40H)     ; ACQUISITION DE LA DONNEE
        OUT     (5),A       ; ENVOI DE LA DONNEE
```

b) Utilisation adressage indirect

```
        LD      C,5         ; ACQUISITION DU N° DE PORT
        LD      A,(40H)     ; ACQUISITION DE LA DONNEE
        OUT     (C),A       ; ENVOI
```

c) Utilisation d'E/S par bloc

```
        LD      C,5         ; ACQUISITION DU N° DE PORT
        LD      HL,40H      ; ACQUISITION DE L'ADRESSE DE DEPART
        OUTI                ; ENVOI
```

5) Chargement des mémoires 0040 à 0047 à partir du port d'entrée 2

a) Utilisation adressage absolu

```
        LD      HL,40H      ; CHARGEMENT DE L'ADRESSE DE DEPART DE LA DONNEE
        LD      B,8         ; COMPTEUR = 8
INOCT : IN      A,(2)       ; LECTURE DE L'OCTET DE DONNEE
        LD      (HL),A      ; STOCKAGE DE L'OCTET DANS LA MEMOIRE
        INC     HL
        DJNZ    INOCT
```

b) Utilisation des E/S par bloc

```
        LD      HL,40H      ; CHARGEMENT ADRESSE DE DEPART
        LD      B,8         ; COMPTEUR D'OCTET = 8
        LD      C,2         ; ACQUISITION DU N° DU PORT
INOCT : INI
        JR      NZ,INOCT
```

c) Utilisation des instructions d'E/S répétitives par bloc

```
        LD      HL,40H      ; ACQUISITION ADRESSE DE DEPART DE LA DONNEE
        LD      B,8         ; COMPTEUR D'OCTET = 8
        LD      C,2         ; ACQUISITION DU N° DE PORT
        INIR                ; TRANSFERT DES OCTETS D'ENTREE DANS LA MEMOIRE
```

6) Envoi le contenu des mémoires 0040 à 0047 vers le port de sortie 5

a) Utilisation adressage absolu

```
        LD      HL,40H      ; ACQUISITION DE L'ADRESSE DE DEPART DE LA DONNEE
        LD      B,8         ; INITIALISATION DU COMPTEUR D'OCTET
OTBYTE: LD      A,(HL)      ; LECTURE D'UN OCTET
        OUT     (5),A       : SORTIE DE L'OCTET
        INC     HL
        DJNZ    OTBYTE
```

b) Utilisation des E/S par bloc

```
        LD      HL,40H      ; ACQUISITION DE L'ADRESSE DE DEPART DE LA DONNEE
        LD      B,8         ; INITIALISATION DU COMPTEUR
        LD      C,5         ; ACQUISITION DU N° DE PORT
OTBYTE: OUTI                ; SORTIE D'UN OCTET A PARTIR DE LA MEMOIRE
        JR      NZ,OTBYTE
```

c) Utilisation des instructions d'E/S répétitives par bloc

```
        LD      HL,40H      ; ACQUISITION ADRESSE DE DEPART DE LA DONNEE
        LD      B,8         ; INITIALISATION ADRESSE DE DEPART DE LA DONNEE
        LD      C,5         ; ACQUISITION DU N° DE PORT
        OTIR                ; SORTIE DES OCTETS A PARTIR DE LA MEMOIRE
```

UTILISATION DES INSTRUCTIONS E/S PAR BLOC

Notons que les instructions d'E/S par bloc opèrent de façon continue. On ne peut insérer aucune temporisation entre deux transferts. Ainsi, ces instructions ne peuvent être utilisées que si le périphérique travaille à la même vitesse que le processeur ou que les temporisations sont gérées séparément par le matériel.
Pour gérer les temporisations par le matériel, on peut soit forcer le processeur dans l'état d'attente (signal « WAIT »), soit mémoriser (« buffer ») les données.
Remarquons également que les instructions d'E/S par bloc mettent le contenu du compteur d'octets (reg. B) sur la partie haute du bus d'adresse durant le transfert d'entrée/sortie. Dans les opérations de sortie, le registre B est décrémenté en premier, la valeur du compteur d'octet est ainsi disponible pour des circuits externes.
Une application typique pour les instructions d'E/S par bloc est la configuration des PIO. Plusieurs octets doivent souvent être envoyés au registre de contrôle pour déterminer le mode d'opération, sélectionner la direction des lignes et établir le système d'interruption. Les PIO opérant à la même vitesse que l'UC, il n'y a aucun problème de compatibilité. Nous discuterons de la configuration des PIO du Z80 et des interfaces sériels (SIO) à l'aide des instructions d'E/S par bloc, plus loin dans ce chapitre et dans le chapitre 12.
Dans les exemples d'E/S qui suivent, on utilisera principalement les instructions avec un adressage absolu. On peut facilement leur substituer les instructions avec un adressage indirect en se rappelant qu'il faut initialiser le registre C.
On indiquera parfois les applications pour ces instructions d'E/S par bloc.

EXEMPLES

But : interfacer un simple bouton poussoir (commutateur unipolaire) à un microprocesseur Z80. Le bouton poussoir est un interrupteur mécanique qui provoque la fermeture d'un contact (c'est-à-dire un zéro logique) lorsqu'il est enfoncé.

Schéma : la figure 11.10 montre le matériel nécessaire pour interfacer le bouton poussoir. Il utilise un des bits du PIO du Z80 qui agit comme amplificateur ; il est inutile de mémoriser, car le bouton poussoir reste fermé pendant plusieurs cycles d'horloge. En pressant le bouton, on amène à zéro le bit d'entrée du PIO. La résistance de polarisation assure que le bit d'entrée est à un si le bouton n'est pas pressé.

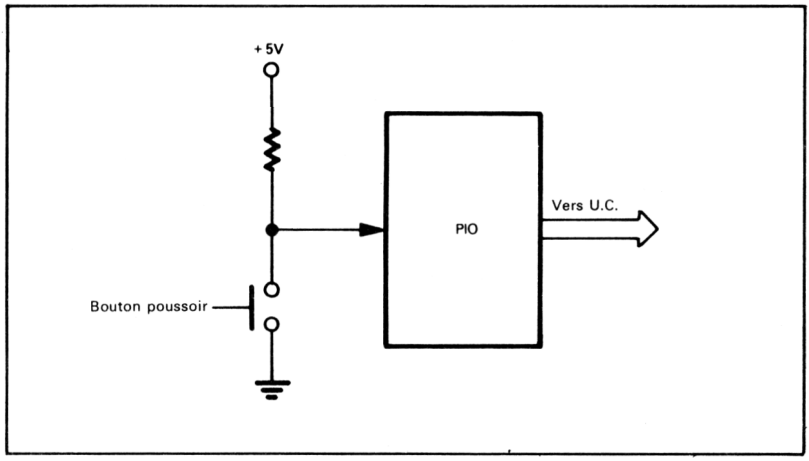


Fig. 11.10. — Un circuit à bouton poussoir

Exemple de programmation :

Ce circuit va nous permettre d'effectuer les deux tâches suivantes :

a) Positionner une mémoire suivant l'état du bouton.
b) Compter le nombre de fois que le bouton est pressé.

Tache 1 : Déterminer la fermeture du contrat.

But : mettre un un dans la mémoire 0040 si le bouton n'est pas enfoncé et un zéro dans le cas contraire.

Exemple :

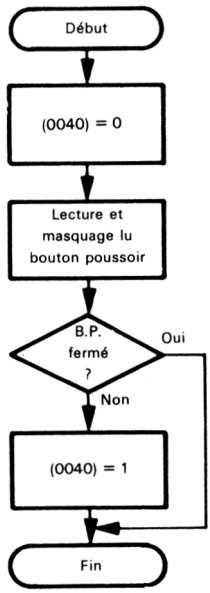
1) Contact ouvert (c'est-à-dire pas enfoncé)
Résultat : (0040) = 01
2) Contact fermé (c'est-à-dire enfoncé)
Résultat : (0040) = 00

Organigramme : (voir page 379)

Programme source :

```
        LD      A,01001111B     ; MISE DU PORT A EN ENTREE
        OUT     (PIOCRA),A
        LD      HL,40H          ; INITIALISATION DU COMPTE RENDU A ZERO
        LD      (HL),0
        IN      A,(PIODRA)      ; LECTEUR DE LA POSITION DU BOUTON POUSSOIR
        AND     MASK            ; EST-CE QUE LE CONTACT EST FERME ?
        JR      Z,FIN           ; OUI, FIN
        INC     (HL)            ; NON, LE MOT DE COMPTE RENDU = 1
FIN :   HALT
```

Organigramme



Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3E	LD	A,01001111B
0001	4F		
0002	D3	OUT	(PIOCRA),A
0003	PIOCRA		
0004	21	LD	HL,40H
0005	40		
0006	00		
0007	36	LD	(HL),0
0008	00		
0009	DB	IN	A,(PIODRA)
000A	PIODRA		
000B	E6	AND	MASK
000C	MASK		
000D	28	JR	Z,FIN
000E	01		
000F	34	INC	(HL)
0010	76	HALT	

Les adresses des ports PIOCRA et PIODRA (adresse du registre de contrôle et registre donnée) dépendent de la façon dont le PIO est connecté au micro-ordinateur. Les lignes de contrôle du PIO sont inutilisées dans cet exemple. En fait, on aurait pu mettre la partie A du PIO dans le mode contrôle avec la séquence d'initialisation suivante * :

```
LD      A,11001111B     ; MISE DU PORT A EN CONTROLE
OUT     (PIOCRA),A
LD      A,0FFH          ; TOUS LES BITS EN ENTREE
OUT     (PIOCRA),A
```

La valeur MASK dépend du bit sur lequel le bouton poussoir est relié ; elle a un 1 sur la position du bouton poussoir et des zéros ailleurs.

Position du bouton poussoir (n° bit)	MASK	
	Binaire	Hexa
0	00000001	01
1	00000010	02
2	00000100	04
3	00001000	08
4	00010000	10
5	00100000	20
6	01000000	40
7	10000000	80

Si le bouton poussoir est relié au bit 7 ou 0 du port d'entrée, le programme peut utiliser une instruction de décalage pour positionner la retenue et déterminer ainsi la position du bouton poussoir. Par exemple :

```
Bit 7
IN      A,(PIODRA)      ; LECTURE DE LA POSITION DU CONTACT
RLA                     ; EST-CE QUE LE CONTACT EST FERME (ZERO) ?
JR      NC, FIN         ; OUI, FIN
Bit 0
IN      A,(PIODRA)      ; LECTURE DE LA POSITION
RRA                     ; EST-CE QUE LE CONTACT EST FERME (ZERO) ?
JR      NC, FIN         ; OUI, FIN
```

La procédure pour le bit 7 est même plus simple si nous avons l'adresse du registre de donnée du PIO dans le registre C. En effet, les instructions d'entrée utilisant l'adressage indirect de registre (IN A,(C)) affectent l'indicateur de signe (S). La séquence devient alors :

```
Bit 7 (PIODRA dans le registre C)
IN      A,(C)           ; LECTURE DE LA POSITION DU CONTACT
JP      P,FIN           ; FIN SI LE CONTACT EST FERME (ZERO)
```

* NDT : on ne saurait effectivement trop recommander le mode 3 pour cette application, l'utilisation du mode 1 impliquant la mise en œuvre impérative du signal STROBE, ce qui n'est pas le cas ici.

Si le bouton poussoir est relié aux bits 6 ou 7 du port d'entrée, le programme peut utiliser le bit de signe pour déterminer l'état du contact, exemple :

```
Bit 7
IN      A,(PIODRA)   ; LECTURE DE LA POSITION DU CONTACT
AND     A            ; EST-CE QUE LE CONTACT EST FERME (ZERO)
JP      P,FIN        ; OUI, FIN
```

L'instruction IN A,(port) n'affecte pas les indicateurs ; cependant nous utilisons l'instruction AND A pour positionner les indicateurs sans modifier l'accumulateur.

```
Bit 6
IN      A,(PIODRA)   ; LECTURE DE LA POSITION DU CONTACT
ADD     A,A          ; EST-CE QUE LE CONTACT EST FERME (ZERO) ?
JP      P,FIN        ; OUI, FIN
```

L'instruction RLA ne peut être utilisée car elle n'affecte pas le bit de signe.

Tâche 2 : comptage des fermetures du contact

But : compter le nombre de fermetures du bouton poussoir en incrémentant la mémoire 0040 après chaque fermeture.

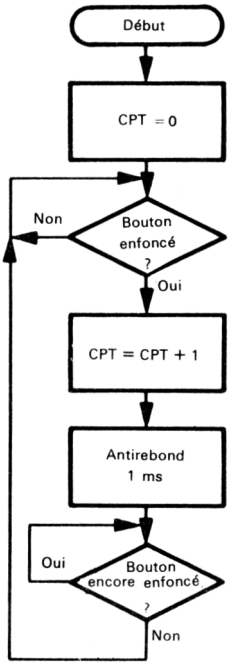
Exemple :

En pressant le bouton dix fois après le départ du programme, on doit obtenir (0040) = 0A.

REBONDS Remarque : Si l'on désire compter le nombre de fois que le bouton a été pressé, on doit être sûr que chaque fermeture provoque une seule transition. Cependant un bouton poussoir mécanique ne génère pas une simple transition pour une fermeture, en effet les contacts mécaniques rebondissent avant d'atteindre leurs positions finales. On doit soit utiliser un circuit (one shot, par exemple) pour éliminer le rebond, soit le gérer par le logiciel.

ANTIREBONDS PAR LOGICIEL Le programme peut supprimer les rebonds en attendant après chaque transition une fermeture. Cette temporisation est appelée la constante d'antirebonds et est une donnée spécifique du bouton poussoir, de l'ordre de quelques millisecondes. Durant cette période, le programme ne devra pas examiner l'état du bouton sinon il pourrait interpréter les rebonds comme de nouvelles fermetures. Le programme peut soit exécuter un sous-programme de temporisation, soit exécuter d'autres tâches pendant un temps donné.
Après la phase d'antirebond, le programme devra attendre encore la réouverture du contact, de manière à être sûr de ne pas compter deux fois le même événement. Le programme suivant utilise une temporisation de 1 ms pour gérer les rebonds du bouton poussoir. On peut essayer de diminuer cette temporisation ou de l'éliminer complètement pour voir ce qui se passe. Pour exécuter ce programme, on doit aussi entrer le programme de temporisation à partir de l'adresse 0030.

Organigramme :



Programme source :

```
        LD      A,01001111B     ; MISE EN ENTREE DU PORT A
        OUT     (PIOCRA),A
        LD      HL,40H
        LD      (HL),0          ; COMPTEUR DE FERMETURE = ZERO
VERFER: IN      A,(PIODRA)      ; LECTURE DE LA POSITION DU BOUTON POUSSOIR
        AND     MASK            ; EST-CE QUE LE BOUTON EST ENFONCE (0) ?
        JR      NZ,VERFER       ; NON, ATTENTE QU'IL LE SOIT
        INC     (HL)            ; OUI, INCREMENTATION DU COMPTEUR DE FERMETURE
        CALL    TEMPO           ; ATTENTE 1 MS POUR LES REBONDS
VEROUV: IN      A,(PIODRA)      ; LECTURE DE LA POSITION DU BOUTON POUSSOIR
        AND     MASK            ; EST-CE QUE LE BOUTON EST ENCORE ENFONCE
        JR      Z,VEROUV        ; OUI, ATTENTE RELACHEMENT
        JR      VERFER          ; NON, CONTROLE DE LA PROCHAINE FERMETURE
```

Programme objet : (voir page 383)

Les trois instructions suivant l'étiquette VEROUV sont utilisées pour déterminer quand le bouton poussoir se réouvre.
Evidemment on n'a pas réellement besoin d'un PIO pour réaliser un tel interface. Un amplificateur adressable 3 états réaliserait ce travail aussi bien à moindre coût. Remarque : CF NDT page 380.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)		Instruction (Mnémonique)	
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	21		LD	HL,40H
0005	40			
0006	00			
0007	36		LD	(HL),0
0008	00			
0009	DB	VERFER :	IN	A,(PIODRA)
000A	PIODRA			
000B	E6		AND	MASK
000C	MASK			
000D	20		JR	NZ,VERFER
000E	FA			
000F	34		INC	(HL)
0010	CD		CALL	TEMPO
0011	30			
0012	00			
0013	DB	VEROUV :	IN	A,(PIODRA)
0014	PIODRA			
0015	E6		AND	MASK
0016	MASK			
0017	28		JR	Z,VEROUV
0018	FA			
0019	18		JR	CHKCL
001A	EE			

Un commutateur (interrupteur)

But : interfacer un commutateur à un microprocesseur Z80. L'interrupteur est un appareil mécanique qui est dans un ou l'autre de deux états stables, l'un appelé ouvert (0), l'autre appelé fermé (F).

ANTIREBONDS AVEC DES PORTES (NAND)

Schéma :

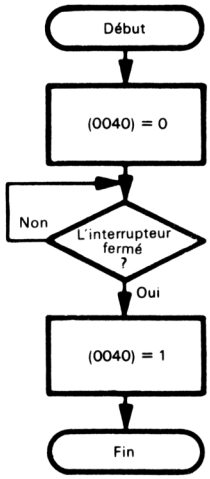
La figure 11.11 montre la circuiterie nécessaire pour interfacer l'interrupteur. Comme pour le bouton poussoir, l'interrupteur utilise un bit du PIO du Z80 qui sert d'amplificateur adressable, il peut de même occuper n'importe quelle position. Les tâches typiques des programmes sont de déterminer la position de l'interrupteur et de voir si celle-ci à changé.
Pour éliminer les rebonds, on peut soit entrecroiser une paire de portes (NAND) (figure 11.12), soit utiliser un monostable qui génère une impulsion de quelques millisecondes. Ces circuits produisent une simple impulsion en réponse à un changement de position de l'interrupteur même si celui-ci rebondit avant de parvenir à sa nouvelle position.

Exemple de programmation :
Ce circuit va nous permettre d'effectuer les deux tâches suivantes :
1) Positionner une mémoire à un quand le circuit est fermé.
2) Positionner une mémoire à un lorsque l'état de l'interrupteur change.

Tâche 1 : Attente de la fermeture de l'interrupteur

But : la mémoire 0040 est à zéro jusqu'à ce que l'interrupteur soit fermé puis est mise à un ; ainsi le processeur efface la mémoire 0040, attend tant que l'interrupteur se ferme puis positionne à un la mémoire 0040.
On peut considérer cet interrupteur comme interrupteur de marche-arrêt puisque le processeur n'exécutera d'autre programme que lorsque le contact sera fermé.

Organigramme :



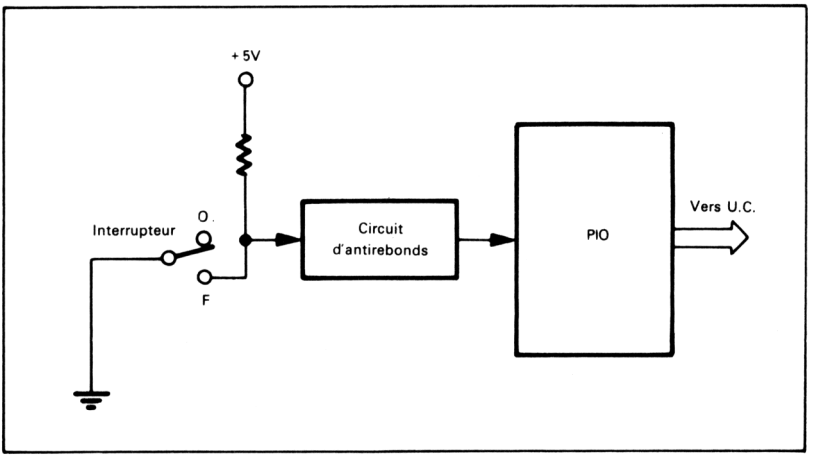


Fig. 11.11. — Interfaçage d'un interrupteur.

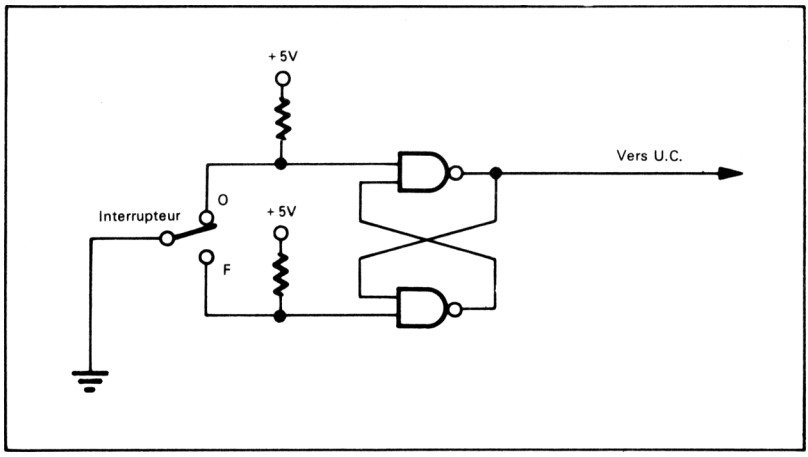


Fig. 11.12. — Un circuit d'antirebonds basé sur des portes (NAND) rétrocouplées.

Programme source :

```
        LD      A,01001111B     ; MISE DU PORT A EN ENTREE
        OUT     (PIOCRA),A
        LD      HL,40H
        LD      (HL),0          ; INITIALISATION A ZERO
ATTFER: IN      A,(PIODRA)      ; LECTURE DE LA POSITION DE L'INTERRUPTEUR
        AND     MASK            ; LA POSITION EST FERMEE ?
        JR      NZ,ATTFER       ; NON, ATTENTE DE LA FERMETURE
        INC     (HL)            ; OUI, MISE A 1 DU COMPTE RENDU
        HALT
```

Programme objet :

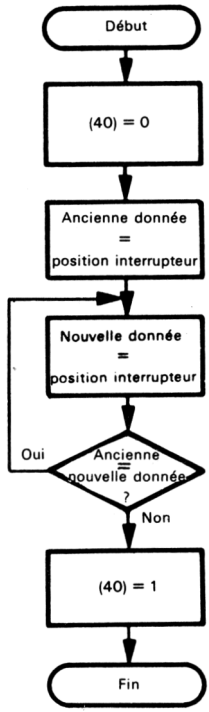
Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3E	LD	A,01001111B
0001	4F		
0002	D3	OUT	(PIOCRA),A
0003	PIOCRA		
0004	21	LD	HL,40H
0005	40		
0006	00		
0007	36	LD	(HL),0
0008	00		
0009	DB	ATTFER : IN	A,(PIODRA)
000A	PIODRA		
000B	E6	AND	MASK
000C	MASK		
000D	20	JR	NZ,ATTFER
000E	FA		
000F	34	INC	(HL)
0010	76	HALT	

Remarque : CF NDT page 380.

Tâche : Attente d'un changement d'état de l'interruption

But : la mémoire 0040 reste à zéro tant que la position de l'interrupteur ne change pas, puis elle passe à un lors d'un changement.

Organigramme :



Programme source :

```
        LD      A,01001111B     ; MISE DU PORT A EN ENTREE
        OUT     (PIOCRA),A
        LD      HL,40H
        LD      (HL),0          ; COMPTE RENDU = 0
        IN      A,(PIODRA)      ; ACQUISITION DE L'ANCIENNE POSITION
        AND     MASK
        LD      B,A
RCH :   IN      A,(PIODRA)      ; ACQUISITION DE LA NOUVELLE POSITION
        AND     MASK
        CP      B               ; EST-CE QUE L'ANCIENNE POSITION = NOUVELLE
        JR      Z,RCH           ; OUI, ATTENTE
        INC     (HL)            ; NON, COMPTE RENDU = 1
        HALT
```

Remarque : CF NDT page 380.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)		Instruction (Mnémonique)	
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	21		LD	HL,40H
0005	40			
0006	00			
0007	DB		IN	A,(PIODRA)
0008	PIODRA			
0009	E6		AND	MASK
000A	MASK			
000B	47		LD	B,A
000C	DB	RCH :	IN	A,(PIODRA)
000D	PIODRA			
000E	E6		AND	MASK
000F	MASK			
0010	B8		CP	B
0011	28		JR	Z,RCH
0012	F9			
0013	34		INC	(HL)
0014	76		HALT	

Une instruction de soustraction ou un ou exclusif peut remplacer l'instruction de comparaison (CP) du programme. Ces instructions changent cependant le contenu de l'accumulateur. Le ou exclusif peut être utile si plusieurs interruptions sont reliés au même PIO, ainsi on aurait un bit pour chaque interrupteur qui change d'état. Comment écririez-vous ce programme si les antirebonds devaient être gérés par logiciel ?

Un commutateur à multiple position (roue codeuse, sélecteur)

But : interfacer un commutateur à multiples positions vers un microprocesseur. Le point correspondant à la position du commutateur est connecté à la masse, tandis que les autres points sont au niveau logique un.

Schéma :

La figure 11.13 montre comment interfacer un commutateur à huit positions. Le commutateur utilise tous les huit bits de données d'un port de PIO. Les tâches classiques sont de déterminer la position du commutateur et de vérifier si cette position change ou pas. Deux situations spéciales doivent être gérées :

— Le commutateur est temporairement entre deux positions et aucune lignes n'est reliée à la masse.
— Le commutateur ne s'est pas stabilisé sur sa position finale.
La première situation peut être gérée en attendant jusqu'à ce qu'une entrée ne soit pas un « un », c'est-à-dire jusqu'à ce qu'une ligne soit reliée à la masse. On peut traiter la seconde situation en examinant la position du commutateur après une temporisation (1 ou 2 seconde) et accepter seulement l'entrée si elle est identique. Cette temporisation

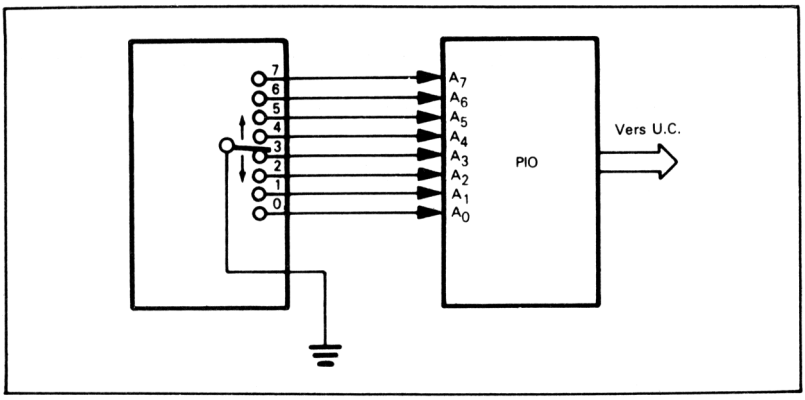


Fig. 11.13. — Commutation à multiples positions.

n'affectera pas le temps de réponse du système au commutateur. On peut aussi utiliser un autre interrupteur (c'est-à-dire un interrupteur de validation) pour dire au processeur quand le commutateur devra être lu.

Exemples de programmation :

Le circuit de la figure 11.13 va nous permettre d'effectuer les deux tâches suivantes :
a) Tester le commutateur jusqu'à ce qu'il soit dans sa position définitive, puis déterminer la position et stocker sa valeur binaire dans une mémoire.
b) Attendre un changement de la position du commutateur, puis stocker la nouvelle position dans une mémoire.
Si le commutateur est dans une position donnée, la ligne de cette position est connectée à la masse. Des résistances de polarisation sur les lignes d'entrées évitent les problèmes dus au bruit.

Tableau 11.3. — Données lues en fonction de la position du commutateur

Position du commutateur	Données lues	
	Binaire	Hexa
0	11111110	FE
1	11111101	FD
2	11111011	FB
3	11110111	F7
4	11101111	EF
5	11011111	DF
6	10111111	BF
7	01111111	7F

Tâche 1 : Détermination de la position du commutateur

But : le programme attend que le commutateur soit dans une position donnée et puis met le numéro de la position dans la mémoire 0040.
Le tableau 11.3 contient les données lues correspondant aux différentes positions du commutateur. Ce schéma est inefficace s'il est nécessaire de distinguer plus de huit positions différentes.

UTILISATION D'ENCODEUR TTL

Un encodeur TTL ou MOS peut réduire le nombre de bits nécessaires. La figure 11.14 décrit une utilisation du 74LS148, un encodeur 8 vers 3 [4]. On relie les sorties du commutateur dans l'ordre inverse, car le 74LS148 a des sorties et des entrées actives basses. La sortie de l'encodeur est une représentation sur 3 bits, de la position du commutateur. Beaucoup de commutateurs comprennent un encodeur de telle sorte que leurs sorties sont codées, généralement comme un chiffre BCD (en logique négative).

L'encodeur génère des sorties actives basses, ainsi par exemple la position 5 du commutateur laquelle est reliée à l'entrée 2, produit une sortie de 2 en logique négative (ou 5 en logique positive).

Vous pouvez vérifier la double complémentation par vous-même.

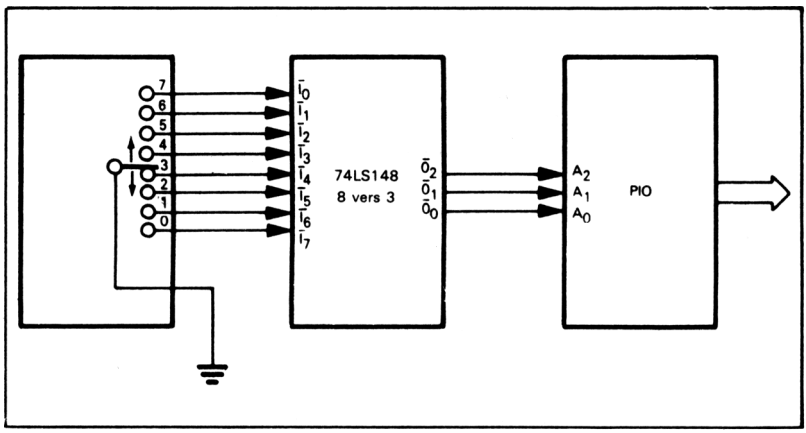


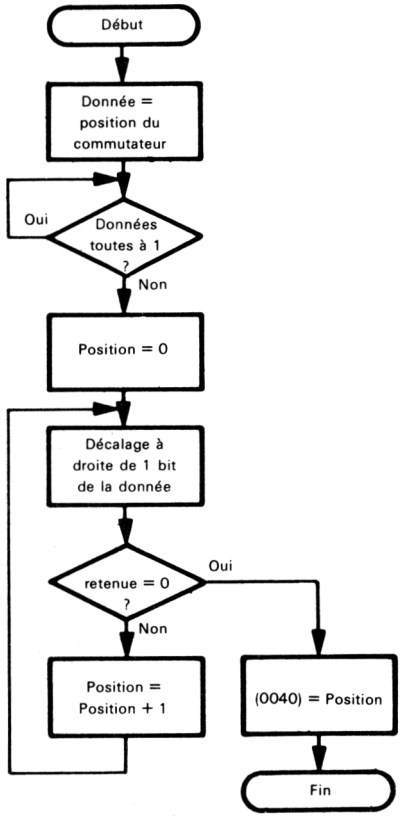
Fig. 11.14. — Un commutateur à positions multiples avec un encodeur

Organigramme : (voir page 390)

Programme source :

```
        LD      A,01001111B     ; MISE EN ENTREE DU PORT A
        OUT     (PIOCRA),A
VERPOS: IN      A,(PIODRA)      ; ACQUISITION DE LA DONNEE DU COMMUTATEUR
        CP      0FFH            ; EST-CE QUE LE COMMUTATEUR EST DANS UNE POSITION
        JR      Z,VERPOS        ; NON, ATTENTE D'UNE POSITION
        LD      B,0             ; POSITION DU COMMUTATEUR = 0
RCPOS : RRA                     ; EST-CE QUE LE PROCHAIN BIT EST A LA MASSE
        JR      NC,FIN          ; OUI, POSITION DU COMMUTATEUR TROUVEE
        INC     B               ; NON, INCREMENTE LA POSITION DU COMMUTATEUR
        JR      ROPOS
FIN :   LD      HL,40H          ; STOCKAGE DE LA POSITION
        LD      (HL),B
        HALT
```

Remarque : CF NDT page 380



Organigramme

Supposons que le commutateur ou un PIO défectueux font que les données d'entrées soient toujours à un (FF_{16}). Comment modifieriez-vous le programme pour qu'il détecte cette erreur ?
Il y a un saut inconditionnel, JR RCPOS, dans le programme source. Pouvez-vous modifier les conditions initiales pour rendre inutile cette instruction. Cet exemple suppose que le matériel gère les antirebonds. Comment modifieriez-vous le programme pour gérer les antirebonds en logiciel.

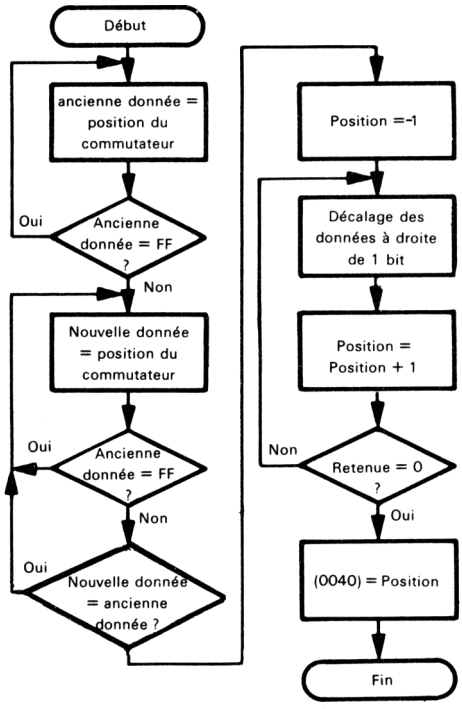
Tâche 2 : attente d'un changement de position de commutateur

But : le programme attend un changement de position du commutateur et met la nouvelle position (décodée) dans la mémoire 0040. Le programme attend que le commutateur atteigne sa nouvelle position.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	DB	VERPOS :	IN	A,(PIODRA)
0005	PIODRA			
0006	FE		CP	0FFH
0007	FF			
0008	28		JR	Z,VERPOS
0009	FA			
000A	06		LD	B,0
000B	00			
000C	1F	RCPOS :	RRA	
000D	30		JR	NC,FIN
000E	03			
000F	04		INC	B
0010	18		JR	RCPOS
0011	FA			
0012	21	FIN :	LD	HL,40H
0013	40			
0014	00			
0015	70		LD	(HL),B
0016	76		HALT	

Organigramme :



Programme source :

```
        LD      A,01001111B  ; MISE EN ENTREE DU PORT A
        OUT     (PIOCRA),A
CHFST:  IN      A,(PIODRA)   ; ACQUISITION DE LA DONNEE DU COMMUTATEUR
        CP      0FFH         ; EST-CE QUE LE COMMUTATEUR EST SUR UNE POSITION ?
        JR      Z,CHFST      ; NON, ATTENTE
        LD      B,A
CHSEC:  IN      A,(PIODRA)   ; ACQUISITION D'UNE NOUVELLE DONNEE DU COMMUTATEUR
        CP      0FFH         ; EST-CE QUE LE COMMUTATEUR EST SUR UNE POSITION
        JR      Z,CHSEC      ; NON, ATTENTE
        CP      B            ; EST-CE QUE LA POSITION EST IDENTIQUE A LA PRECEDENTE?
        JR      Z,CHSEC      ; OUI, ATTENTE D'UN CHANGEMENT
        LD      B,0FFH       ; POSITION DU COMMUTATEUR = - 1
CHPOS:  INC     B            ; INCREMENTATION DE LA POSITION DU COMMUTATEUR
        RRA                  ; EST-CE QUE LE PROCHAIN BIT EST A ZERO ?
        JR      C,CHPOS      ; NON, ATTENTE
        LD      HL,40H       ; STOCKAGE DE LA POSITION DU COMMUTATEUR
        LD      (HL),B
        HALT
```

Remarque : CF NDT page 380

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	DB	CHFST:	IN	A,(PIODRA)
0005	PIODRA			
0006	FE		CP	0FFH
0007	FF			
0008	28		JR	Z,CHFST
0009	FA			
000A	47		LD	B,A
000B	DB	CHSEC:	IN	A,(PIODRA)
000C	PIODRA			
000D	FE		CP	0FFH
000E	FF			
000F	28		JR	Z,CHSEC
0010	FA			
0011	B8		CP	B
0012	28		JR	Z,CHSEC
0013	F7			
0014	06		LD	B,0FFH
0015	FF			
0016	04	CHPOS:	INC	B
0017	1F		RRA	
0018	38		JR	C,CHPOS
0019	FC			
001A	21		LD	HL,40H
001B	40			
001C	00			
001D	70		LD	(HL),B
001E	76		HALT	

Une autre méthode pour déterminer si le commutateur est sur une position est :

```
VERPOS :  IN    A,(PIODRA)
          INC   A
          JR    Z,VERPOS
```

Pourquoi ? Qu'arrive-t-il à la donnée d'entrée ?
Cela marche-t-il ?

Une simple diode électroluminescente (LED)

But : interfacer une diode électroluminescente (LED : Light-Emitting Diode) à un microprocesseur Z80. La diode pourra être commandée soit par un un logique, soit par un zéro logique.

CONTRÔLE DE LA LED

Schéma :

La figure 11.15 montre le matériel nécessaire pour interfacer une LED. La diode émet de la lumière quand son anode et positive par rapport à sa cathode (figure 11.15a). De plus, on peut allumer la diode soit en reliant la cathode à la masse et en commandant un un à l'anode par le processeur (fig. 11.15b), soit en connectant l'anode au +5 V et en commandant un zéro à la cathode par le processeur (fig. 11.15c). L'utilisation de la cathode commune est l'approche la plus fréquente. L'utilisation d'impulsion de courant de 10 à 50 ma et ce à une fréquence de quelques centaines de fois par seconde, permet d'obtenir une luminosité optimale. Les LED ont un temps de commutation très court (de l'ordre de la microseconde), aussi on peut les multiplier facilement (commander plusieurs LED à partir d'un seul port). Les circuits de commandes des diodes requièrent des circuits de puissances (transistors) et des résistances de limitation du courant. Les circuits MOS ne peuvent pas être reliés directement à des LED ni leur fournir un courant suffisant pour les allumer.

Remarque : le PIO a des amplificateurs de sorties sur chaque port. Cependant le port B est utilisé généralement en sortie car il possède des capacités de sortance supérieures. En particulier les sorties du port B sont capables de commander des transistors Darlington (en fournissant 1,5 ma à 1,5 V). Les transistors Darlington sont des transistors à haut gain capables de commuter de forts courants à haute rapidité, ils sont très utilisés en commande de relais, solénoïdes, et autres appareils.

Tâche : allumer ou éteindre la diode

But : le programme allume ou éteint la diode.

a) Envoyer un un logique à la diode (allume un afficheur cathode à la masse, ou éteint un afficheur anode au +5 V).

Programme source :

(Envoi de la donnée initiale)

```
LD      A,00001111B     ; MISE EN SORTIE DU PORT B
OUT     (PIOCRB),A
LD      A,MASKP         ; ACQUISITION DE LA DONNEE
OUT     (PIODRB),A      ; ENVOI DE LA DONNEE VERS LA DIODE
HALT
```

Autre méthode utilisant le mode contrôle :

```
LD      A,11001111B     ; MISE DU PORT B EN MODE CONTROLE
OUT     (PIOCRB),A
SUB     A               ; MISE DES LIGNES DU PORT B EN SORTIE
OUT     (PIOCRB),A
LD      A,MASKP         : ACQUISITION DE LA DONNEE
OUT     (PIODRB),A      ; ENVOI DE LA DONNEE VERS LA LED
HALT
```

Mise à jour :

```
IN      A,(PIODRB)      ; ACQUISITION DE L'ANCIENNE DONNEE
SET     LED,A           ; ALLUMAGE DU BIT DE LA DIODE
OUT     (PIODRB),A      ; ENVOI DE LA DONNEE VERS LA LED
HALT
```

Le mot MASKP a un bit à un dans la position de la LED et des zéros ailleurs. On notera que l'on peut lire le registre de sortie du PIO quand le PIO est dans le mode sortie. On peut aussi bien lire n'importe quelle combinaison du registre d'entrée ou de sortie quand le PIO est dans le mode contrôle ; la combinaison est définie par les assignations des entrées et des sorties.

Remarque : cf. NDT page 380

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
	Envoi de la donnée initiale		
0000	3E	LD	A,00001111B
0001	0F		
0002	D3	OUT	(PIOCRB),A
0003	PIOCRB		
0004	3E	LD	A,MASKP
0005	MASKP		
0006	D3	OUT	(PIODRB),A
0007	PIODRB		
0008	76	HALT	
	Mise à jour des données		
0009	DB	IN	A,(PIODRB)
000A	PIODRB		
000B	CB	SET	LED,A
000C	LED		
000D	D3	OUT	(PIODRB),A
000E	PIODRB		
000F	76	HALT	

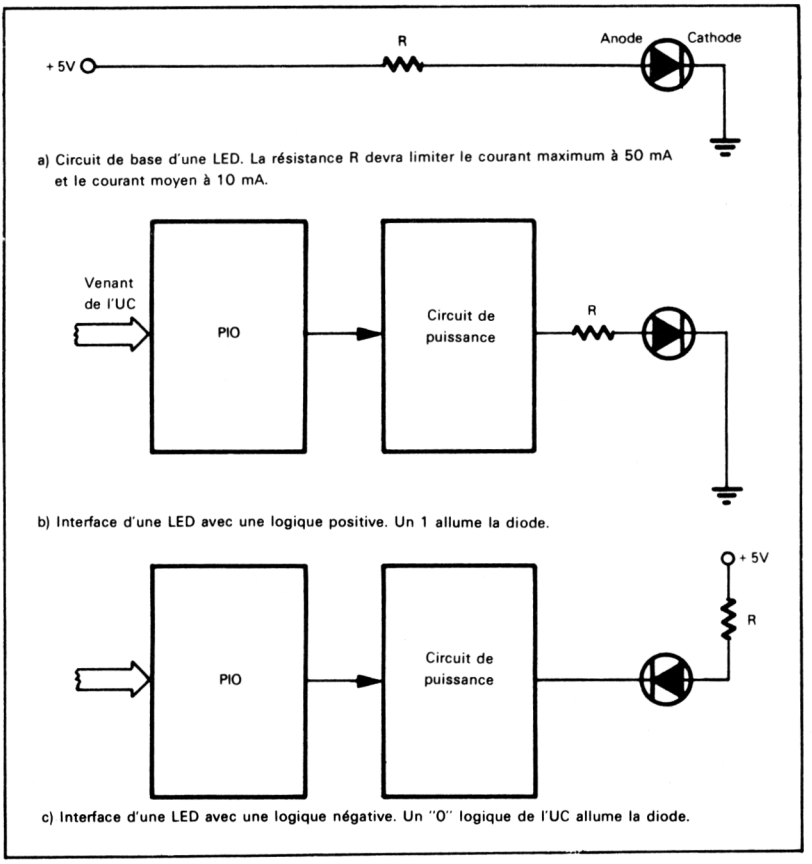


Fig. 11.15. — Interfaçage d'une diode électroluminescente.

B. Envoi d'un zéro logique à la diode (éteint un afficheur cathode à la masse, ou allume un afficheur à anode au + 5 V).
Les différences sont les suivantes :

— MASKP doit être remplacé par son complément logique MASKN
— L'instruction SET LED,A doit être remplacée par RES LED,A.
Remarquons que le second octet du code objet de SET LED,A et de RES LED,A dépend de la position du bit, lequel correspond au symbole LED.
MASKN a un zéro dans le bit correspondant à la position de la LED et des 1 ailleurs.

Affichage sept segments à diode

But : interfacer un affichage sept segments à diodes à un microprocesseur Z80. L'affichage peut être à cathode commune (logique positive) ou anode commune (logique négative).

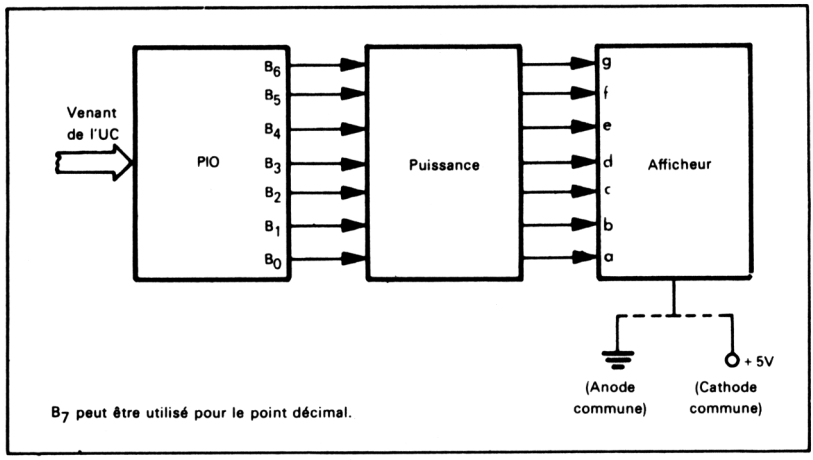


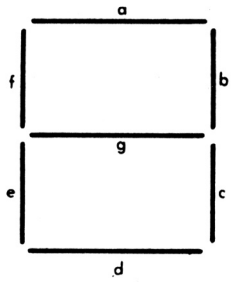
Fig. 11.16. — Interface d'un affichage sept segments.

AFFICHEURS A ANODE OU CATHODE COMMUNES

Schéma :

La figure 11.16 mondre le matériel nécessaire pour interfacer un afficheur sept segments. Il y a deux manières de connecter les afficheurs ; l'une est de relier toutes les cathodes ensemble à la masse (fig. 11.17a), c'est un afficheur à cathode commune ; l'autre est de relier toutes les anodes à la tension positive (fig. 11.17b), c'est un afficheur à anodes commune, et dans ce cas un zéro logique à une cathode allume un segment. Ainsi les afficheurs à cathodes communes utilisent une logique positive, et les afficheurs à anode commune une logique négative. Chaque afficheur nécessite des résistances et les circuits de puissance appropriés.

La ligne commune de l'afficheur est reliée soit au + 5 soit à la masse. Les segments des afficheurs sont couramment nommés.



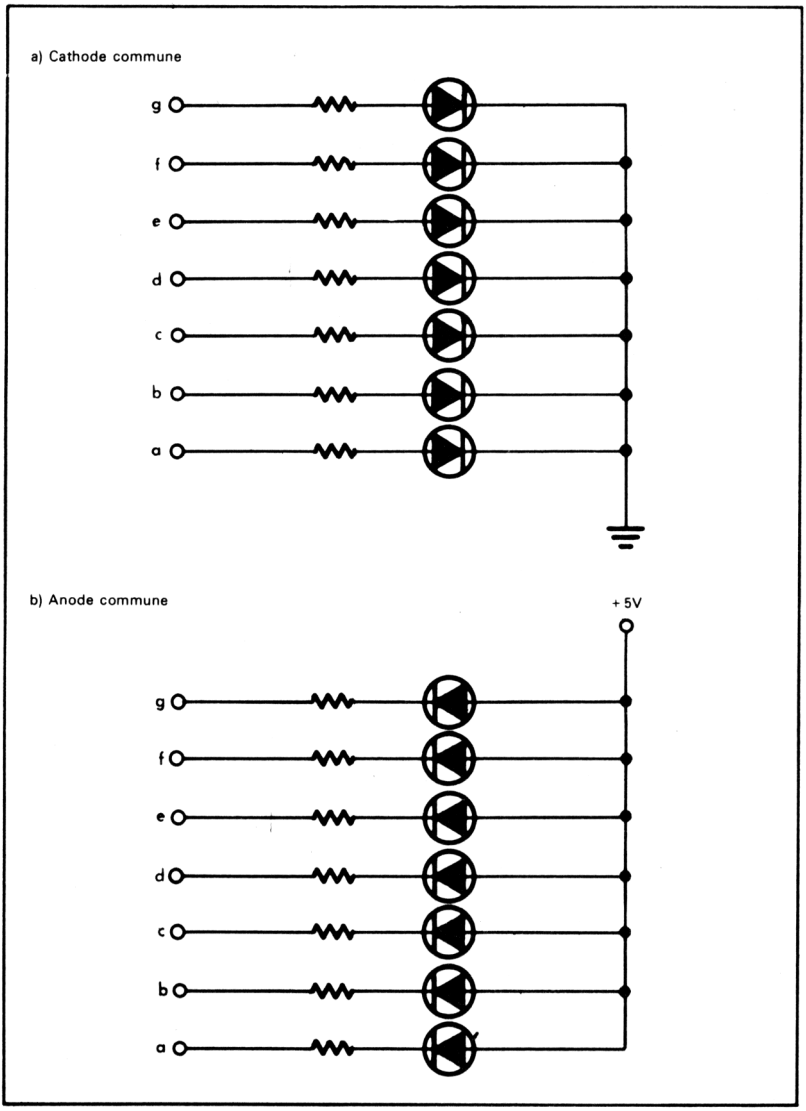


Fig. 11.17. — Organisation des afficheurs sept segments.

Tableau 11.4. — Représentation des nombres décimaux en sept segments.

Nombre	Représentation hexadécimale	
	Cathode commune	Anode commune
0	3F	40
1	06	79
2	5B	24
3	4F	30
4	66	19
5	6D	12
6	7D	02
7	07	78
8	7F	00
9	67	18

Bit 7 est toujours à zéro et les autres sont g, f, e, d, c, b, et a dans l'ordre décroissant.

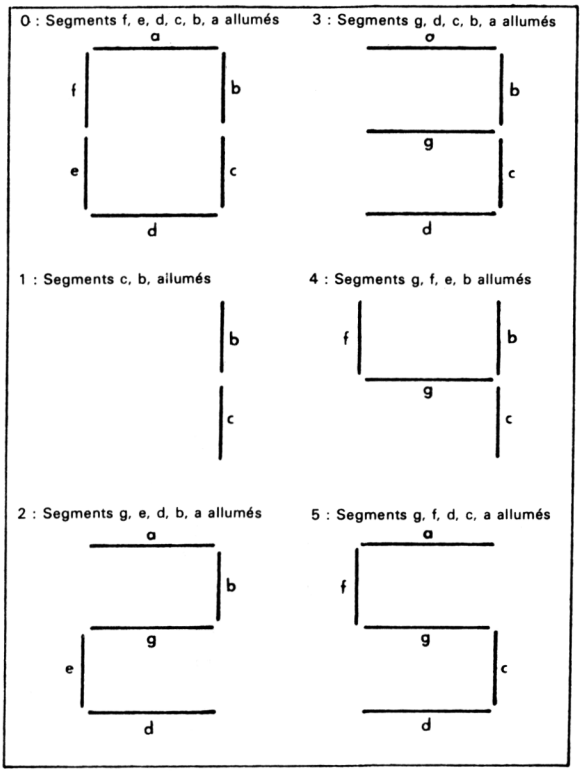


Fig. 11.18. — Représentation sept-segments des chiffres décimaux.

REPRÉSENTATIONS SEPT-SEGMENTS

Note : les afficheurs sept-segments sont largement utilisés parce qu'ils fournissent le plus petit nombre de segments qui peuvent contrôler la représentation des chiffres décimaux (figure 11.18 et tableau 11.4). Les afficheurs sept-segments peuvent aussi représenter quelques lettres et autres caractères (tableau 11.5). Une meilleure représentation demande sensiblement plus de segments et plus de circuits (5). Ainsi les afficheurs sept-segments sont très répandus et les décodeurs/amplificateurs sont peu coûteux et disponibles. Les circuits les plus utilisés sont les 7447, amplificateurs à anode communes et les 7448, amplificateurs à cathodes communes (6). Ces circuits possèdent une entrée de test qui permet d'allumer tous les segments et les entrées et sorties d'effacement.

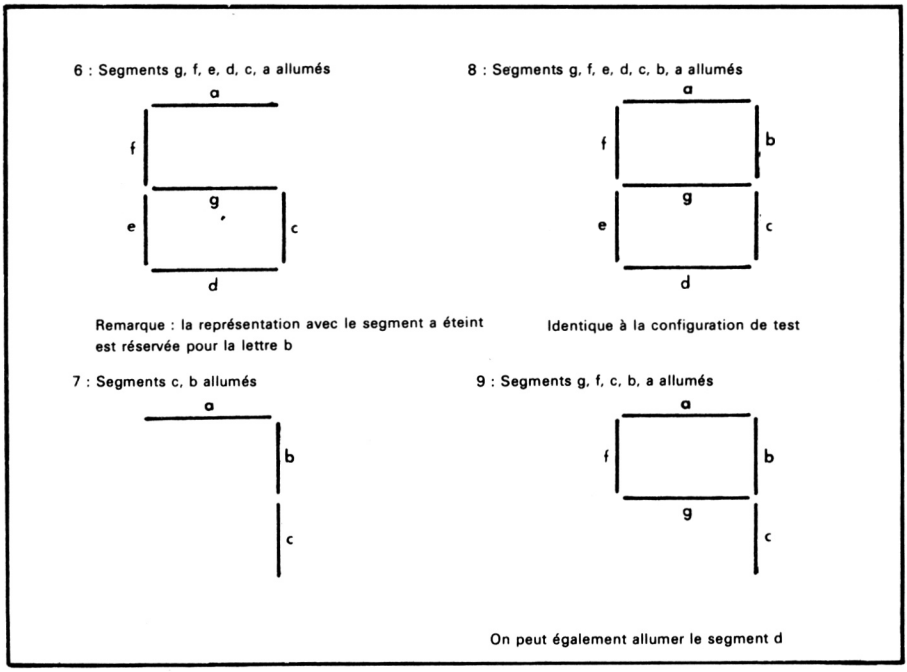


Fig. 11.18. — Représentation sept-segments des chiffres décimaux (suite).

Tâche 1 : affichage d'un chiffre décimal

But : afficher le contenu de la mémoire d'adresse 0040 sur un afficheur sept-segments, si son contenu est un chiffre décimal. Autrement éteindre l'afficheur.

Exemples pratiques :

a. (0040) = 05
   Afficher 5 sur l'afficheur.
b. (0040) = 66
   Eteindre l'afficheur

Tableau 11.5. — Représentation sept-segments des lettres et des symboles.

Lettres majuscules

Lettre	Représentation hexadécimale	
	Cathode commune	Anode commune
A	77	08
C	39	46
E	79	06
F	71	0E
H	76	09
I	06	79
J	1E	61
L	38	47
O	3F	40
P	73	0C
U	3E	41
Y	66	19

Lettres minuscules et caractères spéciaux

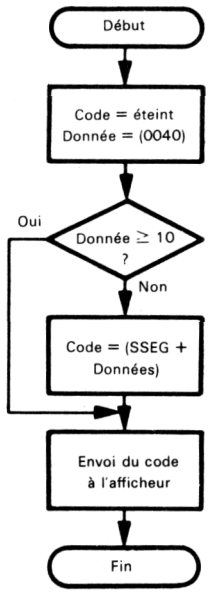
Caractère	Représentation hexadécimale	
	Cathode commune	Anode commune
b	7C	03
c	58	27
d	5E	21
h	74	0B
n	54	2B
o	5C	23
r	50	2F
u	1C	63
-	40	3F
?	53	2C

Organigramme : (voir page 401)

Programme source :

```
        LD      A,00001111B  ; MISE DU PORT B EN SORTIE
        OUT     (PIOCRB),A
        LD      B,ETEINT     ; ACQUISITION DU CODE ETEINT
        LD      A,(40H)      ; ACQUISITION DE LA DONNEE
        CP      10           ; LA DONNEE EST UN CHIFFRE DECIMAL ?
        JR      NC,DSPLY     ; NON, AFFICHEUR ETEINT
        LD      DE,SSEG      ; ACQUISITION DE L'ADRESSE DE BASE DE LA TABLE DE
                             ; SEPT-SEGMENTS
        LD      H,0          ; MISE DE LA DONNEE DANS UN INDEX A 16 BITS
        LD      L,A
        ADD     HL,DE        ; ACCES A L'ELEMENT DE LA TABLE
        LD      B,(HL)       ; ACQUISITION DU CODE SEPT SEGMENTS
DSPLY:  LD      A,B
        OUT     (PIODRB),A   ; ENVOI DU CODE A L'AFFICHEUR
        HALT
```

Organigramme



Le code ETEINT est 00 pour un afficheur à cathode commune, et FF pour un afficheur à anode commune. Un autre moyen est de mettre le code ETEINT à la fin de la table et de remplacer toutes les données impropres par 10 soit :

```
        LD    A,(40H)     ; ACQUISITION DE LA DONNEE
        CP    10          ; EST-CE UN CHIFFRE DECIMAL
        JR    C,CNVRT     ; OUI, CONVERSION EN CODE SEPT-SEGMENTS
        LD    A,10        ; NON, ACQUISITION INDEX POUR LE CODE ETEINT
CNVRT:  LD    DE,SSEG     ; ACQUISITION DE L'ADRESSE DE BASE DE LA TABLE
                          ; SEPT-SEGMENTS
```

La table SSEG est une représentation de la table 11.4 soit en cathode commune, soit en anode commune.

Remarque : cf. NDT page 380

Plusieurs afficheurs peuvent être multiplexés, comme décrit dans la figure 11.19. Un bref signal d'échantillonnage sur la ligne BRDY incrémente le compteur et dirige les données vers le prochain afficheur. Notons que BRDY est relié directement à BSTB, c'est-à-dire que la ligne prêt génère son propre acquittement. Les diagrammes de temps du PIO sont tels que cette connexion provoque un signal d'échantillonnage (STROBE) dont la durée est d'une période d'horloge. Un tel signal convient parfaitement au compteur. Le signal « RESET » RAZ démarre le compteur décimal, à neuf de telle sorte que la 1re opération de sortie remet à zéro le compteur et dirige la donnée vers le premier afficheur.

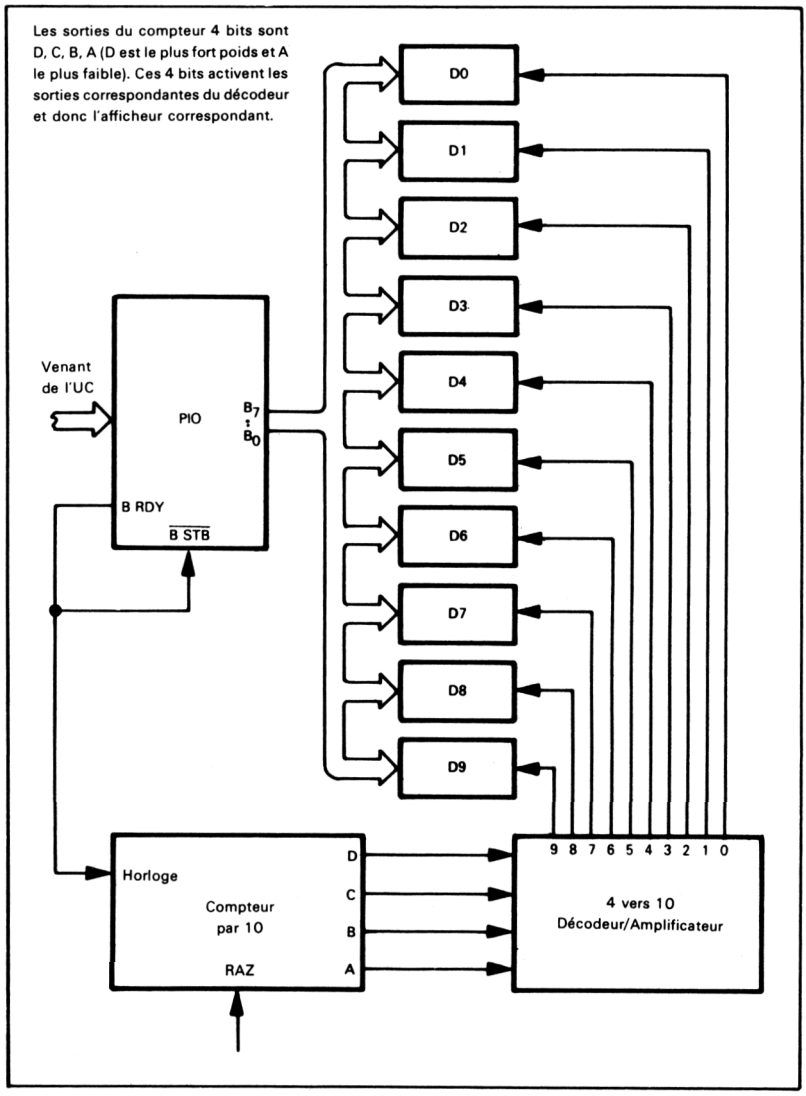


Fig. 11.19. — Afficheurs 7-segments Multiplexés.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3E	LD	A,00001111B
0001	0F		
0002	D3	OUT	(PIOCRB),A
0003	PIOCRB		
0004	06	LD	B, ETEINT
0005	BLANK		
0006	3A	LD	A,(40H)
0007	40		
0008	00		
0009	FE	CP	10
000A	0A		
000B	30	JR	NC,DSPLY
000C	08		
000D	11	LD	DE,SSEG
000E	20		
000F	00		
0010	26	LD	H,0
0011	00		
0012	6F	LD	L,A
0013	19	ADD	HL,DE
0014	46	LD	B,(HL)
0015	78	DSPLY: LD	A,B
0016	D3	OUT	(PIODRB),A
0017	PIODRB		
0018	76	HALT	
0020-0029		SSEG:	(TABLE DU CODE SEPT SEGMENTS)

Tâche 2 : afficher dix chiffres décimaux

But : affiche le contenu des mémoires 40 à 49 sur dix afficheurs sept segments qui sont multiplexés avec un compteur et un décodeur.

Exemple pratique :

(0040) = 66
(0041) = 3F
(0042) = 7F
(0043) = 7F
(0044) = 06
(0045) = 5B
(0046) = 07
(0047) = 4F
(0048) = 6D
(0049) = 7D
Affichage 4088127356

Programme source :

```
        LD      A,00001111B   ; MISE DU PORT B EN SORTIE
        OUT     (PIOCRB),A
DAFFI : LD      HL,40H        ; POINTER SUR LA DONNEE DE DEPART
        LD      B,10          ; NOMBRE D'AFFICHEURS = 10
        LD      C,PIODRB      ; ACQUISITION DU NUMERO DU PORT
DSPLY:  OUTI                  ; ENVOI DE LA DONNEE AUX AFFICHEURS
        CALL    TEMPO         ; ATTENTE DE 1 MS
        JR      NZ,DSPLY      ; COMPTEUR AFFICHAGE
        JR      DAFFI         ; BALAYAGE SUIVANT
```

Nous devons ici sélectionner le mode sortie du PIO, car le circuit utilise les signaux de dialogues (« handshake »).
Remarque : l'instruction OUTI envoie la donnée avec port de sortie adressé par le registre C, incrémente l'adresse dans la paire de registre HL et décrémente le compteur dans le registre B. On suppose que le sous-programme TEMPO n'affecte pas le flag Z, ainsi il peut être utilisé après coup pour un branchement conditionnel.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3E	LD	A,00001111B
0001	0F		
0002	D3	OUT	(PIOCRB),A
0003	PIOCRB		
0004	21	DAFFI : LD	HL,40H
0005	40		
0006	00		
0007	06	LD	B,10
0008	0A		
0009	0E	LD	C,PIODRB
000A	PIODRB		
000B	ED	DSPLY : OUTI	
000C	A3		
000D	CD	CALL	TEMPO
000E	30		
000F	00		
0010	20	JR	NZ,DSPLY
0011	F9		
0012	18	JR	DAFFI
0013	F0		

EXERCICES :

1. Un bouton-poussoir (marche-arrêt)

But : chaque fermeture du bouton-poussoir complémente tous les bits dans la mémoire 40. La mémoire contient initialement 0. Le programme devra examiner continuellement le bouton-poussoir et complémenter la mémoire 40 à chaque fermeture. On peut utiliser un port de sortie et le complémenter à la place de la mémoire de façon à visualiser plus facilement le résultat.

Exemple pratique :

La mémoire 40 contient initialement zéro.
Le premier appui provoque la complémentation de la mémoire 40, FF(hexa), le second appui la ramène dans son état initial, le 3e la remet à FF (hexa) etc. On suppose que les rebonds sont gérés par le matériel. Comment incluriez-vous la gestion des antirebonds dans le logiciel ?

2. Gestion des antirebonds par logiciel

But : gérer les antirebonds d'un commutateur, en s'assurant qu'après deux lectures successives séparées par un délai programmable donnent le même résultat. On suppose que la constante de temporisation (en ms) est contenue dans la mémoire 40 et l'on met la position du commutateur dans la mémoire 41.

Exemple :

(0040) = 03 provoque une attente de 3 ms entre deux lectures.

3. Contrôle d'un commutateur à multiples positions

But : un autre interrupteur sert de validation pour un commutateur quatre positions. L'UC attend que l'interrupteur de validation soit fermé (zéro) et lit, alors, la position du commutateur multiple. Cette procédure permet à l'opérateur de déplacer le commutateur sur sa position finale avant que l'UC acquière cette position. Le programme met la valeur de la position du commutateur dans la mémoire 40. Les antirebonds de l'interrupteur de validation sont gérés par logiciel.

Exemple pratique :

Mettre le commutateur sur la position 2. Fermer l'interrupteur de validation.

Résultat : (0040) = 2

4. Affichage de la position des interrupteurs

But : un ensemble de huit interrupteurs doivent avoir leurs positions reflétées sur huit diodes électroluminescentes. Si l'interrupteur est fermé (zéro), la diode s'allume, autrement la diode est éteinte. On suppose que le port de sortie de l'UC est connecté aux cathodes des diodes.

Exemple :

```
INTERRUPTEUR 0 FERME
INTERRUPTEUR 1 OUVERT
INTERRUPTEUR 2 FERME
INTERRUPTEUR 3 OUVERT
INTERRUPTEUR 4 OUVERT
INTERRUPTEUR 5 FERME
INTERRUPTEUR 6 FERME
INTERRUPTEUR 7 OUVERT
```

Résultat :

```
LED 0 ALLUMEE
LED 1 ETEINTE
LED 2 ALLUMEE
LED 3 ETEINTE
LED 4 ETEINTE
LED 5 ALLUMEE
LED 6 ALLUMEE
LED 7 ETEINTE
```

Comment modifieriez-vous le programme pour qu'un interrupteur relié au bit 7 du port A d'un PIO détermine si oui ou non les afficheurs sont actifs (c'est-à-dire si l'interrupteur de contrôle est fermé, les afficheurs reliés au port B reflètent les positions des interrupteurs reliés au port A ; si l'interrupteur est ouvert, les afficheurs sont éteints) ? Un tel interrupteur de contrôle est utile lorsque les afficheurs ne doivent pas attirer l'attention de l'opérateur, comme dans un avion.
Comment modifieriez-vous le programme pour utiliser un bouton-poussoir en tant qu'interrupteur de contrôle ; ainsi à chaque fermeture, il y a inversion de l'état de l'afficheur.
On suppose que les afficheurs sont actifs au départ, et que le programme examine et gère les antirebonds du bouton-poussoir avant d'envoyer les données aux afficheurs.

5) Comptage sur un afficheur sept-segments

But : le programme compte continuellement de 0 à 9 sur un afficheur sept-segments ; en commençant à 0.
Conseil : essayez différentes temporisations pour l'affichage et remarquez ce qui se passe. Quand l'affichage devient-il visible ? Qu'arrive-t-il si l'afficheur est éteint pendant un certain temps ?

CIRCUITS D'E/S PLUS COMPLEXES

Les circuits d'E/S complexes diffèrent des simples claviers, commutateurs et afficheurs par les points suivants :

1) Ils transfèrent les données à des vitesses plus rapides.
2) Ils peuvent avoir leurs propres horloges et diagrammes temporels.
3) Ils nécessitent des informations d'état ainsi que des informations de contrôle en plus des données transférées.

A cause de leurs grandes vitesses de transfert, on ne peut pas gérer ces circuits simplement. Si le processeur ne possède pas une gestion appropriée, le système peut manquer une donnée en entrée, ou produire une donnée en sortie erronée. On doit alors travailler avec beaucoup plus de précision que dans la gestion des circuits plus simples.
Les interruptions conviennent bien pour gérer ces circuits d'E/S complexes, comme nous le verrons dans le chapitre 12.

SYNCHRONISATION DES CIRCUITS D'E/S

Les périphériques comme les claviers, télé-imprimeurs, magnétophones et disques souples ont leurs propres vitesses de travail. Ces appareils génèrent des séquences de données, séparées par des laps de temps précis. Le calculateur doit synchroniser la première opération d'entrée/sortie avec l'horloge du périphérique, puis gérer le laps de temps entre deux opérations. Une simple boucle de temporisation comme celles vues précédemment, peut produire ces temporisations. La synchronisation exige l'utilisation d'un ou plusieurs des procédés suivants :

1) Attente d'une transition sur une ligne d'horloge ou de validation, provoquée par le périphérique. Une approche simple serait de connecter la validation à l'entrée STB du PIO et d'attendre une transition sur la sortie INT. Cependant, il n'est pas possible d'adresser directement la sortie INT (et de déterminer ainsi sa valeur en la lisant), ni de désactiver ce signal autrement que dans un sous-programme d'interruption. Ainsi pour utiliser un PIO dans un système en scrutation, on doit avoir le signal de validation disponible sur un port d'entrée et le mémoriser si nécessaire. Si la validation doit être mémorisée, un circuit doit permettre la désactivation de ce signal dès le transfert suivant.
2) Recherche du centre de l'intervalle de temps durant lequel la donnée est stable. On préfère déterminer la valeur d'une donnée au centre de l'impulsion plutôt près des fronts de transition où la donnée est instable. La recherche du centre demande une temporisation d'une demi-période de transmission (durée d'un bit) après le front de départ. Echantillonner la donnée au centre signifie aussi que les petites erreurs de synchronisation n'ont pas d'effet sur la valeur reçue.
3) Reconnaissance d'un code spécial de départ. Ceci est facile si le code est un simple bit ou si on dispose d'un moyen de synchronisation. Le procédé est plus complexe si le code est long et peut démarrer n'importe quand. Des décalages seront nécessaires pour déterminer où le transmetteur démarre ses bits, caractères ou messages (cela s'appelle la recherche de la trame (« framing »).
4) Echantillonnage multiple. Cela réduit la probabilité de recevoir une donnée incorrecte sur des lignes de mauvaises qualités. Une logique majoritaire (telle que 3 parmi 5, ou 5 parmi 8) peut être utilisée pour décider la valeur la plus probable.
La réception est, évidemment, plus difficile que la transmission car le périphérique contrôle la réception et le calculateur doit interpréter les informations de cadencement générées par le périphérique. En transmission, le calculateur produit son propre cadencement et son formattage pour un périphérique donné.

INFORMATION DE CONTRÔLE ET D'ÉTAT

Les périphériques peuvent demander ou produire d'autres informations en plus des données et des cadencements. Nous appellerons les autres informations fournies par le calculateur des informations de contrôle ; ces informations incluent sélections de modes opératoires, arrêt ou départ de processus, registres d'horloges, validations de registres tampons, choix de formats ou protocole, produire un affichage pour l'opérateur, compter des opérations ou identifier le type et la priorité de

l'opération. Nous appellerons les autres informations fournies par le périphérique des informations d'état ; elles peuvent indiquer le mode opératoire, la présence de conditions d'erreurs, l'état du périphérique (prêt, occupé), le format du protocole en fonctionnement et d'autres conditions ou états.
Le calculateur gère les informations de contrôle et d'états comme des données. Ces informations varient, même si les données peuvent être transférées à grande vitesse. Les informations de contrôle ou d'état peuvent être quelques bits, octets, mots ou mots multiples. Souvent les informations sur un ou quelques bits sont groupées et traitées par un même port d'entrée et de sortie. En combinant les informations de contrôle ou d'état dans des octets, on réduit le nombre total d'adresses de port d'E/S nécessaires au périphérique. Cependant, la combinaison signifie que les états individuels des bits d'état en entrée doivent être interprétés séparément, et les bits de contrôle en sorties doivent être déterminés séparément. Les méthodes pour isoler un bit d'état ou pour positionner un bit de contrôle sont les suivantes :

SÉPARATION DES INFORMATIONS D'ÉTAT

Séparation des bits d'état de sortie :

Etape 1 : lecture de la donnée d'état du périphérique.
Etape 2 : ET logique avec un masque (le masque contient des uns dans les positions des bits qui doivent être examinés et des zéros ailleurs).
Etape 3 : décalage des bits séparés vers les positions des bits les moins significatifs.
Si le champ est un bit unique, l'étape 2 n'est pas nécessaire puisque l'on peut tester le bit avec l'instruction BIT. Si le bit unique est dans la position la plus significative (bit 7), la suivante (bit 6) ou la moins significative (bit 0), on peut utiliser les instructions de décalage logique (SL A et SR A) pour déterminer sa valeur. Se souvenir également que les instructions d'entrées avec un adressage indirect de registre (par ex. : IN A,(C) affectent le flag de signe. Ces positions les plus accessibles sont souvent réservées pour les informations d'état les plus fréquentes. Essayez d'écrire les séquences d'instructions nécessaires pour le processeur Z80.
L'étape 3 n'est pas nécessaire si le champ est un bit unique, car l'indicateur zéro contiendra le complément de ce bit après l'étape 2 (essayez).

COMBINAISON DES INFORMATIONS DE CONTRÔLE

Positionnement et effacement des bits de contrôle

Etape 1 : lecture des informations de contrôle.
Etape 2 : ET logique avec masque pour effacer les bits (le masque a des zéros dans les positions des bits à effacer et des uns ailleurs).
Etape 3 : OU logique avec masque pour mettre les bits à un (le masque a des uns dans les bits à positionner, des zéros ailleurs).
Etape 4 : envoi du nouveau mot de contrôle au périphérique.
Ici encore, la procédure est plus simple si le champ est un bit unique et occupe une position à la fin du mot.
Voici quelques exemples de bit d'état séparés et combinés :
1) Un champ de 3 bits, de la position 2 à la position 4 d'un registre de donnée d'un PIO, est un facteur d'échelle.
Mettre le facteur d'échelle dans l'accumulateur.

```
;
; LECTURE DE LA DONNEE D'ETAT A PARTIR DU PORT D'ENTREE
;
        IN      A,(PIODR)       ; LECTURE DE LA DONNEE D'ETAT
;
; MASQUAGE DU FACTEUR D'ECHELLE ET DECALAGE
;
        AND     00011100B       ; MASQUAGE DU FACTEUR D'ECHELLE
        RRCA                    ; DECALAGE DE 2 POUR CADRER
        RRCA
```

2) L'accumulateur contient un champ de 2 bits qui doivent être placés dans la positions 3 et 4 d'un registre donné d'un PIO.

```
;
; ENVOI DES DONNEES EN POSITION VOULUE
;
        RLA                     ; DECALAGE DE LA DONNEE VERS LES POSITIONS DES
        RLA                     ; BITS 3 ET 4
        RLA
        AND     00011000B       ; EFFACEMENT DES AUTRES POSITIONS DE BIT
        LD      B,A             ; SAUVEGARDE DE LA NOUVELLE VALEUR DU CHAMP

;
; COMBINAISON DE LA NOUVELLE VALEUR DU CHAMP AVEC UNE AUTRE DONNEE
;
        IN      A,(PIODR)       ; EFFACEMENT DE L'ANCIENNE VALEUR
        AND     11100111B
        OR      B               ; INSERTION DE LA NOUVELLE VALEUR
        OUT     (PIODR),A
```

DOCUMENTATION DES TRANSFERTS DES CONTRÔLES ET DES ÉTATS

La documentation est un problème sérieux dans la gestion des informations de contrôle et d'état. La signification des états en entrée, ou des contrôles en sortie, est souvent obscure. Le programmeur devra clairement indiquer le but de chaque opération d'entrée ou de sortie, dans les commentaires, exemple : « Vérification si le lecteur est sous tension », « Choix de l'option parité paire », ou « Activation du compteur de transmission ». Autrement la compréhension et la mise au point des instructions de manipulation de bit, de décalage et d'opération logiques seront très difficiles.

EXEMPLES

Un clavier non encodé

But : tester l'enfoncement d'une touche sur un clavier non encodé 3×3 et mettre le numéro de la touche qui est pressée dans l'accumulateur.

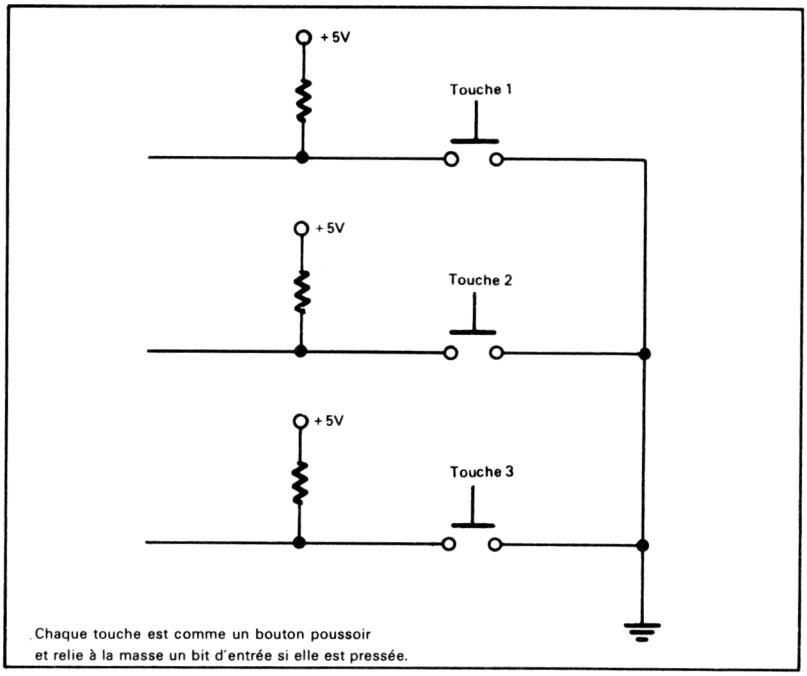


Fig. 11.20. — Un petit clavier.

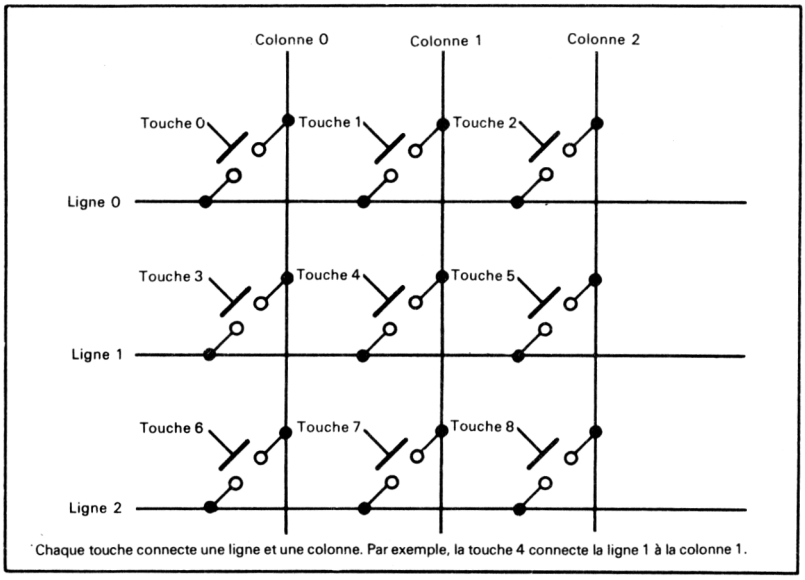


Fig. 11.21. — Un clavier matricé.

Les claviers sont un ensemble de commutateurs (cf. figure 11.20). Les claviers comportant un petit nombre de touches sont plus faciles à gérer, si chaque touche est reliée séparément à un bit d'un port d'entrée. L'interface d'un clavier est alors la même que celle d'un ensemble d'interrupteur.

CLAVIER À MATRICE Les claviers à plus de huit touches demandent plus d'un port d'entrée et donc des opérations sur plusieurs octets. Ceci constitue un véritable gaspillage si les touches sont logiquement distinctes comme dans une calculatrice ou un clavier de terminal, l'utilisateur n'en frappant qu'une à la fois. Le nombre de lignes d'entrées nécessaire peut être réduit en connectant les touches en matrice, comme le montre la figure 11.21. Maintenant chaque touche représente une connexion potentielle entre une ligne et une colonne. Un clavier en matrice demande n + m lignes externes, ou n est le nombre de lignes et m est le nombre de colonnes. A comparer aux n × m lignes externes si chaque touche est séparée. Le tableau 11.6 compare le nombre de lignes exigées par les différents types de claviers.

Tableau 11.6. — Comparaison entre les claviers à connexions matricielles et à connexions indépendantes.

Taille du clavier	Nombre de lignes avec des connexions indépendantes	Nombre de lignes avec une connexion matricielle
3 x 3	9	6
4 x 4	16	8
4 x 6	24	10
5 x 5	25	10
6 x 6	36	12
6 x 8	48	14
8 x 8	64	16

SCRUTATION DU CLAVIER Un programme peut déterminer quelle touche a été appuyée en utilisant les lignes externes de la matrice. La procédure usuelle est une scrutation de clavier. On relie la ligne 0 à la masse et l'on examine les colonnes. Si des colonnes sont à la masse, une touche dans cette colonne a été appuyée, en provoquant une liaison ligne colonne. On peut identifier quelle touche a été appuyée en déterminant quelle colonne a été reliée à la masse, c'est-à-dire quel bit du port d'entrée est à zéro. Si aucune colonne n'est à la masse, on passe à la ligne 1 et on répète le balayage. Remarquons que l'on peut vérifier si une touche a été appuyée en reliant à la masse toutes les lignes à la fois et en examinant les colonnes.
La scrutation du clavier nécessite que chaque ligne soit reliée à un port de sortie et que chaque colonne soit reliée à un port d'entrée. La figure 11.22 montre la disposition du clavier. L'UC peut relier à la masse une ligne particulière en plaçant un zéro dans le bit approprié du port de sortie, et des uns dans les autres bits. L'UC peut déterminer l'état d'une colonne particulière en examinant le bit approprié du port d'entrée.

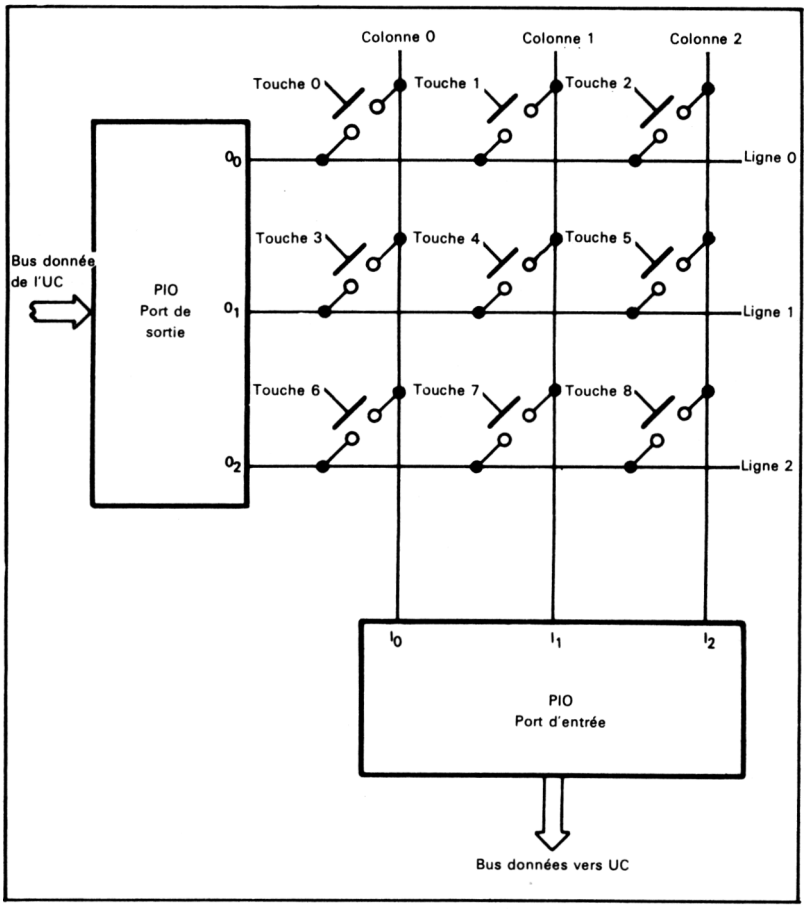


Fig. 11.22. — Disposition d'E/S pour une scrutation de clavier.

ATTENTE D'UN ENFONCEMENT D'UNE TOUCHE

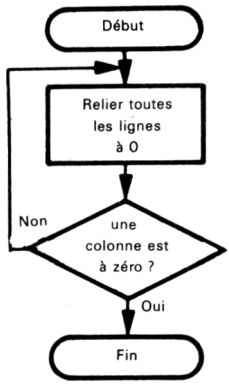
Tâche 1 : déterminer l'enfoncement d'une touche

But : attendre qu'une touche soit appuyée.

La procédure est la suivante :

1) Relier à la masse toutes les lignes en effaçant tous les bits de sortie.
2) Acquérir les entrées des colonnes en lisant le port d'entrée.
3) Retourner à l'étape 1 si toutes les colonnes sont à 1.

Organigramme :



Programme source :

```
        LD      A,01001111B     ; MISE DU PORT A EN ENTREE
        OUT     (PIOCRA),A
        LD      A,00001111B     ; MISE DU PORT B EN SORTIE
        OUT     (PIOCRB),A
        SUB     A               ; RELIER A LA MASSE TOUTES LES LIGNES DU CLAVIER
        OUT     (PIODRB),A
ATTCH : IN      A,(PIODRA)      ; ACQUERIR LES DONNEES DES COLONNES DU CLAVIER
        AND     00000111B       ; MASQUER LES BITS DES COLONNES
        CP      00000111B       ; EST-CE QU'UNE COLONNE EST A LA MASSE ?
        JR      Z,ATTCH         ; NON, BOUCLER
        HALT
```

Remarque : cf. NDT page 380

Programme objet : (voir page 414)

Le port B du PIO est le port de sortie vers le clavier, et le port A est le port d'entrée. Le masquage des colonnes élimine les problèmes qui peuvent être provoqués par les états des lignes inutilisées.
On peut généraliser le sous-programme en nommant les valeurs de sorties et de masque :

```
ALLG    EQU     11111000B
OPEN    EQU     00000111B
```

Ces noms peuvent alors être utilisés dans le programme ; un clavier différent nécessitera seulement une modification des valeurs et un ré-assemblage. Bien évidemment, un seul port d'un PIO est réellement nécessaire pour un clavier 3×3, ou 4×4. Essayez de ré-écrire le programme en utilisant seulement le port A. Le PIO doit être mis dans le mode contrôle ainsi chaque ligne peut être individuellement sélectionnée en entrée ou en sortie.

Programme objet :

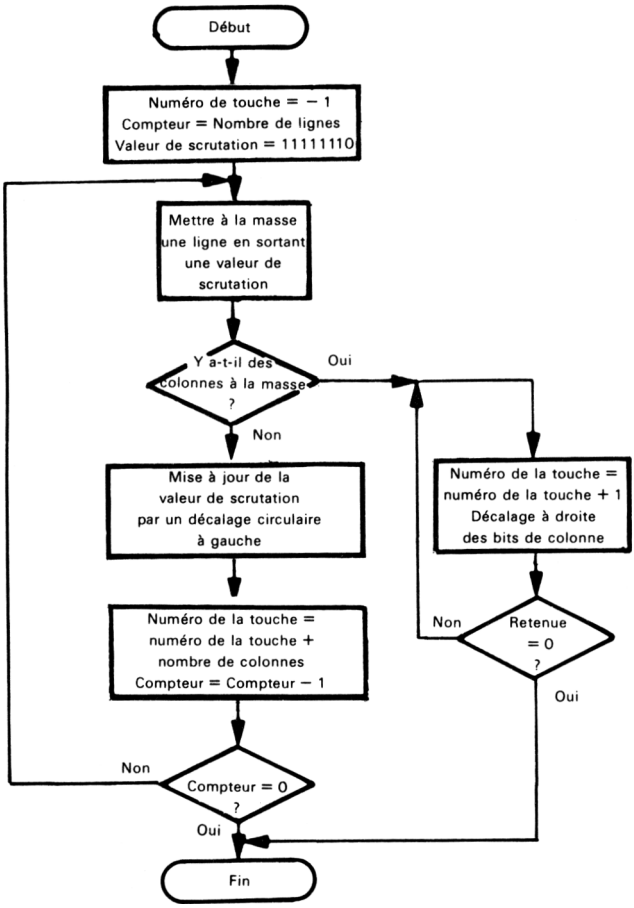
Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3E	LD	A,01001111B
0001	4F		
0002	D3	OUT	(PIOCRA),A
0003	PIOCRA		
0004	3E	LD	A,00001111B
0005	0F		
0006	D3	OUT	(PIOCRB),A
0007	PIOCRB		
0008	97	SUB	A
0009	D3	OUT	(PIODRB),A
000A	PIODRB		
000B	DB	ATTCH : IN	A,(PIODRA)
000C	PIODRA		
000D	E6	AND	00000111B
000E	07		
000F	FE	CP	00000111B
0010	07		
0011	28	JR	Z,ATTCH
0012	F8		
0013	76	HALT	

Tâche 2 : identification de la touche

But : identifier une touche appuyée en plaçant le numéro de la touche dans l'accumulateur.
La procédure est la suivante :

1) Initialiser le numéro de touche à -1, le compteur au nombre de lignes et la valeur de scrutation avec 1 sauf dans le bit 0.
2) Mettre à zéro une colonne en envoyant la valeur de scrutation vers le port de sortie du clavier.
3) Mise à jour de la valeur de scrutation par un décalage à gauche d'une position.
4) Acquisition de l'entrée des colonnes en lisant le port d'entrée.
5) Si une entrée colonne est à zéro, aller à l'étape 8.
6) Additionner le nombre de colonnes au nombre de touches pour chercher la prochaine ligne.
7) Décrémenter le compteur. Allez à l'étape 2 si des lignes n'ont pas été balayées, autrement aller à l'étape 10.
8) Additionner 1 au numéro de touche. Décaler le bit de colonne à droite de 1 bit.
9) Si l'indicateur de retenue = 1 retourner à l'étape 8.
10) Fin du programme.

Organigramme :



Programme source :

```
        LD      A,01001111B     ; MISE DU PORT A EN ENTREE
        OUT     (PIOCRA),A
        LD      A,00001111B     ; MISE DU PORT B EN SORTIE
        OUT     (PIOCRB),A
        LD      B,3             ; COMPTEUR = NOMBRE DE LIGNES
        LD      C,PIODRB        ; ACQUISITION DU NUMERO DE PORT DE SORTIE
        LD      D,3             ; ACQUISITION DU NOMBRE DE COLONNES
        LD      E,11111110B     ; 1re VALEUR DE SCRUTATION

        LD      H,00000111B     ; ACQUISITION DE LA VALEUR DU MASQUE
        LD      L,0FFH          ; NUMERO DE TOUCHE = 1
RLIGN : OUT     (C),E           ; SCRUTATION D'UNE LIGNE
        RLC     E               ; MISE A JOUR DE LA VALEUR DE SCRUTATION POUR LA
                                ; PROCHAINE LIGNE
        IN      A,(PIODRA)      ; ACQUISITION DE LA DONNEE DE LA COLONNE DU CLAVIER
        AND     H               : MASQUE DES BITS DES COLONNES
        CP      H               ; Y A-T-IL ENCORE DES COLONNES A LA MASSE ?
        JR      NZ,FCOL         ; OUI, RECHERCHE DE LAQUELLE
        LD      A,L             ; NON, MISE A JOUR DU NUMERO DE LA TOUCHE POUR LA
                                ; PROCHAINE LIGNE
        ADD     A,D
        LD      L,A
        DJNZ    RLIGN           ; EXAMEN LIGNE SUIVANTE
        INC     L               ; IDENTIFIER LE CAS DANS LEQUEL LA TOUCHE N'EST PAS
                                ; TROUVEE
        JR      FIN
RCOL :  INC     L               ; INCREMENTE LE NUMERO DE LA TOUCHE
        RRA                     ; EST-CE QUE CETTE COLONNE EST A LA MASSE ?
        JR      NC,RCOL         ; NON, EXAMEN COLONNE SUIVANTE
FIN :   HALT
```

Remarque : cf. NDT page 380

Programme objet : (voir page 417)

Chaque fois qu'une scrutation échoue, on doit additionner le nombre de colonnes au numéro de touche pour passer à la ligne suivante (essayer sur le clavier de la figure 12.22). Quel est le résultat de ce programme si aucune touche n'est enfoncée ? Remarquons l'instruction supplémentaire INC L qui fait la différence entre pas de touche appuyée et la dernière touche à avoir été appuyée. Quelle est la valeur finale dans l'accumulateur dans ces deux cas ? Notons que l'indicateur de zéro peut aussi être utilisé pour distinguer le cas où aucune touche n'est appuyée. Pouvez-vous expliquer comment ?
Une autre approche serait d'utiliser le PIO dans son mode contrôle ainsi chaque ligne pourrait être changée d'entrée en sortie. La procédure serait :

1) Mettre à la masse toutes les colonnes et sauvegarder les entrées lignes.
2) Mettre à la masse toutes les lignes et sauvegarder les entrées colonnes.
3) Utiliser les entrées des lignes et des colonnes ensemble, pour déterminer le numéro de la touche à partir d'une table.
Essayez d'écrire un programme pour implanter cette procédure. Ce programme peut être généralisé en mettant le nombre de colonnes, le nombre de lignes et la valeur du masque dans des paramètres étiquetés avec des pseudo-instructions (EQU).

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3E	LD	A,01001111B
0001	4F		
0002	D3	OUT	(PIOCRA),A
0003	PIOCRA		
0004	3E	LD	A,00001111B
0005	0F		
0006	D3	OUT	(PIOCRB),A
0007	PIOCRB		
0008	06	LD	B,3
0009	03		
000A	0E	LD	C,PIODRB
000B	PIODRB		
000C	16	LD	D,3
000D	03		
000E	1E	LD	E,11111110B
000F	FE		
0010	26	LD	H,00000111B
0011	07		
0012	2E	LD	L,0FFH
0013	FF		
0014	ED	RLIGN : OUT	(C),E
0015	59		
0016	CB	RLC	E
0017	03		
0018	DB	IN	A,(PIODRA)
0019	PIODRA		
001A	A4	AND	H
001B	BC	CP	H
001C	20	JR	NZ,RCOL
001D	08		
001E	7D	LD	A,L
001F	82	ADD	A,D
0020	6F	LD	L,A
0021	10	DJNZ	RLIGN
0022	F1		
0023	2C	INC	L
0024	18	JR	FIN
0025	04		
0026	2C	RCOL : INC	L
0027	1F	RRA	
0028	30	JR	NC,RCOL
0029	FC		
002A	76	FIN : HALT	

Un clavier encodé

But : acquisition d'une donnée, quand elle est disponible, à partir d'un clavier encodé qui génère une validation avec chaque transfert de donnée.
Un clavier encodé génère un code unique pour chaque touche. Il possède des circuits électroniques qui assurent les fonctions de scrutation et d'identification de l'exemple précédent. Il existe un compromis pour le choix entre un clavier encodé nécessitant un logiciel plus simple et celui d'un clavier non encodé dont le coût est moins important.

Les claviers encodés utilisent des matrices de diodes, des encodeurs TTL ou MOS. Les codes peuvent être ASCII, EBCDIC ou un code utilisateur. Des PROM sont souvent les circuits d'encodage.

ROLLOVER APPUIS MULTIPLES

Les circuits d'encodage peuvent réaliser d'autres fonctions que celle d'encoder la touche appuyée. Ils peuvent gérer les antirebonds et les appuis multiples (« rollover »), c'est-à-dire si plus d'une touche est appuyée à la fois. Il existe principalement deux types de claviers à « rollover » :

— Les claviers à deux touches simultanées (« two-key-rollover ») et les claviers à n touches simultanées (« n-key-rollover »), où n spécifie le nombre (maximal) de touches acceptées comme des entrées distinctes.

Le clavier encodé envoie aussi un signal de validation à chaque transfert de données. Le signal de validation indique qu'un nouvel appui vient de se produire. La figure 11.23 montre l'interface entre un clavier encodé et un microprocesseur Z80. Le front montant de la validation mémorise les données dans le port d'entrée. On connecte aussi le strobe au port B du PIO, ainsi l'UC peut savoir quand un front montant survient. Naturellement, le port B d'un PIO peut gérer les signaux d'état d'au plus 8 ports. Le logiciel devra alors déterminer quel port est actif à l'aide d'opération de décalage et de masquage.

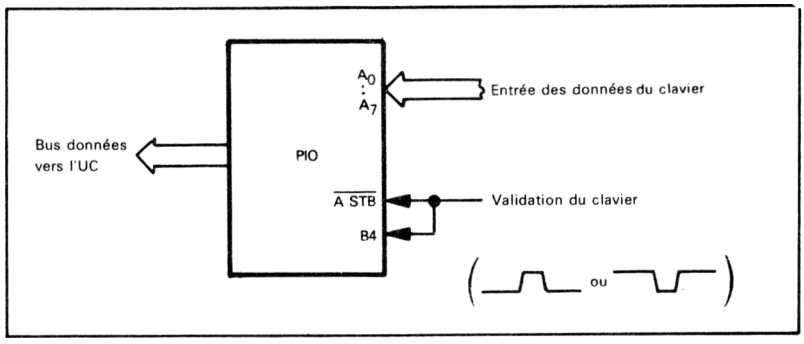


Fig. 11.23. — Interface d'E/S pour un clavier encodé.

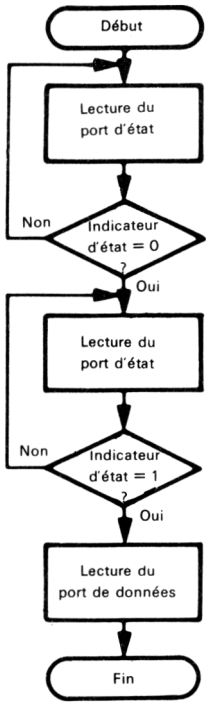
Dans le programme, nous avons supposé que le signal de validation est assez long pour pouvoir être géré par logiciel. Si ce n'est pas le cas, le signal devra être mémorisé et effacé (avec le signal RDY) quand le transfert d'entrée ou de sortie survient.

On doit faire attention à la polarité du signal de validation, car le PIO réagit toujours à un front montant. Un inverseur peut être nécessaire.

Tâche : saisie d'un clavier

But : attendre le front montant du signal de validation sur le port B d'un PIO et placer alors la donnée du port A dans l'accumulateur.

Organigramme :



L'électronique doit maintenir les lignes de contrôle à l'état un durant la réinitialisation (RESET) pour éviter des positionnements accidentels des indicateurs d'état.

Programme source :

```
        LD      A,01001111B     ; MISE DU PORT A EN ENTREE
        OUT     (PIOCRA),A
        LD      A,11001111B     ; MISE DU PORT B EN CONTROLE
        OUT     (PIOCRB),A
        LD      A,0FFH          ; TOUTES LES LIGNES DU PORT B EN ENTREES
        OUT     (PIOCRB),A
SRCHL:  IN      A,(PIODRB)      ; EXAMEN DE L'ETAT DU PORT
        BIT     STB,A           ; LE SIGNAL DE VALIDATION EST BAS ?
        JR      NZ,SRCHL        ; NON, ATTENDRE
SRCHH:  IN      A,(PIODRB)      ; NOUVEL EXAMEN DE L'ETAT DU PORT
        BIT     STB,A           ; EST-CE UN FRONT MONTANT ?
        JR      Z,SRCHH         ; NON, ATTENDRE QUE LE FRONT ARRIVE
        IN      A,(PIODRA)      ; OUI, LECTURE DES DONNEES
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	3E		LD	A,11001111B
0005	CF			
0006	D3		OUT	(PIOCRB),A
0007	PIOCRB			
0008	3E		LD	A,0FFH
0009	FF			
000A	D3		OUT	(PIOCRB),A
000B	PIOCRB			
000C	DB	SRCHL:	IN	A,(PIODRB)
000D	PIODRB			
000E	CB		BIT	STB,A
000F	STB			
0010	20		JR	NZ,SRCHL
0011	FA			
0012	DB	SRCHH:	IN	A,(PIODRB)
0013	PIODRB			
0014	CB		BIT	STB,A
0015	STB			
0016	28		JR	Z,SRCHH
0017	FA			
0018	DB		IN	A,(PIODRA)
0019	PIODRA			
001A	76		HALT	

Si l'UC répète ce programme, il attendra un nouveau front montant sur la ligne d'état ASTB avant de lire un nouveau caractère. Un niveau haut continu sur cette ligne sera ignoré. STB dépend de la position du bit du port B utilisé. La figure 11.23 montre que le bit 4 est utilisé, mais les bits 0, 6 et 7 sont comme d'habitude les plus faciles à examiner. Essayez de ré-écrire le programme en utilisant les positions les plus accessibles. Le second octet de l'instruction BIT dépend de la valeur de STB, mais il n'est pas égal à cette valeur. Par exemple, le second octet est $4F_{16}$ si STB = $1,57_{16}$ si STB = 2, etc.

Convertisseur digital-analogique

But : envoyer un octet à un convertisseur digital analogique, lequel possède un signal de mémorisation actif bas.

Les convertisseurs digitaux-analogiques produisent les signaux continus exigés par les solénoïdes, relais, activateurs, et autres appareils électriques ou mécaniques. Les convertisseurs usuels comprennent des commutateurs et des réseaux de résistances appairées. L'utilisateur doit généralement fournir une tension de référence et quelques autres circuits analogiques ou digitaux, néanmoins des unités complètes sont disponibles à faible coût. La figure 11.24 décrit un convertisseur 8 bits (Signetics NE5018), lequel contient une mémoire d'entrée parallèle 8 bits. Un niveau bas sur le signal d'entrée LE (autorisation de mémorisation = « latch enable ») provoque l'entrée des données dans la mémoire où elles resteront après que le signal redevient haut.

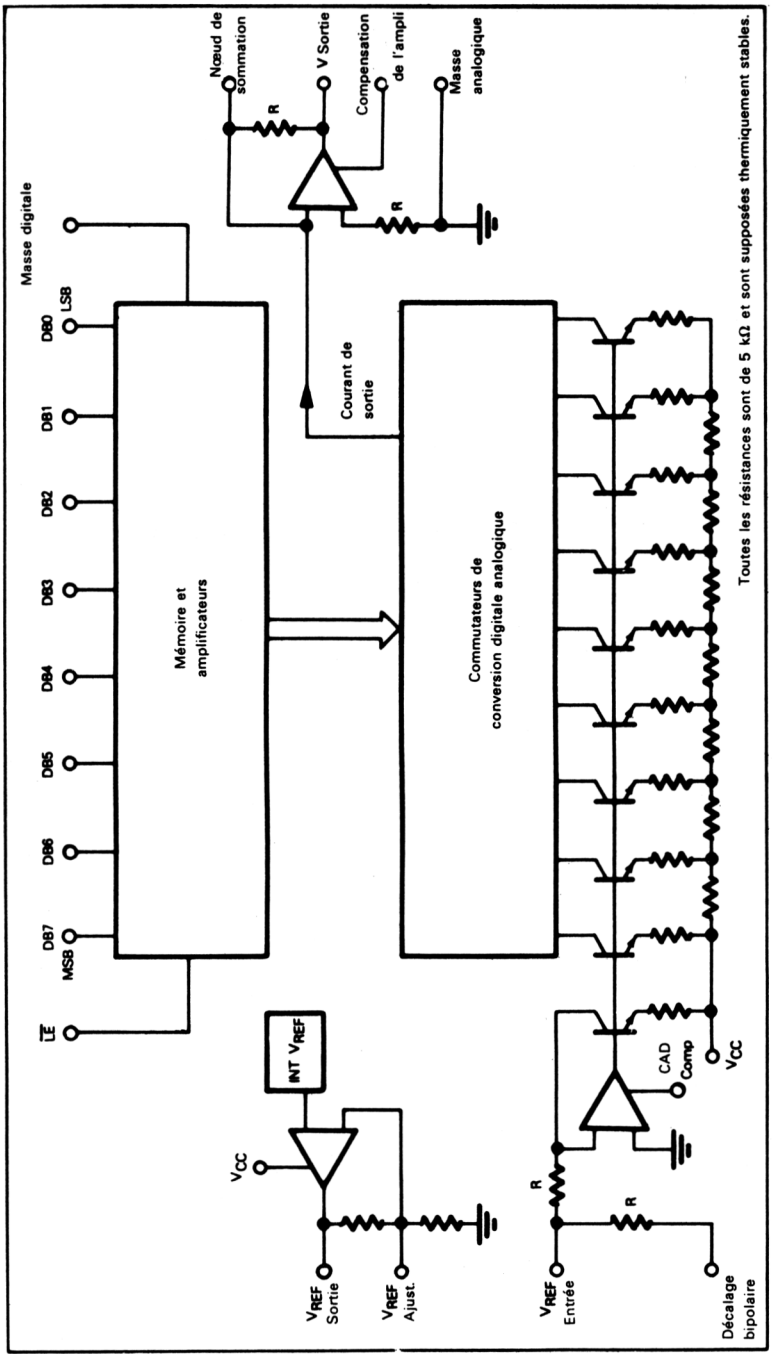


Fig. 11.24. — Convertisseur D/A Signetics NE5018.

La figure 11.25 illustre l'interface de ce circuit avec un microprocesseur Z80. Ici le port A du PIO est utilisé pour générer le signal d'autorisation de mémorisation (LE). La ligne RDY du PIO peut être reliée à la ligne (STB) pour former une impulsion d'un cycle d'horloge. Cependant un seul cycle d'horloge peut être trop court car le NE5018 demande une impulsion de 400 ns. De plus, la polarité est l'inverse de celle exigée par le NE5018.

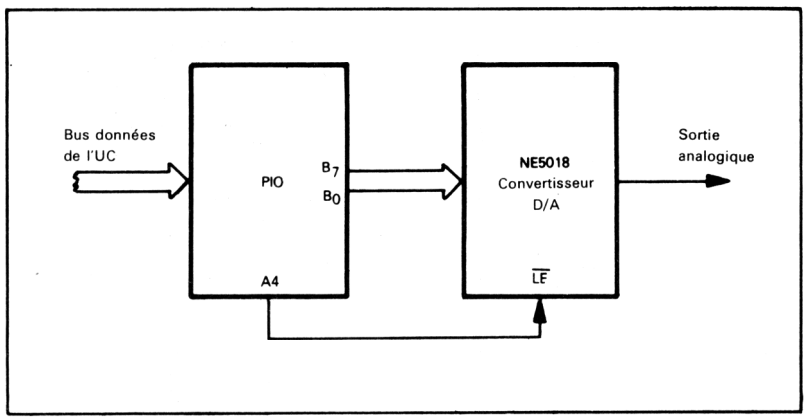


Fig. 11.25. — Interface pour un convertisseur digital/analogique.

Remarquons que le PIO mémorise les données de sorties. La donnée reste donc stable durant et après la conversion. Le convertisseur demande seulement quelques microsecondes pour produire une sortie analogique. Aussi la mémoire du convertisseur peut ne pas être utilisée si le port n'est pas utilisé pour d'autres applications.

Dans les applications où huit bits de résolution ne sont pas suffisants, on peut utiliser des convertisseurs à 10 ou 16 bits. Des circuits logiques supplémentaires seront nécessaires pour transférer tous les bits de données ; certains convertisseurs contiennent une bonne partie de cette logique.

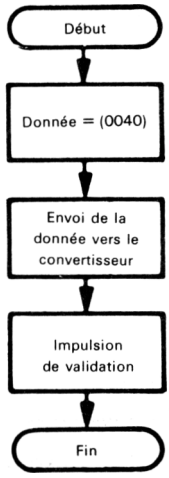
Le PIO sert ici à la fois comme port parallèle de données et comme port de contrôle. Evidemment si le port A est utilisé par le contrôle, il peut gérer jusqu'à huit bits.

Tâche : sortir vers un convertisseur

But : envoyer le contenu de la mémoire 40 vers le convertisseur.

Organigramme : (voir page 423)

Organigramme



Programme source :

```
LD      A,11001111B     ; MISE DU PORT A EN CONTROLE
OUT     (PIOCRA),A
SUB     A               ; TOUT LE PORT A EN SORTIE
OUT     (PIOCRA),A
LD      A,00001111B     ; MISE DU PORT B EN SORTIE
OUT     (PIOCRB),A
LD      A,(40H)         ; LECTURE DE LA DONNEE
OUT     (PIODRB),A      ; ENVOI DE LA DONNEE VERS LE CONVERTISSEUR
IN      A,(PIODRA)      ; ACQUISITION DE L'ANCIEN CONTROLE
RES     4,A             ; METTRE A ZERO LA DEMANDE DE MEMORISATION
OUT     (PIODRA),A
SET     4,A             ; METTRE A UN LA DEMANDE DE MEMORISATION
OUT     (PIODRA),A
HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3E	LD	A,11001111B
0001	CF		
0002	D3	OUT	(PIOCRA),A
0003	PIOCRA		
0004	97	SUB	A
0005	D3	OUT	(PIOCRA),A
0006	PIOCRA		
0007	3E	LD	A,00001111B
0008	0F		
0009	D3	OUT	(PIOCRB),A
000A	PIOCRB		
000B	3A	LD	A,(40H)
000C	40		
000D	00		
000E	D3	OUT	(PIODRB),A
000F	PIODRB		
0010	DB	IN	A,(PIODRA)
0011	PIODRA		
0012	CB	RES	4,A
0013	A7		
0014	D3	OUT	(PIODRA),A
0015	PIODRA		
0016	CB	SET	4,A
0017	E7		
0018	D3	OUT	(PIODRA),A
0019	PIODRA		
001A	76	HALT	

Le bit particulier qui doit être activé et désactivé, dépend bien sûr de comment la validation de mémorisation est câblée au port de contrôle. Le bit 0 est souvent le plus adapté pour ces contrôles, car s'il a été initialement mis à zéro, il peut être activé avec une instruction INC, et désactivé avec une instruction DEC.
On peut utiliser le signal de validation, créé à partir de BACK si la validation de mémorisation est active haute (et si le signal de validation est assez long quand le signal BACK est relié à BSTB). Le programme devient :

```
LD      A,00001111B    ; MISE DU PORT B EN SORTIE
OUT     (PIOCRB),A
LD      A,(40H)        ; LECTURE DE LA DONNEE
OUT     (PIODRB),A     ; ENVOI DE LA DONNEE AU CONVERTISSEUR ET MEMORISATION
HALT
```

Un inverseur pourrait générer un signal actif bas. Notons le petit nombre d'instructions nécessaires.

Convertisseur analogique-digital

But : lire un octet issu d'un convertisseur analogique-digital, qui demande une impulsion de début de conversion et qui possède une ligne indiquant la fin de la conversion et la disponibilité de la donnée.

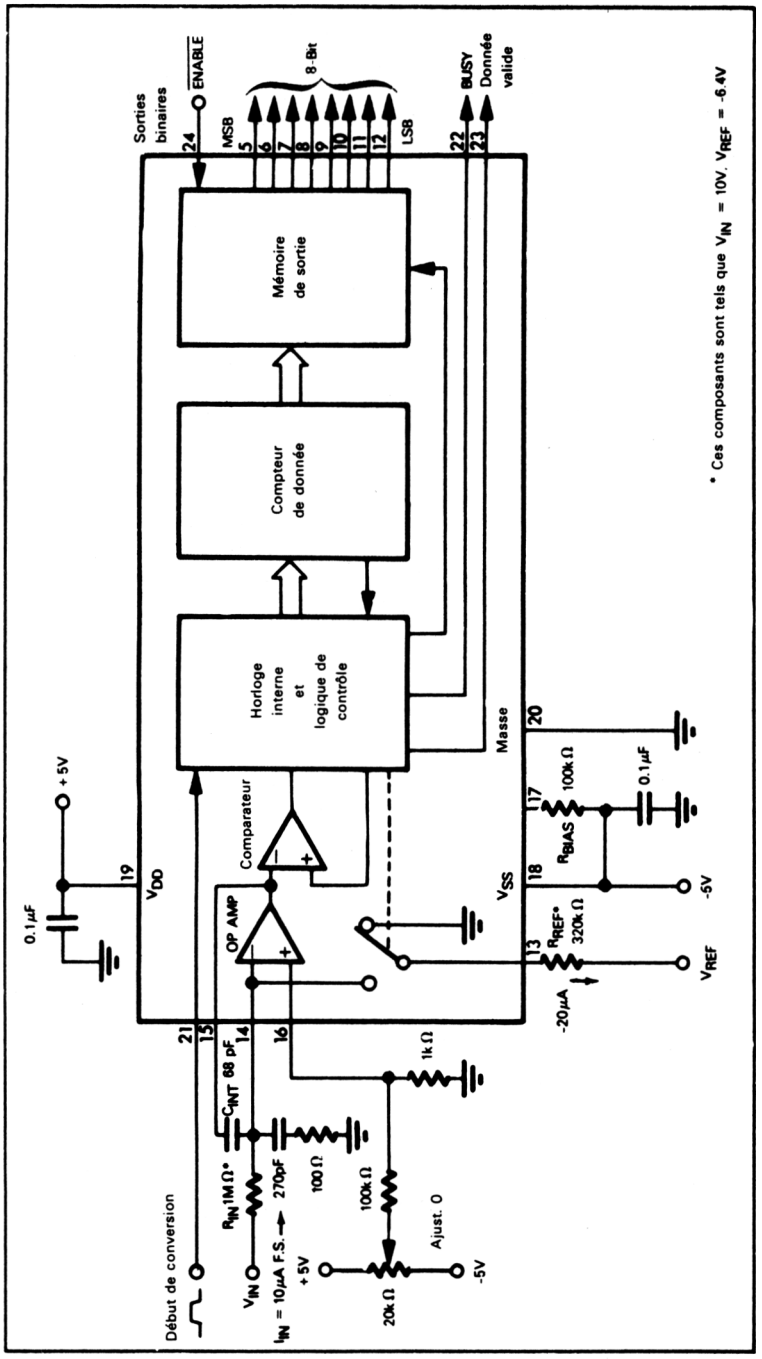


Fig. 11.26. — Convertisseur A/D Teledyne 8703.

Les convertisseurs analogiques-digitaux gèrent des signaux continus produits par différents types de palpeurs et de transducteurs [8]. Le convertisseur produit une entrée digitale qui est transmise au calculateur.
Un type de convertisseur analogique-digital est le convertisseur par approximations successives, qui réalise une comparaison directe de 1 bit durant chaque cycle d'horloge. De tels convertisseurs sont rapides mais ont peu d'immunité au bruit. Les convertisseurs double-rampe constituent un autre type de convertisseur analogique-digital. Ces circuits sont plus lents, mais possèdent une meilleure immunité au bruit. D'autres techniques peuvent être utilisées.
Les convertisseurs analogiques utilisent généralement des circuits externes analogiques et digitaux, cependant des circuits complets sont disponibles à faible coût.
La figure 11.26 décrit le convertisseur A/D huit bits 8703 de Teledyne semiconductor. Le circuit contient une mémoire de sortie et des amplificateurs trois états. Une impulsion sur la ligne de début de conversion démarre la conversion de l'entrée analogique, après deux millisecondes environ. Le résultat ira sur la mémoire de sortie et le signal donnée valide indiquera cela en effectuant une transition basse puis haute. La donnée est lue à partir de la mémoire en appliquant un « 0 » à l'entrée « validation », enable).
La figure 11.27 décrit l'interface entre un microprocesseur Z80 et le convertisseur [9] A/D, 8703. Le port B est utilisé pour générer une impulsion de début de conversion d'une durée suffisante (cette impulsion est active haute). Le signal de donnée valide est relié à ASTB ; ainsi, quand ce signal devient bas puis haut, la donnée convertie sera mémorisée dans le port A. Le signal donnée valide est aussi relié à un bit du port B, de telle sorte que l'UC peut déterminer sa valeur. Le front important de ce signal est la transition, basse-haute, qui indique la fin de la conversion. Comme dans le cas du clavier encodé, des circuits supplémentaires seront nécessaires si l'impulsion de donnée valide est trop courte pour être gérée par le logiciel. Remarquons que l'on utilise le port B à la fois comme port d'état et de contrôle.

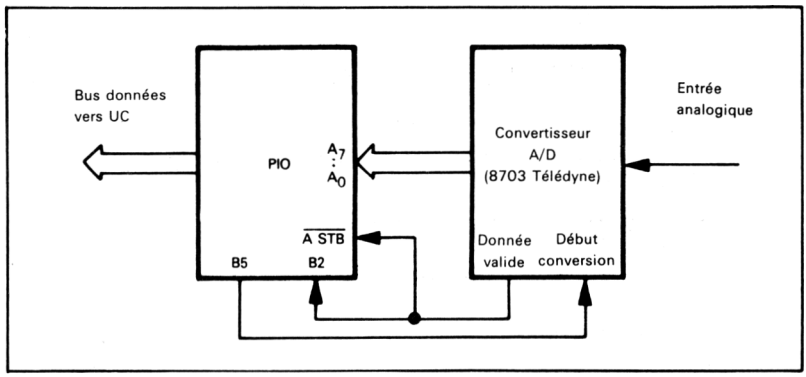
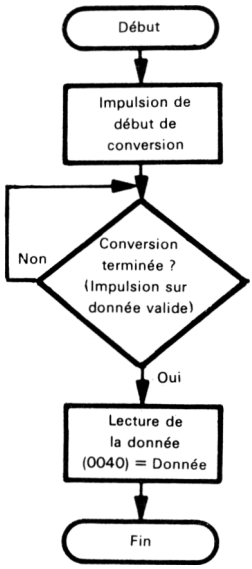


Fig. 11.27. — Interface pour un convertisseur analogique-digital.

Tâche : entrée à partir d'un convertisseur

But : démarrer une conversion, attendre que le signal donnée valide effectue une transition bas-haut, puis lecture de la donnée et stockage dans la mémoire 40.

Organigramme :



Remarquons que le PIO sert ici comme port parallèle, port d'état et comme port de contrôle.

Programme source :

```
        LD      A,01001111B     ; MISE DU PORT A EN ENTREE
        OUT     (PIOCRA),A
        LD      A,11001111B     ; MISE DU PORT B EN CONTROLE
        OUT     (PIOCRB),A
        LD      A,00001111B     ; B4-7 EN SORTIE, B0-3 EN ENTREE
        OUT     (PIOCRB),A
        LD      A,00100000B     ; ENVOI DU DEBUT DE CONVERSION HAUT
        OUT     (PIODRB),A
        SUB     A               ; ENVOI DU DEBUT DE CONVERSION BAS
        OUT     (PIODRB),A
ATTBAS: IN      A,(PIODRB)      ; EST-CE QUE DONNEE VALIDE BAS ?
        BIT     2,A
        JR      NZ,ATTBAS       ; NON, ATTENTE
ATHAUT: IN      A,(PIODRB)      ; EST-CE QUE LA DONNEE EST DISPONIBLE ?
        BIT     2,A
        JR      Z,ATHAUT        ; NON, ATTENTE
        IN      A,(PIODRA)      ; OUI, LECTURE DE LA DONNEE A PARTIR DU CONVERTISSEUR
        LD      (40H),A         ; SAUVEGARDE DE LA DONNEE DU CONVERTISSEUR
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	3E		LD	A,11001111B
0005	CF			
0006	D3		OUT	(PIOCRB),A
0007	PIOCRB			
0008	3E		LD	A,00001111B
0009	0F			
000A	D3		OUT	(PIOCRB),A
000B	PIOCRB			
000C	3E		LD	A,00100000B
000D	20			
000E	D3		OUT	(PIODRB),A
000F	PIODRB			
0010	97		SUB	A
0011	D3		OUT	(PIODRB),A
0012	PIODRB			
0013	DB	ATTBAS :	IN	A,(PIODRB)
0014	PIODRB			
0015	CB		BIT	2,A
0016	57			
0017	20		JR	NZ,ATTBAS
0018	FA			
0019	DB	ATHAUT :	IN	A,(PIODRB)
001A	PIODRB			
001B	CB		BIT	2,A
001C	57			
001D	28		JR	Z,ATHAUT
001E	FA			
001F	DB		IN	A,(PIODRA)
0020	PIODRA			
0021	32		LD	(40H),A
0022	40			
0023	00			
0024	76		HALT	

Une approche pour configurer le PIO est d'utiliser l'instruction répétitive de sortie par blocs OTIR, et une table contenant les octets à envoyer au registre de contrôle. Un programme typique serait :

```
LD      B,LENG      ; COMPTEUR = NOMBRE D'OCTETS DE CONTROLE
LD      C,PIOCR     ; ACQUISITION DU NUMERO DE PORT
LD      HL,CTLTAB   ; ADRESSE DE DEPART DE LA TABLE DE CONTROLE DU PIO
OTIR                ; CONFIGURATION DU PIO
```

En fait, une autre table (ou la pile) peut-être utilisée pour contenir le nombre d'octets de contrôle et le numéro du port pour chaque PIO.

Un télé-imprimeur (TTY)

INTERFACE D'UN TTY

But : transférer des données vers ou à partir d'un télé-imprimeur classique sériel à 10 caratères par seconde.

Les télé-imprimeurs usuels transfèrent des données en modes asynchrone et sériel. La procédure est la suivante :

TTY CLASSIQUE

1) La ligne est normalement à l'état 1.

FORMAT DU CARACTÈRE

2) Un bit de départ (zéro) précède chaque caractère.

3) Le caractère est généralement un caractère ASCII, 7 bits, avec le bit le moins significatif transmis le premier.

4) Le bit le plus significatif est un bit de parité, laquelle peut être paire, impaire ou fixée à un ou à zéro.

5) Deux bits de stop (un logique) suivent chaque caractère.

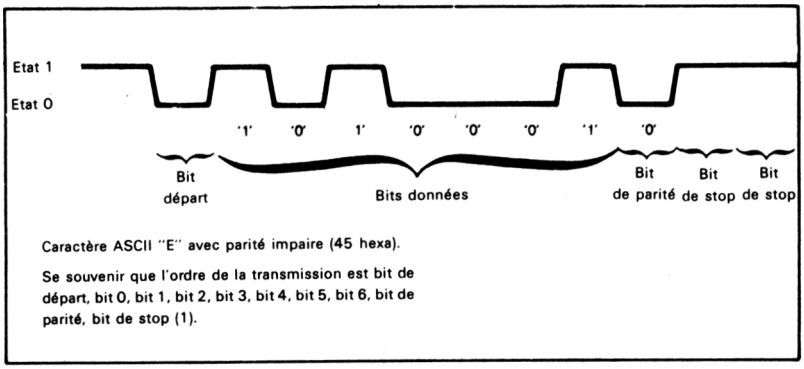


Fig. 11.28. — Format de donnée d'un télé-imprimeur.

La figure 11.28 montre le format. Remarquons que chaque caractère demande la transmission de onze bits, qui contiennent seulement sept bits d'informations. Comme la vitesse de transmission est de dix caractères par seconde, le taux de transmission sera de $10 \times 11 = 110$ bauds (bit par seconde). Chaque bit a donc une durée de 1/110 seconde ou 9,1 millisecondes. Cette durée est une valeur moyenne, le télé-imprimeur ne la maintenant pas très précisément.

Pour qu'un télé-imprimeur communique avec un calculateur, les opérations suivantes sont nécessaires :

MODE RÉCEPTION DU TTY

Réception (organigramme fig. 11.29)

Etape 1 : attente d'un bit de départ (zéro logique) sur la ligne des données.

Etape 2 : centrer la réception en attendant une demi période d'un bit (4,55 ms).

Etape 3 : lecture des bits de données, en attendant la durée d'un bit entre chaque bit. Assembler les bits dans un octet en décalant le bit dans la retenue puis en décalant circu-

lairement la donnée avec la retenue. Se rappeler que le bit le moins significatif est reçu en premier.
Etape 4 : générer la parité à recevoir et la comparer avec celle reçue. S'il n'y a pas égalité, indiquer erreur de parité.
Etape 5 : lire les bits de stop (attente d'une durée d'un bit entre les entrées). S'ils ne sont pas correct (si les deux bits stop ne sont pas à un), indiquer une erreur de trame (« framing error »).

Tâche 1 : lecture de donnée

But : lire une donnée venant d'un télé-imprimeur à l'aide du bit 7 d'un PIO et mettre la donnée dans la mémoire 0060. Pour l'organigramme, voir la figure 11.29.

Programme source :

On suppose que le port série est le bit 7 du PIO et que la vérification de la trame et de la parité sont inutiles.

```
        LD      A,01001111B     ; MISE EN ENTREE DU PORT A
        OUT     (PIOCRA),A
ATDEB : IN      A,(PIODRA)      ; LECTURE DE LA LIGNE ZERO
        RLA                     ; EST-CE UN BIT DE DEPART ?
        JR      C,ATDEB         ; NON, ATTENDRE
        CALL    DEMI            ; OUI, TEMPORISATION D'UNE DUREE D'UN DEMI BIT
        LD      D,10000000B     ; COMPTEUR AVEC UN « UN » DANS LE BIT LE PLUS
                                ; SIGNIFICATIF
RCVB :  CALL    PLEIN           ; ATTENTE D'UNE DUREE D'UN BIT
        IN      A,(PIODRA)      ; LECTURE DE LA LIGNE SERIE
        RLA                     ; TRANSFERT DU BIT VERS LA RETENUE
        RR      D               ; TRANSFERT DU BIT DANS L'OCTET D'ASSEMBLAGE
        JR      NC,RCVB         ; CONTINUER, SI LE BIT DU COMPTEUR N'EST PAS
                                ; DANS LA RETENUE
        LD      A,D
        LD      (60H),A
        HALT
```

Programme de temporisation :

```
        ORG     30H
DEMI :  PUSH    DE              ; SAUVEGARDE DES ANCIENS REGISTRES
        LD      D,8             ; COMPTEUR D'UNE DUREE D'UN DEMI BIT
        JR      DLY16
PLEIN : PUSH    DE              ; SAUVEGARDE DES ANCIENS REGISTRES
        LD      D,16            ; COMPTEUR D'UNE DUREE D'UN BIT
DLY16:  LD      E,8DH           ; TEMPORISATION DE 1/16 DE BIT
DLY1:   DEC     E
        JR      NZ,DLY1
        DEC     D
        JR      NZ,DLY16
        POP     DE              ; RESTAURATION DES ANCIENS REGISTRES
        RET
```

Se souvenir que le bit 0 de la donnée est reçu le premier.

Remarque : cf. NDT page 380

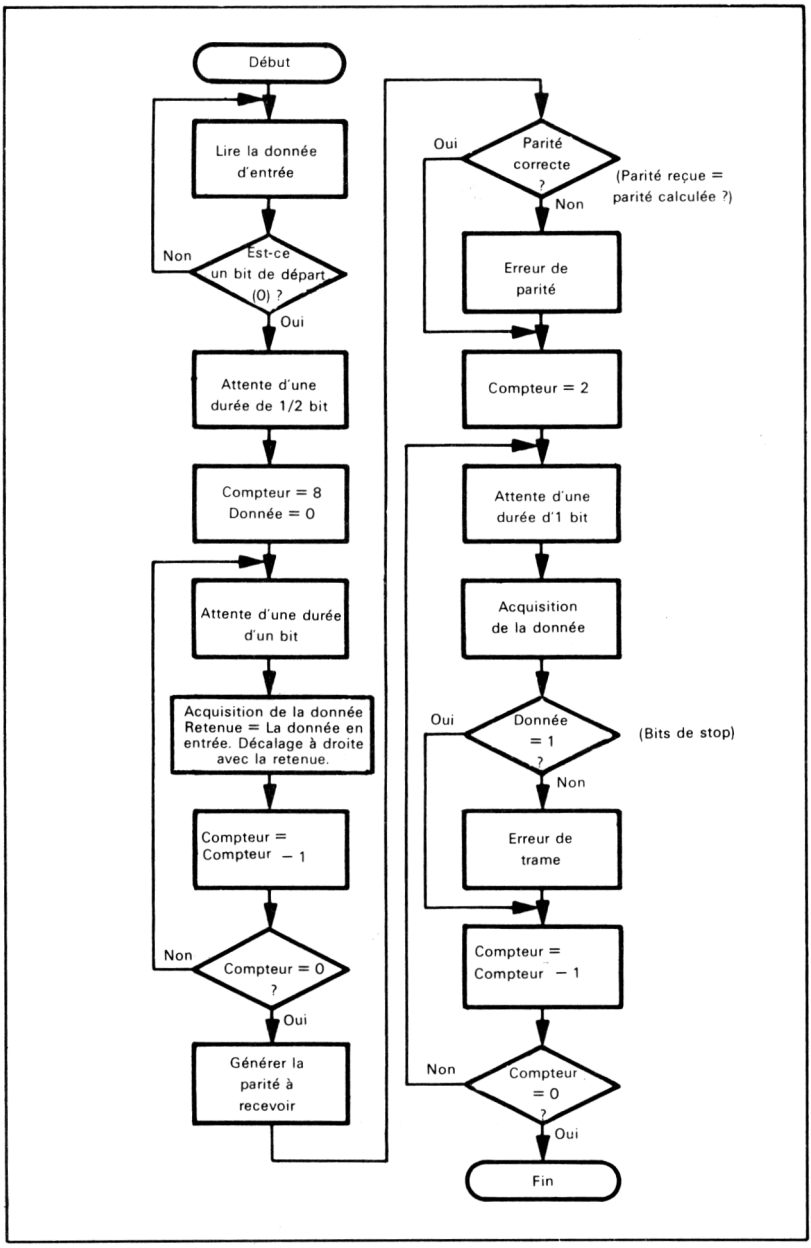


Fig. 11.29. — Organigramme d'une réception.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
0000	3E		LD	A,01001111B
0001	4F			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	DB	ATDEB :	IN	A,(PIODRA)
0005	PIODRA			
0006	17		RLA	
0007	38		JR	C,WTSTB
0008	FB			
0009	CD		CALL	DHALF
000A	30			
000B	00			
000C	16		LD	D,10000000B
000D	80			
000E	CD	RCVB :	CALL	PLEIN
000F	35			
0010	00			
0011	DB		IN	A,(PIODRA)
0012	PIODRA			
0013	17		RLA	
0014	CB		RR	D
0015	1A			
0016	30		JR	NC,RCVB
0017	F6			
0018	7A		LD	A,D
0019	32		LD	(60H),A
001A	60			
001B	00			
001C	76		HALT	
0030	D5	DEMI :	PUSH	DE
0031	16		LD	D,8
0032	08			
0033	18		JR	DLY16
0034	03			
0035	D5	PLEIN :	PUSH	DE
0036	16		LD	D,16
0037	10			
0038	1E	DLY16:	LD	E,8DH
0039	8D			
003A	1D	DLY1:	DEC	E
003B	20		JR	NZ,DLY1
003C	FD			
003D	15		DEC	D
003E	20		JR	NZ,DLY16
003F	F8			
0040	D1		POP	DE
0041	C9		RET	

Ce programme suppose que la pile peut être utilisée pour des appels de sous-programme, c'est-à-dire que le moniteur doit initialiser le pointeur de pile. Autrement on doit initialiser le pointeur de pile comme décrit dans le chapitre 10.
Les constantes pour les sous-programmes de temporisation ont été calculées comme nous l'avons expliqué précédemment dans ce chapitre. Vous pouvez essayer de les trouver par

vous-même. Les temporisations n'ont pas à être très précises car la réception est centrée, les messages sont courts, le taux de transmission lent, et le télé-imprimeur n'est pas lui-même d'une grande précision.
Comment généraliseriez-vous ce programme pour vérifier les deux bits de stop ? Ils doivent être à un sinon une erreur de trame doit être générée. Vous pouvez permettre à ce programme de vérifier la parité impaire en remplaçant l'instruction LD A,D par la séquence :

```
SUB     A
AND     D           ; EST-CE UNE PARITE IMPAIRE
JP      PE,PRERR    ; NON, UNE ERREUR DE PARITE EST SURVENUE
```

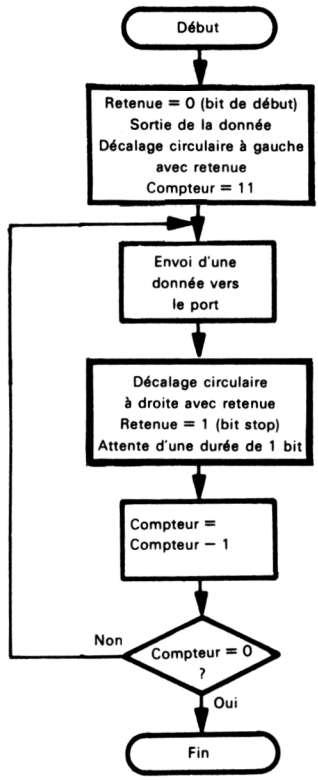


Fig. 11.30. — Organigramme d'une transmission.

Tâche 2 : écriture d'une donnée

But : transmettre une donnée vers un télé-imprimeur à l'aide du bit 0 du registre de donnée du PIO. La donnée est dans la mémoire 0060.

MODE TRANSMISSION VERS TTY

Transmission (organigramme figure 11.30)
Etape 1 : transmission d'un bit de début (c'est-à-dire un zéro).
Etape 2 : transmission des sept bits de donnée, en commençant par le bit le moins significatif.
Etape 3 : génération et transmission du bit de parité.
Etape 4 : transmission de deux bits de stop (c'est-à-dire des un logique).
Le programme de transmission doit attendre la durée d'un bit entre chaque opération.

Programme source : (on suppose que la parité ne doit pas être générée)

```
;
; MISE DU PIO EN MODE SORTIE
;
        LD      A,00001111B     ; MISE DU PORT B EN SORTIE
        OUT     (PIOCRB),A
;
; ACQUISITION DE LA DONNEE ET MISE A ZERO DU BIT DE DEBUT
;
        LD      A,(60H)         ; ACQUISITION DE LA DONNEE
        ADD     A,A             ; DECALAGE A GAUCHE ET FORMATION DU BIT DE DEBUT
        LD      B,11            ; COMPTEUR = 11 BITS
;
; TRANSMISSION D'UN BIT ET MISE A JOUR DE LA DONNEE
;
TBIT:   OUT     (PIODRB),A      ; TRANSMISSION D'UN BIT
        RRA                     ; MISE A JOUR POUR LE PROCHAIN BIT
        SCF                     ; FORMATION D'UN BIT DE STOP (UN)
;
; TEMPORISATION DE 9,1 MS ET COMPTAGE DES BITS
;
        CALL    PLEIN           ; TEMPORISATION 9,1 MS
        DJNZ    TBIT            ; DECOMPTAGE DE 11 BITS
        HALT
```

Le sous-programme PLEIN est le même qu'auparavant. Se rappeler que le bit 0 doit être transféré en premier.
L'instruction ADD A,A efface le bit le moins significatif ainsi il peut être utilisé en tant que bit de début. Le bit le plus significatif est sauvegardé dans la retenue. Dans les applications réelles, le programme de démarrage doit mettre un 1 logique, sur la ligne du téléimprimeur, après la configuration du PIO, car cette ligne est normalement dans l'état 1 (« mark state »).
Chaque caractère comprend 11 bits, commençant par un zéro, et se terminant par deux bits de stop (à 1).
Ce programme peut facilement être modifié pour générer des caractères de 7 bits avec une parité impaire dans le bit le plus significatif. Le programme de génération de la parité est (à insérer après l'instruction LD A,(60H)) :

```
        AND     A               ; EST-CE QUE LA PARITE EST IMPAIRE
        JP      PO,STBIT        ; OUI, PAS DE PROBLEME
        SET     7,A             ; NON, METTRE LA PARITE IMPAIRE EN POSITIONNANT LE
                                ; BIT LE PLUS SIGNIFICATIF
STBIT:  ADD     A,A             ; DECALAGE A GAUCHE ET FORMATION DU BIT DE DEBUT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)		Instruction (Mnémonique)	
0000	3E		LD	A,00001111B
0001	0F			
0002	D3		OUT	(PIOCRB),A
0003	PIOCRB			
0004	3A		LD	A,(60H)
0005	60			
0006	00			
0007	87		ADD	A,A
0008	06		LD	B,11
0009	0B			
000A	D3	TBIT:	OUT	(PIODRB),A
000B	PIODRB			
000C	1F		RRA	
000D	37		SCF	
000E	CD		CALL	PLEIN
000F	35			
0010	00			
0011	10		DJNZ	TBIT
0012	F7			
0013	76		HALT	

Remarque : cf. NDT page 380

UART Comment généreriez-vous une parité paire ?
Ces procédures sont suffisamment complexes et répandues pour mériter un circuit spécial LSI : le UART ou Universal Asynchronous Receiver/Transmitter (Transmetteur-récepteur asynchrone universel) [10]. L'UART accomplira la réception et génèrera les données en parallèle ainsi qu'un signal données prêtes. Il acceptera aussi les données en parallèle et accomplira la transmission et génèrera un signal périphérique prêt quand il pourra gérer d'autres données. Les UART ont d'autres possibilités telles que :

1) Capacité de gérer des longueurs variables de bits (généralement de 5 à 8), des options de parité, et un nombre de bit de stop variable (1, 1-1/2 et 2).
2) Indicateurs d'erreurs de trame, de parité et d'« overrun » (impossibilité de lire un caractère avant que le suivant n'arrive).
3) Compatibilité RS232 [11]* — c'est-à-dire le signal de sortie DPE (« RTS ») (demande pour émettre) qui indique la présence de données à l'équipement de communication, et un signal d'entrée CTS qui indique en réponse à un RTS l'accessibilité à l'équipement de communication. Il peut y avoir d'autres signaux de RS232, tels que signal de qualité de réception, poste de données prêt, terminal de donnée prêt.
4) Sorties trois états et contrôle compatible avec un microprocesseur.
5) Options d'horloges qui permettent à l'UART d'échantillonner les données d'entrée plusieurs fois dans l'intention de détecter des faux bits de début et d'autres erreurs.
6) Gestion et contrôle des interruptions.

*NDT : en France, cette norme correspond à un avis CCITT V24 où CTS et RTS sont remplacés respectivement par les signaux PAE (prêt à émettre) et DPE (demande pour émettre).

Un UART se comporte comme quatre ports parallèles, un port d'entrée de données, un port de sortie de données, un port d'état en entrée, et un port de contrôle en sortie. Les bits d'état comprennent aussi bien les indicateurs d'erreurs que les indicateurs de disponibilité (ready). Les bits de contrôle sélectionnent les différentes options. Les UART sont peu coûteux et faciles d'utilisation.

Le circuit d'entrée-sortie série du Z80 (SIO)*

Le circuit d'entrée-sortie série du Z80 ou SIO (figure 11.31) est un circuit complet de contrôle de communication spécialement conçu pour des micro-ordinateurs basés sur un Z80. Il peut exécuter de multiples fonctions de communication, mais nous discuterons ici seulement de son utilisation en tant que simple UART [12].

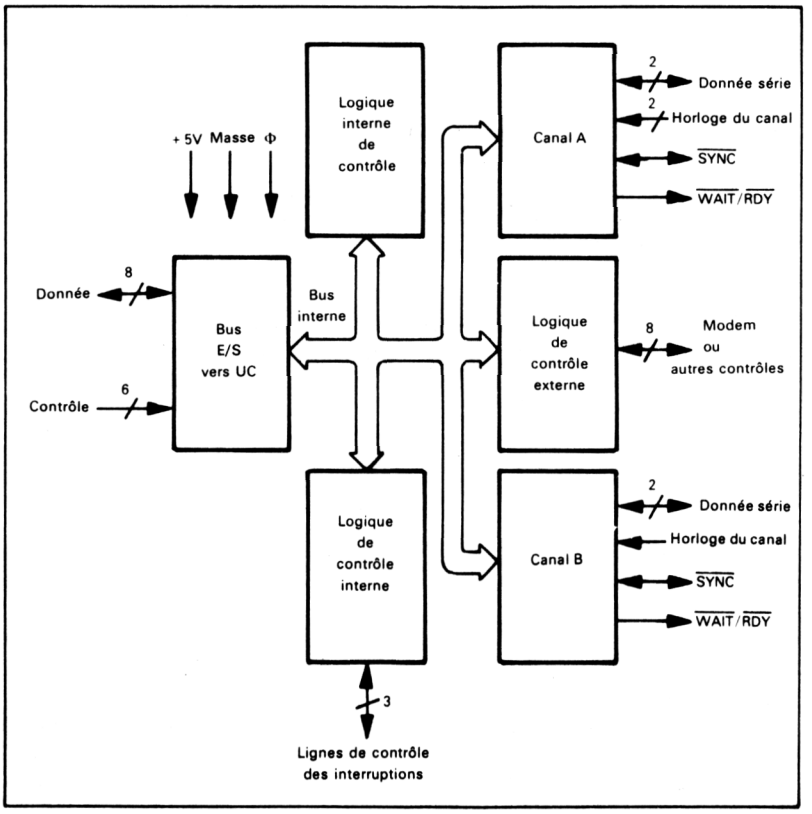


Fig. 11.31. — Diagramme fonctionnel du SIO du Z80.

NDT : ZILOG a également sorti plus récemment un circuit de la famille Z80 spécialisé uniquement en communication asynchrones, par opposition au SIO qui gère également les protocoles synchrones. Ce circuit, le « DART » (pour Dual Asynchronous Receiver-Tramsmitter), compte également 2 canaux indépendants, comme le SIO.

FULL-DUPLEX Le SIO a deux canaux (A et B) qui peuvent à la fois recevoir et transmettre des données en série (fig. 11.32). Les canaux qui peuvent à la fois recevoir et transmettre simultanément sont appelés bidirectionnels (full-duplex), les autres types comprennent les canaux à l'alternat (half-duplex) (capables de recevoir ou de transmettre mais pas simultanément), les récepteurs seuls et les émetteurs seuls.

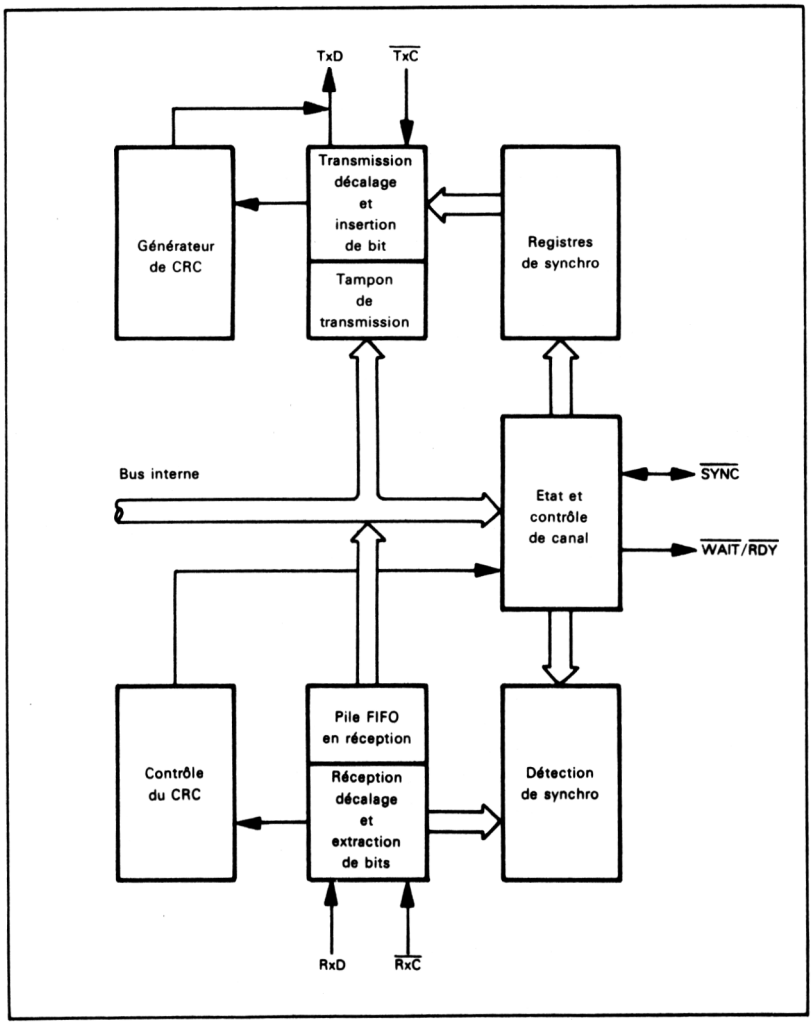


Fig. 11.32. — Diagramme fonctionnel d'un canal du SIO.

ADRESSAGE D'UN SIO

Un SIO occupe quatre adresses de port en entrée et quatre adresses de port en sortie. Le signal B/A (sélection du canal B ou A) et le signal C/D (sélection du contrôle ou de la donnée) choisissent l'un des quatre ports comme le décrit le tableau 11.7. Le plus souvent, les concepteurs relient le bit d'adresse A_0 à 1' entrée B/A et le bit d'adresse A_1 à l'entrée C/D. Le SIO occupe alors quatre adresses consécutives comme décrit dans la dernière colonne du tableau 11.7.

Tableau 11.7. — Adresses du SIO

Sélection donnée ou contrôle	Sélection du canal A ou B	Registre adressé	Adresse du port Début en SIOADD
0	0	Registre donnée A	SIOADD
0	1	Registre donnée B	SIOADD + 1
1	0	Contrôle A	SIOADD + 2
1	1	Contrôle B	SIOADD + 3
Les adresses des ports supposent que $C/\overline{D}$ est relié à A_1 et $B/\overline{A}$ à A_0.			

ADRESSAGE DES REGISTRES DE LECTURE ET D'ÉCRITURE DU SIO

Comme avec le PIO, les SIO ont plus de registre de contrôle que d'adresses. En fait, chaque SIO a huit registres pour chaque canal pour le contrôle et trois registres d'états. La figure 11.33 contient les diagrammes de chaque registre de contrôle et d'écriture, la figure 11.34 contient les diagrammes de chaque registre d'état ou de lecture. Deux transferts sont nécessaires pour lire ou écrire n'importe lequel des registres, exception faite pour le registre d'écriture 0.

Le premier transfert (écrit dans le registre 0) contient 3 bits qui dirigent le prochain transfert vers ou à partir du prochain registre sélectionné.

Remarquons, dans la figure 11.33, que ces trois bits occupent les trois positions de bit les moins significatives et que des zéros dans les autres positions indiquent que l'octet n'a qu'une fonction d'adressage.

CARACTÉRISTIQUES PARTICULIÈRES DU SIO

Remarquons les possibilités intéressantes du SIO :

1) Les instructions d'entrée ou de sortie, adressent des registres physiquement distincts. Il n'est pas possible de lire les registres de contrôle ou d'écrire les registres d'état.

2) Tous les registres d'un canal partagent la même adresse du port. Ainsi, deux octets sont nécessaires pour modifier le contenu de tous les registres sauf le registre 0.

RÉ-INITIALISATION DU SIO

3) Le signal RESET (initialisation) initialise le SIO en écriture du registre 0. Il désactive aussi la réception et la transmission, les signaux de contrôle et les interruptions. Nous discuterons dans le chapitre 12, le système d'interruption du SIO.

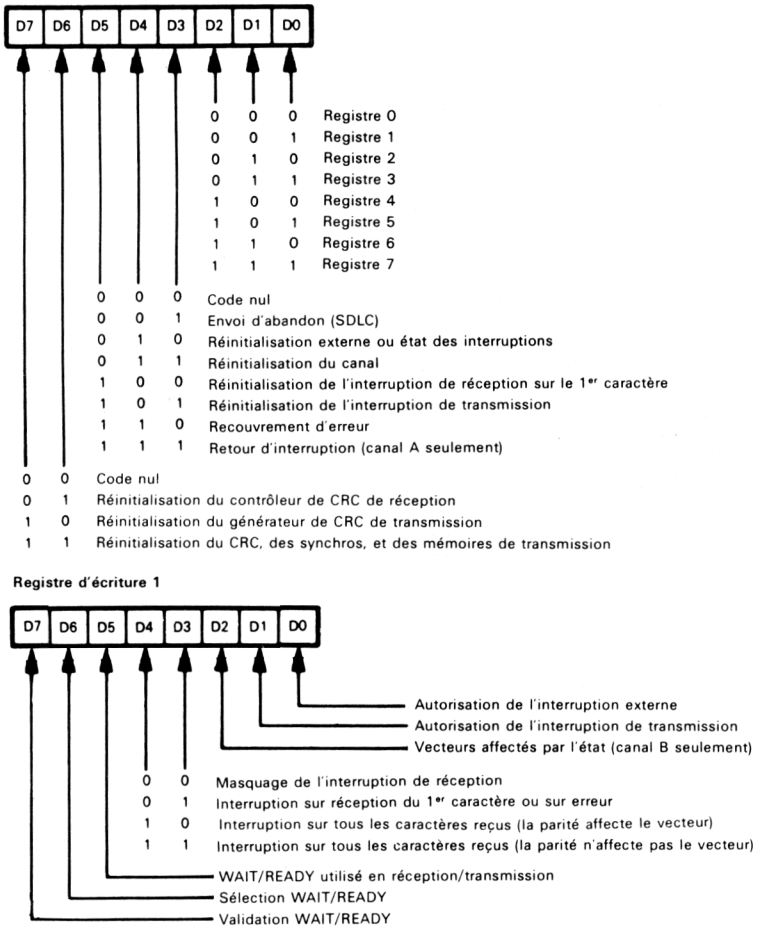
Registre d'écriture

Le SIO du Z80 contient huit registres dans chaque canal qui sont programmés par le logiciel du système qui configure la fonction spécifique de chaque canal. Tous les registres d'écriture, à l'exception du registre d'écriture 0, requièrent deux octets pour être correctement programmés. Le premier octet contient trois bits qui pointent sur le registre désiré (D0-D2) ; le second octet est le mot de contrôle qui doit être écrit dans ce registre pour configurer le SIO.

Le registre d'écriture 0 est un cas spécial. Le signal RESET, réinitialisant (soit par une commande interne soit par une entrée externe) initialisera le SIO en écriture du registre 0. Toutes les commandes de base (CMD2-CMD0) et les contrôles du CRC (CRC0-CRC1) sont accessibles avec un seul octet en utilisant le registre d'écriture C.

En incluant ces commandes dans le premier octet du registre d'écriture, on obtient ainsi une grande souplesse et un contrôle maximum du système.

Registre d'écriture 0



Registre d'écriture 1

Fig. 11.33. — Registres de contrôle et d'écriture du SIO.

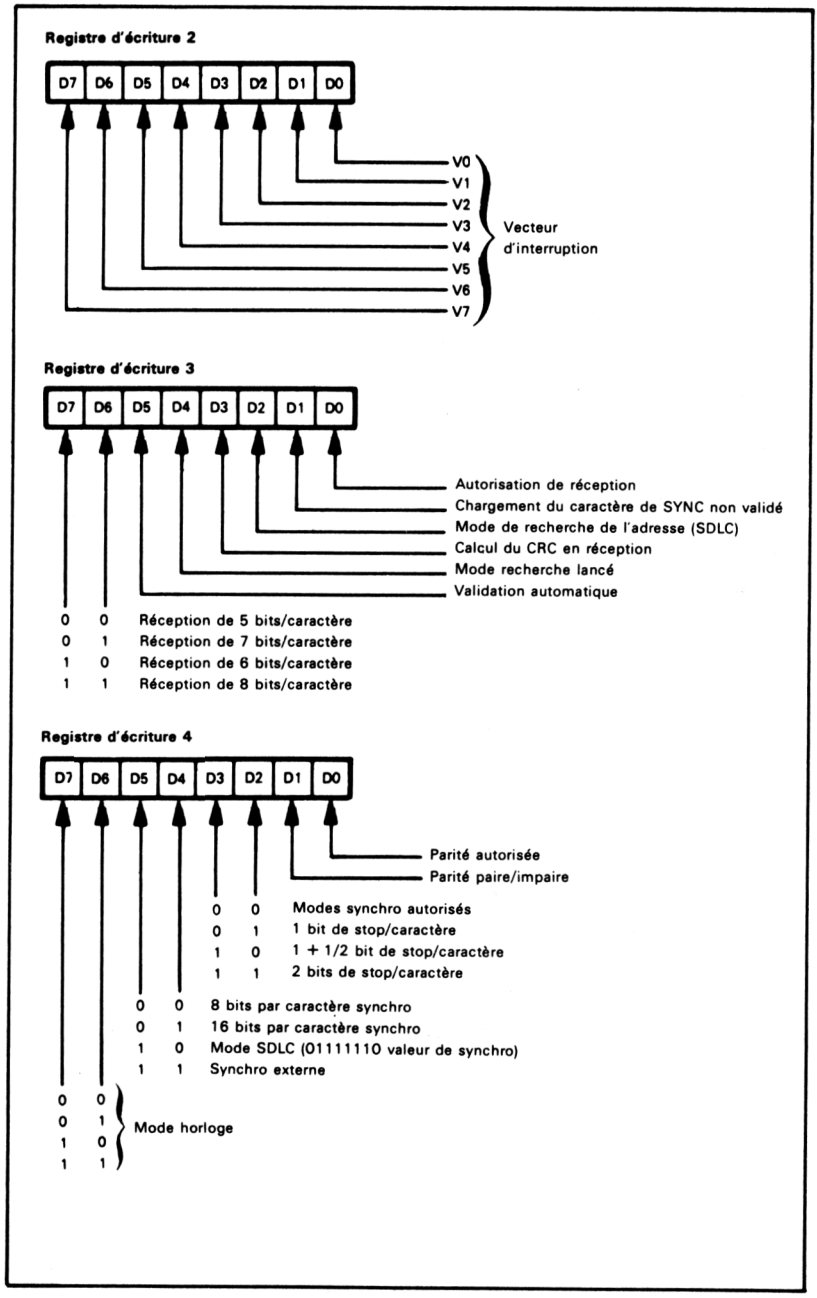


Fig. 11.33. — Registres de contrôle et d'écriture du SIO.

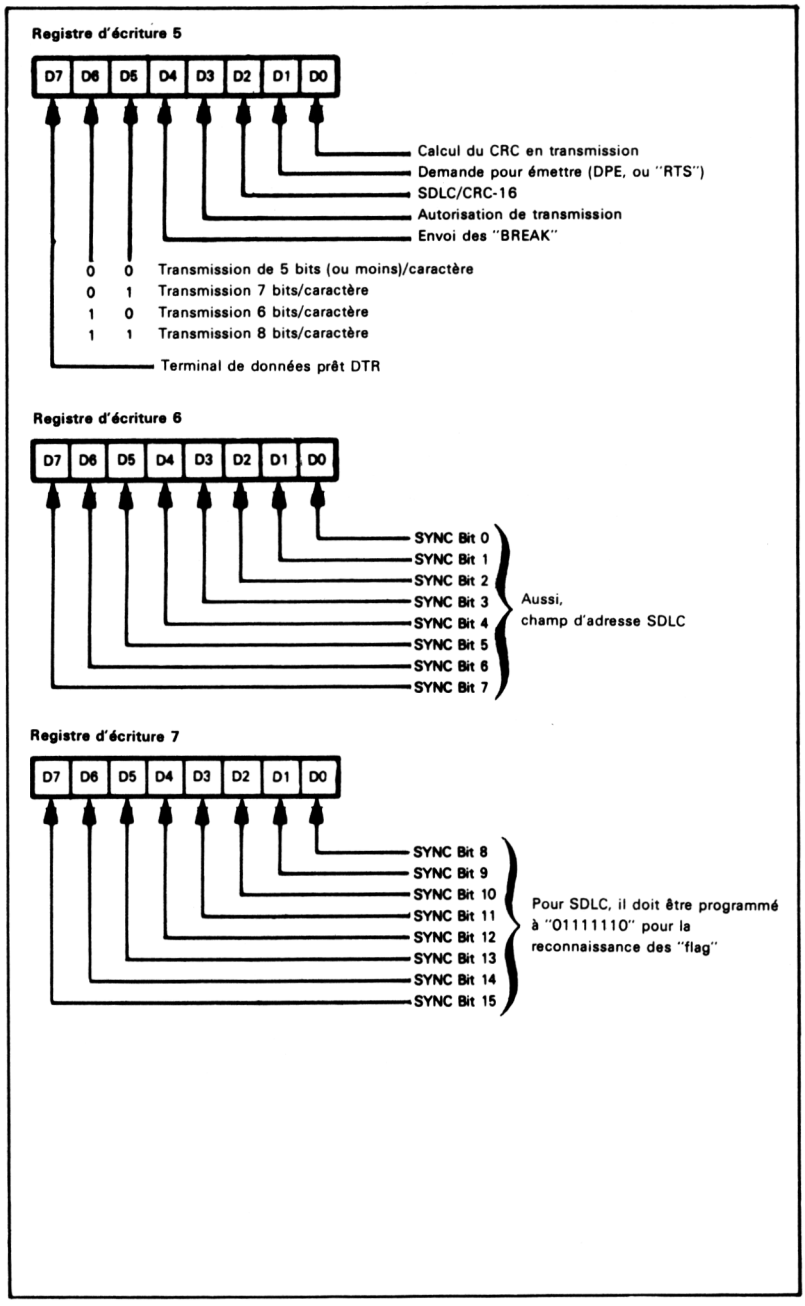


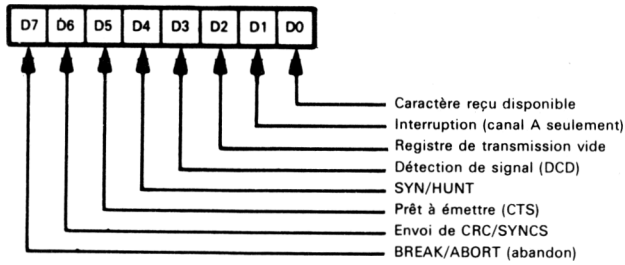
Fig. 11.33. — Registres de contrôle ou d'écriture du SIO (suite).

Registres de lecture

Le SIO du Z80 possède trois registres de lecture qui peuvent être lus pour connaître l'état de chaque canal. Les informations d'état comprennent des conditions d'erreur, vecteur d'interruption, et des signaux d'interface des protocoles de communication standard. Pour lire le contenu du registre de lecture sélectionné, le logiciel doit d'abord écrire dans le SIO l'octet contenant le pointeur, D0-D2, de la même manière que pour une opération sur un registre d'écriture. Puis, en faisant une opération de lecture, le contenu du registre de lecture/d'état peut être lu par le Z80.

La réelle puissance d'une semblable structure de commande est que le programmeur est libre après avoir sélectionné le registre, de le lire ou de l'écrire pour l'initialiser, ou de le tester. En concevant le logiciel d'initialisation du SIO, d'une manière structurée et modulaire, le programmeur peut utiliser avantageusement les instructions d'E/S puissantes du Z80, pour simplifier et diminuer le temps de développement et de mise au point de son logiciel.

Registre de lecture 0



Registre de lecture 1

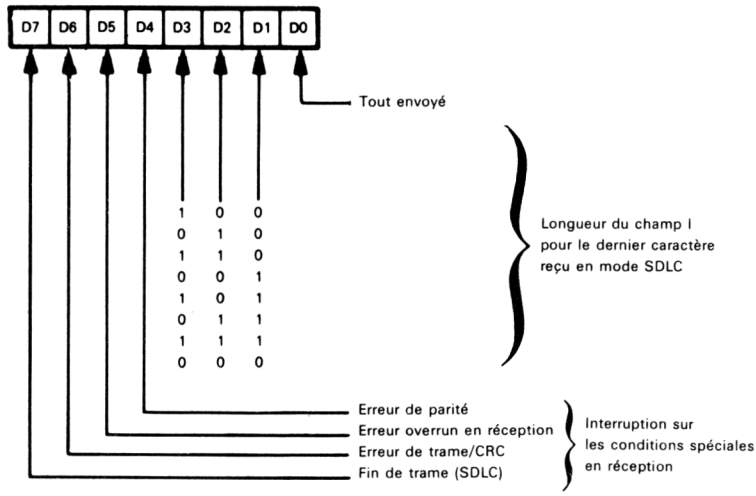


Fig. 11.34. — Registres d'état et de lecture du SIO.

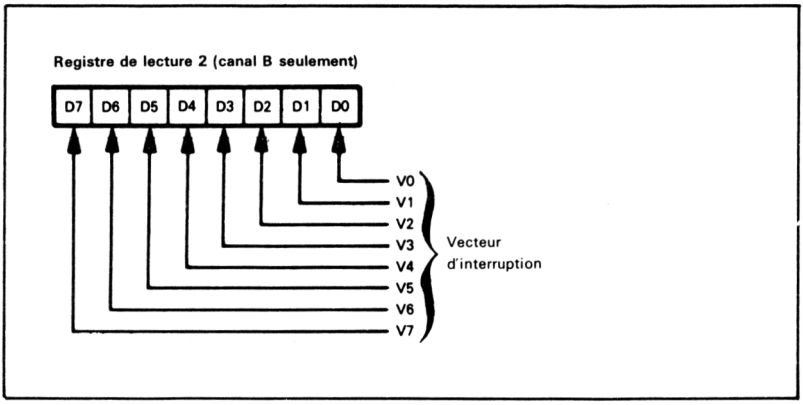


Fig. 11.34. — Registres d'état ou de lecture du SIO (suite).

4) Le SIO doit être configuré avant d'être utilisé. La manière la plus aisée d'accomplir cela est de mettre les octets nécessaires dans une table et d'utiliser les instructions répétitives d'E/S de bloc. La table doit inclure à la fois les octets nécessaires à l'adressage des différents registres et les données à mettre dans ceux-ci. Un programme classique sera :

```
LD      B,LENG      ; NOMBRE D'OCTETS DANS LA TABLE
LD      C,SIOCRA    ; NUMERO DU PORT
LD      HL,CTLTAB   ; DEBUT DE LA TABLE DE CONTROLE
OTIR                ; CONFIGURATION DU SIO
```

5) Les signaux RS232 sont tous actifs bas. Cependant, les bits de contrôle de ces signaux, dans le SIO, sont actifs haut (c'est-à-dire un un logique dans un bit de contrôle, envoie le signal correspondant bas).
6) Le SIO a besoin d'une horloge externe. Dans le mode asynchrone et à 110 bauds, une fréquence de 1 760 Hz est généralement utilisée ainsi que le mode X16. Le SIO échantillonnera les bits à la fréquence d'horloge de synchronisation, et éliminera les faux bits de début, provoqués par le bruit sur la ligne.
7) L'indicateur de donnée prête (data ready) (caractère reçu disponible) est le bit 0 du registre de lecture 0. L'indicateur de périphérique prêt (tampon de transmission vide) est le bit 2 du registre 0.
8) Les bits d'état (parité, overrun et trame) sont dans le registre de lecture 1.

EXEMPLES

E/S d'un télé-imprimeur via un UART

Tâche 1 : lecture d'un télé-imprimeur à l'aide d'un SIO.

But : recevoir une donnée d'un TTY à l'aide d'un SIO et mettre la donnée dans la mémoire 0040. La donnée a 7 bits ASCII avec une parité impaire.

Programme source :

```
        LD      A,4             ; ACCES AU REGISTRE D'ECRITURE 4
        OUT     (SIOCRA),A
        LD      A,01000001B     ; MODE HORLOGE x 16 ET PARITE IMPAIRE
        OUT     (SIOCRA),A
        LD      A,3             ; ACCES AU REGISTRE D'ECRITURE 3
        OUT     (SIOCRA),A
        LD      A,01000001B     ; 7 BITS/CARACTERE, AUTORISATION DE RECEPTION
        OUT     (SIOCRA),A
        SUB     A               ; ACCES AU REGISTRE DE LECTURE 0
        OUT     (SIOCRA),A
ATTEN : IN      A,(SIOCRA)      ; ACQUISITION DE L'ETAT
        RRA                     ; DONNEE DISPONIBLE ?
        JR      NC,ATTEN        ; NON, ATTENTE
        IN      A,(SIODRA)      ; OUI, ACQUISITION
        LD      (40H),A         ; SAUVEGARDE EN MEMOIRE
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
0000	3E	LD	A,4
0001	04		
0002	D3	OUT	(SIOCRA),A
0003	SIOCRA		
0004	3E	LD	A,01000001B
0005	41		
0006	D3	OUT	(SIOCRA),A
0007	SIOCRA		
0008	3E	LD	A,3
0009	03		
000A	D3	OUT	(SIOCRA),A
000B	SIOCRA		
000C	3E	LD	A,01000001B
000D	41		
000E	D3	OUT	(SIOCRA),A
000F	SIOCRA		
0010	97	SUB	A
0011	D3	OUT	(SIOCRA),A
0012	SIOCRA		
0013	DB	ATTEN : IN	A,(SIOCRA)
0014	SIOCRA		
0015	1F	RRA	
0016	30	JR	NC,ATTEN
0017	FB		
0018	DB	IN	A,(SIODRA)
0019	SIODRA		
001A	32	LD	(40H),A
001B	40		
001C	00		
001D	76	HALT	

EXEMPLE DE CONFIGURATION DU SIO

Le programme positionne le registre d'écriture comme suit :
bit 7 et 6 = 01 pour sélectionner le mode horloge × 16 (1 760 Hz doit être fourni)
bit 1 = 0 pour sélectionner la parité impaire
bit 0 = 1 pour autoriser la vérification de la parité.

Le programme positionne le registre d'écriture 3 comme suit :

bit 7 et 6 = 01 pour 7 bits/caractère
bit 0 = 1 pour autoriser la réception

L'état de la donnée reçue est le bit 0 du registre de lecture 0.
Remarque : toutes les erreurs trouvées seront reportées dans le registre de lecture 1.

ETATS D'ERREUR DU SIO

bit 6 = 1 pour une erreur de trame (pas de bit stop)
bit 5 = 1 pour une erreur d'overrun (une donnée reçue en écrase une autre)
bit 4 = 1 pour une erreur de parité.

Essayez d'ajouter un programme de test d'erreurs. Positionner :

(0061) à 0 si aucune erreur
à 1 si une erreur de parité
à 2 si une erreur d'overrun
à 3 si une erreur de trame

Remarque : le récepteur vérifie toujours qu'il y a un bit de stop.

Tâche 2 : écrire vers un téléimprimeur à l'aide d'un SIO

But : envoyer une donnée contenue dans la mémoire 40 vers un TTY à l'aide d'un SIO. La donnée a 7 bits ASCII avec une parité impaire.

Programme source :

```
        LD      A,4             ; ACCES AU REGISTRE D'ECRITURE 4
        OUT     (SIOCRA),A
        LD      A,01001101B     ; MODE HORLOGE x 16, 2 BITS DE STOP, PARITE IMPAIRE
        OUT     (SIOCRA),A
        LD      A,5             ; ACCES AU REGISTRE D'ECRITURE 5
        OUT     (SIOCRA),A
        LD      A,00101000B     ; 7 BITS/CARACTERE, AUTORISATION DE TRANSMETTRE
        OUT     (SIOCRA),A
        SUB     A               ; ACCES AU REGISTRE DE LECTURE 0
        OUT     (SIOCRA),A
ATTEN   IN      A,(SIOCRA)      ; ACQUISITION DE L'ETAT
        BIT     2,A             ; LE TRANSMETTEUR EST PRET ?
        JR      Z,ATTEN         ; NON, ATTENTE
        LD      A,(40H)         ; OUI, ACQUISITION DE LA DONNEE
        OUT     (SIODRA),A      ; ET TRANSMISSION
        HALT
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)		Instruction (Mnémonique)	
0000	3E		LD	A,4
0001	04			
0002	D3		OUT	(SIOCRA),A
0003	SIOCRA			
0004	3E		LD	A,01001101B
0005	4D			
0006	D3		OUT	(SIOCRA),A
0007	SIOCRA			
0008	3E		LD	A,5
0009	05			
000A	D3		OUT	(SIOCRA),A
000B	SIOCRA			
000C	3E		LD	A,00101000B
000D	28			
000E	D3		OUT	(SIOCRA),A
000F	SIOCRA			
0010	97		SUB	A
0011	D3		OUT	(SIOCRA),A
0012	SIOCRA			
0013	DB	ATTEN :	IN	A,(SIOCRA)
0014	SIOCRA			
0015	CB		BIT	2,A
0016	57			
0017	28		JR	Z,ATTEN
0018	FA			
0019	3A		LD	A,(40H)
001A	40			
001B	00			
001C	D3		OUT	(SIODRA),A
001D	SIODRA			
001E	76		HALT	

Le programme positionne le registre d'écriture 4 comme suit :

bits 7 et 6 = 01 pour sélectionner le mode horloge × 16 (1 760 Hz doit être fourni)
bits 3 et 2 = 11 pour ajouter 2 bits de stop à chaque caractère
bits 1 = 0 pour sélectionner la parité impaire
bit 0 = 1 pour autoriser la génération de la parité.
Le programme positionne le registre d'écriture 5 comme suit :

bits 6 et 5 = 01 pour 7 bits par caractère
bit 3 = 1 pour autoriser la transmission.
Le bit d'état de la transmission est le bit 2 du registre de lecture 1.

INTERFACES STANDARD

INTERFACES STANDARD

D'autres interfaces standard, outre la boucle de courant des TTY et la RS232, peuvent être utilisées pour connecter des périphériques au micro-ordinateur. Les plus répandues sont :

1) Les interfaces séries RS449, RS422 et RS423 [13].
2) Le bus d'interface 8 bits parallèle à usage général, bien connu sous le nom de IEEE-488 ou bus d'interface Hewlett-Packard (HPIB) [14].
3) Le bus des Hobbyist S-100 ou Altair/Imsai [15]. C'est aussi un bus 8 bits.
4) Le multibus Intel [16]. C'est également un bus 8 bits, mais il peut être étendu pour gérer 16 bits parallèles.

EXERCICES :

6) Séparation des touches sur un clavier non encodé

But : le programme doit lire les entrées d'un clavier 3×3 non encodé, et les placer dans un tableau. Le nombre d'entrées demandé est contenu dans la mémoire 40 et le tableau commence en 41.
L'enfoncement d'une touche est séparé de l'enfoncement suivant en attendant la fin du contact en cours. Ne pas oublier de gérer les rebonds (en attendant simplement un délai de 1 ms).

Exemple pratique :

```
          (0040) = 04
Les entrées sont 7, 2, 2, 4
Résultat : (0041) = 07
           (0042) = 02
           (0043) = 02
           (0044) = 04
```

7) Lecture d'une phrase à partir d'un clavier encodé

But : le programme devra lire des entrées à partir d'un clavier ASCII (7 bits avec un bit de parité à zéro) et les placer dans un tableau jusqu'à ce qu'il reçoive un point ASCII ($2E_{16}$). Le tableau démarre à l'adresse 0040. Chaque entrée est signalée par un signal de validation comme dans l'exemple d'un clavier encodé donné dans ce chapitre.

Exemple pratique :

```
Les entrées sont H, E, L, L, O, .
Résultat : (0040) = 48  H
           (0041) = 45  E
           (0042) = 4C  L
           (0043) = 4C  L
           (0044) = 4F  O
           (0045) = 2E
```

8) Un générateur de signal carré d'amplitude variable

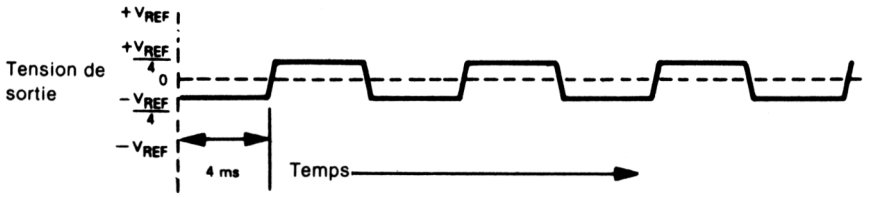
But : Le programme doit générer une onde carrée, comme décrit dans la figure ci-dessous, en utilisant un convertisseur digital-analogique. La mémoire 40 contient l'échelle de l'amplitude du signal, la mémoire 41 contient la longueur d'une demi-période en millisecondes, et la mémoire 42 le nombre de cycles.

On suppose qu'une sortie digitale de 80_{16}, vers le convertisseur provoquera une sortie analogique de zéro volt. En général, une sortie de D provoquera une sortie analogique de Vsortie = – Vréf (D – 80)/80 volts.

Exemple pratique :

(0040) = A0 (hexa)
(0041) = 04
(0042) = 03

Résultat :



Tension de base est 80_{16} = 0 volts.
Pleine échelle 100_{16} = $-V_{REF}$ volts.
Aussi $A0_{16}$ = (A0-80)/80 X $-V_{REF}$ = $-V_{REF}/4$

Le programme génère 3 impulsions d'amplitude Vréf/4 avec une demi période de 4 ms.

9) Moyenne de lectures analogiques

But : le programme doit prendre quatre mesures à l'aide d'un convertisseur A/D, séparées par 10 ms et placer la moyenne dans la mémoire 40. On suppose que le temps de conversion est négligeable.

Exemple pratique :

Les lectures sont, en hexadécimal : 86, 89, 81, 84

Résultat : (0040) = 85

10) Un terminal à 30 caractères par seconde

But : modifier les programmes de réception et de transmission donnés dans ce chapitre pour un TTY, pour gérer un terminal à 30 caractères par seconde qui transfère des données ASCII avec un bit de stop et une parité paire. Comment écririez-vous ce programme pour gérer un terminal ou l'autre suivant un indicateur contenu dans la mémoire 0060, c'est-à-dire (0060) = 0 pour un terminal à 30 cps, (0060) = 1 pour un terminal à 10 cps (TTY) ?

BIBLIOGRAPHIE ET RÉFÉRENCES

1. Barnes J. et V. Gregory : « Utilisation des micro-ordinateurs pour améliorer les performances en milieu perturbé. EDN, 20 août, 1976, pages 71-72.
2. Swanson R. « Comprendre les CRC ». Computer Design, novembre 1975, pages 93-99.
   Mc Namara JE, « Aspects techniques des transmissions de données ». Digital Equipment Corp, Maynard Mass, 1977.
3. Par exemple, le contrôleur Z80 d'accès direct mémoire (ou DMA), pour des micro-ordinateurs basés sur le Z80, est décrit dans : « Une introduction aux microprocesseurs : volume 2 - Des microprocesseurs ».
4. Le guide de l'ingénieur TTL, Texas Instrument Inc., PO box 5012, Dallas, Texas 75222. 1976.
5. Dilatush E., « Dossier spécial : afficheurs numériques et alphanumériques ». EDN, 5 février 1978, pp. 26-35.
6. Cf. référence 4.
7. Hnatek E.R., Manuel pour les utilisateurs de convertisseurs A/D et D/A. Wiley New York, 1976.
8. Cf. référence 7.
9. A consulter aussi, par D. Guzeman, « Marier votre μp aux convertisseurs monolithiques A/D. Electronic Design, 18 janvier 1977, p. 82-86.
10. Pour une discussion à propos des UART, lire P. Rony « The bug book 11a ». E. and L. Instruments Inc., 61 first Street, Derby CT 06418 ; et aussi D.G. Larsen et al., « INWAS : Interfaces avec un mode série synchrone. IEEE Transaction sur l'électronique industrielle et l'instrumentation de contrôle », février 1977, pp. 2-12. Regarder aussi Mc Namara, référence 2.
11. La norme officielle RS232 est disponible en tant que « Interface entre un équipement terminal de donnée et un équipement de communications utilisant des transferts de données binaires sérielles ». EIA RS232C, août 1969. Vous pouvez trouver une introduction aux liaisons en RS232 dans « Qui a peur de la RS232 ? » de G. Pickless dans la revue Kilobaud, mai 1977, pp. 50-54, et dans « Travaux pour micro-ordinateur - 2e partie » de C.A. Ogdin, juillet 1978, pp. 76-80. Ogdin décrit aussi la récente norme RS449.
12. Le SIO est décrit plus complètement dans le volume 3 de « Introduction aux micro-ordinateurs ». Les références suivantes définissent son utilisation comme contrôleur de transmission de données : « Puces de contrôle de transmission de données » par Weissberger A.J.

CHAPITRE 12

INTERRUPTIONS

Les interruptions sont des entrées que l'unité centrale (UC) examine à chaque cycle d'instruction. Ces entrées permettent à l'UC de réagir à des événements asynchrones d'une façon plus efficace qu'en allant sonder chaque périphérique. Les interruptions nécessitent parfois davantage de matériel que les E/S ordinaires, mais fournissent de toutes façons une réponse plus rapide et plus directe [1].

UTILISATION DES INTERRUPTIONS

Pourquoi utiliser les interruptions ? Les interruptions permettent à des événements tels que alarmes, coupure d'alimentation, durée de temps écoulée, périphériques prêts à transmettre ou à recevoir des données, d'attirer immédiatement l'attention de l'UC. Le programmeur n'a pas besoin de tester chaque dispositif, ni de s'inquiéter si le système va laisser passer des événements. Un système d'interruptions est semblable à la sonnerie du téléphone — elle n'intervient que lors d'un appel ; il est donc inutile, en l'absence de sonnerie, d'aller décrocher le combiné pour voir si par hasard quelqu'un ne serait pas au bout du fil. L'UC peut poursuivre son travail normalement. Lorsqu'il se passe quelque chose, l'interruption active l'UC et la force à traiter l'entrée avant de reprendre les opérations courantes. Naturellement, cette description se complique (de la même façon qu'un standard téléphonique), dans le cas de nombreuses interruptions d'importances diverses et de tâches qu'il n'est pas possible d'interrompre.

CARACTÉRISTIQUES DES SYSTÈMES D'INTERRUPTION

La réalisation de systèmes d'interruptions diffère largement d'un système à l'autre. Voici les questions clés sur lesquelles repose un système particulier :

1) Quel est le nombre d'entrées d'interruptions ?
2) Comment l'UC répond-elle à une interruption ?
3) Comment l'UC détermine-t-elle l'origine de l'interruption si le nombre des origines dépasse le nombre des entrées ?
4) L'UC peut-elle différencier les interruptions importantes des autres ?
5) Quand et comment autoriser (« enable ») ou interdire (« disable ») les interruptions ?

Il y a toutes sortes de réponses à ces questions. Cependant, chaque système vise au même but : la réponse rapide de l'UC aux interruptions suivie de la réponse normale de l'activité.
Le nombre d'entrées d'interruption sur le circuit de l'UC détermine le nombre de réponses différentes que l'UC peut donner sans l'apport d'autre logiciel ou matériel. Chaque entrée peut produire une réponse interne différente. Malheureusement, la plupart des microprocesseurs disposent d'un tout petit nombre (en général 1 ou 2) d'entrées séparées. La réponse finale de l'UC à une interruption doit être le transfert du contrôle au bon pro-

gramme gérant l'interruption et la sauvegarde de la valeur en cours du compteur ordinal. Pour ce faire, l'UC doit exécuter une instruction d'appel (Call) ou de saut à un sous-programme (Jump to Subroutine) avec pour adresse le début du programme de gestion d'interruptions. Ceci a pour effet la sauvegarde en pile de l'adresse de retour et le transfert du contrôle au programme d'interruptions. La quantité de matériel nécessaire pour produire cette réponse diffère largement d'une UC à l'autre. Certaines génèrent de façon interne l'instruction et l'adresse, d'autres exigent pour cela des circuits externes. L'UC ne peut que générer une instruction ou une adresse différente par entrée séparée.

SCRUTATION
VECTORISATION

Si le nombre de dispositifs d'interruptions dépasse celui des entrées, l'UC nécessite l'apport externe de matériel ou logiciel pour identifier l'origine de l'interruption. Dans le cas le plus simple, le logiciel consiste en un programme de scrutation qui vérifie l'état des dispositifs pouvant présenter une interruption. Le seul avantage d'un tel système sur une scrutation classique réside dans le fait que l'UC sait qu'au moins un dispositif est actif. L'autre solution consiste à ajouter des circuits pouvant fournir une donnée unique (ou « vecteur ») pour chaque origine. On peut combiner ces deux solutions ; les vecteurs identifient alors des groupes d'entrées à partir desquels l'UC peut identifier une entrée particulière par scrutation.

PRIORITÉ

Un système d'interruptions qui peut distinguer les interruptions suivant leur importance relative est dit « système à priorité » (ou hiérarchisé). Le câblage interne peut fournir autant de niveaux de priorité qu'il y a d'entrées. Le câblage externe peut fournir des niveaux supplémentaires au moyen d'un registre de priorité et de comparateurs. Le câblage externe ne permet à une interruption d'être reconnue que si sa priorité est supérieure à celle du contenu du registre de priorité. Dans un système hiérarchisé, une procédure spéciale peut être nécessaire pour traiter les interruptions de bas niveau qui peuvent être longtemps ignorées.

AUTORISATION ET INTERDICTION DES INTERRUPTIONS

La plupart des systèmes d'interruptions peuvent être autorisés ou interdits. En fait, la plupart des microprocesseurs interdisent automatiquement les interruptions lors d'un RESET (pour permettre au programmeur de configurer le système d'interruptions) ou lorsqu'une interruption est reconnue (pour que l'interruption n'interrompe pas son propre programme de traitement). Le programmeur peut aussi interdire les interruptions pendant le traitement des données, une boucle de temporisation ou une opération sur plusieurs octets.

INTERRUPTION NON MASQUABLE

Une interruption qui ne peut pas être interdite (appelée parfois « interruption non masquable ») peut servir à prévenir d'une coupure d'alimentation, événement dont on doit, à l'évidence, tenir compte avant toute autre chose.

INCONVÉNIENTS DES INTERRUPTIONS

Les avantages des interruptions sont évidents, mais elles comportent aussi quelques inconvénients. Parmi ceux-ci :

1) Les systèmes d'interruptions peuvent nécessiter de nombreux circuits supplémentaires.
2) Le traitement des interruptions passe toujours par l'UC. Il n'y a donc aucun gain de vitesse, comme avec le DMA.

3) Les interruptions sont des entrées aléatoires, ce qui rend la mise au point et les tests difficiles. Des erreurs sporadiques risquent de survenir et sont, de ce fait, difficiles à trouver [2].
4) Les interruptions peuvent être pénalisantes s'il y a de nombreux registres à sauvegarder et si la source est trouvée par scrutation.

SYSTÈME D'INTERRUPTIONS DU Z80

ENTRÉES D'INTERRUPTIONS DU Z80

La réponse interne du Z80 à une interruption est assez complexe puisqu'il existe 3 modes d'exploitation différents. Le système d'interruptions consiste en :

1) Une interruption masquable active à l'état bas (INT) et une interruption non masquable active basse (sur un front descendant) (NMI).
2) Deux bascules d'autorisation (« flip-flop ») (IFF1 et IFF2). IFF1 est positionnée à 1 ou à 0 pour autoriser ou inhiber les interruptions. IFF2 sert au stockage temporaire de IFF1 au cours des interruptions non masquables.

RÉPONSE DU Z80 A UNE INTERRUPTION

Le Z80 vérifie l'état en cours du système d'interruption à la fin de chaque cycle d'instruction. Si l'interruption est active et autorisée, la réponse est donnée comme suit [3] :

1) L'UC interdit les interruptions en remettant IFF1 à zéro. IFF2, cependant, est maintenu dans son état original s'il est survenu une interruption non masquable. Notons que RESET remet à zéro les 2 bascules d'interruptions de telle sorte que le système puisse être configuré avant que les interruptions ne soient autorisées.
2) L'UC exécute un cycle spécial d'acquittement des interruptions. On identifie ce cycle par les états des différents signaux suivants :
M1 (acquisition du code opératoire) actif.
MREQ (demande d'accès mémoire) inactif de façon à ce que l'UC ne procède pas à son accès mémoire habituel.
Et IORQ (demande d'entrées/sorties) actif pour qu'un vecteur de réponse à une interruption puisse être renvoyé sur le bus de données.
Le reste de la réponse dépend du mode d'interruption et de sa source.
Notons en particulier que le Z80 testera les interruptions après chaque transfert ou comparaison dans les instructions de transfert en bloc, de comparaison en bloc ou les instructions répétitives d'entrée/sortie par blocs.

INSTRUCTION D'INTERRUPTION DU Z80

Le Z80 dispose des instructions spéciales suivantes à l'usage du système d'interruption :

1) EI (« Enable Interrupts ») autorise les interruptions masquables en positionnant à 1 les bascules d'interruption.
2) DI (« Disable Interrupts ») interdit les interruptions masquables en mettant à 0 les bascules d'interruption.
3) RST (« Restart ») est une instruction d'appel sur un octet qui sauvegarde dans la pile la valeur en cours du compteur ordinal et effectue un saut à l'adresse spécifiée dans l'instruction. Le tableau 12.1 contient les différentes instructions Restart ainsi que leurs

Tableau 12.1. — Instructions « Restart » (RST)

Instruction RST (mnémonique)	Code opératoire (hexa)	Adresse de destination (hexa)	(décimal)
RST 0	C7	0000	0
RST 8	CF	0008	08
RST 10H	D7	0010	16
RST 18H	DF	0018	24
RST 20H	E7	0020	32
RST 28H	EF	0028	40
RST 30H	F7	0030	48
RST 38H	FF	0038	56

adresses de destination. L'instruction RST est souvent utilisée dans les systèmes d'interruption parce que c'est une instruction sur un octet facile à constituer et à placer sur le bus de données.
4) RETI (« Return from Interrupt ») est tout à fait similaire à la simple instruction RET (« Return »), excepté que les circuits périphériques du Z80 (PIO, SIO, CTC,...) décodent cette instruction et sont ainsi avertis que le programme de traitement de l'interruption en cours est terminé.
5) RETN (« Return from Non-Masquable Interrupt ») est tout à fait similaire à la simple instruction RET (« Return ») excepté qu'elle charge la bascule IFF1 à partir de IFF2 pour revenir à l'état initial du système d'interruptions.
6) LD A,I charge le contenu du registre I (vecteur d'interruption) dans l'accumulateur. Cette instruction (ainsi que LD A,R) met aussi l'état de IFF2 dans le bit P/O du registre d'état. Cet indicateur peut être testé ou sauvegardé dans la pile.
7) LD I,A charge le contenu de l'accumulateur dans le registre I (vecteur d'interruption).
8) IM (Set Interrupt Mode) détermine le mode dans lequel les interruptions sont traitées. Il y a 3 options : 0, 1 et 2 qui seront décrites ultérieurement dans ce chapitre.

Interruption non masquable

INTERRUPTION NON MASQUABLE DU Z80

L'interruption non masquable est déclenchée sur un front d'entrée négatif. Le processeur ne réagit que sur un front d'impulsion sur cette ligne et ainsi l'impulsion n'interrompra pas son propre programme. L'interruption non masquable est très utile dans les applications où il faut répondre à une défaillance de l'alimentation (par exemple, sauvegarder les données dans une mémoire à faible consommation ou connecter une batterie de secours). Les applications typiques concernent les équipements de télécommunications devant retenir des codes et des messages partiels, et les instruments de test devant conserver des résultats partiels ou complets.
Le Z80 répond à une interruption non masquable de la façon suivante :

1) Il met à zéro la bascule IFF1, interdisant ainsi toutes les interruptions (mais en sauvegardant l'ancien état de IFF1 dans IFF2).
2) Il ignore l'instruction suivante cherchée à partir de la mémoire et à la place effectue un saut à l'adresse 0066_{16}, en sauvegardant l'ancienne valeur du compteur ordinal dans la pile.

Rappelons qu'une instruction RETN à la fin du programme de gestion d'interruption remet IFF1 à sa valeur initiale à partir de IFF2.
Nous ne discuterons pas plus longuement de l'interruption non masquable, et supposerons que toutes les interruptions sont connectées à INT.

Modes d'interruption du Z80

MODES D'INTERRUPTION

Le Z80 a 3 modes d'interruption. Le programmeur peut choisir n'importe lequel de ces modes à l'aide de l'instruction IM adéquate. Lors de la réinitialisation du système, le processeur prend toujours le mode 0. Voici les différents modes :

Mode 0

Dans ce mode, l'UC utilise la donnée renvoyée lors d'un cycle d'acquitement comme une instruction à exécuter. Ce mode est identique à celui utilisé dans le 8080 [4].
Habituellement, cette donnée en entrée correspond au code d'une instruction RST (voir tableau 12.1) qui est alors exécutée.

INSTRUCTION RESTART

L'instruction RST est utile dans les systèmes d'interruptions pour les raisons suivantes :

1) C'est une instruction sur un octet qui ne requiert donc qu'un cycle de lecture.
2) Elle peut fournir 8 adresses différentes de destinations (vecteurs).
3) L'espacement de ces vecteurs est suffisant pour permettre l'utilisation d'instruction de saut, qui pointeront les programmes de traitement effectifs.
4) Elle est facile à constituer, puisque 5 des bits sont toujours à 1. Un encodeur 8 vers 3 peut fournir les 3 autres très simplement.
Mais l'instruction RST présente les inconvénients suivants :

1) Elle ne peut pas fournir plus de 8 vecteurs.
2) Ses vecteurs ne sont pas suffisamment espacés pour permettre l'implantation de programmes entiers de gestion d'interruption.
3) Ses vecteurs sont dans une zone mémoire d'adresses fixes.
4) L'instruction RST 0 a la même adresse de destination que l'entrée RESET et est de ce fait, difficile à utiliser. Des circuits supplémentaires sont nécessaires pour distinguer RESET de RST 0 puisque le logiciel seul ne le permet pas.
Rappelons que l'instruction RST sauvegarde en pile l'ancienne valeur du compteur ordinal de la même façon qu'un CALL.

Mode 1

Dans ce mode, l'UC ignore la donnée renvoyée lors d'un cycle d'acquitement et exécute toujours l'instruction RST 38H. Ceci a donc pour effet de se brancher à l'adresse 0038_{16} et de sauvegarder l'ancienne valeur du compteur ordinal dans la pile. Ce mode est équivalent au mode 0 si la donnée en entrée est toujours RST 38H (FF_{16}).

Ce mode présente l'avantage de ne pas nécessiter d'apport externe de matériel. Par contre, il ne permet pas de distinguer de façon directe les sources d'interruption ; de plus, l'adresse de destination est fixe. Le mode 1 est utile dans les applications qui ne comportent qu'une ou deux sources d'interruption et dans lesquelles on recherche un minimum du matériel.

Mode 2

Dans ce mode, l'UC utilise les données en entrée comme une partie de l'adresse permettant d'obtenir l'adresse de départ du programme de gestion d'interruptions. Lorsqu'une interruption est acceptée, l'UC :
1) Interdit les interruptions ultérieures en remettant à zéro IFF1 et IFF2.
2) Range l'ancienne valeur du compteur ordinal en pile.
3) Constitue un pointeur à partir du contenu du registre I (huit bits de poids fort) et de l'entrée du bus de données pendant le cycle de reconnaissance d'interruption (huit bits de poids faible). Le bit le moins significatif de ce pointeur est mis à zéro.
4) Recherche une adresse à partir des 2 emplacements mémoire en commençant par celle indiquée par le pointeur (fig. 12.1).
5) Donne le contrôle à l'adresse obtenue à partir de la mémoire.

Dans ce mode, la réponse à l'interruption nécessite 19 cycles d'horloge.
L'avantage de ce mode est qu'il peut fournir une page entière de 128 vecteurs de service d'interruption situés n'importe où en mémoire. Par contre, ses inconvénients résident dans une réponse d'interruption plus lente et dans une initialisation préalable du système telle que :
1) La table des vecteurs doit être chargée en mémoire si elle ne l'est pas en mémoire morte.
2) Le registre I doit être chargé avec les 8 bits les plus significatifs (ou numéro de la page) de l'adresse de la table. Notez que RESET remet le registre I à zéro. On peut charger une valeur dans I comme suit :

```
LD      A,IPGNO     ; ACQUISITION NUMERO DE PAGE
LD      I,A         ; RANGEMENT DANS REGISTRE DES VECTEURS
```

3) Le mode 2 d'interruption doit être positionné à l'aide de l'instruction IM2.

Le mode 2 est conçu pour marcher avec les PIO, SIO, CTC,... du Z80. Les interruptions des PIO et SIO sont décrites ultérieurement dans ce chapitre.

COMPATIBILITÉ D'INTERRUPTION Z80/8080

Le mode 0 pour le système d'interruption du Z80 est, ainsi qu'il l'a été mentionné, identique à la réponse d'interruption du 8080. Le 8080 ne dispose pas des modes 1 et 2, bien que le mode 1 ne soit en fait qu'un cas particulier du mode 0. Le 8080 ne possède pas non plus d'entrée NMI.
Le 8085 dispose d'entrées d'interruption supplémentaires qui ne sont disponibles ni sur le 8080, ni sur le Z80. Le 8085 possède aussi une interruption non masquable (appelée TRAP) qui force un appel à une adresse différente (24_{16}) de celle utilisée par l'entrée NMI du Z80.

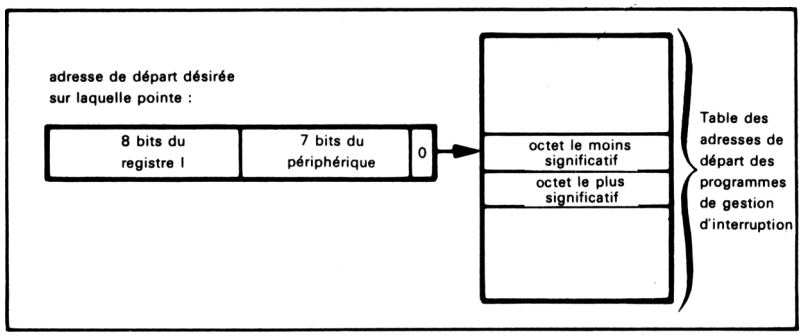


Fig. 12.1. — Constitution d'un vecteur d'interruption en mode 2.

INTERRUPTIONS DU PIO

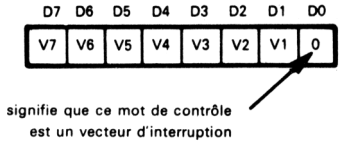
INTERRUPTIONS DU PIO

La plupart des systèmes d'interruption du Z80 comportent des PIO. Chaque port du PIO a les caractéristiques suivantes relatives aux interruptions :

1) Un registre de vecteur d'interruption sur 8 bits servant à contenir les 8 bits les moins significatifs de l'adresse de la table constituée par l'UC en mode 2.
2) Un bit d'autorisation d'interruption.
3) Un registre de contrôle d'interruption servant à déterminer l'opération logique effectuée et la polarité active pour la génération d'interruption dans le mode de contrôle.
4) Un registre de masque d'interruption servant à déterminer quelles lignes de données sont à surveiller pour générer des interruptions dans le mode de contrôle.

VECTEUR D'INTERRUPTION PIO

On peut accéder au registre d'interruption dans chaque port en écrivant un mot de contrôle avec un zéro dans le bit le moins significatif, comme décrit ci-dessous (voir également le tableau 11.2) :



Voici une séquence classique pour mettre la valeur dans ce registre :

```
LD      A,IVECT
OUT     (PIOCR),A
```

où IVECT a un '0' au bit le moins significatif. L'adresse de départ pour le programme de gestion d'interruption se trouve à l'adresse IVECT sur la page assignée à la table des adresses de départ des programmes de gestion d'interruption.

MODE DE CONTRÔLE D'INTERRUPTION DU PIO

On peut positionner l'octet de contrôle d'interruption dans chaque port en écrivant un octet de contrôle avec le format décrit dans la figure 12.2. Si le port est en mode 3, les bits D6, D5 et D4 ont les significations suivantes :

1) D6 = 1 signifie que toutes les lignes d'entrée/sortie programmées doivent devenir actives pour provoquer une interruption (c'est-à-dire un ET logique) tandis que D6 = 0 signifie que si au moins une des lignes d'E/S programmée devient active, elle provoque une interruption (c'est-à-dire un OU logique).
Notons qu'une interruption ne survient que si l'équation logique est vraie lorsque les interruptions sont autorisées.
2) D5 définit la polarité active (haute ou basse) des lignes d'E/S surveillées — D5 = 1 pour la polarité active haute, 0 pour la basse.
3) D4 = 1 signifie que l'octet de contrôle suivant est un masque d'interruption (figure 12.3). Seules les lignes possédant un bit de masque à zéro seront surveillées — D4 = 0 signifie qu'il n'y a pas de masque.

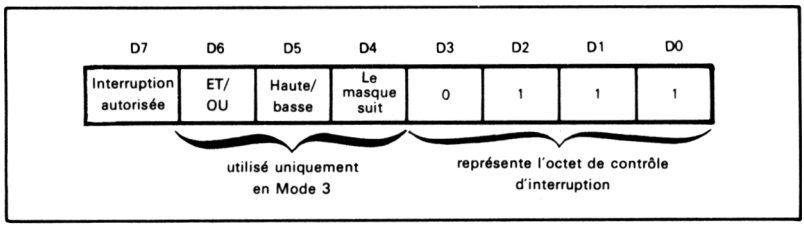


Fig. 12.2. — Format d'un octet de contrôle d'interruption d'un PIO.

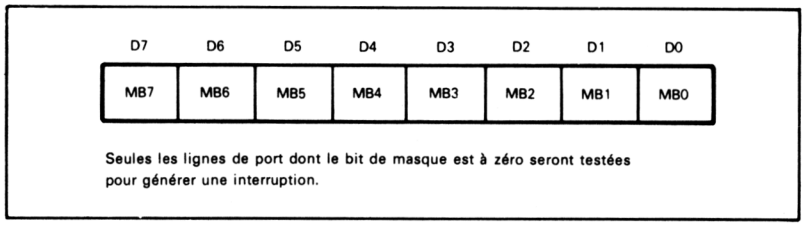


Fig. 12.3. — Format pour un masque d'interruption de PIO.

AUTORISATION ET INTERDICTION DES INTERRUPTIONS DU PIO

Le bit 7 de l'octet de contrôle d'interruption détermine la valeur de la bascule d'autorisation d'interruption du port. Des interruptions peuvent être générées si la bascule est positionnée à un. La mise sous tension réinitialise cette bascule, mais il faut se rappeler que le PIO n'a pas d'entrée RESET.
La bascule d'autorisation d'interruption peut être mise à 1 ou à 0 sans modifier le reste de l'octet de contrôle d'interruption en écrivant un octet de contrôle avec la valeur de la bascule dans le bit 7 et 0011 dans les 4 bits les moins significatifs.
La mise à 1 du bit 4 de l'octet de contrôle d'interruption remet à zéro toutes les interruptions en attente. Ceci peut servir à remettre à zéro toutes les interruptions survenues éventuellement au cours d'une réinitialisation du système.

EXEMPLES

EXEMPLES D'UNE CONFIGURATION D'INTERRUPTION DU PIO

1) Interruption d'un port de sortie avec un vecteur situé à l'adresse 80_{16}. Rappelons que le numéro de la page se trouve dans le registre I de l'UC.

```
LD      A,00001111B     ; PORT B EN SORTIE
OUT     (PIOCRB),A
LD      A,80H           ; ADRESSE DU VECTEUR = 80 HEXA
OUT     (PIOCRA),A
LD      A,10000011B     ; INTERRUPTION DU PIO AUTORISEE
OUT     (PIOCRB),A
```

Voici une autre solution qui remet à zéro les interruptions en attente tout en autorisant les interruptions à partir du port.

```
LD      A,10010111B     ; INTERRUPTION DU PIO AUTORISEE
OUT     (PIOCRA),A
```

Une interruption surviendra sur le front montant de $\overline{\text{STB}}$.

2) Interruption d'un port d'entrée avec un vecteur situé à l'adresse 60_{16}.

```
LD      A,01001111B     ; PORT A EN ENTREE
OUT     (PIOCRA),A
LD      A,60H           ; ADRESSE DU VECTEUR = 60 HEXA
OUT     (PIOCRA),A
LD      A,10000011B     ; INTERRUPTION DU PIO AUTORISEE
OUT     (PIOCRA),A
```

Une interruption surviendra sur le front montant de $\overline{\text{STB}}$.

3) Interruption d'un port de contrôle avec un vecteur situé à l'adresse 48_{16}. Une interruption sera générée si les deux lignes de données A_4 et A_7 deviennent basses ensemble.

```
LD      A,11001111B     ; PORT A = PORT DE CONTROLE
OUT     (PIOCRA),A
LD      A,10001000B     ; LIGNES 4, 7 EN ENTREE, AUTRES EN SORTIE
OUT     (PIOCRA),A
LD      A,48H           ; ADRESSE DU VECTEUR = 48 HEXA
OUT     (PIOCRA),A
LD      A,11010111B     ; INTERRUPTION DU PIO AUTORISEE
OUT     (PIOCRA),A
LD      A,01110111B     ; LIGNES 4 ET 7 TRAITEES
OUT     (PIOCRA),A
```

L'octet de contrôle d'interruption a notamment :

- le bit 7 à 1 pour autoriser l'interruption
- le bit 6 à 1 pour générer une interruption seulement si toutes les lignes surveillées sont ou deviennent actives (un ET logique)
- le bit 5 à 0 pour spécifier qu'un '0' logique est un état actif qui doit être traité

• le bit 4 à 1 pour indiquer qu'un octet de masque suit (et pour remettre à zéro les interruptions en attente).

4) Interruption d'un port de contrôle avec un vecteur situé à l'adresse 28_{16}. Une interruption sera générée si une au moins des lignes devient haute.

```
LD      A,11001111B     ; PORT B = PORT DE CONTROLE
OUT     (PIOCRB),A
LD      A,0FFH          ; TOUTES LES LIGNES EN ENTREE
OUT     (PIOCRB),A
LD      A,28H           ; ADRESSE DU VECTEUR = 28 HEXA
OUT     (PIOCRB),A
LD      A,10110111B     ; INTERRUPTIONS AUTORISEES
OUT     (PIOCRB),A
SUB     A               ; SURVEILLANCE DE TOUTES LES LIGNES
OUT     (PIOCRB),A
```

L'octet de contrôle d'interruption a notamment :

• le bit 7 à 1 pour autoriser l'interruption
• le bit 6 à 0 pour générer une interruption si au moins une des lignes devient active (OU logique)
• le bit 5 à 1 pour spécifier qu'un '1' logique est l'état actif qui doit être testé
• le bit 4 à 1 pour indiquer qu'un octet de masque suit (et pour remettre à zéro les interruptions en attenté).

On pourrait utiliser, à l'évidence, une instruction répétitive de sortie par blocs pour raccourcir considérablement ces programmes.

INTERRUPTIONS A PARTIR D'UN PIO « DAISY CHAIN »

Le PIO comme tous les périphériques Z80 comporte également deux signaux d'entrée et de sortie permettant le chaînage en guirlande (« daisy chain »). La sortie $\overline{INT}$ est active — basse lorsque le PIO présente une demande d'interruption. Les signaux d'autorisation sont :

IEI (« Interrupt Enable In ») interruption autorisée en entrée — ce signal est haut si aucun autre périphérique de priorité supérieure n'a son IEO bas.
IEO (« Interrupt Enable Out ») interruption autorisée en sortie. Ce signal est bas si IEI est bas ou si une interruption a été générée ou est en cours de traitement *.
IEI et IEO peuvent servir à constituer une chaîne en « guirlande » (voir volume 1 de « An Introduction to Microcomputer »). Les PIO et les autres périphériques qui sont reliés plus près de l'UC peuvent bloquer les demandes d'interruption à partir des périphériques les plus éloignés de l'UC **. La chaîne en guirlande présente les avantages suivants :

---

* NDT : il existe cependant un cas particulier où IEO remonte bien que le périphérique ait généré une interruption : c'est le cas où IEI est haut et où le 2e octet de l'instruction RETI se trouve sur le bus. Ceci pour permettre à un périphérique de moindre priorité de « reconnaître » l'instruction RETI qui lui est destinée, bien qu'un circuit plus prioritaire tente de générer une interruption (momentanément masquée).

** NDT : ou plus précisément, des périphériques dont les entrées IEI dépendent d'un plus grand nombre d'IEO. (Le plus prioritaire voit sa broche IEI reliée au +5 V).

AVANTAGES ET INCONVÉNIENTS DES INTERRUPTIONS AVEC CHAÎNAGE EN GUIRLANDE

1) Elle identifie chaque source de façon unique.
2) Elle ne demande pas de matériel supplémentaire.
3) Elle est facile à étendre ou à réorganiser par le matériel.
Mais voici ses inconvénients :

1) Elle ne peut évoluer ou être modifiée que par le matériel.
2) Les interruptions de priorité les plus basses peuvent ne jamais être traitées.
3) Elle nécessite plus de temps car les signaux doivent se propager à travers la chaîne.
Le Z80 insère un retard automatique de sorte à permettre une propagation du signal, à travers une chaîne d'au plus quatre périphériques lorsqu'il opère en mode d'interruption 2. Pour des plus longues chaînes, des circuits supplémentaires sont nécessaires.

UTILISATION D'UN PÉRIPHÉRIQUE DANS UNE CHAÎNE EN « GUIRLANDE »

Notons qu'un périphérique particulier dans la chaîne fonctionne de la façon suivante :

1) Il place son vecteur d'interruption sur le bus pendant un cycle d'acquittement d'interruption à condition que :

a) il ait généré une interruption,
b) son entrée IEI soit haute (indiquant qu'aucun circuit de plus haute priorité n'est en cours de service).
IEO est également forcé bas. Dans un PIO, le port A a une plus grande priorité que le port B.
2) Il ne force ensuite son signal IEO en position haute que si une instruction RETI est exécutée pendant que son signal IEI est haut.
Ainsi, un périphérique particulier ne sera servi que s'il présente la demande de priorité la plus haute ; il bloque alors les demandes de priorité inférieure jusqu'à ce que son programme de service soit terminé. Un périphérique peut interrompre le programme de service d'un périphérique de priorité inférieure sans aucune difficulté. Notons qu'une instruction RETI à la fin d'un programme de haute priorité ne sera pas reconnue par un périphérique de priorité inférieure.

INTERRUPTIONS D'UN SIO

INTERRUPTIONS D'UN SIO

Le SIO peut aussi servir de source pour les interruptions. Notons les caractéristiques suivantes des systèmes d'interruption basés sur les SIO :

1) L'interruption du transmetteur est autorisée en positionnant à 1 le bit 1 du registre d'écriture 1 sur chaque canal.
2) Le vecteur d'interruption est modifié par les bits 2, 3 et 4 du registre d'écriture 1 d'après les tableaux 12.2 et 12.3.
3) Le vecteur d'interruption se trouve dans le registre d'écriture 2 sur le canal B uniquement. Il peut être lu dans le registre de lecture 2 (canal B).
4) Le bit D1 du registre de lecture 0 sur le canal A est à 1 si au moins une condition d'interruption est présente dans tout le SIO.

Tableau 12.2. — Vectorisation spéciale des interruptions du SIO (bit 2 du resgistre d'écriture 1 du canal B vaut 1)

Bit de vectorisation d'état (D2) (Canal B seulement).

Si ce bit est à 1, le vecteur renvoyé lors d'un cycle d'acquittement sera variable en fonction de ce qui suit :

	V3	V2	V1	
Canal B	0	0	0	Buffer de transmission vide
	0	0	1	Changement d'état externe
	0	1	0	Caractère reçu disponible
	0	1	1	Condition spéciale en réception
Canal A	1	0	0	Buffer de transmission vide
	1	0	1	Changement d'état externe
	1	1	0	Caractère reçu disponible
	1	1	1	Condition spéciale en réception

* Conditions réception spéciales → Erreur de parité ou overrun en réception ou erreur de CRC ou de trame ou fin de trame (en SDLC).

Si ce bit est à 0, c'est le vecteur programmé dans le registre du vecteur d'interruption qui sera envoyé tel quel.

Tableau 12.3. — Modes d'interruption du SIO en réception (bit 3 et 4 du registre d'écriture 1)

Rec Int Mode 0 (D3), Rec Int Mode 1 (D4)

Les modes d'interruption sur les caractères en réception sont définis comme suit :

D4	D3	
0	0	Interruptions en réception inhibées
0	1	Interruption sur premier caractère reçu seulement
1	0	Interruption sur tous caractères reçus (une erreur de parité modifie le vecteur)
1	1	Interruption sur tous caractères reçus (une erreur de parité ne modifie pas le vecteur)

A l'intérieur d'un SIO, les interruptions du canal A ont priorité sur celles du canal B, les interruptions du récepteur sur celles du transmetteur et les interruptions du transmetteur sur toute autre interruption externe ou d'état.

SYSTEME D'INTERRUPTION PAR SCRUTATION AVEC SIO

Les SIO peuvent être utilisées dans un système d'interruption par scrutation. L'UC doit contrôler l'activité de chaque SIO en examinant le bit 1 du registre de lecture 0 sur le canal A ; c'est-à-dire :

```
SUB     A           ; ACCES AU REGISTRE DE LECTURE 0
OUT     (SIOCRA),A
IN      A,(SIOCRA)  ; ACQUISITION ETAT DU SIO
BIT     1,A         ; INTERRUPTION EN ATTENTE ?
JR      NZ,SERVE    ; OUI, INTERRUPTION ACTIVE
```

Les caractéristiques importantes que présente le système de scrutation du Z80 sont :

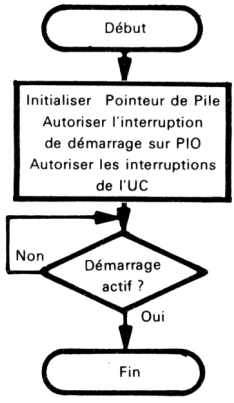
1) La première interruption examinée a la plus haute priorité, puisque les interruptions qui restent ne seront pas examinées tant que la première est active. La seconde interruption a la seconde priorité et ainsi de suite.
2) Le programme de gestion d'interruptions doit mettre à zéro l'interruption du SIO en lisant ou en écrivant le registre approprié même si un transfert de données est inutile autrement.

EXEMPLES D'INTERRUPTIONS

Interruption de démarrage

But : le calculateur attend une interruption du PIO avant de commencer à travailler. De nombreux systèmes restent inactifs tant que l'opérateur ne les initialise pas ou qu'un signal « PRET » n'est pas reçu. Après un « RESET », de tels systèmes doivent initialiser le pointeur de pile, autoriser l'interruption de démarrage et exécuter une instruction « HALT ». Rappelons que le signal RESET inhibe (interdit) les interruptions et qu'à la mise sous tension, les interruptions du PIO sont inhibées. Dans l'organigramme, la décision pour savoir si le démarrage est effectué, est prise par le matériel (c'est-à-dire par les circuits internes de l'UC) plutôt que par le logiciel.

Organigramme :



Programme source :

```
Programme principal :
RESET   EQU     0
        ORG     RESET
        LD      SP,100H         ; PILE EN FIN DE MEMOIRE
        LD      A,01001111B     ; PIO EN MODE D'ENTREE
        OUT     (PIOCRA),A
        LD      A,10000111B     ; AUTORISER INTERRUPTION DU PIO
        OUT     (PIOCRA),A
        EI                      ; AUTORISER INTERRUPTIONS
        HALT                    ; ET ATTENDRE
Programme de service :
        ORG     INTRP
        LD      SP,100H         ; REINITIALISER POINTEUR DE PILE
        JP      DEPART          ; COMMENCER PROGRAMME PRINCIPAL
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
Programme principal :			
0000	31	LD	SP,100H
0001	00		
0002	01		
0003	3E	LD	A,01001111B
0004	4F		
0005	D3	OUT	(PIOCRA),A
0006	PIOCRA		
0007	3E	LD	A,10000111B
0008	87		
0009	D3	OUT	(PIOCRA),A
000A	PIOCRA		
000B	FB	EI	
000C	76	HALT	
Programme de service :			
INTRP	31	LD	SP,100H
INTRP+1	00		
INTRP+2	01		
INTRP+3	C3	JP	DEPART
INTRP+4			
INTRP+5	DEPART :		

Le programme principal doit initialiser le pointeur de pile, puisque la réponse à l'interruption provoque toujours le stockage de l'ancienne valeur du compteur ordinal dans la pile. Ici, le programme initialise simplement le pointeur de pile avant que le véritable programme de départ ne soit exécuté. Une autre solution serait d'incrémenter de 2 le pointeur de pile avant de se brancher au programme de départ. Rappelons que le Z80 est au départ en mode d'interruption 0. Tout autre mode nécessiterait l'utilisation d'une instruction IM.

INTERRUPTIONS SUR CERTAINS ORDINATEURS

L'emplacement mémoire exact du programme de traitement d'interruption varie d'un calculateur à l'autre. Si votre ordinateur n'a pas de moniteur, vous pouvez placer votre programme de traitement là où l'électronique (le matériel), où les tables de vecteur (logiciel) ont accès.

TRAITEMENT D'INTERRUPTION PAR MONITEUR

S'il dispose d'un moniteur, ce dernier occupera vraisemblablement les adresses de traitement du RESET ou des interruptions. Il fournira alors les programmes de traitement ou leurs adresses. L'initialisation par moniteur se fera généralement de la façon suivante :

```
MONIN:  PUSH    HL              ; SAUVEGARDE REGISTRE HL
        LD      HL,USRINT       ; CHARGEMENT ADRESSE DU TRAITEMENT
        JP      (HL)            ; BRANCHEMENT AU PROGRAMME DE TRAITEMENT
```

On doit alors placer l'adresse du programme de traitement dans les mémoires d'adresses USRINT et USRINT + 1, en utilisant comme d'habitude le format Z80, c'est-à-dire les bits de plus faible poids à l'adresse basse. Ne pas oublier que MONIN est une adresse dans le programme moniteur. On peut fort bien inclure le chargement des mémoires USRINT et USRINT + 1 dans le programme principal.

```
        LD      HL,INTRP        ; CHARGEMENT ADRESSE DE DEPART
                                ;
        LD      (USRINT),HL     ; STOCKAGE EN TANT QU'ADRESSE UTILISATEUR
```

Ces instructions doivent être exécutées avant d'autoriser les interruptions.
Dans cet exemple, l'adresse de retour que le Z80 stocke dans la pile n'est pas utile. Cependant, le programme principal doit quand même initialiser le pointeur de pile pour stocker cette adresse quelque part. On peut ne pas utiliser l'instruction LD SP si le moniteur s'occupe de l'initialisation du pointeur de pile.
Le programme principal n'autorise que les interruptions venant du PIO. Le PIO peut, bien entendu, se trouver dans n'importe quel mode. L'interruption est autorisée en mettant à 1 le bit 7 d'un octet de contrôle et en envoyant cet octet sur le port de contrôle du PIO. L'interruption du PIO est autorisée avant que le système global d'interruption ne le soit par l'instruction EI.
Rappelons que le signal RESET ou toute autre interruption inhibe automatiquement le système d'interruption. Ceci permet la configuration de toutes les sources d'interruptions sans être interrompu. Aucune action n'est nécessaire dans le programme de traitement, puisque l'interruption est automatiquement annulée lors du cycle d'acquittement.
La mise en œuvre des instructions EI (Enable Interrupts : autorisation des IT) et DI (Disable Interrupts : masquage des IT) n'est pas la même sur le Z80. DI prend effet dès son exécution, alors qu'EI n'est prise en compte qu'après l'exécution de l'instruction suivante. La raison de tout cela est discutée au chapitre 3, lors de la description de l'instruction EI.

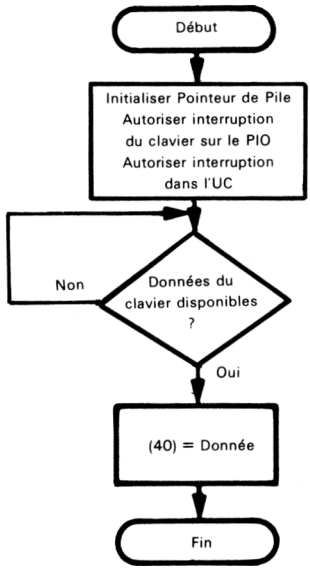
Interruption de clavier

INTERRUPTION DE CLAVIER **But :** l'ordinateur attend une interruption venant d'un clavier et place la donnée venant de ce clavier à l'adresse 0040.

Exemple pratique :

Donnée du clavier = 06
Résultat : (0040) = 06

Organigramme :



Programme source :

Programme principal :

```
RESET   EQU     0
        ORG     RESET
        LD      SP,100H         ; METTRE PILE EN FIN DE MEMOIRE
        LD      A,01001111B     : METTRE PIO EN MODE D'ENTREE
        OUT     (PIOCRA),A
        LD      A,10000111B     ; AUTORISER INTERRUPTIONS SUR PIO
        OUT     (PIOCRA),A
        EI                      ; AUTORISER INTERRUPTIONS DANS L'UC
ICI :   JR      ICI             ; PROGRAMME PRINCIPAL
```

Programme de service :

```
        ORG     INTRP
        EX      AF,AF'          ; SAUVEGARDER ACCUMULATEUR, INDICATEURS
        IN      A,(PIODRA)      ; LIRE DONNEE DU CLAVIER
        LD      (40H),A         ; SAUVEGARDER DONNEE DU CLAVIER
        EX      AF,AF'          ; RESTAURER ACCUMULATEUR, INDICATEURS
        EI                      ; RE-AUTORISER INTERRUPTIONS
        RETI
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)		Instruction (Mnémonique)	
Programme principal :				
0000	31		LD	SP,100H
0001	00			
0002	01			
0003	3E		LD	A,01001111B
0004	4F			
0005	D3		OUT	(PIOCRA),A
0006	PIOCRA			
0007	3E		LD	A,10000111B
0008	87			
0009	D3		OUT	(PIOCRA),A
000A	PIOCRA			
000B	FB		EI	
000C	18	ICI :	JR	ICI
000D	FE			
Programme de service :				
INTRP	08		EX	AF,AF'
INTRP+1	DB		IN	A,(PIODRA)
INTRP+2	PIODRA			
INTRP+3	32		LD	(40H),A
INTRP+4	40			
INTRP+5	00			
INTRP+6	08		EX	AF,AF'
INTRP+7	FB		EI	
INTRP+8	ED		RETI	
INTRP+9	4D			

JR ICI est une instruction de boucle infinie qui est ici utilisée pour représenter le programme principal. Après autorisation des interruptions, ce programme fait ce qu'il a à faire jusqu'à ce qu'une interruption arrive, et reprend son exécution dès que le programme de traitement de l'interruption est terminé.

CHANGEMENT DE L'ADRESSE DE RETOUR

L'instruction RETI redonne la main à l'instruction JR ICI. Si l'on désire éviter cela, on peut simplement incrémenter le compteur ordinal sauvegardé dans la pile, en utilisant par exemple la séquence suivante :

```
EX      (SP),HL     ; CHARGER ADRESSE DE RETOUR
INC     HL          ; INCREMENTER DE 2 L'ADRESSE DE RETOUR
INC     HL
EX      (SP),HL     ; STOCKER ADRESSE DE RETOUR DANS LA PILE
```

L'instruction RETI donnera alors la main à l'instruction suivant le JR. Noter l'utilisation de l'instruction EX (SP),HL ; cette dernière échange les contenus du double registre HL avec le sommet de pile. On peut ainsi modifier l'adresse de retour sans affecter le contenu de HL.

Comme le Z80 ne sauvegarde pas systématiquement ses registres, on peut les utiliser pour passer des paramètres (ou des résultats) entre le programme de traitement d'interruption et le programme interrompu. On peut ainsi laisser la donnée dans l'accumulateur A au

lieu de la stocker à l'adresse 0040. Ceci peut néanmoins s'avérer dangereux et doit donc être évité si (comme c'est le cas généralement) le processeur utilise ses registres pendant l'exécution du programme interrompu ; changer de façon plus ou moins aléatoire le contenu de ces registres lors du traitement de l'interruption entraînerait certainement quelques (graves) problèmes. D'une manière générale, aucun programme de traitement d'interruption ne doit utiliser quelque registre que ce soit avant de l'avoir sauvegardé, sans oublier de le recharger avec sa valeur initiale en fin de traitement.

Notons que l'on doit ré-autoriser les interruptions en fin de traitement de l'interruption, puisque le processeur inhibe les interruptions dès qu'il prend l'une d'elles en compte. Le traitement d'une interruption du PIO désactive le signal d'interruption de telle sorte que la même interruption ne peut être reprise en compte.

SAUVEGARDE DES PRINCIPAUX REGISTRES

Si les programmes de traitement d'interruption ne sont jamais eux-mêmes interrompus (c'est-à-dire qu'il n'y a qu'un niveau d'interruption), les instructions EX AF,AF' et EXX constituent un moyen pratique (et rapide) de sauvegarde et de récupération des registres. EXX échange les contenus des registres BC, DE et HL avec ceux de leurs homologues auxiliaires (respectivement B'C', D'E' et H'L'). Ces instructions ne prennent que 2 octets mémoire et 8 cycles d'horloge. Cette méthode ne peut cependant pas être utilisée s'il existe plusieurs niveaux d'interruption (puisqu'il n'existe qu'une banque de registres auxiliaires), ou si les registres auxiliaires sont eux-mêmes déjà utilisés dans le programme principal (interrompu) ou dans le programme d'interruption.
Une approche plus générale pour sauvegarder des registres consiste à utiliser la pile. PUSH sauvegarde une paire de registres et POP la récupère. Toutefois, PUSH et POP prennent respectivement 11 et 10 cycles d'horloge, ce qui rend cette approche plus lente. De plus, cela implique l'utilisation de la mémoire de pile. L'avantage de cette méthode est qu'elle peut être utilisée indéfiniment (tant qu'il restera de la place pour la pile), car les programmes de traitement d'interruptions imbriquées ne détruiront pas les données sauvegardées au préalable dans la pile.

CHARGEMENT D'UN BUFFER SOUS INTERRUPTION

On peut également se servir des interruptions pour charger un buffer jusqu'à ce qu'un caractère spécial (fin de ligne ou retour chariot) soit reçu. Ce qui conduit à une programmation de type suivant :

Programme principal :

```
RESET   EQU     0
        ORG     RESET
        LD      SP,100H         ; METTRE PILE EN FIN DE MEMOIRE
        LD      A,01001111B     ; METTRE PIO EN MODE D'ENTREE
        OUT     (PIOCRA),A
        LD      A,10000111B     ; AUTORISER INTERRUPTIONS DU PIO
        OUT     (PIOCRA),A
        LD      HL,70H            INITIALISER LE POINTEUR DU BUFFER
        LD      (40H),HL        ; SAUVEGARDER LE POINTEUR DU BUFFER
        EI                      ; AUTORISER L'INTERRUPTION DANS L'UC
ICI :   JR      ICI             ; PROGRAMME PRINCIPAL
```

Programme de service :

```
        ORG     INTRP
        EX      AF,AF'          ; SAUVEGARDER A, INDICATEURS
        EXX                     ; SAUVEGARDER AUTRES REGISTRES
        LD      HL,(40H)        ; LIRE POINTEUR DU BUFFER
        IN      A,(PIODRA)      ; LIRE DONNEES DU CLAVIER
        LD      (HL),A          ; SAUVEGARDER DONNEES DANS BUFFER
        CP      CR              ; DONNEE = CARACTERE RETOUR CHARIOT ?
        JR      Z,FINL          ; OUI, FIN DE LIGNE
        INC     HL              ; NON, INCREMENTATION POINTEUR BUFFER
        LD      (40H),HL
        EXX                     ; RESTAURER AUTRES REGISTRES
        EX      AF,AF'          ; RESTAURER A, INDICATEURS
        EI                      ; RE-AUTORISER LES INTERRUPTIONS
        RETI
FINL :  JP      LPROC           ; LIGNE DE TRAITEMENT SANS INTERRUPTION
```

Quand le processeur reçoit un caractère de RETOUR CHARIOT, il garde les interruptions inhibées pendant qu'il traite la ligne entrée.

DOUBLE BUFFERING

Une autre technique consiste à remplir un second buffer pendant que l'on traite la ligne entrée dans le premier : cette approche est appelée « DOUBLE BUFFERING ».

Le programme de traitement de la ligne commence à l'adresse LPROC, les interruptions étant inhibées, les anciennes valeurs des registres étant dans la banque auxiliaire et l'adresse de retour en sommet de pile.

Dans une application réelle, le microprocesseur peut effectuer d'autres tâches entre 2 interruptions. Il pourrait, par exemple, éditer, sauvegarder ou transmettre une ligne à partir d'un buffer pendant que le deuxième est rempli sous interruption.

Interruption d'imprimante

But : le calculateur attend une interruption venant d'une imprimante et envoie une donnée (caractère) se trouvant à l'adresse 0040 vers l'imprimante.

Exemple pratique :

(0040) = 51H
Résultat : l'imprimante reçoit un 51H (Q ASCII) quand elle est prête.

Organigramme : (voir page 469)

Programme source :

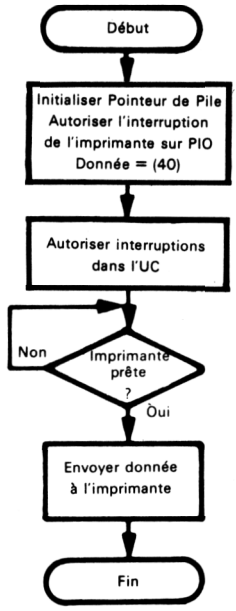
Programme principal :

```
RESET   EQU     0
        ORG     RESET
        LD      SP,100H         ; METTRE PILE EN FIN DE MEMOIRE
        LD      A,00001111B     ; METTRE PIO EN MODE SORTIE
        OUT     (PIOCRA),A
        LD      A,10000111B     ; AUTORISER INTERRUPTIONS DU PIO
        OUT     (PIOCRA),A
        EI                      ; AUTORISER INTERRUPTIONS DANS L'UC
ICI :   JR      ICI             ; PROGRAMME PRINCIPAL
```

Programme de service :

```
        ORG     INTRP
        EX      AF,AF'          ; SAUVEGARDER ACCUMULATEUR, INDICATEURS
        LD      A,(40H)         ; LIRE DONNEES
        OUT     (PIODRA),A      ; ENVOYER DONNEE A L'IMPRIMANTE
        EX      AF,AF'          ; RESTAURER ACCUMULATEUR, INDICATEURS
        EI                      ; RE-AUTORISER LES INTERRUPTIONS
        RETI
```

Organigramme



Programme objet : (voir page 470)

VIDAGE D'UN BUFFER SOUS INTERRUPTION

Ici, comme pour le clavier, l'imprimante peut continuer à envoyer des interruptions jusqu'à ce que la ligne soit entièrement transmise. Ce qui donne :

Programme principal :

```
RESET   EQU     0
        ORG     RESET
        LD      SP,100H         ; METTRE PILE EN FIN DE MEMOIRE
        LD      A,00001111B     ; METTRE PIO EN MODE SORTIE
        OUT     (PIOCRA),A
        LD      A,10000111B     ; AUTORISER INTERRUPTIONS DU PIO
        OUT     (PIOCRA),A
        LD      HL,70H          ; INITIALISER LE POINTEUR DU BUFFER
        LD      (40H),HL        ; SAUVEGARDER POINTEUR DU BUFFER
        EI                      ; AUTORISER INTERRUPTION DANS L'UC
ICI :   JR      ICI             ; PROGRAMME PRINCIPAL
```

Programme de service :

```
        ORG     INTRP
        EX      AF,AF'          ; SAUVEGARDER A, INDICATEURS
        EXX                     ; SAUVEGARDER AUTRES REGISTRES
        LD      HL,(40H)        ; LIRE POINTEUR DU BUFFER
        LD      A,(HL)          ; LIRE UN OCTET DE DONNEE A PARTIR DU BUFFER
        OUT     (PIODRA),A      ; ENVOYER DONNEE SUR IMPRIMANTE
        CP      CR              ; DONNEE = CARACTERE RETOUR CHARIOT ?
        JR      FINL            ; OUI, FIN DE LIGNE
        INC     HL              ; NON, INCREMENTER POINTEUR DU BUFFER
        LD      (40H),HL
        EXX                     ; RESTAURER AUTRES REGISTRES
        EX      AF,AF'          ; RESTAURER A, INDICATEURS
        EI                      ; RE-AUTORISER INTERRUPTIONS
        RETI
FINL :  JP      LCOMP           ; TRAITER LA LIGNE TERMINEE
```

De même, on peut utiliser un « double buffering » pour gérer à la fois les entrées/sorties et le traitement des buffers.

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)		Instruction (Mnémonique)	
Programme principal :				
0000	31		LD	SP,100H
0001	00			
0002	01			
0003	3E		LD	A,00001111B
0004	0F			
0005	D3		OUT	(PIOCRA),A
0006	PIOCRA			
0007	3E		LD	A,10000111B
0008	87			
0009	D3		OUT	(PIOCRA),A
000A	PIOCRA			
000B	FB		EI	
000C	18	ICI :	JR	ICI
000D	FE			
Programme de service :				
INTRP	08		EX	AF,AF'
INTRP+1	3A		LD	A,(40H)
INTRP+2	40			
INTRP+3	00			
INTRP+4	D3		OUT	(PIODRA),A
INTRP+5	PIODRA			
INTRP+6	08		EX	AF,AF'
INTRP+7	FB		EI	
INTRP+8	ED		RETI	
INTRP+9	4D			

Interruption d'horloge temps réel

HORLOGE TEMPS RÉEL

But : le calculateur attend une interruption venant d'une horloge temps réel.

Une horloge temps réel est un simple générateur d'impulsions périodiques. L'intervalle entre 2 impulsions peut ainsi être utilisé comme unité de temps. On peut alors compter ces unités pour produire n'importe quel multiple de ces unités de base. Une horloge temps réel peut être créée à partir de l'horloge de l'UC, d'un compteur-temporisateur externe, tel qu'un CTC pour les ordinateurs à base de Z80, ou à partir d'autres sources externes telles que la tension alternative du secteur à 50 Hz.

FRÉQUENCE D'UNE HORLOGE TEMPS RÉEL

Le compromis sur le choix de la fréquence est le suivant. Une fréquence élevée (10 kHz, par exemple) permet la création d'une vaste gamme d'intervalles de temps avec une grande précision. Par contre, le comptage des interruptions devient fastidieux et pénalisant, et le total pourrait vite excéder la capacité d'un registre 8 bits ou d'une position mémoire. Le choix dépend donc des exigences en temps et précision de votre application. L'horloge peut, évidemment, se composer en partie de matériel : un compteur compte des impulsions à fréquence élevée et n'interrompt le processeur que peu fréquemment, un programme devra lire le compteur pour une mesure du temps plus précise.

SYNCHRONISATION AVEC UNE HORLOGE TEMPS RÉEL

Un problème réside dans la synchronisation des opérations avec l'horloge temps réel. A l'évidence, la précision du cadencement s'en ressentira si l'UC démarre aléatoirement au cours d'une période d'horloge plutôt qu'exactement à son début. Voici quelques façons d'assurer cette synchronisation :

1) Démarrer l'horloge et l'UC simultanément. Un Reset ou une interruption de démarrage peut lancer l'horloge en même temps que l'UC.
2) Autoriser l'UC à lancer et stopper l'horloge sous commande du programme.
3) Utiliser une horloge à fréquence élevée, de façon qu'une erreur de moins d'une période d'horloge soit négligeable.
4) Positionner l'horloge (sur l'arrivée d'un front ou d'une interruption) avant de démarrer une mesure.

PRIORITÉ D'UNE HORLOGE TEMPS RÉEL

Une horloge temps réel doit se voir attribuer un très haut niveau de priorité, puisque la précision des temporisations est affectée par tout retard dans le service de l'interruption. En pratique, on a coutume de lui conférer la priorité la plus haute, avec une seule exception pour la détection d'une défaillance de l'alimentation. La séquence de service de l'interruption d'horloge est généralement très courte, de sorte à ne pas interférer avec les autres fonctions de l'UC.

a) Attente de l'interruption d'une horloge temps réel

Programme source :

Programme principal :

```
RESET   EQU     0
        ORG     RESET
        LD      SP,100H         ; METTRE PILE EN FIN DE MEMOIRE
        LD      A,01001111B     ; METTRE PIO EN MODE D'ENTREE
        OUT     (PIOCRA),A
        LD      A,10000111B     ; AUTORISER INTERRUPTIONS DU PIO
        OUT     (PIOCRA),A
        EI                      ; AUTORISER INTERRUPTIONS DANS L'UC
ICI :   JR      ICI             ; PROGRAMME PRINCIPAL
```

Programme de service :

```
        ORG     INTRP
        HALT                    ; FIN DE L'INTERRUPTION DE L'HORLOGE
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)	
Programme principal :			
0000	31	LD	SP,100H
0001	00		
0002	01		
0003	3E	LD	A,01001111B
0004	4F		
0005	D3	OUT	(PIOCRA),A
0006	PIOCRA		
0007	3E	LD	A,10000111B
0008	87		
0009	D3	OUT	(PIOCRA),A
000A	PIOCRA		
000B	FB	EI	
000C	18	ICI : JR	ICI
000D	FE		
Programme de service :			
INTRP	76		HALT

Le programme de service n'a rien à faire, puisque le traitement de l'interruption du PIO réinitialise automatiquement le signal d'interruption et qu'il n'y a pas de données à recevoir ni à transmettre.
L'interruption de l'horloge est toujours générée sur un front montant si on utilise le signal PIO strobe comme entrée d'horloge.

b) Attente de 10 interruptions de l'horloge temps réel

Programme source :

Programme principal :

```
RESET   EQU     0
        ORG     RESET
        LD      SP,100H         ; METTRE PILE EN FIN DE MEMOIRE
        LD      A,01001111B     ; METTRE PIO EN MODE D'ENTREE
        OUT     (PIOCRA),A
        LD      A,10000111B     ; AUTORISER INTERRUPTIONS DU PIO
        OUT     (PIOCRA),A
        LD      HL,40H          ; COMPTEUR D'HORLOGE = ZERO
        LD      (HL),0
        LD      A,10            ; NOMBRE DE COUPS = 10
        EI                      ; AUTORISER INTERRUPTIONS DE L'UC
ATDIX : CP      (HL)            ; 10 COUPS SONT-ILS PASSES ?
        JR      NZ,ATDIX        ; NON, ATTENDRE
        HALT                    ; OUI, FINI
```

Programme de service :

```
        ORG     INTRP
        EXX                     ; SAUVEGARDER REGISTRES UTILISATEUR
        EX      AF,AF'          ; SAUVEGARDER A, INDICATEURS
        LD      HL,40H          ; INCREMENTER COMPTEUR D'HORLOGE
        INC     (HL)
        EX      AF,AF'          ; RESTAURER A, INDICATEURS
        EXX                     ; RESTAURER REGISTRES UTILISATEUR
        EI                      ; RE-AUTORISER LES INTERRUPTIONS
        RETI
```

Programme objet : (voir page 474)

Une autre solution est d'utiliser la pile pour sauvegarder et restaurer les valeurs des registres. Pour sauvegarder :

```
        PUSH    HL              ; SAUVEGARDER REGISTRES H ET L
        PUSH    AF              ; SAUVEGARDER ACCUMULATEUR ET INDICATEURS
```

Pour restaurer :

```
        POP     AF              ; RESTAURER ACCUMULATEUR ET INDICATEURS
        POP     HL              ; RESTAURER REGISTRES H ET L
```

On peut noter que, si la pile est utilisée, les registres doivent être restaurés dans l'ordre inverse de la sauvegarde. En revanche, l'ordre dans lequel les instructions EXX et EX AF,AF' sont exécutées est indifférent.
Ce programme de service met simplement à jour le compteur qui se trouve à l'adresse 0040. Il est transparent au programme principal.

Programme source :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)		Instruction (Mnémonique)	
Programme principal :				
0000	31		LD	SP,100H
0001	00			
0002	01			
0003	3E		LD	A,01001111B
0004	4F			
0005	D3		OUT	(PIOCRA),A
0006	PIOCRA			
0007	3E		LD	A,10000111B
0008	87			
0009	D3		OUT	(PIOCRA),A
000A	PIOCRA			
000B	21		LD	HL,40H
000C	40			
000D	00			
000E	36		LD	(HL),0
000F	00			
0010	3E		LD	A,10
0011	0A			
0012	FB		EI	
0013	BE	ATDIX :	CP	(HL)
0014	20		JR	NZ,ATDIX
0015	FD			
0016	76		HALT	
Programme de service :				
INTRP	D9		EXX	
INTRP+1	08		EX	AF,AF'
INTRP+2	21		LD	HL,40H
INTRP+3	40			
INTRP+4	00			
INTRP+5	34		INC	(HL)
INTRP+6	08		EX	AF,AF'
INTRP+7	D9		EXX	
INTRP+8	FB		EI	
INTRP+9	ED		RETI	
INTRP+10	4D			

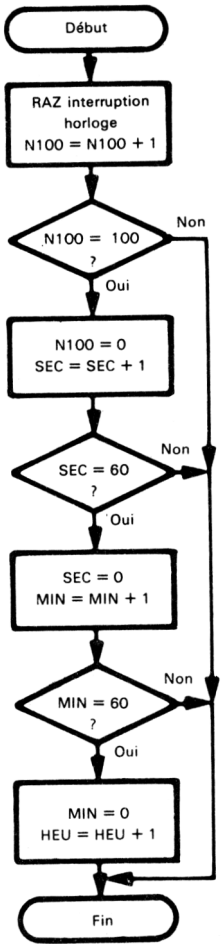
TRAITEMENT DU TEMPS RÉEL

Un programme de service d'horloge plus réaliste effectue la mise à jour du temps réel dans plusieurs emplacements mémoire. Par exemple, le programme suivant utilise les adresses 0040 à 0043 de la façon suivante :

0040 — centièmes de seconde (N100)
0041 — secondes (SEC)
0042 — minutes (MIN)
0043 — heures (HEU)

On suppose que l'horloge du programme est à 100 Hz.

Organigramme :



Programme source :

```
        ORG     INTRP
        PUSH    AF          ; SAUVEGARDE REGISTRES
        PUSH    HL
        LD      HL,40H      ; INCREMENTER CENTIEMES
        INC     (HL)
        LD      A,100
        CP      (HL)        ; RETENUE POUR LES SECONDES ?
        JR      NZ,FINI     ; NON, FINI
        LD      (HL),0      ; OUI, CENTIEMES = 0
        INC     HL          ; INCREMENTER SECONDES
        INC     (HL)
        LD      A,60
        CP      (HL)        ; RETENUE POUR LES MINUTES ?
        JR      NZ,FINI     ; NON, FINI
        LD      (HL),0      ; OUI, SECONDES = 0
        INC     HL          ; INCREMENTER MINUTES
        INC     (HL)
        CP      (HL)        ; RETENUE POUR LES HEURES ?
        JR      NZ,FINI     ; NON, FINI
        LD      (HL),0      ; OUI, MINUTES = 0
        INC     HL          ; INCREMENTER HEURES
        INC     (HL)
        LD      A,24        ; JOUR ACHEVE ?
        JR      NZ,FINI     ; NON, FINI
        LD      (HL),0      ; OUI, HEURES = 0
FINI :  POP     HL          ; RESTAURER REGISTRES
        POP     AF
        EI                  ; RE-AUTORISER INTERRUPTIONS
        RETI
```

Une attente de 300 ms peut maintenant être réalisée dans le programme principal à l'aide du programme suivant :

```
        LD      HL,40H      ; LIRE TEMPS PRESENT (EN CENTIEMES)
        LD      A,(HL)
        ADD     A,30        ; TEMPS DESIRE = 30 COUPS PLUS TARD
        CP      100         ; MODULO 100
        JR      C,AT30
        SUB     100
AT30 :  CP      (HL)        ; TEMPS VOULU ATTEINT ?
        JR      NZ,AT30     ; NON, ATTENDRE
```

Il faut bien noter dans ce programme la différence entre les instructions INC HL et INC (HL). L'instruction INC HL ajoute 1 au contenu (16 bits) du registre HL, alors que INC (HL) ajoute 1 au contenu (8 bits) de la mémoire adressée par HL.
Bien sûr, le programme peut effectuer d'autres tâches et ne vérifier le temps écoulé que de manière occasionnelle. Comment programmeriez-vous un délai de 7 secondes ? de 3 minutes ?
Parfois, on préfère coder le temps écoulé sous forme DCB ou ASCII. Comment modifieriez-vous ce programme pour un tel traitement ?

MASQUAGE DES INTERRUPTIONS

On peut inhiber une interruption d'horloge (ou n'importe quelle autre interruption), lorsqu'on n'en a plus besoin, de l'une des façons suivantes :

1) En exécutant une instruction DI dans le programme principal. Ceci inhibe le système d'interruption dans son ensemble.
2) En mettant à zéro le bit 7 de l'octet de contrôle. Ceci n'inhibe les interruptions que d'un port du PIO.
3) En ne ré-autorisant pas les interruptions lors d'un traitement d'interruption.

Rappelons que l'UC inhibe automatiquement les interruptions dès qu'elle accepte l'une d'entre elles. C'est pourquoi le système d'interruption est inhibé tant que le programme de traitement ne ré-autorise pas les interruptions explicitement. Notons cependant que l'on doit être très prudent lorsque l'on ne ré-autorise pas les interruptions, le programme principal n'étant à priori prévenu en aucune façon que les interruptions sont masquées. D'une manière générale, tout programme de gestion d'interruption doit ré-autoriser les interruptions avant de redonner la main au programme interrompu, tout autre choix signifiant que les programmes de traitement d'interruption ne sont pas transparents au programme principal.

Interruption d'un télé-imprimeur

But : le calculateur attend la réception d'un caractère à partir d'un télé-imprimeur et stocke cette donnée à l'adresse 0040.

TRAITEMENT DES INTERRUPTIONS DU SIO

a) Avec un SIO

(Caractères de 7 bits avec parité impaire et 2 bits de stop).

Programme source :

Programme principal :

```
RESET   EQU     0
        LD      A,4             ; ACCES REGISTRE ECRITURE 4
        OUT     (SIOCRA),A
        LD      A,01000001B     ; MODE HORLOGE x 16, PARITE
        OUT     (SIOCRA),A
        LD      A,3
        OUT     (SIOCRA),A
        LD      A,01000001B     ; CARACTERES 7 BITS, AUTORISATION RECEPTION
        OUT     (SIOCRA),A
        LD      A,1             ; ACCES REGISTRE ECRITURE 1
        OUT     (SIOCRA),A
        LD      A,00011000B     ; AUTORISER INTERRUPTION SUR TOUS CARACTERES REÇUS
        OUT     (SIOCRA),A
        EI                      ; AUTORISER INTERRUPTION DANS UC
HERE:   JR      HERE            ; PROGRAMME PRINCIPAL
```

Programme de service :

```
        ORG     INTRP
        PUSH    AF              ; SAUVEGARDER ACCUMULATEUR, INDICATEURS
        IN      A,(SIODRA)      ; LIRE CARACTERE A PARTIR DU SIO
        LD      (40H),A         ; SAUVEGARDER CARACTERE EN MEMOIRE
        POP     AF              ; RESTAURER ACCUMULATEUR, INDICATEURS
        EI                      ; RE-AUTORISER INTERRUPTIONS
        RETI
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)		Instruction (Mnémonique)	
Programme principal :				
0000	3E		LD	A,4
0001	04			
0002	D3		OUT	(SIOCRA),A
0003	SIOCRA			
0004	3E		LD	A,01000001B
0005	41			
0006	D3		OUT	(SIOCRA),A
0007	SIOCRA			
0008	3E		LD	A,3
0009	03			
000A	D3		OUT	(SIOCRA),A
000B	SIOCRA			
000C	3E		LD	A,01000001B
000D	41			
000E	D3		OUT	(SIOCRA),A
000F	SIOCRA			
0010	3E		LD	A,1
0011	01			
0012	D3		OUT	(SIOCRA),A
0013	SIOCRA			
0014	3E		LD	A,00011000B
0015	18			
0016	D3		OUT	(SIOCRA),A
0017	SIOCRA			
0018	FB		EI	
0019	18	ICI :	JR	ICI
001A	FE			
Programme de service :				
INTRP	F5		PUSH	AF
INTRP+1	DB		IN	A,(SIODRA)
INTRP+2	SIODRA			
INTRP+3	32		LD	(40H),A
INTRP+4	40			
INTRP+5	00			
INTRP+6	F1		POP	AF
INTRP+7	FB		EI	
INTRP+8	ED		RETI	
INTRP+9	4D			

Ce programme de traitement suppose que seule l'interruption en réception d'un canal du SIO a été autorisée. Autrement, on devrait utiliser un encodage vectorisé en chargeant les bits de contrôle D2, D3 et D4 du registre d'écriture 0 (cf. l'utilisation des interruptions du SIO traitées auparavant dans ce chapitre), ou examiner les bits d'états du registre de lecture 0. Les principaux bits d'état sont :

Bit 0 (caractère reçu) : vaut 1 si au moins un caractère est disponible dans un buffer interne.

Bit 1 (interruption générée) : vaut 1 si une quelconque interruption a été générée dans le SIO.

Bit 2 (caractère émis) : vaut 1 si le registre de transmission est vide.

Il serait évidemment beaucoup plus court et plus rapide de configurer le SIO en utilisant une table et l'instruction de transfert de bloc, soit :

```
LD      B,6             ; NOMBRE OCTETS DANS LA CONFIGURATION
LD      C,SIOCRA        ; PORT DE CONTROLE SIO
LD      HL,SIOTBL       ; DEBUT TABLE DE CONFIGURATION
OTIR                    ; CONFIGURER SIO
```

Cette méthode nécessite 9 octets de mémoire programme et 6 octets pour la table, comparés aux 23 utilisés dans l'exemple pour programmer le SIO. Le programme initialise le SIO de la façon suivante :

Registre d'écriture 4 :

Bit 7 = 0 bit 6 = 1 pour diviser l'horloge par 16
Bit 1 = 0 pour sélectionner la parité impaire
Bit 0 = 1 pour autoriser ce calcul de parité

Registre d'écriture 3 :

Bit 7 = 0, bit 6 = 1 pour sélectionner les caractères de 7 bits
Bit 0 = 1 pour le programmer en réception

Registre d'écriture 1 :

Bit 4 = 1, bit 3 = 1 pour générer une interruption pour chaque caractère reçu, les erreurs de parité n'affectant pas le vecteur.

L'UC remet à zéro le bit « caractère reçu » en lisant ce caractère dans le registre de données du SIO. Le bit d'interruption (« INTERRUPT PENDING ») est automatiquement remis à zéro lorsque l'interruption est acquittée.

b) Avec un PIO

INTERRUPTION SUR BIT DE START (Les données sont reçues en série sur le bit 7 du port A du PIO).

Programme source :

Programme principal :

```
        LD      A,11001111B     ; PORT A = PORT DE CONTROLE
        OUT     (PIOCRA),A
        LD      A,10000000B     ; BIT 7 EN ENTREE, AUTRES EN SORTIE
        OUT     (PIOCRA),A
        LD      A,10010111B     ; AUTORISER INTERRUPTION SUR BIT DE START (0)
        OUT     (PIOCRA),A
        LD      A,01111111B     ; MASQUER AUTRES BITS
        OUT     (PIOCRA),A
        EI                      ; AUTORISER INTERRUPTIONS UC
HERE:   JR      HERE            ; PROGRAMME PRINCIPAL
```

Programme de service :

```
ORG     INTRP
PUSH    AF              ; SAUVEGARDER ACCUMULATEUR, INDICATEURS
LD      A,00000111B     : INTERDIRE INTERRUPTION BIT DE START
OUT     (PIOCRA),A
CALL    TTYRCV          ; CHERCHER DONNEE A PARTIR DE TTY
LD      A,10000111B     ; AUTORISER INTERRUPTION BIT DE START
OUT     (PIOCRA),A
POP     AF              ; RESTAURER ACCUMULATEUR, INDICATEURS
EI                      ; RE-AUTORISER INTERRUPTIONS
RETI
```

Programme objet :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)		Instruction (Mnémonique)	
Programme principal :				
0000	3E		LD	A,11001111B
0001	CF			
0002	D3		OUT	(PIOCRA),A
0003	PIOCRA			
0004	3E		LD	A,10000000B
0005	80			
0006	D3		OUT	(PIOCRA),A
0007	PIOCRA			
0008	3E		LD	A,10010111B
0009	97			
000A	D3		OUT	(PIOCRA),A
000B	PIOCRA			
000C	3E		LD	A,01111111B
000D	7F			
000E	D3		OUT	(PIOCRA),A
000F	PIOCRA			
0010	FB		EI	
0011	18	ICI :	JR	ICI
0012	FE			
Programme de service :				
INTRP	F5		PUSH	AF
INTRP+1	3E		LD	A,00000111B
INTRP+2	07			
INTRP+3	D3		OUT	(PIOCRA),A
INTRP+4	PIOCRA			
INTRP+5	CD		CALL	TTYRCV
INTRP+6	TTYRCV			
INTRP+7				
INTRP+8	3E		LD	A,10000111B
INTRP+9	87			
INTRP+10	D3		OUT	(PIOCRA),A
INTRP+11	PIOCRA			
INTRP+12	F1		POP	AF
INTRP+13	FB		EI	
INTRP+14	ED		RETI	
INTRP+15	4D			

Ces programmes supposent que le moniteur initialise le pointeur de pile. Sinon, il doit être initialisé dans le programme principal.
Le sous-programme TTY RCV est le programme de réception décrit au chapitre précédent.
Le front utilisé pour causer l'interruption est ici très important. L'interruption doit en effet se produire lorsque la ligne de transmission passe de l'état 1 (« MARK ») à l'état 0 (« SPACE »), cette transition servant à identifier le début de la transmission.
Le programme de traitement doit inhiber les interruptions venant du PIO, car chaque transition 1→0 causerait une interruption dans le cas contraire. On doit bien sûr ré-autoriser ces interruptions dès qu'un caractère entier a été lu.
Notons l'utilisation du PIO en mode contrôle :

1) Ce mode correspond au mode 3 du PIO.
2) L'octet de contrôle suivant définit les bits utilisés en entrée (1) ou en sortie (0).
3) Cet octet de contrôle a également son bit 7 positionné comme d'habitude.
Le bit 6 vaut 0 pour effectuer le OU logique des différentes lignes (ce qui ici n'est pas utile puisqu'il n'y a qu'une seule ligne en entrée).
Le bit 5 vaut 0 également pour définir l'état actif comme étant l'état bas (pour le bit de start).
Le bit 4 vaut 1 pour indiquer l'envoi du masque dans l'octet suivant.
4) L'octet de contrôle suivant contient le masque d'interruption. Seules les lignes correspondant à des bits nuls seront testées pour la génération d'interruption.
Le résultat de tout ceci est qu'une interruption sera générée si le bit 7 vaut zéro ou passe de l'état 1 à l'état 0. Ici encore, le PIO peut être configuré en utilisant une table et une instruction de transfert par bloc.

PROGRAMMES DE TRAITEMENTS PLUS GÉNÉRAUX

TÂCHES DES PROGRAMMES DE TRAITEMENT GÉNÉRAUX

D'une manière générale, les programmes dans un système entièrement géré par interruptions doivent accomplir les tâches suivantes :

1) Sauvegarder tous les registres utilisés dans la pile, de telle sorte que le programme interrompu puisse reprendre la main correctement.
Rappelons que l'instruction d'empilement du Z80 (« PUSH ») permet la sauvegarde d'un double registre (ou d'un registre d'index) dans la pile. L'instruction PUSH AF (F est le registre d'états) transfère l'accumulateur et les indicateurs dans la pile.
Pour sauvegarder tous les registres dans la pile, on utiliserait la séquence :

```
PUSH    AF          ; SAUVEGARDER ACCUMULATEUR, INDICATEURS
PUSH    BC          ; SAUVEGARDER REGISTRES B,C
PUSH    DE          ; SAUVEGARDER REGISTRES D,E
PUSH    HL          ; SAUVEGARDER REGISTRES H,L
PUSH    IX          ; SAUVEGARDER REGISTRES D'INDEX IX
PUSH    IY          ; SAUVEGARDER REGISTRES D'INDEX IY
EX      AF,AF'
EXX
PUSH    AF          ; SAUVEGARDE ACCUMULATEUR AUXILIAIRE ET INDICATEUR
PUSH    BC          ; SAUVEGARDE REGISTRES AUXILIAIRES B,C
PUSH    DE          ; SAUVEGARDE REGISTRES AUXILIAIRES D,E
PUSH    HL          ; SAUVEGARDE REGISTRES AUXILIAIRES H,L
```

Bien entendu, seuls les registres utilisés dans le programme de traitement d'interruption doivent être sauvegardés.

2) Recharger tous les registres sauvegardés à partir de la pile, avant de retourner au programme interrompu.
Ne pas oublier que les registres doivent être rechargés dans l'ordre inverse de l'ordre de la sauvegarde.

ex. : (PUSH AF / PUSH BC) ⟹ (POP BC / POP AF)

3) Gérer le masquage des interruptions, en se souvenant que l'UC inhibe (masque) automatiquement les interruptions dès qu'elle en prend une en compte.
Plus généralement, les programmes de traitement d'interruption doivent être transparents vis-à-vis des programmes interrompus. Notons également que tout sous-programme utilisé dans un ou plusieurs programmes de traitement d'interruption doit être ré-entrant. Si ce n'est pas le cas, des versions différentes de ce même sous-programme doivent être utilisées [5].

EXERCICES

1) Interruption de test

But : l'UC attend une interruption venant du PIO puis exécute l'instruction :

```
ICI :    JR    ICI        (BOUCLE INFINIE)
```

jusqu'à ce qu'une nouvelle interruption arrive.

2) Interruption de clavier

But : le calculateur attend l'entrée de 4 caractères venant d'un clavier, et les place dans les mémoires d'adresses 0040 à 0043 (le premier en adresse 0040). Chaque caractère envoyé génère une interruption. Le 4e caractère doit également masquer d'éventuelles interruptions suivantes.

Exemple pratique :

```
Données du clavier = 04, 06, 01, 07
Résultat : (0040) = 04
           (0041) = 06
           (0042) = 01
           (0043) = 07
```

3) Interruption d'imprimante :

But : l'ordinateur envoie à l'imprimante 4 caractères stockés dans les mémoires 40 à 43. Chaque caractère est envoyé après une interruption de demande de la part de l'imprimante. Le transfert du 4e octet doit également inhiber de nouvelles interruptions.

4) Interruption d'horloge temps réel

But : au départ, le calculateur met à zéro la mémoire d'adresse 0040, puis la complémente à chaque interruption générée par l'horloge temps réel.
Comment modifier le programme pour que cette complémentation ne se produise que toutes les dix interruptions ? Comment le modifier pour que la mémoire d'adresse 0040 reste à 0 pendant 10 coups d'horloge, puis à FF (hexadécimal) pendant 5 coups d'horloge et ainsi de suite ?
(Utiliser un affichage plutôt que la mémoire 0040 pour un contrôle plus facile.)

5) Interruption d'un télé-imprimeur

But : l'ordinateur reçoit sous interruption une donnée du télé-imprimeur via un SIO et la stocke dans un buffer qui commence à l'adresse 0040. La procédure continue jusqu'à la réception d'un caractère de retour chariot (OD_{16}) également rangé dans le buffer.
On suppose que les données sont des caractères ASCII 7 bits avec parité impaire. Comment modifier le programme pour utiliser un PIO ? Utiliser le sous-programme TTYRCV.

RÉFÉRENCES

1. Pour une révision sur les interruptions, se reporter au volume 1 de « An Introduction to Microcomputer ».
2. Sur le traitement de la conception des interruptions, voir de R.L. Baldridge, « Interrupts Add Power, Complexity to Microcomputer System Design ». EDN, 5 août 1969, pp. 67.73.
3. Se reporter au volume 2 de « An Introduction to Microcomputer ».
4. Voir « An Introduction to Microcomputers, Volume 2 et 8080A/8085 Assembly Language Programming ».
5. Pour un traitement plus poussé et des exemples pratiques des systèmes à partir de Z80 avec interruptions, consulter « Z80 Programming for Logic Design » de la page 5-24 à 5-37, ainsi que :
   Baldridge R.L. « Interrupts Add Power, Complexity to Microcomputer System Design », EDN, 5 août 1977, pp. 67-73.
   Pond R.M. « Let Microprocessors Communicate », Electronic Design, 8 novembre 1977, pp. 88-90.
   Shima M. and R. Blacksher, « Improved Microprocessor Interrupt Capability », Electronic Design, 26 avril 1978, pp. 96-100.
   Weller W.J., Practical Microcomputer Programming : the Z80 », Northern Technology Books, Evanston III, 1978.
   Winston A.W. and T.B. Smith, « use of the Z-80 in Data Collection and Control », IECI 78 Proceedings - Industrial Applications of Microprocessors, 20-22 mars 1978, pp. 208-214.

CHAPITRE 13

DÉFINITION DES PROBLÈMES ET CONCEPTION DES PROGRAMMES

FONCTIONS DU LOGICIEL DE DÉVELOPPEMENT

Nous nous sommes concentrés, dans les chapitres précédents, sur la rédaction de programmes courts en langage d'assemblage. Bien qu'il s'agisse là d'un thème important, il ne constitue qu'une faible partie du problème global du développement logiciel. L'écriture de programmes en assembleur, si elle semble constituer la tâche majeure au débutant, n'en devient pas moins et très vite relativement simple. Maintenant, vous devriez être familiarisé avec la plupart des méthodes standards de programmation en assembleur avec le Z80. **Les quatre prochains chapitres vont décrire comment formuler les tâches sous forme de programmes et comment réunir de courts programmes en un système exploitable.**

ÉTAPES DU DÉVELOPPEMENT DU LOGICIEL

Le développement d'un logiciel passe par plusieurs étapes. La figure 13-1 est l'organigramme du processus ; **ces étapes sont :**

- Définition du problème.
- Conception du programme.
- Codage.
- Mise au point (« debugging » ou « *déverminage* » !).
- Test.
- Documentation.
- Maintenance et reconception.

Chacune de ces étapes est importante dans la réalisation d'un système exploitable. Vous remarquerez que le codage, c'est-à-dire la rédaction du programme dans une forme compréhensible par l'ordinateur, n'est qu'une étape parmi sept.

IMPORTANCE RELATIVE DU CODAGE

En fait, **le codage constitue généralement l'étape la plus facile à définir et à exécuter.** Les règles de rédaction des programmes sont aisées à apprendre. Elles varient quelque peu d'un ordinateur à l'autre mais les techniques de base restent les mêmes. Peu de projets se heurtent à des problèmes de codage ; le codage n'est pas, en fait, l'étape la plus significative dans le développement du logiciel. Les experts estiment qu'un programmeur peut rédiger de une à dix instructions documentées et mises au point à 100 % par jour. En fait, le simple codage de une à dix instructions par jour ne justifie pas le travail d'une journée. Dans la plupart des projets, le codage occupe moins de 25 % du temps du programmeur.

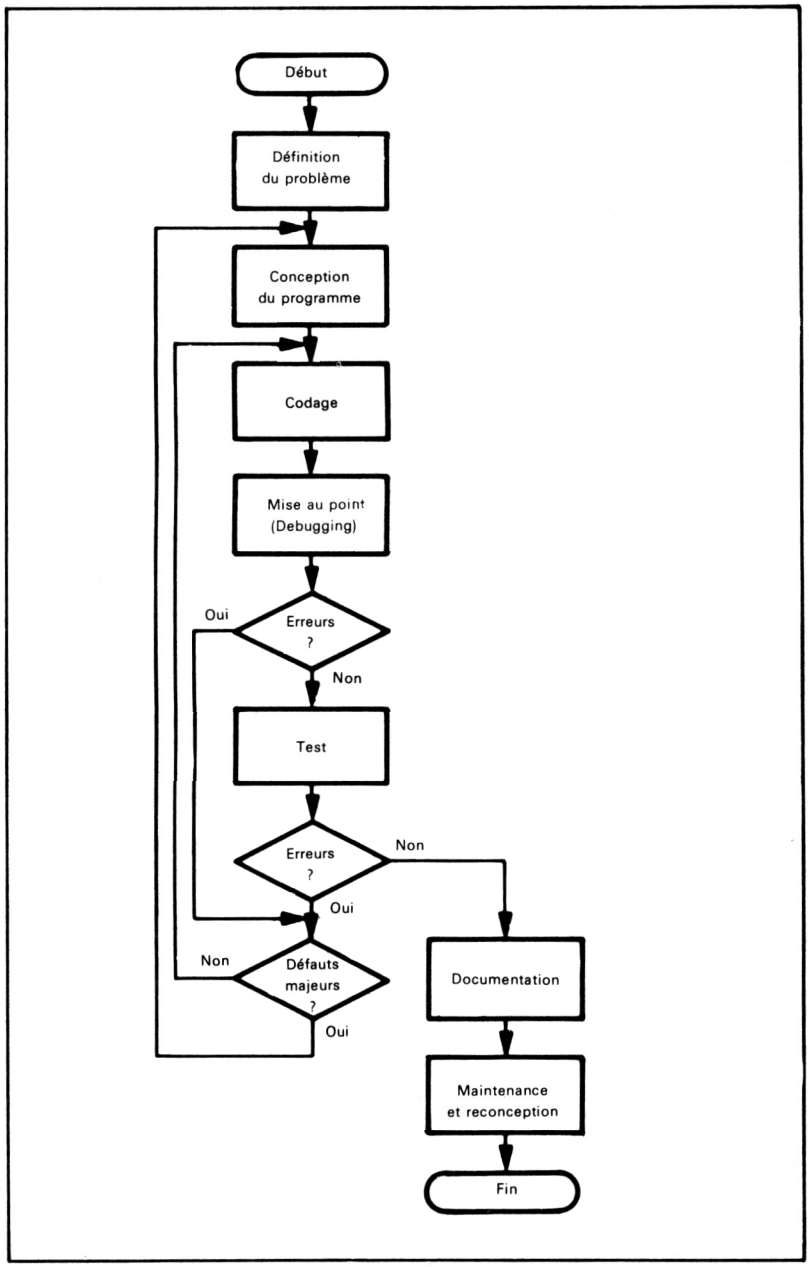


Fig. 13.1. — Organigramme du développement d'un logiciel.

PROGRESSION DANS LES ÉTAPES

Mesurer la progression dans les autres étapes est difficile. On peut affirmer que la moitié d'un programme a déjà été rédigée, mais on ne peut guère assurer que la moitié des erreurs a été éliminée, ou que la moitié du problème a été définie. Des plannings pour des étapes telles que la conception du programme, sa mise au point, ou son test sont difficiles à élaborer. Des jours ou des semaines d'efforts ne se traduiront parfois par aucun progrès. En outre, une étape incomplètement traitée pourra donner lieu à des problèmes ultérieurs énormes. Par exemple, la mauvaise définition d'un programme peut rendre son test et son débugging très difficiles. Le temps économisé dans une étape peut provoquer une dépense de temps plusieurs fois supérieure dans une autre.

DÉFINITION DES ÉTAPES

DÉFINITION DES PROBLÈMES

La définition des problèmes consiste à formuler les tâches à accomplir en termes de besoins de l'ordinateur. Par exemple, qu'est-ce qui sera nécessaire pour qu'un ordinateur commande un outil, dirige une série de tests électriques ou organise des communications entre un contrôleur central et un terminal ? La définition des problèmes demande que l'on détermine la forme et le rythme des entrées et des sorties, l'importance et la vitesse du traitement, le risque d'erreurs et leur traitement. La définition des problèmes représente en gros l'idée de construction d'un système contrôlé par ordinateur et définit les tâches et les besoins de l'ordinateur.

CONCEPTION DU PROGRAMME

Concevoir un programme signifie tracer ses lignes de façon que le programme exécute les tâches qui lui sont assignées. A cette étape, ces tâches sont décrites d'une manière facile à convertir en instructions. **Parmi les techniques usuelles, on trouve l'usage de l'organigramme, de la programmation structurée, de la programmation modulaire, et de la conception verticale descendante (« top-down »).**

CODAGE

Le codage consiste à rédiger le programme sous une forme que l'ordinateur peut directement comprendre et traduire. Cette forme peut être le langage machine, l'assembleur ou un langage haut niveau.

MISE AU POINT

La mise au point, appelée aussi *debugging*, ou déverminage (*), consiste à rendre le programme en mesure d'exécuter les tâches prescrites. On utilisera ici des outils tels que des points d'arrêt, la technique de trace, les simulateurs, les analyseurs logiques ou les émulateurs en ligne. La fin du debugging est difficile à déterminer car on ne sait jamais si l'on vient de détecter la dernière erreur.

(*) Dans le jargon des laboratoires, le mot « debugging » avec ses variantes a acquis droit de cité, et pourrait bien s'imposer. Aussi le trouvera-t-on souvent en synonyme de « mise au point ».

TEST **Le test,** ou validation du programme, **assure que le programme exécute l'ensemble des tâches correctement.** Le concepteur se servira de simulateurs, d'un émulateur, ou des techniques statistiques variées lui procurant quelque mesure des performances du programme.

DOCUMENTATION **La documentation est la description d'un programme destinée aux utilisateurs et au personnel de maintenance.** La documentation permet aussi au concepteur de créer une bibliothèque de programmes afin de simplifier la mise au point des tâches ultérieures. Les organigrammes, commentaires, tables des symboles et manuels d'utilisation seront autant d'outils précieux.

MAINTENANCE ET RECONCEPTION **La maintenance et la reconception servent à améliorer et développer le programme.** En fait, le concepteur doit être prêt à résoudre les problèmes qui se poseraient sur le terrain. Des modes de diagnostics ou des programmes spéciaux, ainsi que divers autres outils de maintenance, peuvent être nécessaires. L'amélioration et l'extension des programmes s'imposera lorsque les conditions changeront ou lorsqu'on voudra inclure de nouvelles tâches.

Le reste de ce chapitre va prendre en compte les seuls problèmes de la définition et de la conception. Le chapitre 14 discutera du debugging et du test, et le chapitre 15 de la documentation, des extensions et de la reconception. On regroupera tout cela à travers quelques exemples dans le chapitre 16.

DÉFINITION DES PROBLÈMES

Les fonctions typiques des microprocesseurs demandent quantité de définitions. Par exemple, que doit être un programme commandant une balance, une caisse enregistreuse ou un générateur de signaux ? Manifestement, la route est longue pour arriver simplement à définir l'ensemble des tâches impliquées.

DÉFINITION DES ENTRÉES

Comment démarrer une définition ? Le plus évident consiste à commencer par les entrées. On devra tout d'abord établir la liste des entrées dont l'ordinateur disposera pour telle application. Des exemples d'entrées sont :

- Des blocs de données venant de lignes de transmission.
- Des mots d'états issus de périphériques.
- Des données provenant de convertisseurs A/D.

FACTEURS LIÉS AUX ENTRÉES **Dans ce cas, on pourra se poser les questions suivantes :**

1) Sous quelle forme se présentent ces entrées, c'est-à-dire quels signaux l'ordinateur va-t-il recevoir ?
2) Quand l'entrée est-elle disponible et comment le processeur le saura-t-il ? Devra-t-il interroger l'entrée avec un signal d'échantillonnage (strobe) ? L'entrée fournit-elle sa propre horloge ?

3) Combien de temps reste-t-elle disponible ?
4) Change-t-elle souvent ? Comment le processeur saura-t-il qu'elle a changé ?
5) L'entrée consiste-t-elle en une séquence ou un bloc de données ? L'ordre est-il important ?
6) Que faire si les données sont entachées d'erreurs ? Celles-ci comprennent les erreurs de transmissions, des données incorrectes, des erreurs de séquencement, des données superflues, etc.
7) L'entrée est-elle dépendante d'autres entrées ou sorties ?

DÉFINITION DES SORTIES

L'étape suivante consiste à définir les sorties. **On dressera la liste des sorties que l'ordinateur doit produire.** Celles-ci comprennent, par exemple :

- Des blocs de données pour lignes de transmission.
- Des mots de commandes pour périphériques.
- Des données pour convertisseurs D/A.

On pourra alors se poser les questions suivantes :
1) Sous quelle forme se présentent ces informations, c'est-à-dire quels signaux l'ordinateur doit-il générer ?
2) Quand ces sorties doivent-elles être disponibles, et comment le périphérique saura-t-il qu'elles sont à sa disposition ?
3) Combien de temps doivent-elles rester disponibles ?
4) Changeront-elles souvent ? Comment le périphérique saura-t-il qu'elles ont varié ?
5) Y a-t-il une séquence de sorties ? L'ordre est-il important ?
6) Que faire pour éviter des erreurs de transmission ou pour détecter des défaillances des périphériques ?
7) Une sortie est-elle dépendante d'autres sorties ou d'entrées ?

SECTION DE TRAITEMENT

FACTEURS LIÉS AU TRAITEMENT

Entre la lecture des entrées et l'émission des résultats en sorties intervient le traitement. **On doit déterminer exactement comment le processeur doit traiter les entrées. Les questions sont alors :**

1) Quelle est la procédure de base (l'algorithme) transformant les données entrées en résultats sortis ?
2) Quelles sont les contraintes de temps ? Elles peuvent comprendre les rythmes de transfert, les temporisations, les constantes de temps des dispositifs d'entrées ou de sorties, etc.
3) Quelles sont les contraintes au niveau des mémoires ? Est-on limité en mémoire de programme, de données, ou sur la taille des buffers ?
4) A quels programmes standards ou tables devra-t-on recourir ? Quelles sont leurs exigences ?

5) Y a-t-il des cas particuliers, et comment devront-ils être traités ?
6) Quelle est la précision requise des résultats ?
7) Comment le programme doit-il traiter les erreurs d'exécution ou les conditions spéciales telles que débordement, sous-emploi, ou perte de signification ?

TRAITEMENT DES ERREURS

CONSIDÉRATIONS SUR LES ERREURS

Dans bien des applications, un facteur important réside dans le traitement des erreurs. Le concepteur devra se réserver les moyens de répondre aux erreurs les plus communes et de diagnostiquer les mauvais fonctionnements. **Parmi les questions qui surgissent se trouvent les suivantes :**

1) Quelles erreurs peuvent survenir ?
2) Quelles sont les plus probables ? Si le système est servi par un opérateur, les erreurs humaines seront les plus probables. Après, on trouvera les erreurs de communications ou de transmission, plus communes que celles dues aux dispositifs mécaniques, électriques, mathématiques, ou au processeur.
3) Quelles erreurs n'apparaîtront pas immédiatement évidentes aux systèmes comme telles ? Un problème particulier s'attache aux erreurs que ni le système, ni l'opérateur, ne reconnaîtront alors.
4) Comment le système se sortira-t-il des erreurs avec le minimum de perte de temps et de données, tout en signalant qu'une erreur s'est produite ?
5) Quelles erreurs se manifesteront de façon semblable ? Comment pourra-t-on les distinguer aux fins d'un diagnostic ?
6) Quelles erreurs demanderont une procédure spéciale au niveau du système ? Par exemple, une erreur de parité exigera-t-elle qu'on retransmette la donnée ?
Une autre question est la suivante : comment le technicien de maintenance après-vente pourra-t-il systématiquement déceler la source d'un mauvais fonctionnement sans être un expert ? On pourra s'aider ici de programmes de tests incorporés de diagnostic ou de l'analyse de signature [1].

FACTEURS HUMAINS

INTERACTION AVEC L'OPÉRATEUR

La plupart des systèmes à microprocesseur impliquent une interaction avec des opérateurs humains. **Les facteurs humains doivent donc être pris en considération tout au long du processus de développement. Les questions qui se posent au concepteur sont, entre autres :**

1) Quelle procédure d'entrée est la plus naturelle à l'opérateur ?
2) L'opérateur pourra-t-il facilement déterminer comment commencer, poursuivre ou terminer une opération d'entrée ?
3) Comment sera-t-il informé d'une erreur de procédure ou d'un mauvais fonctionnement de l'équipement ?
4) Quelles sont les erreurs les plus probables qu'il risque de commettre ?

5) Comment saura-t-il qu'il a introduit les données correctement ?
6) La forme des affichages constituera-elle un bon moyen d'échange (lecture aisée, compréhension facile) ?
7) La réponse du système sera-t-elle bien perçue par l'opérateur ?
8) L'opérateur exploitera-t-il aisément le système ?
9) Le système offre-t-il à l'opérateur débutant un moyen de se guider ?
10) Offre-t-il des raccourcis ou des options raisonnables à un opérateur expérimenté ?
11) L'opérateur pourra-t-il toujours déterminer ou réinitialiser l'état du système après des interruptions ou des distractions ?

Bâtir un système exploité par des opérateurs est difficile. Le microprocesseur peut le rendre plus puissant, plus flexible et plus efficace. Cependant, le concepteur devra toujours imaginer les diverses touches qui le rendront plus utile, plus attrayant et plus efficace pour l'opérateur [2].

EXEMPLES

Réponse à un interrupteur

DÉFINITION D'UN SYSTÈME À INTERRUPTEUR ET VOYANT

La figure 13.2 montre un système simple dans lequel l'entrée provient d'un interrupteur SPST et la sortie se fait sur une LED. En réponse à la fermeture de l'interrupteur, le processeur allume le voyant pendant une seconde. Le système devrait être facile à définir.

ENTRÉE

Examinons tout d'abord l'entrée, et répondons aux questions précédentes :

1) L'entrée est un bit unique pouvant être 0 (interrupteur fermé) ou 1 (interrupteur ouvert).
2) L'entrée est toujours disponible et n'a pas besoin d'être activée.
3) L'entrée est disponible pendant au moins plusieurs millisecondes après la fermeture.
4) L'entrée ne changera que rarement plus d'une fois en quelques secondes. Le processeur doit uniquement éliminer les rebonds du contact. Il doit surveiller cet interrupteur pour déterminer l'instant de sa fermeture.
5) Il n'y a pas de séquence d'entrée.
6) Les erreurs d'entrée manifestes consistent en une panne de l'interrupteur, une coupure du circuit d'entrée, ou une tentative de l'opérateur pour fermer à nouveau l'interrupteur avant qu'un certain temps ne se soit écoulé. Nous y reviendrons.
7) L'entrée est indépendante des autres entrées et sorties.

INTERRUPTEUR ET SORTIE SUR LED

Ensuite, on va examiner les sorties du système. La réponse aux questions précédentes est :

1) La sortie est un unique bit, à 0 pour allumer le voyant, à 1 pour l'éteindre.
2) Il n'y a pas de contraintes de temps sur la sortie. Le périphérique n'a besoin d'aucune information sur la disponibilité des données.

3) La sortie étant une LED, la donnée doit être disponible pendant quelques millisecondes à une fréquence d'environ 100 Hertz. L'observateur verra une lumière continue.
4) La donnée doit changer (s'effacer) après une seconde.
5) Il n'y a pas de séquence en sortie.
6) Les erreurs de sortie possibles sont une défaillance de la LED ou du circuit de sortie.
7) La sortie dépend uniquement de l'interrupteur d'entrée et du temps.
La section traitement est extrêmement simple. Dès que l'entrée passe au zéro logique, l'UC allume le voyant en sortant un zéro logique pendant une seconde. Il n'y a aucune contrainte de mémoire ou de temps.

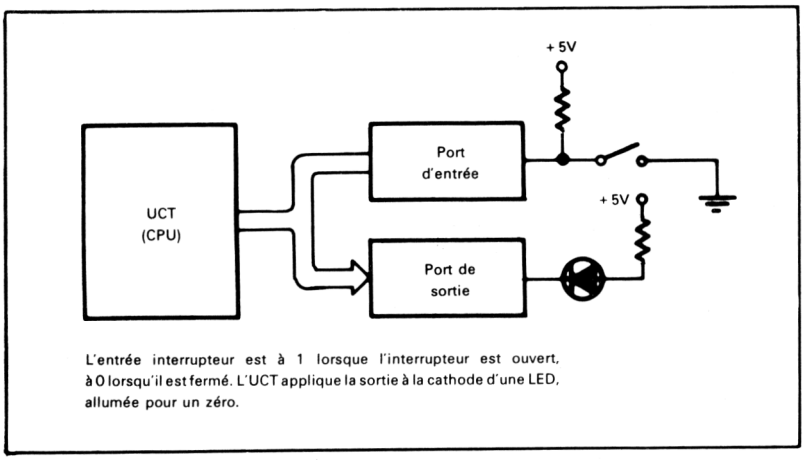


Fig. 13.2. — Le système à interrupteur et voyant.

TRAITEMENT DES ERREURS **Examinons les erreurs et les mauvais fonctionnements possibles.** En voici les causes :

- Une autre fermeture de l'interrupteur avant qu'une seconde ne se soit écoulée.
- Une défaillance de l'interrupteur.
- Une défaillance de l'affichage.
- Une défaillance de l'ordinateur.

La première paraît la plus probable. La solution la plus simple consistera, pour le processeur, à ignorer toute fermeture du contact se produisant avant l'écoulement d'une seconde. La brève période sans réponse sera difficilement remarquée par un opérateur. De plus, ignorer l'interrupteur pendant cette période signifie qu'aucun montage ou logiciel anti-rebonds n'est nécessaire, puisque le système ne réagira plus à une oscillation, de toute façon.
Pratiquement, les trois dernières défaillances peuvent mener à des résultats imprévisibles. L'affichage peut rester allumé, éteint, ou changer d'état aléatoirement. Quelques méthodes pour isoler la cause peuvent consister en :

- Un test matériel de voyant, par exemple un bouton allumant le voyant indépendamment du processeur.
- Une connexion directe à l'interrupteur pour tester son fonctionnement.
- Un programme de diagnostic qui essaie les circuits d'entrée et de sortie.

Si tout à la fois l'interrupteur et le voyant sont en bon état de fonctionnement, c'est donc l'ordinateur qui est fautif. Un technicien de maintenance, équipé du matériel approprié, pourra déterminer sur place la cause de la panne.

Chargeur mémoire à interrupteurs

DÉFINITION D'UN CHARGEUR MÉMOIRE À INTERRUPTEURS

La figure 13.3 montre un système permettant à un utilisateur d'entrer des données dans n'importe quelle case mémoire d'un micro-ordinateur. Un port d'entrée, le DPORT, lit les données fournies par 8 interrupteurs. L'autre port, le CPORT, est employé pour lire l'information de commande ; il comporte trois poussoirs : Adresse haute, Adresse basse, et Données. La sortie est la valeur de la dernière entrée exécutée à partir des interrupteurs de données et est réinitialisé par huit LEDs.

Le système requiert aussi quelques résistances, buffers et drivers, bien entendu.

Nous allons tout d'abord examiner les entrées. Les caractéristiques des interrupteurs sont les mêmes que dans l'exemple ci-dessous ; cependant, il y a maintenant une séquence spécifique d'entrées :

1) L'opérateur forme l'octet de poids fort de l'adresse avec les interrupteurs de données, puis :
2) il enfonce le poussoir Adresse haute. L'octet de poids fort est visualisé par les LEDs et le programme interprète cet octet comme l'octet le plus significatif de l'adresse.
3) L'opérateur forme de la même façon l'octet de poids faible de l'adresse et :
4) Il enfonce le poussoir Adresse basse. Cette adresse est visualisée par les LEDs et le programme l'interprète comme l'octet de poids faible de l'adresse.
5) Finalement, l'opérateur forme la donnée avec les interrupteurs et,
6) Il presse le poussoir Données. Les LEDs la visualisent et le programme la range en mémoire à l'adresse imposée ci-dessus.

L'opérateur peut répéter cette séquence jusqu'à entrer un programme entier. Même dans cette situation simplifiée, on s'aperçoit qu'on a plusieurs séquences possibles à prendre en considération. Comment va-t-on faire face aux les séquences erronées et rendre le système facile d'emploi ?

La sortie ne constitue pas un problème. Après chaque entrée, le programme envoie aux afficheurs le complément de celle-ci (puisque les LEDs sont ici actives au niveau bas). La sortie ne change pas jusqu'à l'opération d'entrée suivante.

TRAITEMENT DES ERREURS AU CHARGEMENT

Le processus de traitement reste tout à fait simple. Il n'y a aucune contrainte de temps ou de mémoire. Le programme peut éliminer les rebonds des contacts mécaniques en temporisant quelques millisecondes, et doit fournir des données complémentées aux voyants.

Les erreurs les plus probables sont celles dues à l'opérateur. Elles comprennent :

- Des entrées incorrectes.
- Un ordre non respecté.
- Des entrées incomplètes ; par exemple, l'oubli d'entrer la donnée.

Le système doit s'avérer capable de gérer ces problèmes de façon raisonnable, puisqu'ils sont probables.

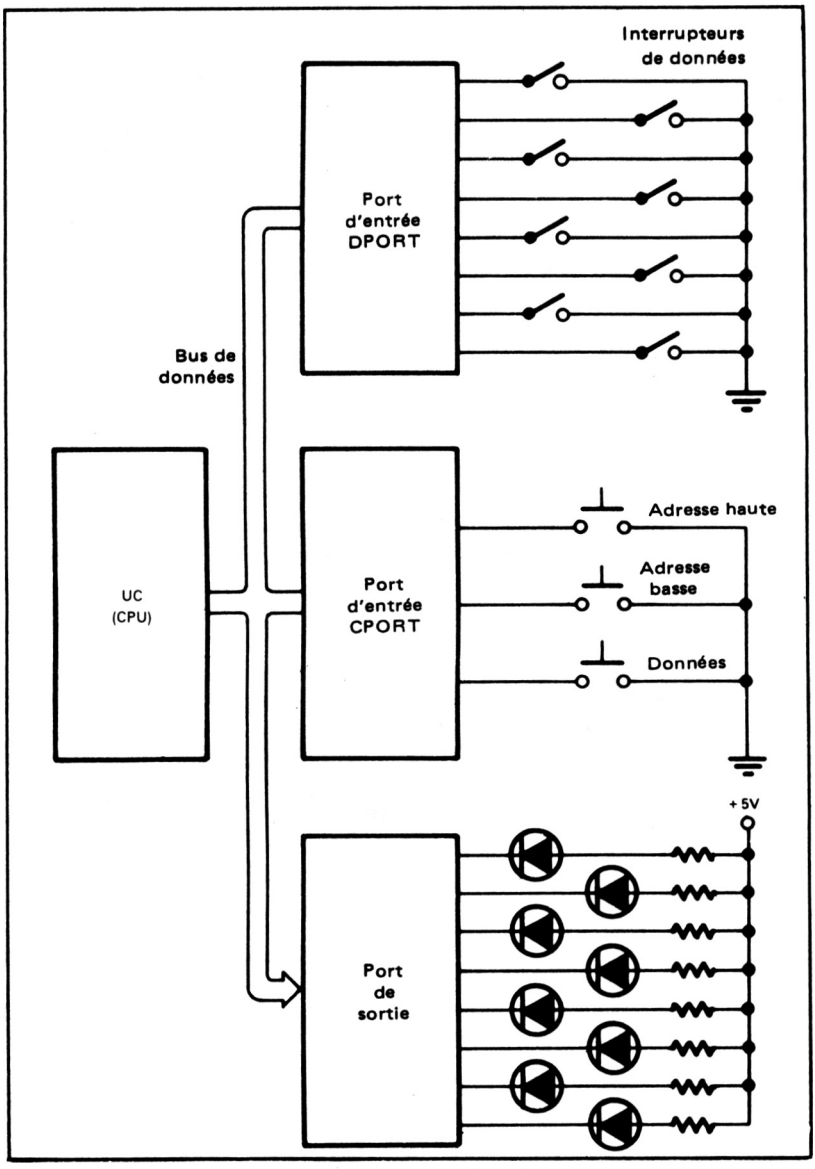


Fig. 13.3. — Le chargeur mémoire à interrupteurs.

Le concepteur doit aussi imaginer les défaillances possibles de l'équipement et leurs effets. Comme précédemment, les difficultés peuvent être :

- Une défaillance d'un interrupteur (ou de plusieurs).
- Une défaillance de l'affichage.
- Une défaillance de l'ordinateur.

On doit cependant accorder davantage d'attention, ici, à la façon dont ces défaillances affectent le système. Une panne de l'ordinateur amènera probablement le système à se comporter de façon inhabituelle, et sera facile à détecter. Une panne de l'affichage pourra ne pas être immédiatement perçue : ici, un dispositif de test des voyants permettra à l'opérateur d'en vérifier le bon fonctionnement. On préférera sans doute tester chaque LED séparément, afin de diagnostiquer un éventuel court-circuit de lignes de sortie. Par ailleurs, l'opérateur pourra aussi ne pas s'apercevoir immédiatement de la défaillance d'un interrupteur ; cependant, il ne tardera pas à remarquer une panne et repérera quel interrupteur en est responsable par un simple processus d'élimination.

CORRECTION DES ERREURS DE L'OPÉRATEUR

Voyons quelques types d'erreurs dues à l'opérateur :

- Les données sont erronées.
- L'ordre des entrées de la manœuvre des interrupteurs n'est pas le bon.
- On tente de passer à l'entrée suivante avant d'en avoir terminé avec la précédente.

L'opérateur remarquera probablement des données erronées dès que celles-ci seront affichées. Quelle pourrait alors être la procédure de correction à suivre ? Des options possibles sont :

1) L'opérateur termine la procédure d'entrée ; par exemple, il introduit l'adresse basse et les données si l'erreur s'est produite avec l'adresse haute. Manifestement, il y a là un gaspillage et cette procédure ne pourra qu'ennuyer l'opérateur.
2) L'opérateur redémarre la séquence en retournant à l'entrée de l'adresse haute. Cette solution est applicable lorsque l'erreur s'est produite à cette étape, mais elle oblige à tout recommencer si l'erreur est intervenue avec l'adresse basse ou la donnée.
3) L'opérateur peut redémarrer en n'importe quel endroit de la séquence et à tout moment, simplement en positionnant les interrupteurs et en actionnant le bon poussoir. Une telle procédure lui permet d'effectuer des corrections en tous points de la séquence. Cette procédure devra toujours être préférée à celles qui ne permettent pas les corrections immédiates d'erreurs, offrent plusieurs variantes d'exécution, ou entrent des données dans le système sans que l'opérateur dispose d'un dernier test. Toute complication du matériel ou du logiciel trouvera sa justification dans une efficacité accrue de l'opérateur. On doit toujours confier au micro-ordinateur les tâches ennuyeuses et le charger de reconnaître les séquences arbitraires ; il ne se fatigue jamais et n'oublie jamais ce qui était indiqué dans le manuel d'utilisation.
Une autre caractéristique intéressante consisterait en des voyants d'états indiquant la signification de l'affichage. Trois voyants d'états, marqués Adresse haute, Adresse basse, et Données, permettraient à l'opérateur de savoir où il en est sans avoir à se souvenir du dernier poussoir actionné. Le processeur piloterait cette séquence mais la complication du logiciel simplifierait alors le travail de l'opérateur. En pratique, trois jeux séparés d'afficheurs, plus la possibilité d'examiner une position mémoire, seraient encore plus utiles à l'opérateur.
On notera que, bien qu'on ait insisté sur les réactions humaines, celles de la machine ou du système présenteront nombre de caractéristiques semblables. Le microprocesseur exécutera le travail. Si, en compliquant ses tâches, on rend la réponse aux erreurs plus simples et la cause des défaillances évidente, le système complet fonctionnera mieux et sera

plus aisé à entretenir. On ne doit d'ailleurs pas remettre les considérations relatives à l'usage du système et à sa maintenance à une étape ultérieure mais les inclure immédiatement dans l'étape du démarrage, celle de définition des problèmes.

Terminal de vérification

DÉFINITION DU TERMINAL DE VÉRIFICATION

La figure 13.4 est le synoptique d'un petit terminal de vérification de cartes de crédit. Un port d'entrées prend les données d'un clavier (fig. 13.5) ; l'autre reçoit celles provenant d'une ligne de transmission fournissant les informations à vérifier. Un port de sorties alimente un jeu d'afficheurs (fig. 13.6), l'autre envoie le numéro de la carte de crédit à l'ordinateur central. Un troisième allume un voyant lorsque le terminal est prêt à accepter une demande, et un autre lorsque l'opérateur émet l'information. Le voyant « Occupé » s'éteint lorsque la réponse revient. Ainsi, les entrées et les sorties de données vont être ici plus complexes que dans les cas précédents, bien que le processus de traitement reste encore relativement simple.
Des afficheurs supplémentaires pourront être utiles pour confirmer la signification de la réponse. Nombre de terminaux utilisent un voyant vert pour « Oui » ; un rouge pour « Non », et un jaune pour « Consulter le gérant » ; ces voyants devront voir leurs fonctions explicitement indiquées, dans le cas où l'opérateur ne distingue pas les couleurs.

ENTRÉES DU TERMINAL DE VÉRIFICATION

Examinons d'abord les entrées clavier. Elles sont différentes des entrées provenant d'interrupteurs puisque l'UC doit pouvoir distinguer de nouvelles données. **On supposera que chaque action sur une touche se traduit par un code hexadécimal (on peut donc coder les 12 touches présentes avec un seul chiffre) et par l'émission**

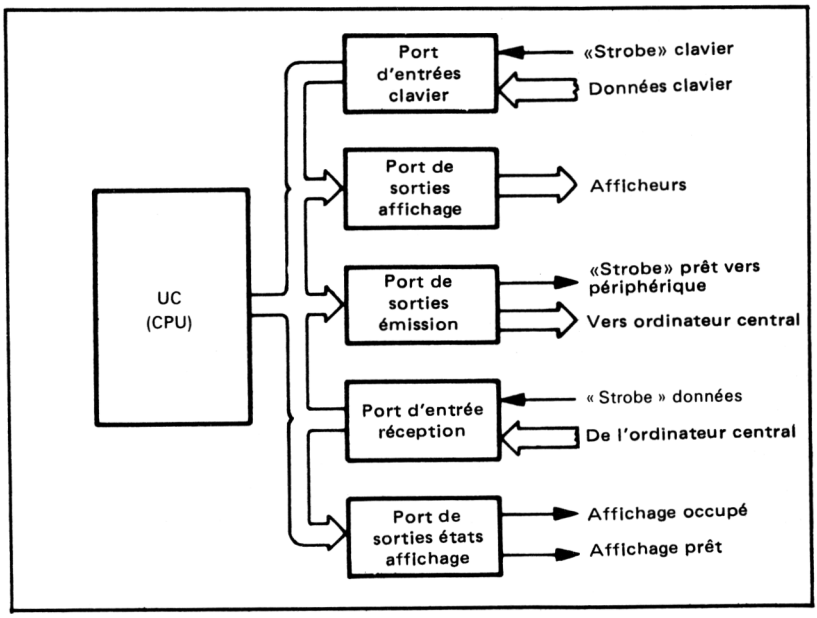


Fig. 13.4. — Synoptique du terminal de vérification de cartes de crédit.

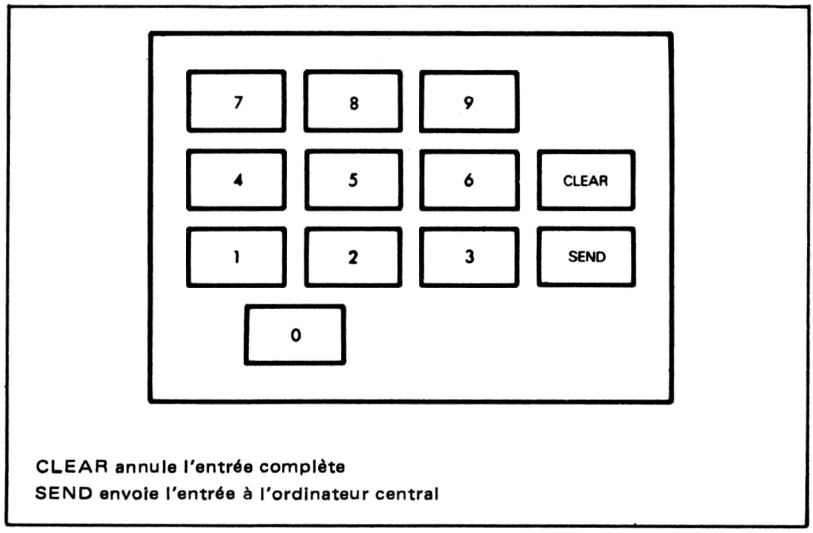


Fig. 13.5. — Le clavier du terminal de vérification.

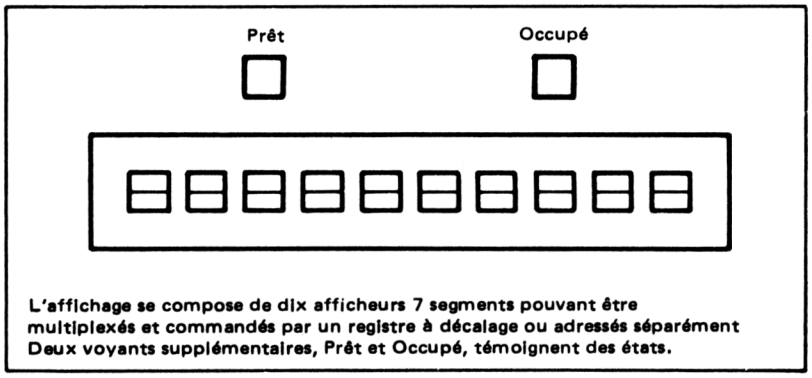


Fig. 13.6. — Afficheur du terminal de vérification.

d'un signal de strobe (échantillonnage). Le programme devra reconnaître le strobe et acquérir le chiffre hexadécimal identifiant la touche. Une contrainte de temps apparaît puisque le programme ne doit manquer aucune donnée ni aucun strobe ; elle n'est pas importante puisque les entrées se succèderont à au moins plusieurs millisecondes.
L'entrée provenant de la ligne de transmission se fait de façon semblable et consiste en une série de caractères identifiés chacun par un strobe (provenant probablement d'une UART). Le programme devra reconnaître chaque strobe et acquérir chaque caractère. Les données transmises ainsi sur les lignes sont généralement organisées en messages, dont le format possible est :

- Caractères d'introduction, ou en-tête.
- Adresse du terminal destinataire.

- Réponse codée « Oui » ou « Non ».
- Caractères de terminaison.

Le terminal examinera l'en-tête, lira l'adresse du destinataire et saura si le message lui est destiné. Si oui, il acceptera des données. Souvent, l'adresse est « câblée » dans le terminal, de façon qu'il ne reçoive que les messages qui lui sont destinés. Cette approche simplifie le logiciel mais sacrifie quelque peu la flexibilité d'emploi.

SORTIES DU TERMINAL DE VÉRIFICATION

Les sorties sont également plus complexes que dans les exemples précédents. Si les afficheurs sont multiplexés, le processeur n'a plus seulement à émettre les données mais il doit en même temps organiser leur distribution. Il nous faudra donc, soit un port de contrôle séparé, soit un compteur et un décodeur pour gérer le multiplexage. Notons que des circuits d'effacement des zéros de tête permettront de ne pas en tenir compte pour autant que le premier chiffre d'un nombre à plusieurs chiffres ne sera jamais un zéro ; le logiciel pourrait en faire autant. Les contraintes de temps portent sur la longueur de l'impulsion et sur la fréquence requise pour que l'affichage émette une lumière apparemment continue.

Pour les télécommunications, la sortie fournira une série de caractères dans un format spécifié. Le programme devra aussi tenir compte du temps requis entre chaque caractère. Un format possible est le suivant :

- En-tête.
- Adresse du terminal.
- Numéro de la carte de crédit.
- Terminaison.

Un ordinateur central de communications pourra scruter les terminaux et les interroger pour savoir s'ils ont des données à fournir.

Plusieurs tâches nouvelles apparaissent donc dans ce traitement.

- L'identification des touches de contrôle par un nombre et l'exécution de la tâche adéquate.
- L'addition de l'en-tête, de l'adresse du terminal et de la terminaison au message en sortie.
- La reconnaissance de l'en-tête et de la terminaison du message reçu en réponse.
- Le test de l'adresse du terminal dans le message reçu.

TRAITEMENT DES ERREURS DU TERMINAL DE VÉRIFICATION

Aucune de ces tâches n'implique d'arithmétique complexe ni ne crée de sérieuses contraintes en temps ou mémoire. **Le nombre des erreurs possibles avec ce système est évidemment bien plus important que dans les cas précédents.** Pour commencer, examinons les erreurs humaines :

- Le numéro de la carte de crédit a été mal introduit.
- On a tenté d'envoyer un numéro incomplet.
- On a voulu envoyer un numéro alors que l'ordinateur était encore en train de traiter le précédent.
- On a annulé (CLEAR) des entrées inexistantes.

Quelques-unes de ces erreurs sont facilements éliminées par une bonne structuration du programme. Par exemple, le programme n'acceptera pas d'ordre d'émission des données

tant que le numéro de la carte de crédit n'aura pas été totalement introduit, et il devra ignorer toute entrée tant qu'une réponse à la précédente n'a pas été fournie par l'ordinateur central. L'opérateur devrait d'ailleurs savoir que l'entrée n'a pas été émise grâce au voyant « Occupé » qui ne s'allumera pas. Il saura aussi que le clavier a été verrouillé (le programme a « déconnecté » le clavier) car les touches frappées ne seront pas affichées et le voyant « Prêt » sera éteint.

CORRECTION DES ERREURS DU CLAVIER

Le risque d'entrées incorrectes est évident. Si l'opérateur reconnaît son erreur, il pourra employer la touche d'effacement « Clear » pour exécuter les corrections ; il souhaitera probablement deux touches « Clear », l'une effaçant la dernière touche frappée et l'autre supprimant l'entrée complète. On pourrait ainsi répondre aux cas où l'opérateur s'aperçoit immédiatement, ou tardivement de son erreur ; il pourrait alors procéder aux corrections avec le minimum de nouvelles frappes. On peut être certain, cependant, que l'opérateur fera des erreurs dont il ne s'apercevra pas. La plupart des numéros de cartes de crédit comprennent un chiffre d'auto-test ; le terminal pourra le tester avant d'autoriser l'envoi à l'ordinateur central. On évitera ainsi à ce dernier la perte d'un temps précieux utilisé à tester ce numéro.

Le terminal devra alors disposer d'un moyen d'alerter l'opérateur qu'il a commis une erreur, peut-être en faisant clignoter un afficheur, ou encore en fournissant une indication qui ne pourra échapper à l'opérateur.

Un autre problème réside dans la manière dont l'opérateur sera informé qu'une entrée a été perdue ou traitée incorrectement. Certains terminaux se déverrouillent simplement après une temporisation maximale donnée. L'opérateur note que le voyant « Occupé » n'est plus actif alors que la réponse n'est pas parvenue. L'opérateur est alors supposé recommencer l'entrée. Après un ou deux essais, il devra alerter le personnel d'entretien.

De nombreuses pannes peuvent affecter l'équipement. Outre les afficheurs, le clavier et le processeur, on risque d'avoir maintenant des problèmes portant sur des erreurs de communication ou des défaillances de l'ordinateur central.

CORRECTION DES ERREURS DE TRANSMISSION

Les données échangées devront probablement inclure une procédure de test et de correction d'erreurs. Parmi les possibilités existantes figurent les suivantes :

1) La parité procure un moyen de détecter une erreur, mais ne fournit aucun mécanisme de correction. Le récepteur devra disposer d'un moyen de demander une ré-émission, et l'émetteur devra sauvegarder une copie de l'émission jusqu'à l'accusé de réception témoignant que les données sont bien parvenues sans erreur. La parité est très simple à introduire et utiliser.

2) Des méthodes plus élaborées pourront être appliquées aux messages courts. Par exemple, la réponse du terminal par « oui » ou « non » pourra être codée de façon à offrir un moyen de détecter et corriger une erreur.

3) Un accusé de réception et un nombre limité de ré-émissions pourraient activer un indicateur signalant à l'opérateur une panne de liaison (impossibilité de transmettre un message sans erreurs), ou une panne de l'ordinateur central (aucune réponse à tous les messages pendant une certaine période de temps). Cette méthode, associée à un test de voyant, permettrait des diagnostics de défaillances simples.

Une panne de liaison ou de l'ordinateur central devrait aussi « déverrouiller » le terminal, c'est-à-dire lui permettre d'accepter de nouvelles entrées. C'est nécessaire pour que le terminal ne refuse pas de nouvelles entrées alors qu'une vérification est en cours. Le

terminal pourra aussi se déverrouiller après un certain laps de temps. Certaines entrées pourront être réservées aux diagnostics ; par exemple, certains numéros de cartes de crédit serviront à tester le fonctionnement interne et l'affichage du terminal.

EN RÉSUMÉ

La définition des tâches constitue un aspect essentiel du développement du logiciel (ou de toute autre tâche d'ingénierie). Elle n'implique pas la connaissance de la programmation ou de l'informatique ; par contre, elle se fonde sur une bonne compréhension du système et relève de jugements sains. Les microprocesseurs offrent au concepteur une flexibilité qu'il peut mettre en œuvre pour fournir des caractéristiques qui n'étaient pas envisageables auparavant.
Les définitions de problèmes sont indépendantes de l'ordinateur qui sera employé, de son langage ou du système de développement. Ils devront cependant guider le choix en ce qui concerne le type d'ordinateur ou la vitesse qui conviendront le mieux à l'application, et en indiquant à quelles sortes de compromis matériel-logiciel il faudra se livrer. En fait, la définition est même indépendante de l'usage ou non d'un ordinateur, bien que la connaissance des possibilités de l'ordinateur envisagé puisse aider le concepteur en lui suggérant des procédures utilisables.

CONCEPTION DU PROGRAMME

La définition ayant été achevée, on passe à l'étape où cette définition est formulée en programme. Si le programme est court et simple, cette étape se bornera à la rédaction d'un organigramme sur une page. S'il est long ou plus complexe, le concepteur devra recourir à des méthodes plus élaborées.
Nous allons discuter de la réalisation des organigrammes, de la programmation modulaire, de la programmation structurée et de la conception descendante (« top-down »). Nous allons essayer d'indiquer les raisonnements qui accompagnent ces méthodes, leurs avantages et leurs inconvénients. Nous ne nous ferons l'avocat d'aucune d'elles, car il n'est pas évident qu'une de ces méthodes soit toujours supérieure à toutes les autres. On doit conserver présent à l'esprit que l'objectif est de réaliser un système fonctionnant bien et qu'il ne s'agit pas de suivre religieusement la doctrine d'une méthodologie ou d'une autre.

PRINCIPES DE BASE DE LA CONCEPTION DU PROGRAMME

Toutes ces méthodologies ont cependant en commun des principes évidents. La plupart sont d'ailleurs les mêmes qui s'appliquent à toutes sortes de conceptions, **tels que :**

1) Procéder par petits pas. N'essayez pas d'en faire trop d'un coup.
2) Diviser des tâches importantes en petites tâches, logiquement séparées. Rendre ces sous-tâches indépendantes les unes des autres de façon à pouvoir les tester séparément et introduire des modifications sans affecter les autres.
3) Rendre la chaîne de contrôle aussi simple que possible afin de faciliter la détection des erreurs.
4) Utiliser autant de schémas et descriptions graphiques que possible. Ils visualisent mieux les choses que de longues descriptions. C'est d'ailleurs le gros intérêt des organigrammes.

5) Mettre l'accent sur la clarté et la simplicité en tout premier lieu. On pourra améliorer les performances (si nécessaire) dès lors que le système fonctionnera.
6) Procéder de façon minutieuse et systématique. Utiliser des listes de test et des procédures standards.
7) Ne pas employer de méthodes dont on n'est pas sûr, ou alors les utiliser avec la plus grande prudence. Veiller aux situations pouvant créer des confusions et les clarifier dès que possible.
8) Garder présent à l'esprit que le système devra être mis au point, testé et maintenu. Préparer ces trois dernières étapes.
9) Employer une terminologie et des méthodes simples. Des répétitions ne sont pas une faute dans la conception des programmes, tout comme la complexité n'est pas une vertu.
10) Formuler complètement la conception avant de commencer le codage. Résistez à la tentation de commencer à rédiger des instructions ; cela n'a pas plus de sens que d'élaborer des listes de composants ou de dessiner un circuit imprimé avant de savoir ce qu'on mettra exactement dans le système.
11) Porter toute son attention aux facteurs qui pourront changer. Concevoir ce qui sera peut-être appelé à changer le plus simplement possible.

LES ORGANIGRAMMES

Les organigrammes constituent probablement la méthode de conception de programmes la plus connue. Les livres de programmation décrivent comment les programmeurs commencent d'abord par rédiger des organigrammes complets, et ensuite, attaquent la rédaction du programme. En fait, rares sont les programmeurs à avoir procédé ainsi, et le tracé des organigrammes a souvent été davantage un jeu ou une nuisance qu'une méthode de conception. Nous allons essayer de décrire les avantages et les inconvénients des organigrammes, et montrer la place de cette technique dans la conception des programmes.

AVANTAGES DES ORGANIGRAMMES

L'intérêt fondamental des organigrammes réside dans une représentation imagée. On trouve cette représentation plus significative que des descriptions écrites. Le concepteur peut visualiser l'ensemble du système et examiner les relations entre ses diverses parties. Les erreurs logiques et les contradictions apparaissent au lieu de rester dissimulées dans une page imprimée. **Au mieux, l'organigramme est une image du système entier.**
Quelques avantages plus spécifiques des organigrammes sont :

1) Des symboles standards existent (fig. 13.7), donc, les formes utilisées sont reconnues par tous.
2) Les organigrammes peuvent être compris par des gens sans connaissance de la programmation.
3) Les organigrammes peuvent servir à diviser le projet entier en sous-tâches. Les organigrammes serviront alors à mesurer les progrès accomplis.
4) Les organigrammes décomposent une séquence d'opérations et aident ainsi à localiser les sources d'erreurs.
5) Les organigrammes sont largement employés dans des domaines autres que la programmation.

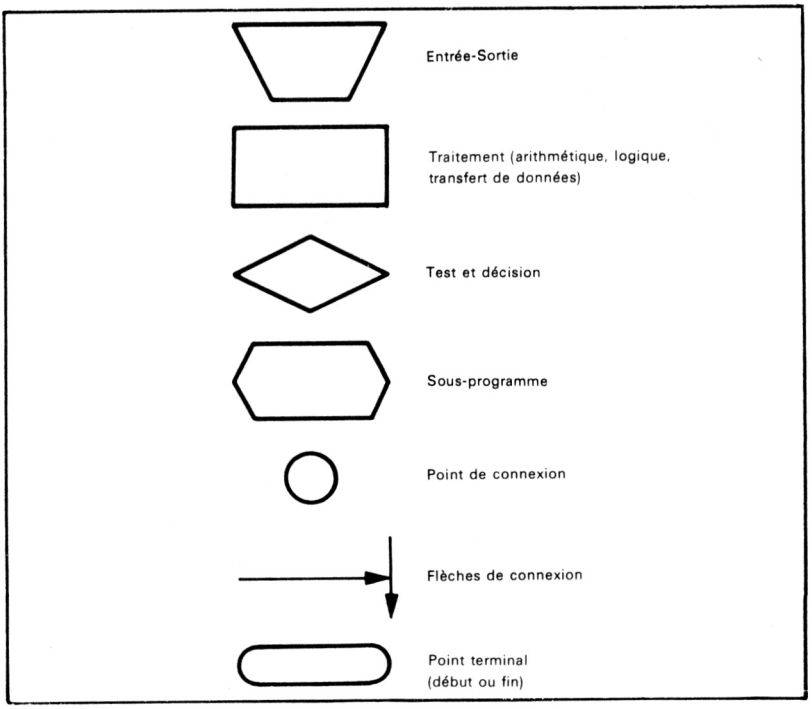


Fig. 13.7. — Symboles standards pour organigrammes.

INCONVÉNIENTS DES ORGANIGRAMMES

Ces avantages sont tous importants. Il est certain que les organigrammes seront encore largement utilisés. **Mais il est important d'en remarquer quelques inconvénients :**

1) Les organigrammes sont difficiles à concevoir, dessiner ou modifier dans tous les cas, sauf les plus simples.
2) Il n'existe pas de méthode simple pour mettre au point ou tester un organigramme.
3) Les organigrammes tendent à devenir encombrants. Il est difficile de décider entre l'introduction de nouveaux détails qui les rendront utiles et le nombre de ces détails, qui les rendraient à peine plus lisibles qu'un listage.
4) Les organigrammes ne montrent que l'organisation du programme. Ils ne montrent pas l'organisation des données ou la structure des modules d'entrées-sorties.
5) Les organigrammes ne sont d'aucun secours pour les problèmes d'ordre matériel ou temporel ; ils ne suggèrent même pas où de tels problèmes pourraient surgir.
6) Les organigrammes autorisent une programmation sans aucune structuration. Les renvois, flèches et boucles sont l'antithèse d'une bonne programmation structurée.

De ce fait, les organigrammes constituent une technique utile à laquelle il ne faudrait pas trop demander. **Ils sont utiles pour la documentation des programmes, puisqu'ils disposent de formes standard et sont compris par des non programmeurs.** En tant qu'outil de conception, ils ne peuvent cependant offrir qu'un moyen de démarrer, **le programmeur ne pourra pas déverminer un organigramme détaillé** et l'organigramme sera souvent plus difficile à concevoir que le programme lui-même.

EXEMPLES

Réponse à un interrupteur

ORGANIGRAMME DU SYSTÈME A INTERRUPTEUR ET VOYANT

Avec cette tâche simple, dans laquelle un seul interrupteur commande la mise en service d'un voyant pour une seconde, le tracé de l'organigramme est facile. C'est d'ailleurs l'exemple typique d'une séquence illustrée par organigramme, bien qu'il ne s'agisse là le plus souvent que d'une petite partie d'un système. La structure de la donnée est si simple qu'elle peut être totalement ignorée. **La figure 13.8 montre cet organigramme.** Il y a peu de difficulté à décider des détails requis. L'organigramme fournit une image linéaire de la procédure et tout le monde peut le comprendre.

Notez que les plus utiles des organigrammes peuvent ignorer les variables du programme et poser directement les questions. Naturellement, des compromis sont souvent nécessaires. **Deux versions de l'organigramme seront souvent les bienvenues : l'une dans le langage du concepteur, qui sera utile aux non programmeurs, et l'autre pour le programmeur, en terme de variables du programme, qui servira à d'autres programmeurs.**

ORGANIGRAMME DE DONNÉES

Un troisième type d'organigramme, un organigramme de données pourra aussi être utile. Il servira de références aux autres, en montrant comment le programme traite les divers types de données. Les organigrammes classiques montrent ce que fait le programme, traitant des données diverses en des points différents. L'organigramme de données, lui,

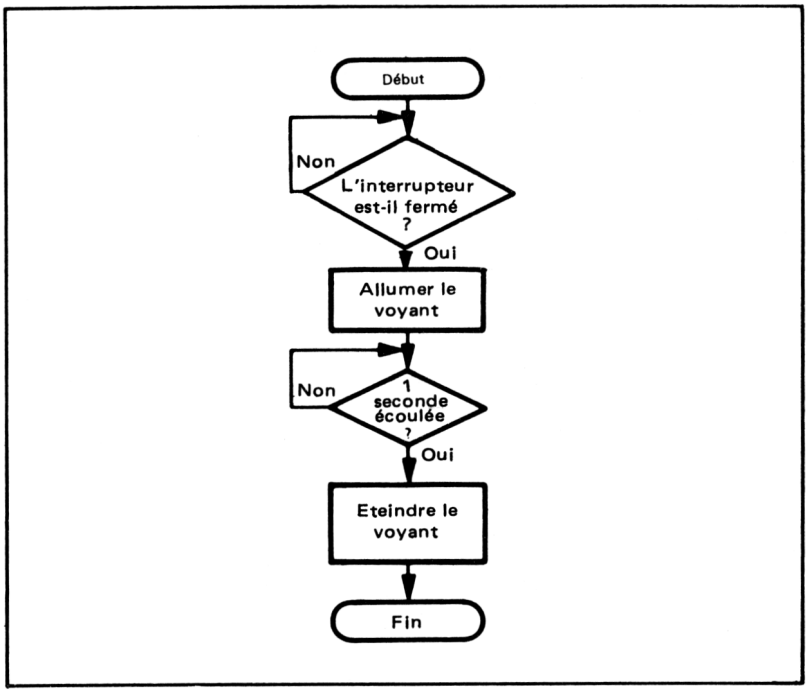


Fig. 13.8. — Organigramme du programme à interrupteur et voyant.

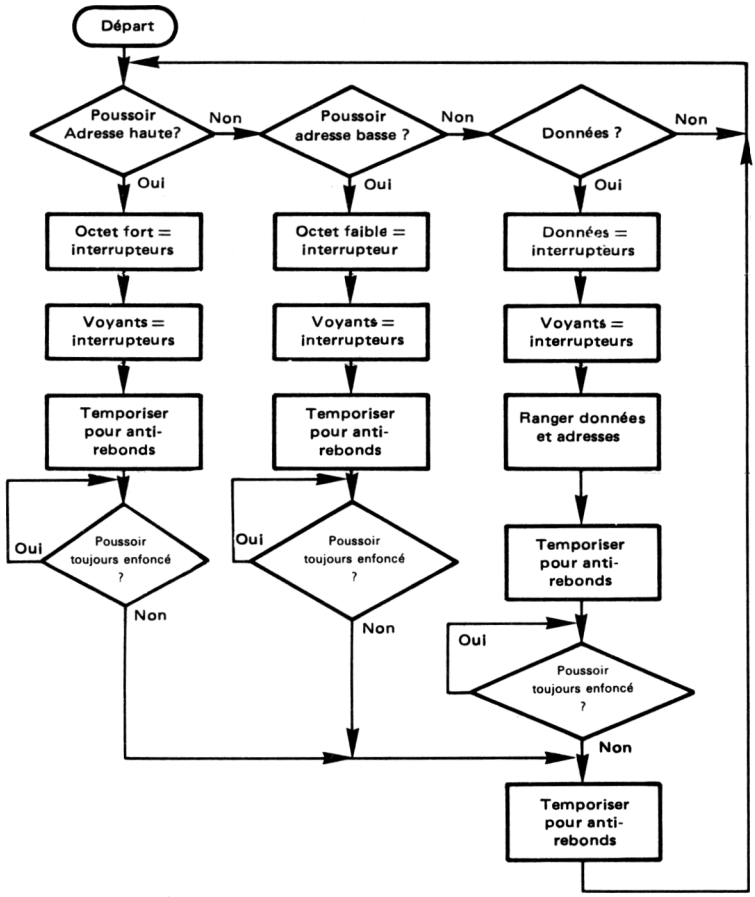


Fig. 13.9. — Organigramme du chargeur à base d'interrupteurs.

montrera comment des types particuliers de données évoluent dans le système, passant d'une section à l'autre. De tels organigrammes seront très utiles lors de la mise au point et de la maintenance, car les erreurs surviennent le plus souvent sur un type particulier de données traité incorrectement.

Le chargeur de mémoire à interrupteurs

ORGANIGRAMME POUR LE PROGRAMME CHARGEUR À INTERRUPTEURS

Ce système (voir figure 13.3) est considérablement plus complexe que le précédent et implique plusieurs décisions. **L'organigramme (fig. 13.9) est plus difficile à préparer et n'est plus aussi évident que dans l'exemple précédent.** On s'aperçoit maintenant qu'il n'y a pas moyen de mettre au point ou tester l'organigramme.

Celui de la figure 13.9 incorpore les améliorations que nous avons suggérées dans la définition du problème. Manifestement, **cet organigramme devient confus et perd ses avanta-**

ges sur une description écrite. On le rendrait encore plus complexe en lui ajoutant d'autres caractéristiques, pour définir la signification des entrées avec les voyants d'états et permettre à l'opérateur de tester les entrées après exécution. Tracer l'organigramme complet dès le départ pourrait devenir rapidement une tâche colossale. Cependant, et dès lors que le programme a été rédigé, il devient utile au titre de la documentation.

Terminal de vérification de cartes de crédit

ORGANIGRAMME DU PROGRAMME POUR TERMINAL DE CARTE DE CRÉDIT

Dans cette application (voir figures 13.4 à 13.6), l'organigramme sera encore plus complexe que précédemment. **La meilleure idée consisterait à tracer des sections d'organigramme séparément, de façon à leur garder une dimension raisonnable.** Cependant, la présence des structures des données (multi-chiffres pour l'affichage et les messages) accroîtra encore le fossé entre l'organigramme et le programme.

SECTIONS DE L'ORGANIGRAMME

Examinons quelques-unes des sections. **La figure 13.10 montre le processus pour les entrées des touches numériques du clavier.** Le programme doit acquérir les données après

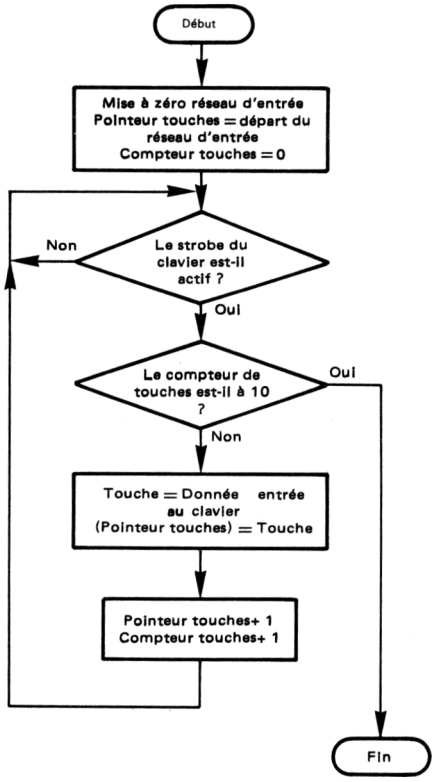


Fig. 13.10. — Organigramme de l'entrée du clavier.

chaque strobe et placer les caractères dans le réseau des afficheurs s'il reste de la place. Si les dix caractères du réseau sont déjà employés, le programme ignore simplement les entrées.

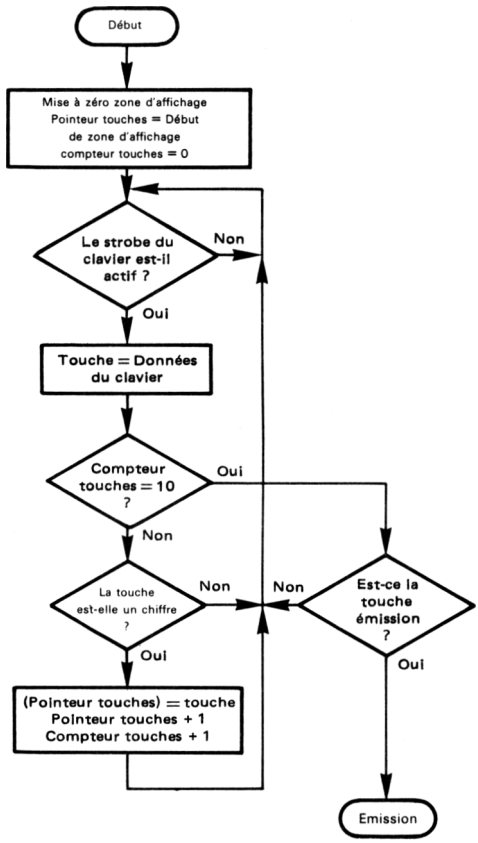


Fig. 13.11. — Organigramme du processus d'entrées du clavier avec la touche de fonction « Emission ».

Le programme réel devra traiter les afficheurs en même temps. Notez que le matériel ou le logiciel doivent désactiver le strobe clavier après que le processeur a lu un caractère. **La figure 13.11 ajoute à cet organigramme l'intervention de la touche émission.** Cette touche est, naturellement, optionnelle. Le terminal pourrait émettre les données dès que l'opérateur a introduit un nombre complet. Cependant, une telle procédure ne donnerait à l'opérateur aucun moyen de vérifier l'entrée complète. L'organigramme avec touche émission est plus complexe car deux voies se dessinent.

1) Si l'opérateur n'a pas introduit dix chiffres, le programme ignore l'ordre d'émission donné par la touche émission.

2) Si l'opérateur a introduit dix chiffres, le programme doit répondre à la touche d'émission en transférant le contrôle à la séquence de programme traitant cette émission, et ignorer toutes les autres touches.

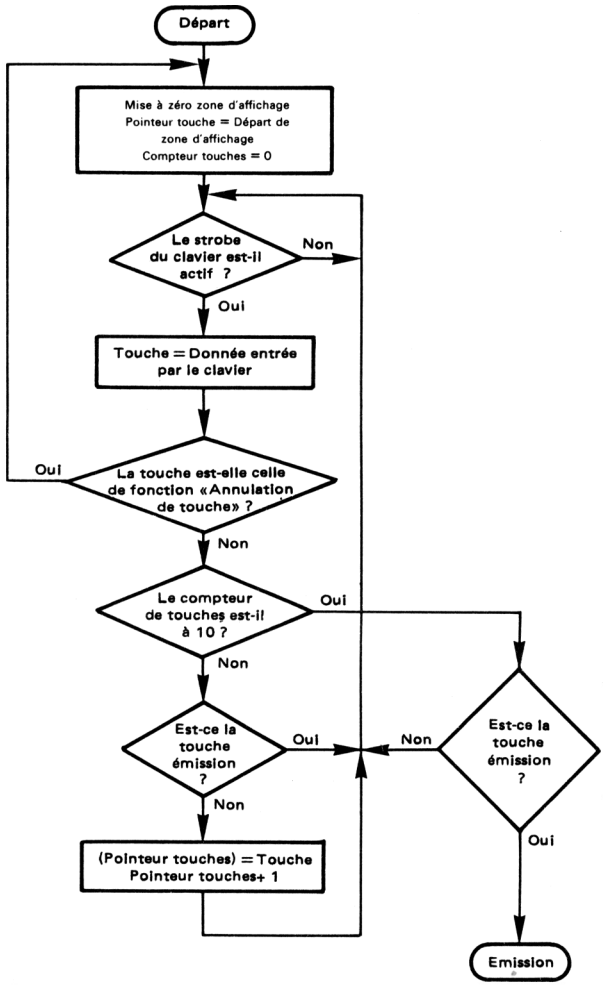


Fig. 13.12. — Organigramme du processus d'entrées du clavier avec les touches de fonctions.

L'organigramme devient bien plus difficile à organiser et à suivre. Il n'existe pas de chemin évident pour le tester.

La figure 13.12 donne l'organigramme du processus d'entrées du clavier avec toutes les touches de fonctions. Dans cet exemple, suivre le cheminement des contrôles n'est plus du tout aisé. A l'évidence, une description écrite est indispensable. L'organisation et le dessin d'organigrammes complexes demande un plan soigneux. Nous avons choisi d'ajouter des caractéristiques une par une, mais cette méthode impose un travail considérable de redessin. A nouveau, il faut se souvenir que, tout au long du processus d'entrées du clavier, le programme doit rafraîchir continuellement l'affichage s'il est multiplexé et non commandé par des registres à décalage ou autres circuits.

La figure 13.13 montre l'organigramme de réception. On suppose que la conversion série-parallèle et le test d'erreurs sont exécutés par matériel (par exemple, par une UART). Le processeur doit :

1) Surveiller l'arrivée de l'en-tête (on suppose qu'il s'agit d'un caractère unique).
2) Lire l'adresse de destination (supposée être de trois caractères) et vérifier si le message est destiné au terminal (c'est-à-dire si les trois caractères correspondent à l'adresse du terminal).
3) Attendre le caractère de terminaison.
4) Si le message est destiné au terminal, éteindre le voyant « Occupé » et passer au programme d'affichage de la réponse.
5) En cas d'erreur, demander une re-transmission en passant un programme « RTRAN ».

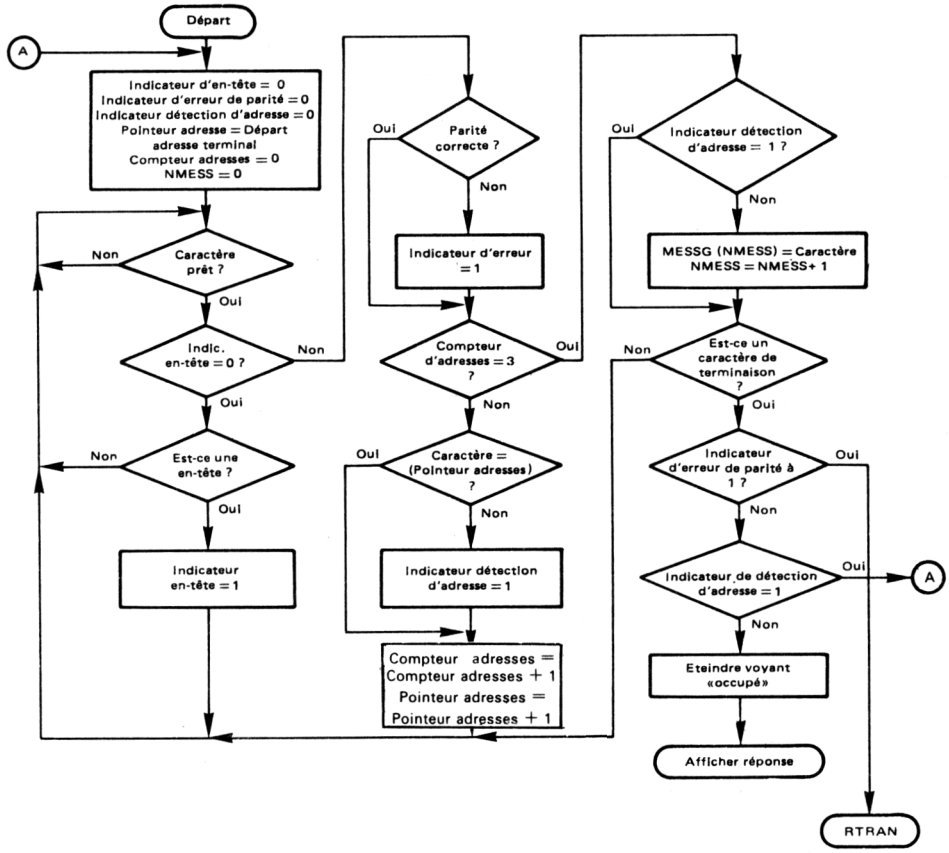


Fig. 13.13. — Organigramme de la section réception.

Tout ceci implique un grand nombre de décisions et l'organigramme n'est plus ni simple, ni évident.

On a ainsi parcouru un grand chemin depuis l'organigramme simple de la figure 13.8 de notre premier exemple. Un jeu complet d'organigrammes pour ce terminal de transmissions constituerait un travail important. Il consisterait en plusieurs schémas interconnectés avec une logique complexe, ce qui représenterait réellement un gros investissement en travail. Ce travail serait aussi difficile qu'écrire un programme préliminaire, mais pas aussi utile, puisqu'on ne pourrait pas le vérifier sur un ordinateur.

PROGRAMMATION MODULAIRE

Dès lors que les programmes deviennent longs et complexes, les organigrammes ne conviennent plus. Cependant, la définition des problèmes et l'organigramme peuvent donner quelques idées sur la façon de subdiviser rationnellement le programme. **La division d'un programme complet en sous-tâches, ou modules, est appelée « programmation modulaire ».** Nombre des programmes que nous avons proposés antérieurement constituent des modules typiques d'un système important. **Les problèmes qui surgissent et que le programmeur doit résoudre en programmation modulaire sont la manière de fractionner le programme en modules, et la façon de réunir ces modules.**

AVANTAGES DE LA PROGRAMMATION MODULAIRE

Les avantages de la programmation modulaire sont évidents :

1) Un simple module est plus facile à écrire, mettre au point et tester qu'un programme entier.
2) Un module peut être employé à divers moments d'un programme et dans d'autres programmes, surtout s'il est raisonnablement général et exécute une tâche courante. On peut ainsi bâtir une bibliothèque de modules standard.
3) La programmation modulaire permet au programmeur de fractionner les tâches et de recourir à des modules rédigés antérieurement.
4) Les modifications sont introduites dans un unique module et non plus dans le système entier.
5) Les erreurs peuvent souvent être isolées et attribuées à un module unique.
6) La programmation modulaire donne une idée du progrès accompli et de ce qui reste à faire.

INCONVÉNIENTS DE LA PROGRAMMATION MODULAIRE

Le concept de la programmation modulaire paraît si évident qu'on en ignore souvent les inconvénients :

1) Adapter les divers modules entre eux peut constituer un problème majeur, particulièrement lorsqu'ils ont été écrits par différentes personnes.
2) Les modules exigent une documentation soignée, puisqu'ils peuvent affecter d'autres parties du programme, par exemple les structures de données utilisées par tous les modules.
3) Le test et la mise au point des modules distincts est difficile car d'autres modules peuvent produire les données utilisées par celui qui est testé, ou employer les résultats obte-

nus. Vous aurez peut-être à rédiger des programmes spéciaux (appelés « drivers ») qui produiront uniquement les échantillons de données nécessaires et testeront les programmes. Ces drivers requièrent donc un effort supplémentaire de programmation sans rien ajouter au système.
4) Certains programmes sont très difficiles à modulariser d'une façon rationnelle. Si cette modularisation est mal conduite, il sera très difficile de rassembler, « d'intégrer » les modules car presque toutes les erreurs et les modifications consécutives toucheront plusieurs modules.
5) La programmation modulaire exige souvent un supplément de temps et de mémoire, puisque des fonctions peuvent se répéter dans plusieurs modules.
Ainsi, et bien que la programmation modulaire constitue sans aucun doute un progrès sur les tentatives de rédiger d'emblée le programme, elle n'en présente pas moins des désavantages certains.
Des considérations importantes font intervenir une restriction du volume des informations partagées par les modules, une limitation des décisions à prendre dans un unique module si elles sont sujettes à changements, et des restrictions sur les accès entre modules [3].

PRINCIPES DE LA MODULARISATION

Il n'existe pas de méthode systématique, pour modulariser les programmes. Nous devons mentionner les principes suivants [4] :

1) Des modules à références communes doivent faire partie d'un même module plus général.
2) Deux modules, dont le premier emploie ou dépend du second, mais non l'inverse, doivent être séparés.
3) Un module utilisé par plus d'un autre module doit appartenir à un modèle plus général distinct.
4) Deux modules, dont le premier est employé par de nombreux autres et dont le second par quelques autres seulement, doivent être séparés.
5) Deux modules utilisés avec des fréquences très différentes doivent appartenir à des modules distincts.
6) La structure ou l'organisation de données interdépendantes doit être incorporée dans un seul module.

Si un programme paraît trop difficile à modulariser, il y a de fortes chances qu'il soit mal défini et qu'une redéfinition soit nécessaire. De trop nombreux cas particuliers ou de trop nombreuses variables qui exigent des traitements spécifiques constituent des signes typiques d'une mauvaise adéquation de la définition du problème.

EXEMPLES

Réponse à un interrupteur

MODULARISATION DU SYSTÈME À INTERRUPTEUR ET VOYANT

Ce simple programme peut être scindé en deux modules :

- **Le module 1 attend que l'interrupteur soit déclenché et allume le voyant.**
- **Le module 2 temporise une seconde.**

Le module 1 restera spécifique du système puisqu'il dépend

des modes de branchement de l'interrupteur et du voyant. Le module 2 sera d'usage plus général car de nombreuses tâches font intervenir des temporisations. Or, il est intéressant de disposer d'une temporisation standard servant de base à la création de délais de différentes durées. Ce module devra être soigneusement déconnecté afin de savoir comment spécifier la durée de la temporisation, comment appeler le module, et quels registres et mémoires il utilise.
Une version générale du module 1 serait bien moins utile car elle devrait faire intervenir tous les modes de connexion de l'interrupteur et du voyant.
On trouvera probablement plus simple de rédiger un module pour une configuration particulière de l'interrupteur et du voyant plutôt que de passer par une séquence standard. Remarquez la différence entre cette situation et le module 2.

Le chargeur mémoire à interrupteurs

MODULARISATION DU CHARGEUR À INTERRUPTEURS

Le programme chargeur à interrupteurs est difficile à modulariser, car toutes les tâches dépendent de la configuration matérielle et sont tellement simples que cela ne semble pas en valoir la peine. L'organigramme de la figure 13.9 suggère que l'un des modules pourrait être celui qui attend que l'opérateur actionne l'un des trois poussoirs.
Les autres modules pourraient être :

- Un module de temporisation fournissant le retard requis pour éliminer les rebonds des interrupteurs.
- Un module interrupteurs et affichage lisant les données formées par les interrupteurs et les envoyant à l'affichage.
- Un module de test des voyants.

Il est peu probable que des modules tels que les deux précédents soient utiles d'une façon générale. Cet exemple n'est pas de ceux qui révèlent l'intérêt de la programmation modulaire.

Le terminal de vérification

MODULARISATION DU TERMINAL DE VÉRIFICATION

Par contre, le terminal de vérification de cartes de crédit se prête facilement à la programmation modulaire. Le système complet peut aisément se subdiviser en trois modules principaux.

- **Le module clavier et affichage.**
- **Le module d'émission des données.**
- **Le module de réception des données.**

Un module général pour le clavier et l'affichage pourrait être en mesure de gérer de nombreux claviers et affichages de divers systèmes ; les sous-modules exécuteraient des tâches telles que :

- Reconnaissance d'une nouvelle entrée du clavier et acquisition de la donnée.
- Effacement en réponse à une action sur la touche « Clear ».
- Introduction des caractères (chiffres) en mémoire.
- Surveillance du code terminaison ou du poussoir d'émission.
- Affichage des chiffres.

Bien que l'interprétation de toutes les touches et que le nombre de chiffres varient alors d'un programme à l'autre, les processus fondamentaux d'entrée, rangement et affichage

des données, pourraient rester identiques. Certaines clés de fonctions, telles que « Clear », peuvent aussi être standards. **Aussi le concepteur doit-il considérer les modules qui pourront resservir dans d'autres applications et leur accorder une grande attention.**

Le module d'émission pourrait, lui aussi, se subdiviser en sous-modules tels que :

1) Addition de l'en-tête.
2) Emission des caractères dès que la ligne de sortie peut les accepter.
3) Création des délais entre bits ou caractères.
4) Addition du caractère de fin de message.
5) Test des défaillances de transmission, telles que : absence d'accusé de réception, ou incapacité d'assurer une transmission sans erreur.

Le module de réception pourrait inclure les sous-modules suivants :

1) Recherche du caractère d'en-tête.
2) Test de l'adresse du destinataire du message et comparaison avec l'adresse propre du terminal.
3) Rangement et interprétation du message.
4) Recherche du caractère de fin de message.
5) Création des délais entre bits ou caractères.

PRINCIPE DE L'INFORMATION CACHÉE

Il est important, ici, que chaque décision (tels que débit des bits, format du message, procédure de test d'erreur) soit conçue et incorporée dans un seul module. Pour modifier l'une de ces décisions, il suffira alors d'intervenir dans ce seul module. Les autres modules doivent être écrits de façon à pouvoir se passer de la connaissance des valeurs retenues ou des méthodes utilisées pour réaliser les modules. **Un concept important est ici le « concept de l'information cachée » [5],** selon lequel les modules ne doivent se partager que les informations qui leur sont absolument nécessaires pour exécuter leur tâche. Les autres informations sont cachées dans un unique module. Le traitement des erreurs constitue un contexte typique dans lequel ce principe doit être utilisé. Lorsqu'un module détecte une erreur fatale, il ne doit pas essayer de la récupérer ; au contraire, il doit informer le module appelant et lui permettre de décider de la suite à donner. La raison réside dans le fait que le module de niveau inférieur manque souvent d'une information suffisante pour établir la procédure de correction. Par exemple, supposons que le module de niveau inférieur est celui qui accepte l'entrée numérique d'un utilisateur. Ce module attend une chaîne de chiffres se terminant par un retour chariot. L'entrée d'un caractère non numérique provoque une fin anormale. Puisque ce module ne connait pas le contexte (c'est-à-dire qu'il ne sait pas si cette entrée est un opérande, un numéro de ligne, la référence d'une unité d'E/S ou la longueur d'un fichier), il ne peut pas décider du traitement de l'erreur. Si le module faisait toujours suivre l'erreur de procédure de correction, il perdrait sa généralité et ne serait exploitable que dans les cas où cette procédure spécifique serait requise.

RÉSUMÉ DE LA PROGRAMMATION MODULAIRE

RÈGLES DE LA PROGRAMMATION MODULAIRE

La programmation modulaire peut se révéler très utile si l'on respecte les règles suivantes :

1) Employer des modules de 20 à 50 lignes. Des modules plus courts gaspillent généralement du temps ; de plus longs risquent de ne plus être généraux et peuvent se révéler difficiles à intégrer.
2) Tenter de rendre ces modules suffisamment généraux. Bien distinguer les caractéristiques générales, telles que le code ASCII ou les formats d'échanges asynchrones qui restent les mêmes pour bien des applications, et l'identification des touches, le nombre d'afficheurs ou le nombre de caractères dans un message qui sont souvent spécifiques d'une application particulière. Rendre facile la modification de ces derniers paramètres. Des changements importants, tels que des codes de caractères différents, doivent faire l'objet de modules distincts.
3) Prendre son temps avec les modules de temporisation, de gestion d'affichage, de clavier, etc., **qui servent dans d'autres projets ou en différents endroits du programme en cours.**
4) Séparer le plus logiquement possible les modules et essayer de les rendre aussi distincts que faire se peut. Réduire le flot des informations entre les modules et exécuter chaque décision de conception dans un unique module.
5) Ne pas essayer de modulariser des tâches simples pour lesquelles une ré-écriture entière peut être plus facile qu'un assemblage de modules ou des modifications dans l'un d'eux.

PROGRAMMATION STRUCTURÉE

Comment conserver des modules distincts et les empêcher de réagir les uns avec les autres ? Comment rédiger un programme développant une séquence claire des opérations, de façon que l'on puisse isoler et corriger les erreurs ?
Une réponse consiste à utiliser la méthode dite de « programmation structurée », pour laquelle chaque partie du programme se compose d'éléments organisés en un nombre limité de structures de base, chacune de celles-ci ne disposant que d'une entrée et d'une sortie.
La figure 13.14 donne l'organigramme d'un programme non structuré. Si une erreur se manifeste dans le module B, on peut lui découvrir cinq sources. On ne doit pas seulement tester chacune des séquences possibles, mais on doit aussi s'assurer qu'une modification quelconque introduite pour corriger cette erreur n'affectera pas les autres séquences. Le résultat courant est que la mise au point du programme ressemble à une tentative de démêlage d'un écheveau. Chaque fois que l'on pense que la structure est rétablie, on découvre d'autres nœuds.

STRUCTURES DE BASE DE LA PROGRAMMATION STRUCTURÉE

Pour répondre à ces problèmes, on fait appel à des séquences claires permettant d'isoler les erreurs. Une telle séquence dispose d'une seule entrée et d'une seule sortie. **Les modules de base de ces séquences sont :**

1) Une séquence ordinaire, c'est-à-dire linéaire dans laquelle les ordres sont exécutés consécutivement :

S1
S2
S3

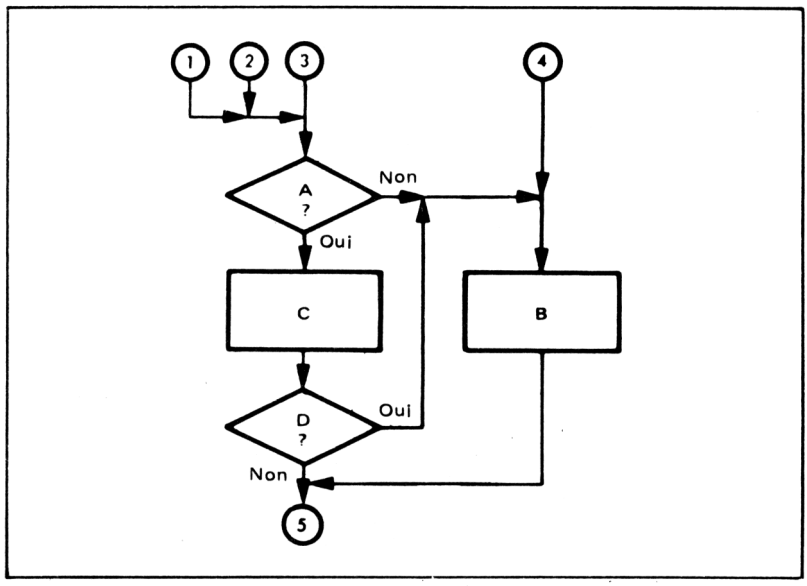


Fig. 13.14. — Organigramme d'un programme non structuré.

L'ordinateur exécute S1 d'abord, puis S2 en second, et enfin S3 en dernier. S1, S2 et S3 peuvent être de simples instructions ou des modules entiers.

2) Une structure conditionnelle.

La structure classique est : « si C alors S1, sinon S2 », où C est la condition et S1 et S2 sont des instructions ou des modules. L'ordinateur exécute S1 si C est vraie, S2 sinon. La figure 13.15 montre la logique de cette structure. Remarquons qu'il n'y a qu'une entrée

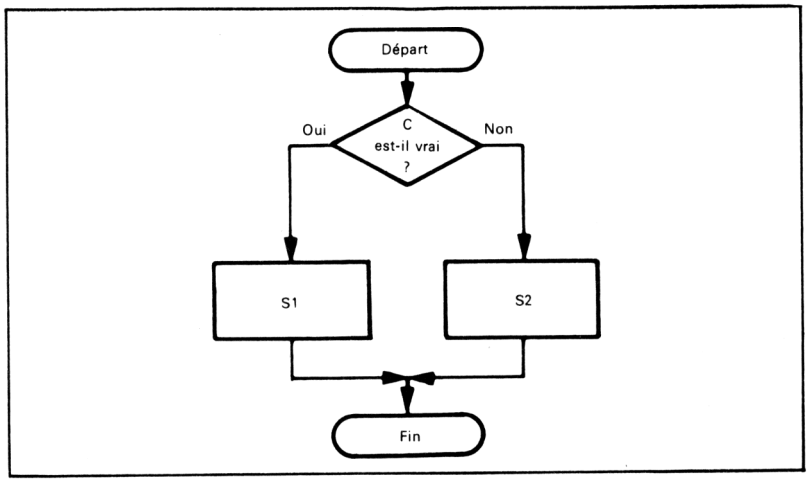


Fig. 13.15. — Organigramme de la structure « si, alors, sinon » (ou « if-then-else », en anglais).

et qu'une sortie ; on ne peut accéder à S1 ou S2 qu'en passant par cette structure. Notons qu'elle s'écrit « if-then-else » en anglais.

3) Une structure de boucle.

La structure commune de boucle est « tant que C, faire S », où C est une condition et S, une instruction ou un module. L'ordinateur teste C et exécute S tant que C est vraie. Cette structure (fig. 13.16) ne dispose que d'une seule entrée et d'une seule sortie, elle aussi. Remarquons que l'ordinateur n'exécutera jamais S si d'entrée de jeu, C est fausse, puisque la valeur de C est testée avant l'exécution de S. Cette structure est appelée « do-while » en anglais*.

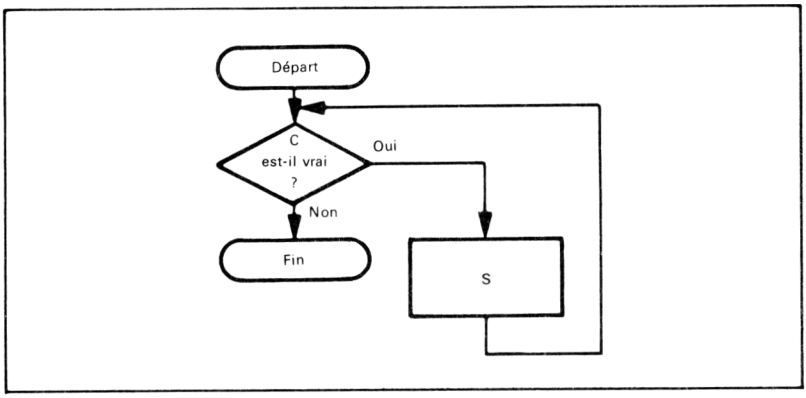


Fig. 13.16. — Organigramme de la structure « tant que, faire » (ou « do-while », en anglais).

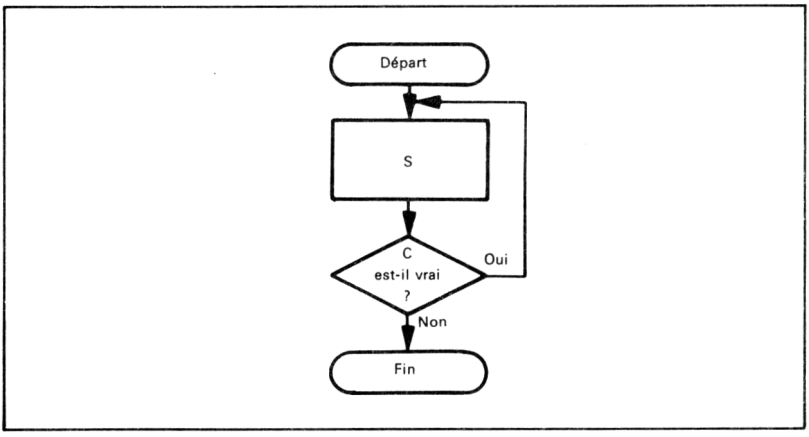


Fig. 13.17. — Organigramme de la structure « exécuter jusqu'à » (« do-until », en anglais).

*NDT : en fait, la terminologie « while-do » serait plus correcte ici, de manière à ne pas confondre cette séquence avec l'instruction « do S while C » (que l'on retrouve par exemple dans le langage C) équivalente à « repeat S until not C » (soit répéter S jusqu'à ce que C ne soit plus vraie).

Dans la plupart des langages structurés de programmation, une variante de cette structure est disponible, et connue sous le nom de « exécuter jusqu'à » (« do-until »). Sa structure de base est « exécuter S jusqu'à ce que C ». où C est la condition et S, une instruction ou un module. Elle est semblable au « do-while », sauf que le test de la condition C est exécuté à la fin de la boucle. De ce fait, la boucle sera au moins exécutée une fois. C'est ce qu'illustre la figure 13.17. Les célèbres boucles « DO » FORTRAN ou les boucles classiques contrôlées par des index peuvent être facilement mises en œuvre à l'aide de ces deux structures de base.

4) Une structure de cas

Bien qu'il ne s'agisse pas là d'une structure primitive, telle que les trois précédentes, la structure de cas est tellement employée que nous l'avons ajoutée ici. Cette structure est la suivante : « dans le cas I, séquence S0, S1,... Sn », où I est un index et S0, S1,... Sn des instructions ou séquences d'instructions. Si I est égal à zéro, S0 est exécuté ; si I est égal à 1, c'est S1 qui est exécuté ; etc.* Une seule des *n* séquences est exécutée. Après exécution, le contrôle est passé à l'instruction suivante. Si I est plus grand que *n* (c'est-à-dire que le nombre de séquences de l'instruction de cas), aucune n'est exécutée et le contrôle passe à l'instruction suivante. C'est ce qu'illustre le schéma de la figure 13.18.
Notons les caractéristiques suivantes de la programmation structurée :

1) Seules, les trois structures de base sont autorisées, avec quelques structures auxiliaires.
2) Ces structures peuvent être emboîtées jusqu'à n'importe quel niveau de complexité, de façon qu'un programme puisse, à son tour, contenir n'importe laquelle de ces structures.
3) Chaque structure n'a qu'une seule entrée et une seule sortie.

EXEMPLES DE STRUCTURES

Voici quelques exemples de la structure conditionnelle donnée dans la figure 13.15 (nous les donnerons avec l'ordre anglais if-then-else, puisque c'est celui qu'on trouvera en programmant réellement ; if = si, then = alors, et else = autrement, sinon — *N. de l'E.).*

1) S2 inclus :

if $X \geq 0$ then NPOS = NPOS + 1 else NNEG = NNEG + 1
S1 et S2 sont de simples instructions.

2) S2 omis :

If $X \neq 0$ then $Y = 1/X$
Ici, aucune action n'est proposée si C est fausse ($X \neq 0$). S2 et « else » peuvent être omis dans ce cas.
Voici quelques exemples de la structure de boucle donnée figure 13.16 (donnés également avec l'ordre anglais « do-while » — *N. de l'E.) :*

---

*NDT : il existe même dans les langages les plus évolués (PL/1, PASCAL,...) une possibilité de définir les comparaisons qui déterminent les traitements exécutés ; par exemple case I of

2,K/5 : S1
N + M : S2

Si I vaut 2 ou K/5, effectuer S1 ; si I vaut N + M, effectuer S2 ; etc.

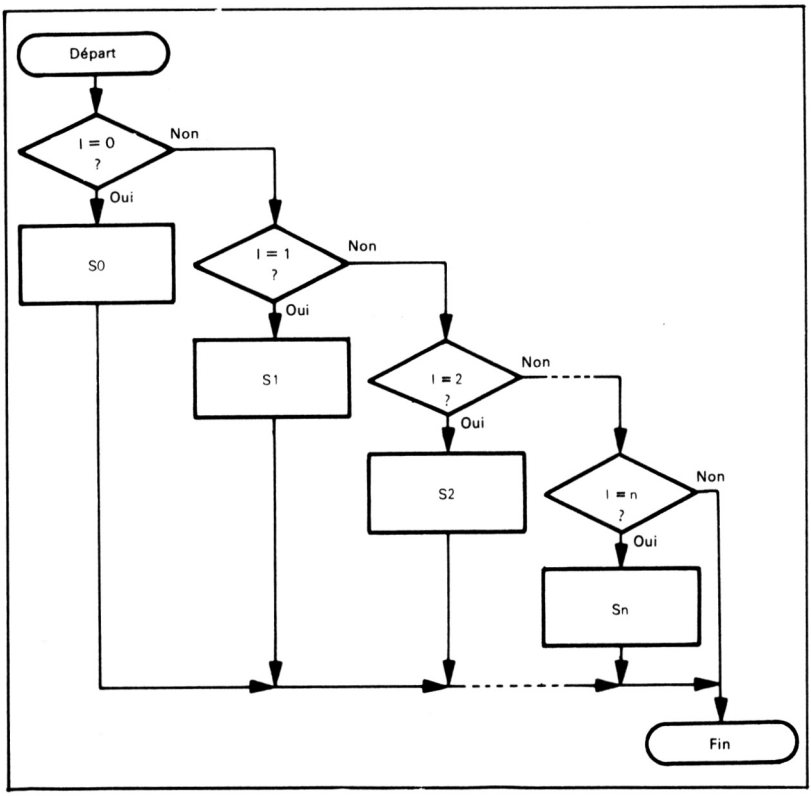


Fig. 13.18. — La structure « cas ».

1) Former la somme des entiers de 1 à N :

```
I = 0
SUM = 0
do while I<N
    I = I + 1
    SUM = SUM + I
end
```

L'ordinateur exécute la boucle tant que I<N. Si N = 0, le programme compris dans le « do-while » n'est plus exécuté. Le mot « end » (fin, en anglais) marque la fin.

2) Compter les caractères de la liste « PHRASE » jusqu'à l'arrivée d'un point codé en ASCII.

```
NCARACT = 0
do-while PHRASE (NCARACT) ≠ POINT
    NCARACT = NCARACT + 1
end
```

L'ordinateur exécute la boucle aussi longtemps que le caractère dans PHRASE n'est pas un point ASCII. Le compte est zéro si le premier caractère est un point.

AVANTAGES DE LA PROGRAMMATION STRUCTURÉE

Les avantages de la programmation structurée sont :

1) La séquence des opérations est simple à tracer. Cela vous permet de la tester et de la mettre au point facilement.
2) Le nombre de structures est limité et la terminologie est standardisée.
3) Les structures peuvent aisément se transformer en modules.
4) Les théoriciens ont prouvé que le jeu donné des structures est complet, c'est-à-dire que tous les programmes peuvent être rédigés en termes de ces structures.
5) La version structurée d'un programme est en partie auto-documentée et très facile à lire.
6) Les programmes structurés sont faciles à décrire avec des organigrammes ou d'autres méthodes graphiques.
7) La programmation structurée a montré, en pratique, qu'elle accroissait la productivité du programmeur.

INCONVÉNIENTS DE LA PROGRAMMATION STRUCTURÉE

La programmation structurée oblige à une plus grande discipline que la programmation modulaire. Le résultat est plus systématique et les programmes sont mieux organisés.
Les inconvénients de la programmation structurée sont :

1) Seuls, quelques langages haut niveau (par exemple, PL/M, Pascal) acceptent directement les structures. Le programmeur doit, par conséquent, passer par une étape supplémentaire de traduction pour convertir les structures en code d'assemblage. La version structurée du programme restera cependant utile pour la documentation*.
2) Les programmes structurés sont souvent plus lents d'exécution et emploient davantage de mémoire que leurs versions non structurées.
3) La limitation aux quatre structures de base rend parfois certaines tâches embarrassantes.
Dire que les trois structures sont complètes signifie seulement que tous les programmes peuvent être conçus avec elles : cela ne veut pas dire qu'un programme donné sera élaboré de façon plus efficace ou plus commode.
4) Les structures standards introduisent souvent quelque confusion ; par exemple, des « if-then-else » emboîtées peuvent être très difficiles à lire car il peut ne pas y avoir d'indication claire précisant quand la structure interne se termine. Une série de boucles « do-while » emboîtées peut aussi être difficile à lire. L'indentation reste cependant un bon moyen de contrecarrer cet inconvénient.
5) Les programmes structurés ne considèrent que la séquence des opérations du programme, et non le flot des données. Par conséquent, les structures traiteront parfois les données maladroitement.
6) Peu de programmeurs ont pris l'habitude de la programmation structurée. Nombreux sont ceux qui trouvent les structures gênantes et restrictives.

---

*NDT : ce défaut est de moins en moins sensible à l'heure actuelle, tous les langages de la nouvelle génération (C, ADA par exemple) étant structurés. ZILOG propose même un assembleur structuré pour ses microprocesseurs, et pour le Z80 en particulier.

QUAND EMPLOYER LA PROGRAMMATION STRUCTURÉE

Nous ne nous faisons ni l'avocat, ni l'accusateur de la programmation structurée et ne voulons décourager personne de l'utiliser. Elle constitue une façon de systématiser la conception des programmes, et sera très intéressante dans les cas suivants :

- Dans les grands programmes, excédant probablement les 1000 instructions.
- Dans les applications où l'occupation mémoire n'est pas critique.
- Dans les applications en faible série, où les coûts du développement du logiciel, particulièrement le test et la mise au point, sont des facteurs importants.
- Dans les applications impliquant des traitements de chaînes, des contrôles de processus et autres algorithmes, plutôt que dans celles traitant seulement des bits.

A l'avenir, le prix des mémoires devrait encore décroître et la taille moyenne des programmes pour microprocesseurs s'allonger, ce qui accroîtrait les coûts de développement du logiciel. Aussi, des méthodes telles que la programmation structurée, qui réduit les coûts du logiciel pour de grands programmes mais exige davantage de mémoire, devraient être plus intéressantes.

Le fait que ces structures sont généralement définies en langage haut niveau ne signifie pas qu'elles ne sont pas applicables en assembleur. Au contraire, **le langage assembleur, avec la liberté complète d'expression que ce niveau de programmation procure, a besoin des concepts que fournit la programmation structurée. La création de modules à une seule entrée et une seule sortie, l'emploi de structures simples et le maintien de chaque module à un niveau de complexité minimale, rendent le codage en langage d'assemblage plus efficace.**

EXEMPLES

Réponse à un interrupteur

La version structurée de ce programme est :

PROGRAMMATION STRUCTURÉE DU SYSTÈME À INTERRUPTEUR ET VOYANT

```
INTER = OFF
do while INTER = OFF
    LECTURE INTER
end
VOYANT = ON
DELAI (1)
VOYANT = OFF
```

Les mots ON (en marche) et OFF (arrêté) doivent disposer de définitions propres à l'interrupteur et au voyant. On suppose que DELAI est un module fournissant une temporisation dont la durée est spécifiée, en secondes, par le paramètre.

Une instruction, dans un programme structuré, peut en fait être un sous-programme. Cependant, et pour se conformer aux règles édictées, le sous-programme ne peut avoir d'autre sortie que vers le programme appelant.

Puisque « do-while » teste la condition avant l'exécution de la boucle, on positionne la condition INTER sur OFF avant de démarrer. Le programme structuré est directement lisible et aisé à tester à la main. Cependant, il occupera probablement davantage de mémoire que sa version non structurée, où on n'aurait pas eu à initialiser INTER et où l'on aurait pu combiner les procédures de lecture et de test.

Le chargeur à interrupteurs

PROGRAMMATION STRUCTURÉE DU PROGRAMME CHARGEUR DE MÉMOIRE

Le programme chargeur de mémoire à interrupteurs est un exemple bien plus complexe de programmation structurée. On peut réaliser l'organigramme de la figure 13.9 comme suit (les astérisques indiquant des commentaires) :

```
★
★ INITIALISATION DES VARIABLES
★
ADRESHAUTE = 0
ADRES BASSE = 0
★
★ CE PROGRAMME UTILISE UNE STRUCTURE DO-WHILE SANS CONDITION,
★ APPELEE SIMPLEMENT « DO-FOREVER » (FAIRE ETERNELLEMENT). LE
★ SYSTEME EXECUTE CONTINUELLEMENT LE PROGRAMME CONTENU
★ DANS LA BOUCLE DO-WHILE
★
do forever
★
★ TEST DU POUSSOIR ADRESHAUTE, EXECUTION DU PROCESSUS
★ REQUIS SI CE POUSSOIR EST ACTIONNE
★
  if POUSSADRESHAUT = 1 then begin
    ADRESHAUTE = INTER
    VOYANTS = INTER
    do
      DELAI (ANTIREBONDS)
    until POUSSADRESHAUTE = 1
    end
  end
★
★ TEST DU POUSSOIR CHARGERADRES ; EXECUTER TRAITEMENT
★ ADRESSE BASSE S'IL EST ACTIONNE
★
  if POUSSOIRCHARGE = 1 then begin
    CHARGERADRES = INTER
    VOYANTS = INTER
    do
      DELAI (ANTIREBONDS)
    until POUSSOIRCHARGE = 1
    end
```

```
★
★ TEST POUR BOUTONDONNEES ET RANGEMENT DONNEES EN
★ MEMOIRE S'IL EST ACTIONNE
★
  if BOUTONDONNEES = 1 then begin
    DONNEES = INTER
    VOYANTS = INTER
    (ADRESHAUTE, ADRESBASSE) = DONNEES
    do
      DELAI (ANTIREBONDS)
    until BOUTONDONNEES = 1
      DELAI (ANTIREBONDS)
    end
★ end
★
★ LE DERNIER end TERMINE LA BOUCLE do forever.
```

Les programmes structurés ne sont pas faciles à rédiger mais ils peuvent procurer une vue considérable sur la logique globale du programme. On peut tester la logique du programme structuré à la main, avant même de codifier la première instruction.

Terminal de vérification des cartes de crédit

PROGRAMMATION STRUCTURÉE POUR LE TERMINAL DE CARTES DE CRÉDIT

MODULE STRUCTURÉ POUR CLAVIER

Examinons maintenant les entrées du clavier de notre terminal de transaction. On suppose que la zone d'affichage est AFFICH, le strobe du clavier est STROBE, et la donnée du clavier est DONNÉE. **Le programme structuré sans les touches de fonctions est :**

```
NTOUCHES = 10
★
★ INITIALISATION AFFICH
★
  do while NTOUCHES >0
    NTOUCHES = NTOUCHES -1
    AFFICH (NTOUCHES) = 0
  end
★
★ ACQUISITION D'UNE ENTREE COMPLETE DU CLAVIER
★
  do while NTOUCHES <10
    if STROBE = ACTIF then begin
      STROBE = INACTIF
      AFFICH (NTOUCHES) = DONNEE
      NTOUCHES = NTOUCHES +1
  end
```

L'adjonction de la touche d'émission signifie que le programme devra ignorer les chiffres suivant l'acquisition d'une entrée complète, et devra ignorer la touche d'émission tant que l'entrée n'est pas complète. Le programme structuré devient :

```
   NTOUCHES = 10
★
★ INITIALISATION AFFICH
★
   do while NTOUCHES >0
      NTOUCHES = NTOUCHES - 1
      AFFICH (NTOUCHES) = 0
   end
★
★ ATTENDRE UNE ENTREE COMPLETE SUIVIE DE LA
★ FRAPPE DE LA TOUCHE EMISSION
★
   do while TOUCHE ≠ EMISSION OU NTOUCHES ≠ 10
      if STROBE = ACTIF then begin
        STROBE = INACTIF
        TOUCHE = DONNEE
        if NTOUCHES ≠ 10 et TOUCHE ≠ EMISSION then begin
          AFFICH (NTOUCHES) = TOUCHE
          NTOUCHES = NTOUCHES + 1
          end
        end
   end
```

Notons les particularités suivantes de ce programme structuré.

1) Le second if-then est emboîté dans le premier puisque les données provenant des touches ne sont introduites qu'après qu'un strobe ait été reconnu. Si le second if-then était placé sur le même niveau que le premier, la valeur d'une seule touche pourrait remplir la zone AFFICH, puisqu'elle serait stockée à chaque itération de la boucle do-while.
2) TOUCHE n'a pas besoin d'être préalablement défini, puisque NTOUCHES part de zéro à l'initialisation de la zone.
L'adjonction du poussoir RAZ (remise à zéro) permet au programme de supprimer l'entrée en simulant une pression de RAZ, c'est-à-dire en initialisation NTOUCHES à 10 et TOUCHE à RAZ avant de démarrer. Le programme structuré ne devra alors effacer que les chiffres préalablement stockés. **Le nouveau programme structuré est :**

```
★
★ SIMULATION DE L'EFFACEMENT
★
NTOUCHES = 10
TOUCHE = RAZ
★
★ ATTENDRE UNE ENTREE COMPLETE ET LA TOUCHE EMISSION
★
do while TOUCHE ≠ EMISSION ou NTOUCHES ≠ 10
★
★ EFFACER L'ENTREE COMPLETE SI TOUCHE RAZ FRAPPEE
★
★ if TOUCHE = RAZ then begin
     TOUCHE = 0
     do while NTOUCHES >0
     NTOUCHES = NTOUCHES -1
     AFFICH (NTOUCHES) = 0
     end
  end
★
★ ACQUISITION CHIFFRE SI ENTREE INCOMPLETE
★
★ if STROBE = ACTIF then begin
     STROBE = INACTIF
     TOUCHE = DONNEE
     if TOUCHE <10 ET NTOUCHES ≠ 10 then begin
       AFFICH (NTOUCHES) = TOUCHE
       NTOUCHES = NTOUCHES +1
       end
     end
end
```

Remarquons que le programme remet TOUCHE à zéro après avoir effacé la zone, de telle façon que l'opération ne soit pas répétée.

MODULE DE RÉCEPTION STRUCTURÉ

De façon similaire, on peut élaborer un programme structuré pour la réception. Un programme initial effectue uniquement la recherche des caractères d'en-tête et de fin. On suppose que RSTB est l'indicateur d'un caractère prêt. **Le programme structuré est :**

```
★
★ INITIALISATION INDICATEUR D'EN-TETE
★
INDET = 0
★
★ ATTENTE D'EN-TETE ET FIN DE MESSAGE
★
do while INDET = 0 ou CAR ≠ FIN
★
★ ACQUISITION CARACTERE SI PRET, RECHERCHE EN-TETE
★
   if RSTB = ACTIF then begin
     RSTB = INACTIF
     CAR = ENTREE
   if CAR = EN-TETE then INDET = 1
   end
end
```

On peut maintenant ajouter la section testant l'adresse du message et comparer celle-ci aux trois chiffres d'adresse du terminal (TERMADR). S'il n'y a pas correspondance parfaite, l'indicateur ADRESOK est mis à 1.

```
★
★ INITIALISATION INDICATEURS D'EN-TETE ET DE CORRESPONDANCE
★ ET COMPTEUR D'ADRESSE
★
INDET = 0
ADRESOK = 0
ADRESCPTR = 0
★
★ ATTENTE D'EN-TETE, ADRESSE DESTINATAIRE ET FIN DE MESSAGE
★
do while INDET ≠ 0 ou CAR = FIN ou ADRESCPTR ≠ 3
★
★ ACQUISITION CARACTERE SI DISPONIBLE
★
   if RSTB = ACTIF then begin
     RSTB = INACTIF
     CAR = ENTREE
   end
```

```
★
★ TEST DE L'ADRESSE TERMINAL ET EN-TETE
★
   if INDET = 1 et ADRESCPTR ≠ 3 then begin
        ADRESOK = 1
        ADRESCPTR = ADRESCPTR +1
      end
   if CAR = EN-TÊTE then INDET = 1
end
```

Le programme attend maintenant l'en-tête, un code d'identification sur 3 chiffres, et une terminaison. Soyez attentif à ce qui se passe durant les itérations, lorsque le programme trouve l'en-tête, et à ce qui arriverait si un caractère erroné du code d'identification était le même que la fin de message.
Une nouvelle addition rangera le message à MESSG ; NMESS est le nombre de caractères du message ; si sa valeur finale n'est pas nulle, le programme sait que le terminal a reçu un message valide. On n'a pas essayé de minimiser les expressions logiques dans ce programme :

```
★
★ INITIALISATION INDICATEURS, COMPTEURS
★
INDET = 0
ADRESOK = 0
ADRESCPTR = 0
NMESS = 0
★
★ ATTENTE D'EN-TETE, ADRESSE DESTINATAIRE ET TERMINAISON
★
do while INDET = 0 ou CAR ≠ FIN ou ADRESSCPTR ≠ 3
★
★ ACQUISITION CARACTERE SI DISPONIBLE
★
   if RSTB = ACTIF then begin
     RSTB = INACTIF
     CAR = ENTREE
   end
★
★ LIRE MESSAGE SI ADRESSE DESTINATAIRE = ADRESSE TERMINAL
★
   if INDET = 1 AND ADRES CPTR = 3 then
     if ADRESOK = 0 et CAR = FIN then begin
       MESSG (NMESS) = CAR
       NMESS = NMESS + 1
   end
```

```
★
★ TEST DE L'ADRESSE DU TERMINAL
★
  if INDET = 1 etADRESCPTR ≠ 3 then
    CAR ≠ TERMADR (ADRESCPTR) then begin
    ADRESOK = 1
    ADRESCPTR = ADRESCPTR +1

★
★ RECHERCHE EN-TETE
★
  if CAR = EN-TETE then INDET = 1
end
```

Le programme ne teste le code d'identification que s'il a trouvé une en-tête pendant l'itération précédente. Il n'accepte le message que s'il avait d'abord trouvé une en-tête et une adresse destinataire complète, correspondant parfaitement. Le programme doit fonctionner correctement pendant les itérations, alors qu'il détecte l'en-tête, la fin de message et le dernier chiffre de l'adresse du destinataire. Il ne doit pas tenter de comparer l'en-tête avec l'adresse du terminal, ou placer la terminaison ou le chiffre final de l'adresse du destinataire dans le message. On peut ajouter le reste de la logique en s'inspirant de l'organigramme de la figure 13.13. Notons que l'ordre des opérations est souvent critique. On doit être sûr que le programme ne termine pas une phrase en démarrant la suivante au cours de la même itération.

RÉSUMÉ DE LA PROGRAMMATION STRUCTURÉE

La programmation structurée introduit de la discipline dans la conception des programmes. Elle contraint à s'en tenir aux structures imposées dans les séquences d'opérations. Elle fournit des structures à une seule entrée et une seule sortie que l'on peut tester afin de contrôler leur logique. La programmation structurée permet souvent de s'apercevoir des incompatibilités des entrées ou de leurs combinaisons possibles. La programmation structurée n'est pas une panacée, mais elle apporte de l'ordre dans un processus qui, autrement, pourrait être chaotique. Elle doit également aider à la mise au point, au test et à la documentation.

Cependant, la programmation structurée n'est pas simple. Le programmeur a non seulement à définir le problème de façon adéquate mais il doit encore travailler la logique avec le plus grand soin. C'est astreignant et difficile, mais il pourra en résulter un programme fonctionnel et clairement écrit.

TERMINAISONS POUR STRUCTURES

Les structures particulières ci-dessus ne sont pas idéales et sont souvent embarrassantes. De plus, il peut être difficile de distinguer la fin de l'une et le début de l'autre, en particulier lorsqu'elles sont emboîtées. Les théoriciens pourraient proposer des structures meilleures, à l'avenir, les utilisateurs pouvant dans l'immédiat en ajouter qui leur soient propres. Un moyen de terminer chaque structure paraît souhaitable, car l'indentation ne clarifie pas toujours la situation. « End » (fin) est une terminaison logique pour une boucle

« do-while ». Il n'y a cependant pas de terminaison évidente pour l'ordre « if-then-else » ; quelques théoriciens on préconisé le « endif » ou le « fi » (« if » inversé), mais ils sont tous deux peu agréables et ne contribuent pas à la lisibilité des programmes.

RÈGLES POUR LA PROGRAMMATION STRUCTURÉE

Nous suggérons les règles suivantes pour la programmation structurée :

1) Commencer par rédiger un organigramme de base afin de définir la logique du programme.

2) Commencer par les structures « séquentielle », « if-then-else » et « do-while ». Elles constituent un jeu complet, c'est-à-dire qu'elles permettent de rédiger tout programme en termes de ces structures.

3) Indenter chaque niveau en utilisant des espaces pour le distinguer du niveau précédent, de façon à savoir à quelle structure appartient chaque instruction.

4) Utiliser des terminaisons pour chaque structure, par exemple « end » pour la structure « do-while » et « endif » ou « fi » pour la structure « if-then-else ». Les terminaisons plus l'indentation devraient rendre le programme raisonnablement clair.

5) Insister sur la simplicité et la lisibilité. Laisser des espaces, utiliser des noms significatifs et rendre les expressions aussi claires que possible. Ne pas essayer de minimiser la logique au prix de la clarté.

6) Commenter le programme de façon organisée.

7) Tester la logique. Essayer les cas extrêmes ou les conditions spéciales, ainsi que quelques cas d'exécution. Les erreurs logiques que l'on découvre à ce niveau ne reviendront plus vous empoisonner ultérieurement.

CONCEPTION DESCENDANTE

CONCEPTION ASCENDANTE

Il subiste le problème qui consiste à savoir comment tester et intégrer les modules ou les structures. Nous désirons évidemment diviser une tâche importante en sous-tâches. Mais comment tester les sous-tâches isolées et ensuite les rassembler ? La procédure dite ascendante (« bottom-up ») demande un travail supplémentaire de test et de mise au point et reporte l'intégration complète à la fin. Ce dont on a besoin est une méthode qui permette au test et à la mise au point d'intervenir dans l'environnement réel du programme et qui modularise l'intégration du système.

CONCEPTION DESCENDANTE

« STUBS » DU PROGRAMME

Cette méthode est la « conception descendante » (« top-down »). On commence par rédiger le programme superviseur global. On remplace les sous-programmes non définis par des unités factices de simulation d'enchaînement, auxquelles on conservera leur nom générique anglais de « stubs », faute d'un équivalent français ; il s'agit plus généralement de programmes temporaires enregistrant des entrées, fournissant la réponse à un problème de test sélectionné, ou ne faisant rien du tout. On teste alors le superviseur afin de vérifier si la logique est correcte.

EXTENSION DE STUBS DU PROGRAMME

On continue par l'extension des stubs. Chaque stub contiendra souvent des sous-tâches qui se présenteront temporairement comme des stubs. Ce processus d'extension, mise au point et test, continue jusqu'à ce que les stubs soient remplacés par des programmes réels. Notons que le test et l'intégration se produisent à chaque niveau plutôt qu'à la fin. On n'a plus besoin de drivers spéciaux ou de programmes de création de données. On a une idée précise de la situation exacte de la conception. **La méthode descendante suppose une programmation modulaire et est aussi compatible avec la programmation structurée.**

AVANTAGE DE LA MÉTHODE DESCENDANTE

INCONVÉNIENTS DE LA CONCEPTION DESCENDANTE

Les inconvénients de la conception descendante sont :
1) La conception globale peut ne pas s'harmoniser parfaitement avec le matériel du système.
2) On ne profitera peut-être pas des logiciels pré-existants.
3) Un stub convenable peut être difficile à rédiger, particulièrement si le même stub doit fonctionner correctement en des endroits différents.
4) La programmation descendante peut ne pas aboutir à des modules d'usage général.
5) Des erreurs au niveau supérieur peuvent amener des effets catastrophiques, alors que les erreurs en conception ascendante n'ont généralement que des effets limités à un module particulier.

Dans les projets de programmation importants, la conception descendante a montré qu'elle améliorait grandement la productivité des programmeurs. Cependant, presque tous ces projets ont utilisé quelques concepts de la programmation ascendante lorsque la conception descendante aurait abouti à imposer une grande quantité de travail supplémentaire.
La conception descendante est un outil utile qui ne doit pas être poussé à ses limites. Elle fournit la même discipline pour les tests et l'intégration des systèmes que la programmation structurée pour la conception des modules. La méthode est cependant d'application plus générale parce qu'elle ne suppose pas réellement l'utilisation de la logique programmée. Cependant, la conception descendante peut ne pas se révéler comme la méthode la plus efficace de conception.

EXEMPLES

Réponse à un interrupteur

CONCEPTION DESCENDANTE DU SYSTÈME À INTERRUPTEUR ET VOYANT

Le premier exemple de programmation structurée montrait également une conception descendante. Ce programme est ré-écrit ici :

```
INTER = OFF
do while INTER = OFF
   LECTURE INTER
end
VOYANT = ON
DELAI 1
VOYANT = OFF
```

Dans ce programme, OFF = ouvert ou éteint, et ON = fermé, ou allumé. **Tous ces ordres sont, en fait, des stubs puisqu'aucun n'est entièrement défini.** Par exemple, que signifie LECTURE INTER ? Si l'interrupteur constituait un bit du port d'entrée SPORT, il signifierait réellement :

INTER = SPORT AND SMASQ

où SMASQ est le masque du port S, avec un bit à 1 à la position appropriée.
De même, DELAI 1 signifie réellement (si c'est le processeur qui fournit la temporisation) :

```
REG = COMPTE
do while REG ≠ 0
   REG = REG - 1
end
```

COMPTE est la valeur menant à une temporisation d'une seconde. **La version développée du programme est alors :**

```
INTER = 0
do while INTER = 0
   INTER = SPORT AND SMASK
end
VOYANT = ON
REG = COMPTE
do while REG ≠ 0
   REG = REG - 1
end
VOYANT = NOT (VOYANT)
```

où NOT est l'ordre d'inversion logique. Ce programme est certainement plus explicite et pourrait plus aisément être traduit en instructions.

Le chargeur à interrupteurs

PROGRAMMATION DESCENDANTE DU CHARGEUR A INTERRUPTEURS

Cet exemple est plus complexe que le précédent, aussi devons-nous être plus systématiques. A nouveau, **le programme structuré contient des stubs.**
Par exemple, si le poussoir d'Adresses Hautes est un bit du port d'entrée CPORT « if POUSADRESHAUTE = 1 » signifie réellement :

1) Entrée du CPORT
2) Complément
3) ET logique avec AHMASK

où AHMASK comporte un 1 à la position appropriée et des 0 ailleurs. De même, « if BOUTONDONNEE = 1 » signifie réellement :

1) Entrée du CPORT
2) Complément
3) ET logique avec DAMASK

Ainsi, les stubs initiaux pourraient juste affecter des valeurs aux poussoirs, et par exemple :

POUSADRESBASSE = 0
POUSADRESHAUTE = 0
BOUTONDONNEE = 0

Un lancement du programme superviseur montrerait qu'on prend le chemin « else » implicite des structures « if-then-else » sans jamais lire les interrupteurs. De même, si le stub était :

POUSADRESHAUTE = 1

le programme superviseur resterait dans la boucle « do while POUSADRESHAUTE = 1 », attendant que le poussoir soit relâché. Ces simples essais testent la logique générale du programme.

On peut maintenant étendre chaque stub et voir si cette extension mène à un résultat général rationnel. Notez que la mise au point et le test procèdent désormais de façon directe et modulaire. Etendons le stub POUSADRESHAUTE = 1 à l'aide des instructions :

LECTURE CPORT
POUSADRESHAUTE = NOT (CPORT) AND AHMASK

Le programme doit attendre que le poussoir d'adresses hautes se ferme. Il doit alors afficher les valeurs des interrupteurs. Cette exécution teste la réponse propre du bouton-poussoir d'adresses hautes.

Puis, on développe le module du poussoir d'adresses basses :

LECTURE CPORT
POUSADRESBASSE = NOT (CPORT) AND ABMASQ

Avec le poussoir d'adresses basses fermé, le programme doit afficher sur les voyants les valeurs des interrupteurs. On teste ainsi la réponse propre du poussoir d'adresses basses. De la même manière, on pourra développer le module du poussoir de données et tester sa réponse. Le programme complet aura alors été testé de façon systématique.

Lorsque tous les stubs auront été développés, le codage, la mise au point et le test seront terminés. Naturellement, on doit savoir quels sont exactement les résultats à attendre de chaque stub. Cependant, nombreuses seront les erreurs de logique à être détectées à chaque niveau, avant extension ultérieure.

Le terminal de transaction

CONCEPTION DESCENDANTE DU TERMINAL DE VÉRIFICATION

Cet exemple dispose de davantage de niveaux de détails. **On pourrait démarrer par le programme suivant** (voir fig. 13.19 pour l'organigramme page 382) :

```
CLAVIER
ACK = 0
do while ACK = 0
   TRANSMIT
   RECEIVE
   end
AFFICH
```

Ici, CLAVIER, TRANSMIT (émission), RECEIVE (réception) et AFFICH sont des stubs de programmes qui devront être développés. CLAVIER, par exemple, pourra simplement placer un nombre de dix chiffres dans le buffer approprié.

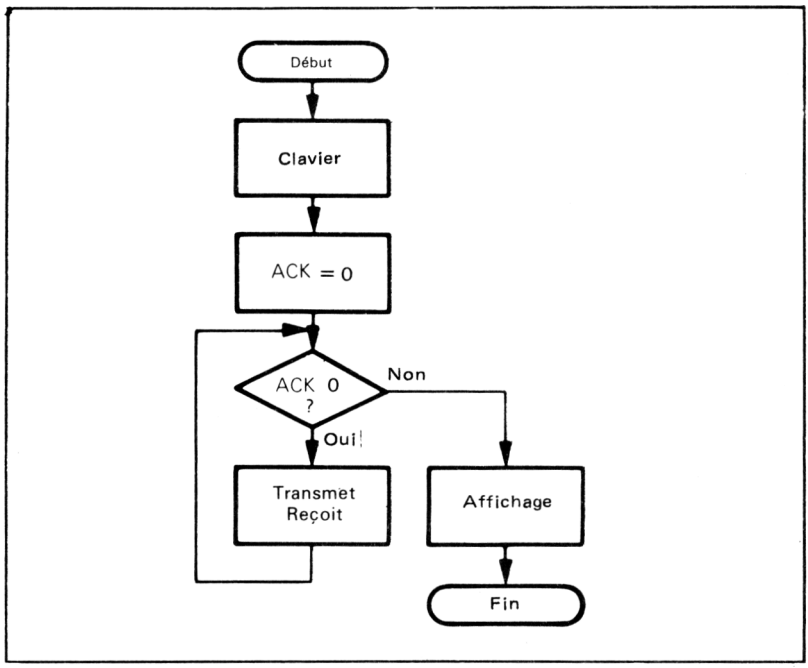


Fig. 13.19. — Organigramme initial du programme du terminal de transaction.

EXTENSION DE LA SÉQUENCE CLAVIER

La prochaine étape d'extension donnera par exemple le programme clavier suivant (fig. 13.20) :

```
VER = 0
do while VER = 0
   TERMINE = 0
   do while TERMINE = 0
   CLAENT
   CLAFF
   end
   VERIF
end
```

Ici, VER = 0 signifie que l'entrée n'a pas été vérifiée ; TERMINE = 0 signifie que l'entrée est incomplète. CLAENT et CLAFF sont les séquences d'entrées du clavier (CLAENT) et d'affichage (CLAFF). VERIFY teste les entrées. Un stub possible pour CLAENT consisterait simplement en une entrée aléatoire (provenant d'un générateur de nombres aléatoires), placée dans un buffer et mettant TERMINE à 1.

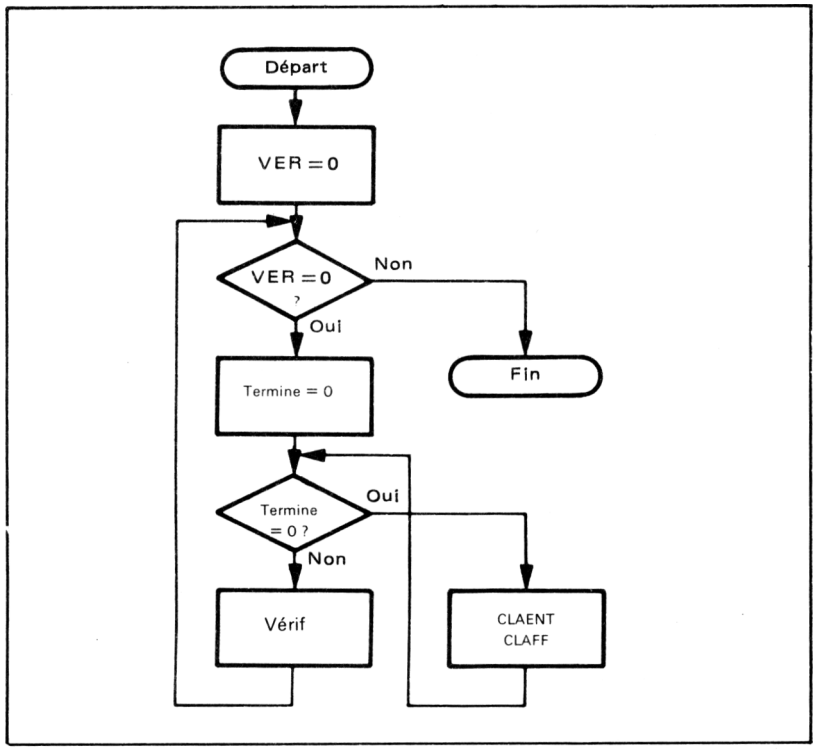


Fig. 13.20. — Organigramme de la séquence développée de clavier.

On peut continuer de façon similaire le développement, la mise au point et le test de TRANSMIT, RECEIVE et AFFICH. Notons que l'on doit étendre chaque module pas à pas et non intégrer le programme entier en une seule fois. C'est à vous de définir chaque étape. Un pas trop faible fait perdre du temps s'il est trop important, on retrouve les problèmes d'intégration que la procédure descendante était précisément censée résoudre.

RÉSUMÉ DE LA CONCEPTION DESCENDANTE

La conception descendante introduit de la discipline dans les phases de test et d'intégration lors de la conception d'un programme. Elle fournit une méthode systématique d'extension d'un organigramme ou d'une définition du problème jusqu'au niveau requis pour la rédaction du programme. Avec la programmation structurée, elle constitue un ensemble complet de techniques.

FORMAT POUR STRUCTURE DESCENDANTE

Tout comme la programmation structurée, la conception n'est pas simple. Le concepteur doit avoir soigneusement défini son problème et doit traiter systématiquement chaque niveau. A nouveau, cette méthodologie pourra paraître fastidieuse, mais les bénéfices peuvent être substantiels si l'on en respecte les règles.

Nous recommandons l'adoption de l'approche suivante :

1) Démarrer avec un organigramme de base.
2) Rendre les stubs aussi complets et distincts que possible.
3) Définir avec précision toutes les conséquences de chaque stub et sélectionner un jeu de tests.
4) Vérifier soigneusement et systématiquement chaque niveau.
5) Utiliser les structures **de la programmation structurée.**
6) Assurer l'extension de chaque stub pas à pas. Ne pas essayer d'en faire trop à la fois.
7) Surveiller attentivement les tâches communes et les structures des données.
8) Tester et mettre au point après chaque extension de stub.
9) Connaître les possibilités du matériel. Ne pas hésiter à s'arrêter et à faire un peu de conception ascendante là où cela paraît nécessaire.

RÉSUME DE LA DÉFINITION DES PROBLÈMES ET DE LA CONCEPTION DES PROGRAMMES

On remarquera qu'au cours de ce chapitre, on ne s'est référé à aucun microprocesseur spécifique ni à aucun langage assembleur, et qu'on n'a même pas rédigé une ligne de code. Heureusement, vous en connaissez bien davatange maintenant à travers les exemples que si vous aviez dû écrire les programmes à leur démarrage. Bien que la rédaction des instructions constitue une phase clé dans le développement du logiciel, elle en est réellement l'une des plus faciles.
Dès lors que vous aurez écrit quelques programmes, la phase de codage deviendra simple. Vous apprendrez vite le jeu d'instructions, vous reconnaîtrez les plus utiles, et vous vous souviendrez des séquences courantes qui participent à l'élaboration de grands programmes. Vous vous apercevrez alors que bien d'autres étapes dans le développement du logiciel restent difficiles et disposent de peu de règles claires.
Nous avons suggéré ici quelques méthodes pour systématiser les premières étapes. Dans la phase de définition des problèmes, vous devez définir chaque caractéristique du système — ses entrées, ses sorties, le traitement, les contraintes de temps et de mémoires, le traitement des erreurs. Vous devez considérer, en particulier, la façon dont le système répondra à un système plus vaste dont il constituerait une partie, et si ce vaste système comporte des équipements électriques, mécaniques, ou un opérateur. Vous devrez même démarrer à ce niveau afin de rendre le système facile à utiliser et à entretenir.
Dans l'étape de conception du programme, plusieurs techniques peuvent contribuer à systématiser les spécifications et la documentation logiques de votre programme. La programmation modulaire oblige à fractionner le programme complet en petits modules distincts. La programmation structurée fournit une méthode systématique de définition de la logique de ces modules, alors que la conception descendante est une méthode systématique d'intégration et de test. Naturellement, personne ne peut vous contraindre à suivre toutes ces techniques ; il s'agit, en fait, surtout de guides. Mais elles procurent une approche homogène aux tâches de définition et de conception, et vous devriez en tenir compte comme d'une base à partir de laquelle vous développerez votre propre approche.

RÉFÉRENCES

1. Voir par exemple, V.P. Srini, « Fault Diagnosis of Microprocessor Systems », *Computer,* janvier 1977, pp. 60-65. Pour une description de l'analyse de signature, voir G. Gordon et H. Nadig, « Hexadecimal Signatures Identify Trouble-spots in Microprocessor Systems », *Electronics,* 3 mars 1977, pp. 89-96. Il existe aussi une note d'application (#222) intitulée « A Designer's Guide to Signature Analysis » disponible chez Hewlett-Packard.
2. Pour une brève discussion de facteurs humains, voir G. Morris, « Make Your Next Instrument Design Emphasize User Needs and Wants », *EDN,* 20 octobre 1978, pp. 100-105.
3. D.L. Parnas (références ci-dessous) a été un leader de la programmation modulaire.
4. Rassemblé par B.W. Unger (voir référence ci-dessous).
5. Formulé par D.L. Parnas.

Les références suivantes fournissent des informations complémentaires :

Chapin, N., *Flowcharts,* Auerbach, Princeton, N.J. 1971.

Dahl O.J., C.A.R. Hoave et E.W. Dijkstra, *Strutured Programming,* Academic Press, New York, N.Y., 1972.

Dalton, W.F., « Design Microcomputer Software like Other Systems — Systematically », *Electronics,* 19 janvier 1978, pp. 97-101.

Dijkstra, E.W., *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, N.J., 1976.

Halstead, M.H., *Elements of Software Science,* American Elsevier, New York 1977.

Hughes, J.K. et J.I. Michtom, *A Structured Approach to Programming,* Prentice-Hall, Englewood Cliffs, N.J., 1977.

Morgan, D.E. et D.J. Taylor, « A Survey of Methods for Achieving Reliable Software », *Computer,* février 1977, pp. 44-52.

Myers, W., « The Need for Software Engineering », *Computer,* février 1978, pp. 12-25.

Parnas D.L., « On the Criteria to be Used in Decomposing Systems into Modules », *Communications of the ACM,* decembre 1972, pp. 1053-1058.

Parnas, D.L. « A Technique for the Specification of Software Modules with Examples », *Communications of the ACM,* mai 1973, pp. 330-336.

Schneider, V., « Prediction of Software Effort and Project Duration — Four New Formulas », *SIGPLAN Notices,* juin 1978, pp. 49-59.

Schneiderman, B. et al., « Experimental Investigations of the Utility of Detailed Flowcharts in Programming », *Communications of the ACM,* juin 1977, pp. 373-381.

Ulrickson R.W., « Software Modules Are the Building Blocs », *Electronic Design, 1er février 1977, pp. 62-66.*

Ulrickson R.W., « Solve Software Problems Step-by-Step » Electronic Design, 18 janvier 1977, pp. 54-58.

Unger, B.W., « Programming Languages for Computer System Simulation », *Simulation,* avril 1978, pp. 101-110.

Wirth, N., *Algorithms + Data Structures = Programs,* Prentice-Hall, Englewood Cliffs, N.J., 1976.

Wirth, N., *Systematic Programming : an Introduction,* Prentice-Hall, Englewood Cliffs, N.J., 1973.

Yourdon, E.U., *Techniques of Program Structure and Design,* Prentice-Hall, Englewood Cliffs, N.J., 1975.

CHAPITRE 14

MISE AU POINT ET TEST

Ainsi que nous l'avons noté au commencement du chapitre précédent, la mise au point ou « debugging » et le test constituent les étapes qui consomment le plus de temps dans le développement du logiciel. **Même en utilisant des méthodes telles que la programmation modulaire, la programmation structurée ou la conception descendante qui peuvent contribuer à simplifier les programmes et réduire la fréquence des erreurs, la mise au point et le test restent difficiles** car ils sont mal définis. La sélection d'un jeu convenable de données de test est rarement un processus clair et scientifique. La recherche des erreurs ressemble parfois à la recherche d'une aiguille dans une meule de foin. Rares sont certainement les tâches aussi frustrantes que le « debugging » de programmes.

Dans ce chapitre, on va d'abord décrire les outils disponibles d'aide à la mise au point. On discutera ensuite des procédures de base de « debugging », on présentera les types d'erreurs les plus communes, et on proposera quelques exemples de mise au point de programmes. La dernière section décrira comment sélectionner les données et les programmes de test.

On ne fera guère plus que décrire les objectifs des outils de debugging. La standardisation ne règne pas en ce domaine, et nous ne disposons pas d'assez d'espace pour passer en revue tous les systèmes et les programmes actuellement disponibles. Les exemples devraient vous fournir quelques idées sur l'utilisation, les avantages et les limites des aides particulières, matérielles et logicielles.

OUTILS SIMPLES DE MISE AU POINT

Les outils de « debugging » (mise au point, déverminage) les plus simples sont :

- **Une possibilité de fonctionnement en pas à pas.**
- **La possibilité d'introduire des points d'arrêt.**
- **Un programme de vidage de registres (« Dump »).**
- **Un programme de vidage de mémoire (« Dump »).**

PAS À PAS **Le pas à pas sert à exécuter une instruction à la fois. La plupart des micro-ordinateurs à base de Z80 offrent cette possibilité, car la circuiterie reste simple. Naturellement, les seules choses que l'on peut tester lorsque la machine travaille en pas à pas, ce sont les états des lignes de sorties que l'on surveille.** Les plus importantes de ces lignes sont :

- Le bus de données.
- Le bus d'adresses.
- Les lignes de contrôle MREQ (« Memory Request », accès mémoire), IORQ (« Input/Output Request », demande d'entrée/sortie), RD (« Read », lecture) et WR (« Write », écriture).

En surveillant ces lignes (par matériel ou par logiciel), on sera à même de constater la progression des adresses, des instructions et des données au fur et à mesure de l'exécution du programme. On peut ainsi déterminer quel type d'opération l'UC est en train d'exécuter. Ces informations seront suffisantes pour permettre de découvrir des erreurs telles que des branchements (JUMP) incorrects, des adresses incorrectes ou omises, des codes opérations erronés, ou des valeurs fausses pour les données. Cependant, on ne pourra examiner le contenu des registres et indicateurs sans possibilités supplémentaires de debugging, ou une séquence particulière d'instructions. La plupart des opérations du programme ne pourront être testées en temps réel.

LIMITES DU PAS À PAS

Le pas à pas ne pourra pas vous aider à déceler de nombreux types d'erreurs. Parmi celles-ci, les erreurs de cadencement et les erreurs sur les systèmes d'interruption et d'accès direct à la mémoire. En outre, le pas à pas est très lent ; il exécute généralement un programme à une vitesse moins rapide de un million de fois que le processeur lui-même. Cela signifie qu'un pas à pas dans une seconde de programme réel demandera plus de dix jours. Aussi, le pas à pas n'est-il utile que pour tester la logique de courtes séquences d'instructions.

POINTS D'ARRÊT

Un point d'arrêt est un point auquel le programme s'arrêtera automatiquement, de façon que l'utilisateur puisse examiner les états courants du système. Le programme ne repartira généralement pas tant que l'opérateur n'aura pas relancé l'exécution. Les points d'arrêt permettent de tester ou parcourir une section entière d'un programme. Ainsi, pour vérifier qu'une séquence d'initialisation est correcte, on peut introduire un point d'arrêt à son issue et lancer le programme. On vérifie alors les contenus des cases mémoires et des registres afin de s'assurer que la section complète est correcte. Noter que, si cette section ne fonctionnait pas comme espéré, il resterait encore à piéger l'erreur, soit avec d'autres points d'arrêts antérieurs, soit en mode pas à pas. Le point d'arrêt se dit « breakpoint », en anglais, d'où l'usage courant du mot abrégé BRKPT.

Les points d'arrêt sont un complément au pas à pas. On peut employer les points d'arrêt soit pour localiser une erreur, soit pour franchir une section de programme que l'on sait correcte. On peut alors opérer un « debugging » détaillé en pas à pas. Noter que les points d'arrêt n'affectent pas le séquencement du programme ; ils peuvent, par conséquent, servir à tester les entrées-sorties et les interruptions.

RST EN TANT QUE POINT D'ARRÊT

Souvent, les points d'arrêt utiliseront tout ou partie du système d'interruptions du microprocesseur. Certains microprocesseurs disposent d'une instruction spéciale d'« interruption logicielle » ou d'une « trappe » pouvant faire office de point d'arrêt.

Dans le Z80, si l'on n'utilise pas déjà tous les vecteurs des instructions RST dans un programme, on peut employer l'instruction RST (pour « RESTART », redémarrage) comme point d'arrêt. Le tableau 14.1 donne les adresses de destinations des diverses instructions RST. Le chapitre 12 en discute plus en détail.
Le programme de point d'arrêt peut commander l'impression des contenus des registres et mémoires, ou simplement attendre (en exécutant une instruction HALT ou un branchement conditionnel dépendant d'un interrupteur d'entrée) jusqu'à ce que l'utilisateur permette à l'ordinateur de poursuivre l'exécution. Si l'on n'utilise pas l'interruption masquable (INT) ou l'interruption non masquable (NMI), on peut utiliser ces vecteurs

Tableau 14.1. — Instruction RST du Z80 et adresses d'interruption

Instruction ou entrée externe (mnémonique)	Instruction code objet (hexa)	Adresse de destination (hexa)
RST 00H	C7	0000
RST 08H	CF	0008
RST 10H	D7	0010
RST 18H	DF	0018
RST 20H	E7	0020
RST 28H	EF	0028
RST 30H	F7	0030
RST 38H ou $\overline{\text{INT}}$ en Mode 1	FF	0038
$\overline{\text{NMI}}$		0066

comme des points d'arrêt à contrôle externe. Il faut cependant se rappeler que les interruptions (y compris NMI) et l'instruction RST recourent à la pile et à son pointeur pour sauvegarder l'adresse de retour. La figure 14.1 montre une séquence dans laquelle l'instruction RST aboutit à une boucle infinie. Le programmeur doit alors annuler ce point d'arrêt à l'aide d'un RESET ou d'un signal d'interruption.

```
        ORG     18H
RST18   EQU     18H
        JR      RST18     ; ATTENDRE SUR PLACE
```

Fig. 14.1. — Programme simple de point d'arrêt.

La méthode la plus simple d'insertion de points d'arrêt consiste à remplacer le 1er octet de l'instruction par une instruction RST ou de remplacer l'instruction par un branchement ou l'instruction CALL. L'emploi d'une instruction RST est préférable sur le Z80, puisqu'elle n'implique que le remplacement d'un octet, tandis que les instructions JP et CALL sont relatives à 3 octets. L'instruction JR ne convient pas pour des points d'arrêt parce qu'on ne peut garantir que le logiciel de test se trouve dans la zone − 126, + 129 octets de l'instruction sur laquelle a été placée le point d'arrêt. L'utilisation d'instructions sur plusieurs octets pour traiter des points d'arrêt peut poser des problèmes pour le Z80 à cause de l'existence d'instructions sur un octet.
Pour illustrer ce fait, examinons le segment de programme ci-après :

Adresse mémoire (Hexa)	Contenu mémoire (Hexa)	Instruction (Mnémonique)		
100	7B		LD	A,E
101	87	L1:	ADD	A,A
102	87	L2:	ADD	A,A

Si l'on désire placer un point d'arrêt à l'adresse 100_{16} en utilisant une instruction CALL ou JP sur 3 octets, le code des adresses 101_{16} et 102_{16} sera aussi « écrasé » par l'instruction CALL ou JP. C'est pourquoi le programmeur doit être conscient que ces adresses ont aussi été modifiées.
Un transfert de contrôle quelconque à L1 ou L2 en présence du point d'arrêt produit des résultats inattendus, sauf si le cas de figure a spécialement été prévu. Cette complication supplémentaire peut être évitée si l'on utilise l'instruction RST.

INSERTION DE POINTS D'ARRÊT

De nombreux moniteurs permettent facilement l'insertion ou le retrait de points d'arrêt obtenus par une instruction de branchement. De tels point d'arrêt ne modifient pas le cadencement du programme jusqu'à l'exécution du point d'arrêt. Cependant, cette procédure n'est pas valable si une partie du programme ou son entier se trouve en ROM ou en PROM. D'autres moniteurs réalisent des points d'arrêt en vérifiant effectivement des lignes d'adresse ou le compteur ordinal soit par logiciel, soit par matériel. Cette méthode autorise des points d'arrêt sur les adresses en ROM ou en PROM, mais elle peut modifier le cadencement du programme si l'adresse doit être vérifiée par logiciel. Un moyen plus puissant consisterait à ce que l'utilisateur rentre une adresse à laquelle le processeur donnerait le contrôle. Une autre possibilité serait de placer un retour dépendant de l'état d'un commutateur :

```
        ORG     18H
RST18   EQU     18H
        PUSH    AF          ; SAUVEGARDE ACCUMULATEUR, INDICATEURS
ATTNT:  IN      A,(PIODRA)  ; TEST COMMUTATEUR
        BIT     SW.A        ; COMMUTATEUR FERME ?
        JR      NZ,ATTNT    ; NON, ATTENDRE QU'IL LE SOIT
        POP     AF          ; RESTAURER ACCUMULATEUR, INDICATEURS
        RET
```

Il ne faut pas oublier de sauvegarder l'état du masque d'interruption si ce programme utilise une entrée d'interruption externe.

VIDAGE DE REGISTRES

Un vidage de registres consiste en un programme listant le contenu de tous les registres de l'UC. On ne peut généralement pas obtenir cette information directement. **Le programme suivant imprime le contenu de tous les registres sur une imprimante,** si l'on suppose que PRTHEX commande l'impression du contenu de l'accumulateur A sous forme de deux chiffres hexadécimaux. La figure 14.2 est l'organigramme du programme et la figure 14.3 montre un résultat typique.
On suppose que cette section de programme est appelée à l'aide d'une instruction CALL, qui range l'ancienne valeur du compteur ordinal au sommet de la pile.

```
;
; METTRE EN PILE LE CONTENU DE TOUS LES REGISTRES DE L'UC (PC DEJA SUR LA PILE)
;
        PUSH    AF      ; SAUVEGARDE REGISTRES UTILISATEUR
        PUSH    BC
        PUSH    DE
        PUSH    HL
        PUSH    IX      ; SAUVEGARDE REGISTRES D'INDEX
        PUSH    IY
        EX      AF,AF'  ; ACCES ET SAUVEGARDE BANQUE AUXILIAIRE
        EXX
        PUSH    AF
        PUSH    BC
        PUSH    DE
        PUSH    HL


;
; UTILISATION POINTEUR DE PILE COMME ADRESSE DE DEPART
;
        LD      HL,0    ; ACQUISITION POINTEUR PILE ORIGINAL
        ADD     HL,SP
        LD      DE,20   ; CALCUL POINTEUR PILE ORIGINAL
        ADD     HL,DE
        PUSH    HL      ; SAUVEGARDE EN PILE DU POINTEUR ORIGINAL


;
; IMPRESSION DU CONTENU DES REGISTRES
; ORDRE : PC (HAUT), PC (BAS), A,F,B,C,D,E,H,L,IX (HAUT), IX (BAS),
;     IY (HAUT), IY (BAS), A',F',B',C',D',E',H',L',SP (HAUT), SP (BAS)
;
        LD      B,22    ; NOMBRE OCTETS = 22
PRNT1:  DEC     HL
        LD      A,(HL)  ; ACQUISITION OCTET A PARTIR DE LA PILE
        CALL    PRTHEX  ; ET IMPRESSION DE L'OCTET
        DJNZ    PRNT1


;
; RESTAURATION DES REGISTRES A PARTIR DE LA PILE
;
        POP     HL      ; RETRAIT POINTEUR DE PILE (INUTILISE)
        POP     HL      ; RESTAURATION BANQUE AUXILIAIRE
        POP     DE
        POP     BC
        POP     AF
        EX      AF,AF'
        EXX
        POP     IY      ; RESTAURATION REGISTRES D'INDEX
        POP     IX
        POP     HL      ; RESTAURATION REGISTRES UTILISATEUR
        POP     DE
        POP     BC
        POP     AF
        RET
```

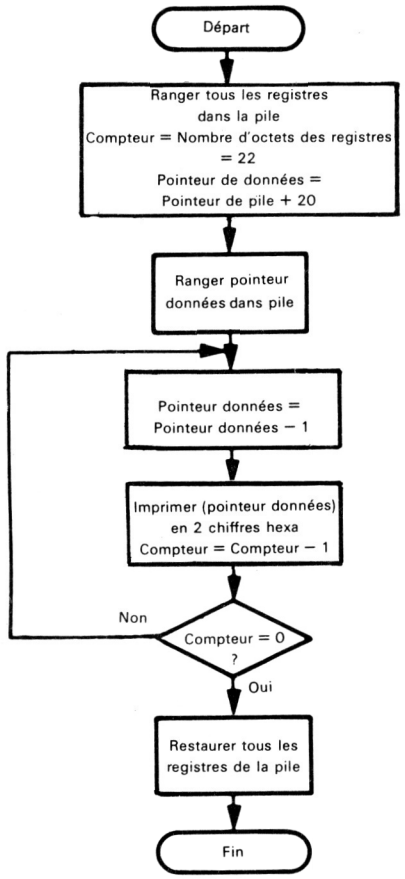


Fig. 14.2. — Organigramme d'un vidage de registres.

VIDAGE MÉMOIRE **Le programme de vidage mémoire liste le contenu de la mémoire sur un périphérique (une imprimante, par exemple).** Il est bien plus efficace d'examiner des zones entières de données ou des programmes entiers que de lire certaines cellules ça ou là. Cependant, le vidage de mémoires de grand volume n'est pas utile (sauf pour faire marcher le commerce de papier) en raison de la masse d'informations produite ; il peut aussi prendre beaucoup de temps si on utilise une imprimante lente. **De petits vidages fourniront cependant au programmeur un volume raisonnable d'informations qu'il pourra exa-**

```
1D      (A)
42      (F)
07      (B)
3E      (C)
23      (D)
01      (E)
17      (H)
01      (L)
D3      (IX)
58
E2      (IY)
A2
36      (A')
67      (F')
E8      (B')
11      (C')
EB      (D')
09      (E')
D7      (H')
66      (L')
68      (Pointeur de pile)
E2
```

Fig. 14.3. — Résultat d'un vidage typique de registres du Z80.

miner comme un tout. Des relations telles que des répétitions régulières d'organisations de données ou des décalages de listes complètes pourront devenir évidentes.
Un vidage général est souvent plutôt difficile à rédiger. Le programmeur doit se méfier des situations suivantes :

1) La taille de la mémoire excède 256 octets : un simple compteur 8 bits n'y suffira pas.
2) La position finale est une adresse plus petite que celle de départ : ce cas sera traité comme une erreur, car l'utilisateur désirera rarement lister entièrement la mémoire dans un ordre inverse.
L'efficacité du programme de vidage est peu importante, car la vitesse de vidage dépend de la vitesse du périphérique de sortie employé. **Le programme suivant s'arrêtera si l'adresse du départ est supérieure à l'adresse d'arrivée et traite des blocs de toutes longueurs.** On suppose que l'adresse de départ se trouve dans la paire de registres DE et l'adresse finale dans HL.

Remarque : la seule soustraction 16 bits est SBC qui soustrait le contenu d'un double registre et de l'indicateur de retenue de contenu du double registre HL. L'instruction SBC, comme les autres instructions de soustraction, met à 1 l'indicateur de retenue s'il y a effectivement une retenue (contrairement à ce que l'on trouve écrit dans quelques manuels du Z80).
Ce programme gère correctement les cas d'égalité des adresses de départ et de fin. L'utilisateur devra interpréter soigneusement les résultats si la zone de vidage inclut la pile, car le sous-programme de vidage utilise lui-même cette pile. PRTHEX peut aussi modifier les valeurs en mémoire et dans la pile.
Dans un vidage mémoire, les données peuvent être représentées de bien des façons différentes. Les formats courants sont les caractères ASCII ou bien deux chiffres hexadécimaux pour les valeurs 8 bits et 4 chiffres hexadécimaux pour les valeurs 16 bits. L'utilisation que l'on veut faire du vidage doit déterminer le choix du format. Cependant, il est

```
;
; ARRET SI ADRESSE FINALE < ADRESSE DEPART
;
        AND     A           ; RAZ RETENUE
        SBC     HL,DE       ; ADRESSE FINALE<ADRESSE DEPART ?
        JR      C,FIN       ; OUI, AUCUN VIDAGE
        EX      DE,HL       ; METTRE ADRESSE DEPART DANS HL
        INC     DE          ; COMPTEUR = NOMBRE D'ADRESSES QUI SONT
                            ;     VIDEES
;
; IMPRESSION DU CONTENU DES ADRESSES
;
DUMP:   LD      A,(HL)      ; LECTURE CONTENU D'UNE ADRESSE
        CALL    PRTHEX      ; ET IMPRESSION
        INC     HL
        DEC     DE
        LD      A,E         ; TOUTES LES ADRESSES VIDEES ?
        OR      D
        JR      NZ,DUMP     ; NON, POURSUIVRE VIDAGE
FIN:    HALT
```

```
23  1F  60  54  37  28  3E  00
6E  42  38  17  59  44  98  37
47  36  23  81  E1  FF  FF  5A
34  ED  BC  AF  FE  FF  27  02
```

Fig. 14.4. — Résultats d'un vidage typique de la mémoire.

presque toujours plus facile d'interpréter un code objet sous une forme hexadécimale plutôt qu'en ASCII.
Voici un exemple de format courant et utile d'un vidage :

```
1000 54 68 65 20 64 75 6D 70        THE DUMP
```

Chaque ligne se compose de 3 parties. La ligne commence par l'adresse hexadécimale du premier octet imprimé. Ensuite viennent 8 ou 16 octets représentés en hexadécimal. Enfin se trouve la représentation ASCII de ces mêmes 8 ou 16 octets. Essayez de ré-écrire le programme de vidage pour qu'il imprime l'adresse et les caractères ASCII ainsi que la forme hexadécimale du contenu de la mémoire.

OUTILS DE MISE AU POINT PLUS ÉVOLUÉS

Les outils de « debugging » les plus utilisés sont :
- **Les programmes de simulation pour tester le logiciel.**
- **Les analyseurs logiques pour tester les signaux et le cadencement.**

De nombreuses variantes de ces outils existent, aussi allons-nous discuter uniquement de leurs caractéristiques standard.

SIMULATION LOGICIELLE

Le simulateur est l'équivalent informatisé du papier et du crayon. **C'est un programme qui parcourt le cycle d'opération d'un autre ordinateur, conservant en mémoire les contenus de tous les registres, indicateurs et adresses en mémoire.** On pourrait, bien sûr, faire tout ceci à la main mais il faudrait y consacrer pas mal d'efforts et prêter une attention constante aux effets exacts de chaque instruction. Le programme simulateur ne se fatigue et ne se trompe jamais ; il n'oublie aucun registre ni instruction, et fournit des résultats imprimés.
La plupart des simulateurs sont des programmes importants rédigés en FORTRAN. Ils peuvent être achetés, ou utilisés sur les réseaux en temps partagé. Le simulateur du Z80 existe en plusieurs versions, selon les sources qui le proposent.

Les caractéristiques typiques des simulateurs sont :

1) La possibilité d'introduire des points d'arrêt. Généralement, les points d'arrêt sont installés après l'exécution d'un nombre particulier de cycles, lorsqu'une adresse mémoire ou une cellule d'un groupe de cellules mémoires a été altérée, ou sur d'autres conditions.
2) Des possibilités de vidage de registres ou mémoires, qui affichent les valeurs des adresses, des registres et des ports d'entrée/sortie.
3) Le fonctionnement en « trace », qui imprimera les contenus de registres ou cellules mémoires particulières dès lors que le programme a modifié leur contenu ou s'en est servi.
4) Des fonctions de chargement permettant d'établir des valeurs initiales ou de les changer au cours de la simulation.
Certains simulateurs peuvent aussi simuler des entrées-sorties, des interruptions et même des accès directs à la mémoire.

Le simulateur offre de nombreux avantages :

1) Il peut fournir une description complète des états d'un ordinateur, puisque le programme de simulation n'est pas limité par le nombre de broches ou d'autres considérations relatives aux circuits.
2) Il offre des points d'arrêt, des fonctions de vidage, de trace et autres, sans recourir aux mémoires ou au système de contrôle du processeur sous test. Ces avantages n'interfèrent donc pas avec les programmes de l'utilisateur.
3) Les programmes, les points de départ et d'autres conditions sont aisés à modifier.
4) Toutes les possibilités d'un ordinateur puissant, incluant les périphériques et le logiciel, sont mises à la disposition du concepteur du micro-ordinateur.
Par ailleurs, le simulateur souffre, lui aussi, de restrictions dues à sa base logicielle et à son indépendance à l'égard du micro-ordinateur réel. Voici les principales :

1) Le simulateur ne peut aider à résoudre les problèmes de cadencement, car il fonctionne bien plus lentement qu'en temps réel et ne modélise pas les matériels réels ni les interfaces employées dans l'application réelle.
2) Le simulateur ne peut recréer complètement le système d'entrées-sorties.
3) Le simulateur est généralement lent. Reproduire une seconde de traitement du processeur peut demander des heures de travail machine. L'utilisation du simulateur peut se révéler onéreuse.
Le simulateur représente l'aspect logiciel du « debugging » ; il offre les avantages typiques mais aussi les limites d'une approche totalement logicielle. Le simulateur permet de

voir ce qui se passe dans la logique interne d'un programme, mais il ne peut aider à résoudre des problèmes de cadencements, d'entrées-sorties, et d'autres problèmes matériels.

L'ANALYSEUR LOGIQUE

L'analyseur logique est la solution matérielle au « debugging ». Fondamentalement, l'analyseur est la version parallèle de l'oscilloscope. Il affiche des informations en binaire, hexa-décimal ou mnémonique sur un écran cathodique, et dispose de systèmes de déclenchement variés sur événements, seuils et entrées. La plupart des analyseurs comportent également une mémoire de façon à pouvoir afficher ce qui se passait auparavant sur les bus.

La procédure standard consiste à établir un événement de déclenchement, tel que l'apparition d'une adresse particulière sur le bus d'adresse, ou d'une instruction sur le bus de données. Par exemple, on pourra commander le déclenchement de l'analyseur lorsque le micro-ordinateur tentera de ranger une donnée dans une adresse particulière, ou d'exécuter une instruction d'entrée ou de sortie, ou, alors, on pourra surveiller la séquence d'événements précédant un point d'arrêt. **Les problèmes que l'on découvrira ainsi comprennent les impulsions parasites, des séquences incorrectes de signaux, des ondes se chevauchant, ainsi que d'autres erreurs de signaux ou de séquencement. Naturellement, un simulateur logiciel n'aurait pu être employé pour diagnostiquer de telles erreurs, pas plus d'ailleurs qu'un analyseur logique ne pourrait servir pour détecter une erreur de logique dans un programme.**

CARACTÉRISTIQUES IMPORTANTES DES ANALYSEURS LOGIQUES

Les analyseurs logiques varient grandement, sur des points parmi lesquels :

1) Le nombre de lignes d'entrées. Il en faut au moins 24 pour contrôler les 8 lignes du bus de données et les 16 du bus d'adresses. Il en faut encore pour les signaux de commande, les horloges, et d'autres entrées importantes.
2) Le volume de la mémoire. Chaque état antérieur mémorisé occupera plusieurs octets.
3) La fréquence maximale. Elle doit être de plusieurs mégahertz pour pouvoir suivre les processeurs les plus rapides.
4) La largeur minimale du signal détectable. C'est important lors de la capture d'impulsions parasites.
5) Le type et le nombre d'évènements de déclenchement prévus. Les caractéristiques importantes sont les délais de pré et post-déclenchement ; ils permettent à l'utilisateur d'afficher des évènements survenant avant ou après le déclenchement.
6) Les méthodes de connexion au micro-processeur. Elles peuvent requérir des interfaces plutôt complexes.
7) Le nombre de canaux affichés.
8) La forme d'affichage : binaire, hexadécimale, ou mnémonique.
9) Les formats d'affichage.
10) Les exigences sur les temps de maintien des signaux.
11) La capacitance de la sonde.
12) Les seuils, simples ou doubles.

Tous ces facteurs sont importants lorsqu'on compare divers analyseurs logiques (ou analyseurs de microprocesseurs) car ces instruments sont récents et non standardisés. Une variété énorme de produits est déjà disponible, et elle n'ira qu'en se développant à l'avenir.

Les analyseurs logiques ne sont nécessaires, en fait, qu'avec des séquencements complexes. Des applications simples avec des périphériques à faible vitesse se caractériseront par peu de problèmes matériels qui ne sauraient être traités avec un oscilloscope standard.

« DEBUGGING » AVEC LISTES DE TEST

Le concepteur ne peut manifestement pas tester un programme complet à la main ; cependant, certains points sensibles peuvent, eux, être aisément testés. **On peut tester systématiquement à la main ces points afin de détecter un grand nombre d'erreurs et ce, sans recourir aux outils de « debugging ».**

QU'INCLURE DANS UNE LISTE DE TEST ?

La question est : que faut-il tester ? La réponse est qu'il faut tester les points qui peuvent être traités avec des réponses oui-non, ou des calculs arithmétiques simples. N'essayez pas de faire des calculs complexes, de tester tous les indicateurs, ni même d'essayer tous les cas imaginables. Limitez votre test manuel à ce qui peut être établi facilement. Les problèmes complexes seront étudiés à l'aide des divers outils de « debugging ». Mais procédez systématiquement, établissez votre liste de test et assurez-vous que le programme exécute correctement les opérations de base.

La première étape consiste à comparer l'organigramme ou le programme structuré au code rédigé. Assurez-vous que tout ce qui apparaît dans l'un figure aussi dans l'autre. Une simple liste de test fera l'affaire. Il est facile d'oublier complètement un branchement ou une section de traitement.

Puis, concentrez-vous sur les boucles de programme. Vérifiez que tous les registres et toutes les cases mémoires participant aux boucles ont été initialisés avant usage. C'est là une source d'erreur commune ; à nouveau, une simple liste de test suffira.

Maintenant, examinez chaque branchement conditionnel. Sélectionnez un cas d'exécution produisant un branchement, et un autre n'en produisant pas. Le branchement s'effectue-t-il correctement ou est-il inversé ? Si le branchement implique le test d'un nombre pour savoir s'il est en-deça ou au-delà d'un seuil, essayez-le cas où il y a égalité. Se produit-il alors correctement ? Assurez-vous que votre choix reste compatible avec la définition du problème.

Considérez les boucles comme un ensemble. Expérimentez la première et la dernière itération à la main ; elles sont souvent sources d'ennuis particuliers. Que se passerait-il si le nombre d'itérations était nul, c'est-à-dire s'il n'y avait pas de données ou d'éléments dans une table ? Le programme s'en sortirait-il correctement ? Souvent, les programmes exécuteront une itération inutile ou, pire, décrémenteront le compteur, après avoir passé le zéro, avant de le tester.

Testez tout, soigneusement, jusqu'à la dernière instruction. Ne supposez pas (avec espoir) que la première erreur est la seule du programme. Le test manuel vous permettra de tirer le maximum de bénéfices des outils de « debugging », puisque vous vous serez déjà débarassé de nombreuses erreurs simples.

NDT : notons tout de même la disponibilité de systèmes de développement particuliers appelés émulateurs, qui permettent de remplacer l'UC d'une maquette à tester par une bande reliée à un système (l'émulateur) dont l'UC a ceci de particulier qu'elle permet un contrôle de ses signaux en temps réel.

QUESTIONS DU TEST MANUEL

Voici une revue rapide des questions du test manuel :

1) Retrouve-t-on dans le programme chaque élément figurant dans sa conception (et vice-versa, pour la documentation) ?
2) Les registres et cases mémoires intervenant dans les boucles ont-ils été initialisés avant usage ?
3) Tous les branchements conditionnels sont-ils corrects ?
4) Toutes les boucles commencent-elles et se terminent-elles normalement ?
5) Les cas d'égalité sont-ils judicieusement traités ?
6) Les cas banals sont-ils traités correctement ?

RECHERCHE DES ERREURS

ERREURS COURANTES

Bien sûr, en dépit de toutes ces précautions (ou si vous en sautez quelques-unes), vos programmes ne tourneront souvent pas encore. Le concepteur se retrouve alors avec le souci de trouver comment déceler les erreurs. La liste du test manuel constitue un bon début si vous ne l'avez pas encore utilisé ; quelques-unes des erreurs que vous n'auriez pas encore éliminées peuvent être :

1) Avoir omis d'initialiser des variables dans des compteurs, pointeurs, sommes, etc. Ne supposez pas que les registres ou cellules mémoires, ou indicateurs contiennent nécessairement zéro avant usage.

2) Avoir interverti un branchement conditionnel, par exemple, avoir utilisé un branchement sur retenue (carry) à 1 alors que vous vouliez un branchement sur carry = 0. Rappelez-vous les effets d'une comparaison ou d'une soustraction (A est le contenu de l'accumulateur, M celui d'un registre ou d'une mémoire) :

Zero	= 1	si $A = M$
	= 0	si $A \neq M$
Carry	= 1	si $A < M$
	= 0	si $A \geq M$

Notez particulièrement que CARRY = 0 si A = M, c'est-à-dire sur égalité. Ainsi, un JUMP sur carry à 1 signifie un saut sur $A<M$; un saut sur non carry (carry à 0) signifie un saut sur $A \geq M$. Si vous voulez l'égalité dans l'autre sens, essayez d'intervertir les rôles de A et M, on d'ajouter 1 à M. Par exemple, pour un JUMP sur $A \geq 10$, faites :

```
CP      10
JR      NC,ADRES
```

Si vous voulez un saut sur $A>10$, faites :

```
CP      11
JR      NC,ADRES
```

3) Avoir mis à jour des compteurs ou pointeurs à la mauvaise place (ou pas du tout). Assurez-vous qu'aucun chemin d'une boucle ne supprime ou ne répète les instructions d'initialisation.

4) Avoir omis d'imaginer ce qui se passe dans le cas trivial où, par exemple, il n'y a pas de données dans le buffer, il n'y a pas de test à effectuer, ou il n'y a pas d'entrée dans une transaction. Ne supposez jamais que de tels cas ne se produiront pas, sauf si le programme les élimine spécifiquement.

D'autres problèmes sur lesquels il vous faudra reporter votre attention sont :

5) Une inversion de l'ordre des opérandes. Rappelez-vous que l'instruction LD transfère le second opérande dans le premier. Par exemple, LD B,A met A dans B et non l'inverse.

6) Une modification des indicateurs avant leur emploi.

Rappelez-vous que les instructions INC et DEC, lorsqu'elles s'appliquent à un simple registre ou à une adresse, modifient tous les indicateurs excepté celui de retenue (carry). Rappelez-vous aussi que les instructions POP AF et EX AF,AF' modifient tous les indicateurs, et que les instructions logiques remettent l'indicateur de retenue à zéro.

7) Oublier de modifier les indicateurs alors qu'il fallait le faire.

Les indicateurs de zéro et de signe peuvent ne pas représenter le contenu de l'accumulateur car plusieurs instructions (LD spécialement) ne modifient pas les indicateurs. Il faut remarquer que l'incrémentation ou la décrémentation des doubles registres (par exemple INC HL ou DEC BC) et la complémentation de l'accumulateur (CPL) ne modifient en rien l'état des indicateurs.

8) Confondre les données et les adresses.

Rappelez-vous que LD HL, 1000H charge 1000 (hexa) dans HL, alors que LD HL,1000H charge le *contenu* des positions 1000 et 1001 dans HL. Une distinction semblable s'applique à LD A,COMPT et LD A,(COMPT).

9) Une ré-initialisation accidentelle d'un registre ou une cellule mémoire. Assurez-vous qu'aucune instruction de branchement ne transfère le contrôle encore une fois aux ordres d'initialisation.

10) Une confusion entre nombres et caractères.

Souvenez-vous que les représentations ASCII et EBCDIC des chiffes diffèrent des chiffres eux-mêmes. Le 7 en ASCII se code 37 en hexadécimal alors que 07 est le code ASCII de la sonnerie.

11) Une confusion entre binaire et décimal.

La représentation binaire des nombres est différente de la représentation BCD. Par exemple, 36 en BCD vaut 54 en décimal lorsqu'il est traité comme une constante hexadécimale.

12) Une inversion dans un ordre de soustraction. Soyez prudent avec les autres opérations aussi, avec la division par exemple, et veillez à ne pas intervertir les opérandes. Rappelez-vous que SUB M et CP M exécutent A – M et non M – A.

13) Avoir ignoré les effets des sous-programmes et des macros. Ne supposez pas que les appels de sous-programmes ou des références à des macros n'affectent pas les indicateurs ou les contenus des registres ou cases mémoires. Cernez avec précision leurs effets. Notez aussi qu'il est très important de bien les documenter, de façon que l'utilisateur puisse les déterminer sans avoir à analyser tout le listage.

14) Mal utiliser les instructions de décalage

Rappelez-vous les effets précis des instructions RLC, RL, RRC, RR, SLA, SRA et SRL. Elles représentent toutes des décalages sur 1 bit. Les deux instructions SLA et SRL mettent à zéro le bit vide. SRA préserve le signe (bit le plus significatif) en le décalant à droite. RLC et RRC sont des décalages circulaires qui n'incluent pas l'indicateur de retenue (carry) dans le registre circulaire ; par contre les décalages circulaires RL et RR

l'incluent. Rappelez-vous que ces instructions modifient tous les indicateurs, même si elles s'appliquent à une donnée en mémoire. Cependant les décalages sur un mot RLCA, RLA, RRCA et RRA ne modifient que l'indicateur de retenue.

15) Calculer incorrectement la longueur d'un tableau.

Rappelez-vous qu'entre les adresses 100 et 104, inclusivement, il y a 5 positions mémoire et non 4.

16) Confondre les registres et les paires de registres.

Gardez à l'esprit que les registres de l'UC et les paires de registres sont physiquement identiques. On peut les utiliser « simples » pour des données 8 bits ou doubles (par paire) pour des données 16 bits, mais jamais les 2 en même temps. On peut noter que l'instruction INC HL incrémente effectivement le registre L et ne modifie H que si L s'annule.

17) Confondre les registres 8 et 16 bits

L'accumulateur et d'autres registres de l'UC sont sur 8 bits alors que les registres d'index, le compteur ordinal, le pointeur de pile et les paires de registres sont sur 16 bits. On ne peut pas transférer le contenu d'un registre 16 bits dans un registre 8 bits et vice-versa.

18) Oublier que les nombres ou les adresses 16 bits occupent 2 positions mémoire.

L'instruction LD HL,(40H) charge le contenu des adresses 0040 et 0041 dans le double registre HL. De façon similaire, l'instruction PUSH DE range le contenu du double registre DE dans deux positions de la pile. Rappelez-vous aussi que le Z80 range toutes les valeurs sur 2 octets dans l'ordre bas/haut. Par exemple, l'instruction LD (40H),HL range le contenu du registre L à l'adresse 0040 et celui du registre H à l'adresse 0041.

19) Confondre la pile et le pointeur de pile.

Les instructions DEC, INC et LD modifient le pointeur de pile mais non le contenu de la pile. Les instructions PUSH et POP transfèrent les données en destination ou à partir de la pile. Rappelez-vous que les instructions CALL, RET, RETI, RETN et RST utilisent aussi la pile pour sauvegarder ou restaurer le compteur ordinal. La réponse à une interruption implique toujours la sauvegarde de l'ancienne valeur du compteur ordinal dans la pile, même si aucune instruction explicite n'est obtenue de façon externe (comme dans la réponse à NMI ou à INT en mode d'interruption 1 ou 2). Il faut noter que des instructions telles que EX (SP),HL ne modifient pas le pointeur de pile ; elles échangent les 2 positions du haut de la pile avec le contenu d'un double registre ou d'un registre d'index sans faire varier la longueur de la pile.

20) Oublier d'initialiser le pointeur de pile.

Rappelez-vous que vous devez placer l'adresse mémoire correcte dans le pointeur de pile avant tout appel à un sous-programme ou toute opération portant sur cette pile.

21) Modifier un registre ou une position mémoire avant emploi.

Rappelez-vous que l'instruction LD modifie le contenu du destinataire (et non de la source). Faites attention aux instructions qui utilisent implicitement certains registres — par exemple, DJNZ décrémente le registre B ; les instructions LDI, LDIR, LDD, LDDR, CPI, CPIR, CPD et CPDR décrémentent toutes le compteur d'octets dans le double registre BC et incrémentent ou décrémentent le double registre HL. Les instructions LDI, LDIR, LDD et LDDR incrémentent ou décrémentent aussi le double registre DE. Les instructions INI, INIR, IND, INDR, OUTI, OUTIR, OUTD et OTDR décrémentent toutes le registre B et incrémentent ou décrémentent le double registre HL.

22) Oublier de transférer le contrôle à des sections antérieures du programme ne devant être exécutées que dans des cas particuliers.

Rappelez-vous que le calculateur traite la mémoire programme de façon séquentielle sauf si un ordre spécifique le contraint à agir autrement.

MISE AU POINT (« DEBUGGING ») DE PROGRAMMES AVEC INTERRUPTIONS

Les programmes pilotés par des interruptions sont difficiles à mettre au point, car les erreurs peuvent survenir aléatoirement. Si, par exemple, le programme autorise les interruptions quelques instructions trop tôt, une erreur ne surviendra que si une interruption se manifeste lorsque le programme est en train d'exécuter ces quelques instructions. **En fait, on peut généralement supposer que des erreurs aléatoires sont provoquées par le système d'interruptions [2]. Les erreurs typiques des programmes avec interruptions sont :**

1) L'oubli de ré-autoriser les interruptions après en avoir accepté et servi une.
Le processeur inhibe automatiquement le système d'interruptions sur un reset ou en acceptant une interruption. Rappelons que l'on doit non seulement ré-autoriser les interruptions mais également souvent réinitialiser le signal cause d'interruption. Si cela n'est pas fait, la ligne d'interruption sera constamment prise en compte et considérée comme active.
2) Utiliser l'accumulateur avant de l'avoir sauvegardé, l'instruction PUSH AF doit précéder toute opération d'entrée ou de sortie relative à l'accumulateur.
3) Oublier de sauvegarder et de restaurer l'accumulateur et les indicateurs (double registre AF).
4) Restaurer les registres dans le mauvais ordre.
Si l'ordre de sauvegarde était :

```
PUSH    AF
PUSH    BC
PUSH    DE
PUSH    HL
```

l'ordre de restitution devra être :

```
POP     HL
POP     DE
POP     BC
POP     AF
```

5) Autoriser les interruptions avant d'en avoir établi les conditions nécessaires, telles que priorités, indicateurs, configurations PIO et SIO, pointeurs, compteurs, etc.
Une liste de test s'avèrera utile.
6) Laisser des résultats dans les registres et les détruire au cours du processus de restauration.
Ainsi qu'il l'a déjà été dit précédemment, les registres ne doivent pas servir à passer l'information entre le programme régulier et les programmes de gestion d'interruption.
7) Oublier que RST (et NMI) laisse une adresse dans la pile, qu'on l'utilise ou pas.
On aura peut-être à ré-initialiser ou remettre à jour le pointeur de pile.
8) Ne pas inhiber les interruptions lors de transferts multi-mots ou de séquences d'instructions.
Faire particulièrement attention aux situations où le programme de gestion d'interruption se sert des mêmes adresses que le programme régulier.
On peut espérer que ces listes vous donneront quelques idées sur la façon d'orienter vos recherches. Malheureusement, même un debugging systématique peut encore laisser subsister des problèmes vraiment embarrassants, surtout lorsque le système d'interruptions est impliqué [3].

EXEMPLE DE « DEBUGGING » 1 : Conversion décimale en 7-segments

MISE AU POINT D'UN PROGRAMME DE CONVERSION

Ce programme convertit un nombre décimal stocké à l'adresse mémoire 0040 en un code à 7 segments rangé en 0041. Il éteint l'affichage si l'adresse 0040 ne contient pas un nombre décimal.

Programme initial (organigramme de la figure 14.5) :

```
        LD      A,40H       ; ACQUISITION DONNEE
        CP      9           ; DONNEE : CHIFFRE DECIMAL ?
        JR      C,FIN       ; NON, GARDER CODE ERREUR
        LD      HL,(SSEG)   ; CHARGER ADRESSE BASE TABLE 7 SEGM.
        LD      D,A
        ADD     HL,DE       ; TROUVER ELEMENT PAR INDEXATION
        LD      A,(HL)      ; ACQUISITION CODE 7 SEGM.
FIN:    LD      (41H),A     ; SAUVEGARDER CODE 7 SEGM. OU ERREUR
        HALT
SSEG:   DEFB    3FH
        DEFB    06H
        DEFB    5BH
        DEFB    4FH
        DEFB    66H
        DEFB    6DH
        DEFB    7DH
        DEFB    07H
        DEFB    7DH
        DEFB    6FH
```

La procédure de la liste de test montrera alors que :

1) Le bloc mettant le résultat à zéro a été omis.
2) Le branchement conditionnel est incorrect.
Par exemple, si la donnée est 00, CP 9 met à 1 le carry car 0<9. Cependant le JUMP sur la condition inverse, (c'est-à-dire JR NC,FIN), ne produira toujours pas le bon résultat. Ici, c'est le cas d'égalité qui ne convient plus : cas où la donnée serait 9. CP 9 mettrait le carry à zéro et provoquerait le branchement. La version correcte est :

```
        CP      10          ; DONNEE : CHIFFRE DECIMAL ?
        JR      NC,FIN      ; NON, GARDER CODE ERREUR
```

Second programme :

```
        LD      B,0         ; ACQUISITION CODE ERREUR POUR ETEINDRE AFFICHAGE
        LD      A,40H       ; ACQUISITION DONNEE
        CP      10          ; DONNEE : CHIFFRE DECIMAL ?
        JR      NC,FIN      ; NON, GARDER CODE ERREUR
        LD      HL,(SSEG)   ; ACQUISITION ADRESSE BASE TABLE 7-SEGMENTS
        LD      D,A
        ADD     HL,DE       ; TROUVER ELEMENT PAR INDEXATION
        LD      A,(HL)      ; ACQUISITION CODE 7 SEGMENTS A PARTIR DE LA TABLE
FIN :   LD      (41H),A     ; SAUVEGARDER CODE 7 SEGMENTS OU ERREUR
        HALT
```

```
SSEG:   DEFB    3FH
        DEFB    06H
        DEFB    5BH
        DEFB    4FH
        DEFB    66H
        DEFB    6DH
        DEFB    7DH
        DEFB    07H
        DEFB    7DH
        DEFB    6FH
```

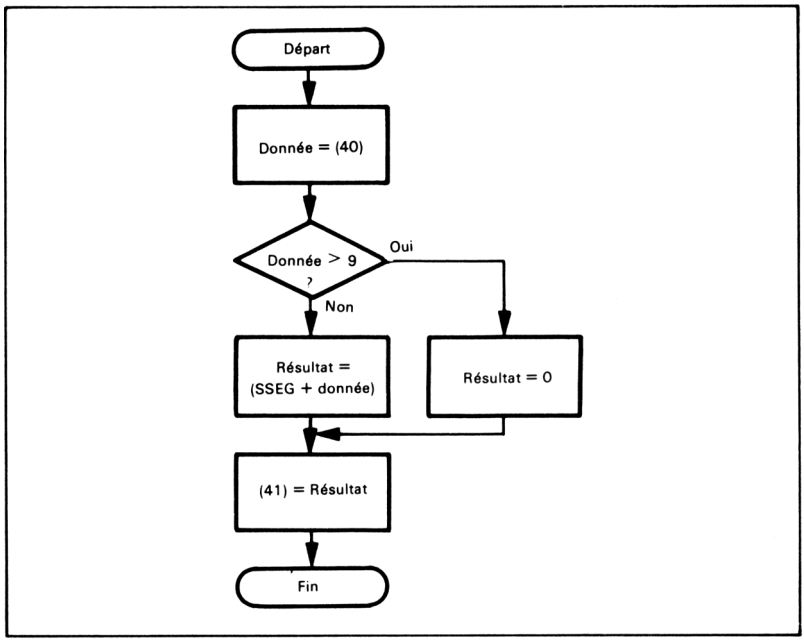


Fig. 14.5. — Organigramme de la conversion décimal en 7-segments.

Cette version a été testée manuellement avec succès.
Puisque ce programme est simple, la prochaine étape consiste à le parcourir en pas à pas avec de vraies données. Pour cet essais, on a choisi les données suivantes :

0 (le plus petit nombre)
9 (le plus grand nombre)
10 (cas limite, en dehors)
6B (hexa) (au hasard)

Le premier essai commence avec zéro à l'adresse 0040 (hexa). La première erreur est évidente — l'instruction LD A,40H charge le nombre 40 dans A et non le contenu de l'adresse 0040. L'instruction correcte est LD A,(40H). Après cette correction, le programme se déroule sans erreur apparente jusqu'à ce qu'il tente d'exécuter l'instruction LD A,(HL).

Le contenu du bus d'adresse durant la recherche de donnée est 0647, adresse qui n'existe même pas dans le système. A l'évidence, quelque chose ne va pas.
Il est alors temps de procéder à un test manuel. Puisque nous savons que l'instruction JR NC,FIN est correcte, l'erreur se trouve au-delà de cette instruction mais avant LD A,(HL). Un test manuel montre que :
1) LD HL,(SSEG) met 3F (hexa) dans L et 06 (hexa) dans H. C'est évidemment une erreur. Il nous faut LD HL,SSEG et non LD HL,(SSEG) puisque nous voulons charger l'adresse SSEG et non le contenu de cette adresse, dans le double registre HL.
2) LD D,A met 0 dans le registre D.
Ceci est faux — la donnée doit être placée en E, puisque nous voulons l'ajouter aux bits de poids faible de l'adresse du tableau. En fait, une instruction doit remettre à zéro le registre D, puisque le programme erroné n'initialise ni ne modifie l'autre moitié du double registre DE.

Troisième programme :

```
        LD      B,0         ; ACQUISITION CODE ERREUR POUR EFFACER AFFICHAGE
        LD      A,(40H)     ; ACQUISITION DONNEE
        CP      10          ; DONNEE : CHIFFRE DECIMAL ?
        JR      NC,FIN      ; NON, GARDER CODE ERREUR
        LD      HL,SSEG     ; ACQUISITION ADRESSE BASE DE LA TABLE 7 SEGM.
        LD      E,A
        LD      D,0         ; DONNEE = INDEX 16 BITS
        ADD     HL,DE       ; TROUVER ELEMENT PAR INDEXATION
        LD      B,(HL)      ; ACQUISITION CODE 7 SEGMENTS A PARTIR DE TABLE
FIN :   LD      A,B
        LD      (41H),A     ; SAUVEGARDE CODE 7 SEGMENTS OU CODE ERREUR
        HALT
SSEG:   DEFB    3FH
        DEFB    06H
        DEFB    5BH
        DEFB    4FH
        DEFB    66H
        DEFB    6DH
        DEFB    7DH
        DEFB    07H
        DEFB    7FH
        DEFB    6FH
```

Ce programme fournit les résultats suivants :

Donnée	Résultat
00	3F
09	6F
0A	0A
6B	6B

Ce programme ne remet donc pas le résultat à zéro si la donnée n'est pas valide, c'est-à-dire si elle est supérieure à 9. Le programme ne transfère pas, non plus, le code d'effacement de l'affichage dans le registre B. Parce que ce programme reste simple, il peut être testé pour tous les chiffres décimaux, ce qui donne :

Donnée	Résultat
0	3F
1	06
2	5B
3	4F
4	69
5	6D
6	7D
7	07
8	7D
9	6F

Notons que le chiffre 8 fournit un résultat faux : on devrait obtenir 7F. Parce que tout semble maintenant correct par ailleurs, on peut supposer que l'erreur réside dans le tableau. En fait, l'entrée du 8 dans le tableau a été mal recopiée.
Le programme final est ainsi :

```
; CONVERSION DECIMAL-7 SEGMENTS

        LD      B,0         ; ACQUISITION CODE ERREUR POUR EFFACER AFFICHAGE
        LD      A,(40H)     ; ACQUISITION DONNEE
        CP      10          ; DONNEE : CHIFFRE DECIMAL ?
        JR      NC,FIN      ; NON, GARDER CODE ERREUR
        LD      HL,SSEG     ; ACQUISITION ADRESSE DE BASE DE TABLE 7 SEGMENTS
        LD      E,A
        LD      D,0         ; DONNEE = INDEX 16 BITS
        ADD     HL,DE       ; TROUVER ELEMENT PAR INDEXATION
        LD      B,(HL)      ; ACQUISITION CODE 7 SEGMENTS A PARTIR DE LA TABLE
FIN :   LD      A,B
        LD      (41H),A     ; SAUVEGARDE CODE 7 SEGMENTS OU ERREUR
        HALT
SSEG :  DEFB    3FH
        DEFB    06H
        DEFB    5BH
        DEFB    4FH
        DEFB    66H
        DEFB    6DH
        DEFB    7DH
        DEFB    07H
        DEFB    7FH
        DEFB    6FH
```

Les erreurs rencontrées ici sont des erreurs types auxquelles le programmeur utilisant l'assembleur du Z80 devra s'attendre. Elles comprennent :

1) La non initialisation des registres ou adresses.
2) L'inversion des branchements conditionnels ou logiques.
3) Le branchement incorrect sur égalité.
4) Une confusion entre adressage immédiat et direct, donc entre données et adresses.
5) La non-distinction entre des données sur 8 bits et des adresses sur 16.
6) Un branchement à une mauvaise adresse, qui rend un chemin du programme incorrect.
7) Une erreur de transcription de liste.

Notons que les instructions de base telles que ADD, SUB, AND, etc., occasionnent rarement des problèmes. Une erreur particulièrement agaçante et à laquelle on doit porter toute son attention est l'inversion des opérandes dans les instructions LD. Maintes erreurs peuvent être éliminées à l'aide d'un langage de programmation d'un système de bas niveau tel que le PLZ/ASM [4].

EXEMPLE DE « DEBUGGING » 2 : tri par ordre décroissant

MISE AU POINT PROGRAMME DE TRI

Ce programme ordonne une liste de nombres binaires sur un octet non signé ; l'ordre est décroissant. La liste commence à l'adresse mémoire 41 et sa longueur est notée en 40.

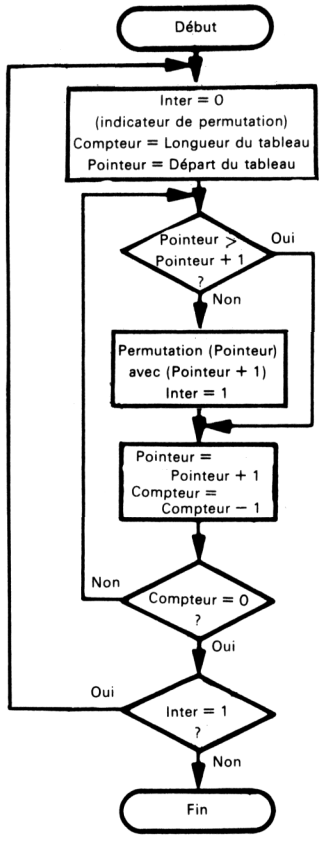


Fig. 14.6. — Organigramme de tri.

Programme initial (organigramme de la figure 14.6) :

```
        LD      C,0         ; INDICATEUR PERMUTATION = 0
        LD      A,(40H)     ; COMPTEUR = LONGUEUR TABLEAU
        LD      C,A
        LD      HL,41H      ; DEPART DU TABLEAU
PASS1:  LD      A,(HL)      ; ACQUISITION D'UN ELEMENT
        INC     HL
        CP      (HL)        ; EST-IL < ELEMENT SUIVANT ?
        JR      C,CNT       ; NON, PAS DE PERMUTATION
        LD      (HL),A      ; OUI, PERMUTATION ELEMENTS
        INC     HL
CNT:    DJNZ    PASS1
        DEC     C           ; INDICATEUR PERMUTATION = 1 ?
        JR      NZ,PASS1    ; OUI, AUTRE PASSE
        HALT
```

Le test manuel montre que tous les blocs de l'organigramme se retrouvent dans le programme et que tous les registres ont été initialisés. Les branchements conditionnels doivent être examinés avec soin. L'instruction JR C,CNT doit créer un branchement si la nouvelle valeur est inférieure ou égale à l'ancienne. Notons que le cas d'égalité ne doit pas provoquer de permutation, car on entrerait alors dans un processus sans fin.
Essayons avec un exemple :

```
(0040) = 30
(0041) = 37
```

L'instruction CP HL calcule 30 – 37 ; la retenue (carry) est 1. Cet exemple devrait provoquer une permutation, mais il n'en est rien. JR NC,CNT fournit le branchement correct dans ce cas. Si les deux nombres sont égaux, la comparaison met à zéro le carry et l'instruction JR NC,CNT est à nouveau correcte.
Qu'en est-il de JR NZ,PASS1 à la fin du programme ? S'il n'y a pas de permutation, B est à zéro et ainsi, le branchement n'est pas bon. L'instruction devrait être JR Z,PASS1.
Exécutons un premier essai. Les résultats de l'initialisation sont :

```
 A = COMPTEUR (CNT)
 B = COMPTEUR (CNT)
 C = 0
HL = 0041
```

Les instructions de la boucle donnent :

```
        LD      A,(HL)      ;A = (0041)
        INC     HL          ;HL = 0042
        CP      (HL)        ;(0041)-(0042)
        JR      NC,CNT
        LD      (HL),A      ;(0042) = (0041)
        INC     HL          ;HL = 0043
CNT:    DJNZ    PASS1       ;B = COMPTEUR – 1
```

Notons qu'on a déjà testé les instructions de saut (JUMP) conditionnel. A l'évidence, la logique est incorrecte. Si les deux premiers nombres sont dans le désordre, le résultat de la première itération devrait donner :

(0041) = ANCIEN (0042)
(0042) = ANCIEN (0041)
(HL) = 0042
(B) = COMPTEUR – 1

Or, nous avons :

(0041) = INCHANGE
(0042) = ANCIEN (0041)
(HL) = 0043
(B) = COMPTEUR – 1

L'erreur dans HL est facile à corriger. Le second INC HL est superflu et doit être supprimé. La permutation demande un peu plus de soin, ainsi qu'un registre temporaire :

```
        LD      D,(HL)
        LD      (HL),A
        DEC     HL
        LD      (HL),D
        INC     HL
```

Une permutation exige toujours une position intermédiaire dans laquelle un nombre sera rangé pendant qu'on transfère le second.

Avec les nouvelles modifications, le programme devient :

```
        LD      C,0         ; RAZ INDICATEUR PERMUTATION
        LD      A,(40H)     ; COMPTEUR = LONGUEUR DU TABLEAU
        LD      C,A
        LD      HL,41H      ; POINTEUR = VALEUR DEPART TABLEAU
PASS1:  LD      A,(HL)      ; ACQUISITION ELEMENT DU TABLEAU
        INC     HL
        CP      (HL)        ; ELEMENT < ELEMENT SUIVANT ?
        JR      NC,CNT      ; NON, PAS DE PERMUTATION
        LD      D,(HL)      ; OUI, PERMUTATION ELEMENTS
        LD      (HL),A
        DEC     HL
        LD      (HL),D
        INC     HL
CNT:    DJNZ    PASS1
        DEC     C           ; INDICATEUR PERMUTATION POSITIONNE ?
        JR      NZ,PASS1    ; OUI, AUTRE PASSE
        HALT
```

Que donne la dernière itération ? Supposons qu'il y ait 3 éléments :

(0040) = 03
(0041) = 02
(0042) = 04
(0043) = 06

A chaque tour, le programme incrémente HL de 1. Aussi, au départ de la 3e itération :

(HL) = 0041 + 2 = 0043

Les effets des instructions de la boucle sont les suivants :

```
LD      A,(HL)      ; A = (0043)
INC     HL          ; HL = (0044)
CP      (HL)        ; (0043) – (0044)
```

Ceci est incorrect, le programme testant une donnée hors de la table (0044). La correction consiste à diminuer de 1 le nombre d'itérations, ce qui peut être réalisé en ajoutant une instruction DEC B après LD A,(40H).

Examinons maintenant les cas banals, triviaux. Que se passe-t-il lorsque le tableau ne contient rien, ou seulement un élément ? Dans ce cas, le programme ne fonctionnera pas correctement et pourra changer tout un bloc de données improprement sans vous en alerter (essayez ce cas !). Les corrections sont, ici, simples mais essentielles : il en coûtera seulement quelques cases mémoires pour éviter des problèmes qu'il serait très difficile de situer ultérieurement.
Le nouveau programme est :

```
        LD      C,0         ; RAZ INDICATEUR PERMUTATION
        LD      A,(40H)     ; COMPTEUR = LONGUEUR DU TABLEAU
        CP      2           ; NOMBRE ELEMENTS TABLEAU > 2 ?
        JR      C,DONE      ; NON, PAS D'ACTION NECESSAIRE
        LD      B,A
        DEC     B           ; NOMBRE DE PAIRES D'ELEMENTS = COMPTEUR – 1
        LD      HL,41H      ; POINTEUR = VALEUR DEPART TABLEAU
PASS1:  LD      A,(HL)      ; LECTURE D'UN ELEMENT DU TABLEAU
        INC     HL
        CP      (HL)        ; ELEMENT < = ELEMENT SUIVANT ?
        JR      NC,CNT      ; NON, PAS DE PERMUTATION NECESSAIRE
        LD      D,(HL)      ; OUI, PERMUTATION ELEMENTS
        LD      (HL),A
        DEC     HL
        LD      (HL),D
        INC     HL
CNT:    DJNZ    PASS1
        DEC     C           ; INDICATEUR PERMUTATION POSITIONNE ?
        JR      NZ,PASS1    ; OUI, AUTRE PASSE
        HALT
```

Il est temps, maintenant, de tester le programme sur l'ordinateur ou le simulateur. Un examen de données serait :

```
(0040) = 02
(0041) = 00
(0042) = 01
```

Cet exemple consiste en deux éléments dans le désordre. Le programme demandera deux passes. La première produira :

```
(0041) = 01
(0042) = 00
  C    = 01
```

La seconde passe devrait terminer l'opération et produire :

(C) = 00

L'exécution serait plutôt longue en pas à pas, aussi allons-nous introduire des points d'arrêt. Chacun d'eux stoppe le déroulement et commande l'impression du contenu de tous les registres. Les points d'arrêt seront introduits :

1) Après LD HL,41H pour tester les conditions initiales.
2) Après CP (HL) pour tester la comparaison.
3) Après le second INC HL (juste avant le label CNT) pour tester la permutation.
4) Après DEC C pour tester l'exécution d'une passe complète.

Au premier point d'arrêt, les contenus des registres sont :

REGISTRE	CONTENU
A	02
B	01
C	00
H	00
L	41

Tout est correct ; le programme calcule bien les conditions initiales.

Au second point d'arrêt, on trouve :

	REGISTRE	CONTENU
	A	00
	B	01
	C	00
	H	00
	L	42
RETENUE	(CARRY)	1

C'est également correct. Le troisième point d'arrêt donne :

REGISTRE	CONTENU
	00
B	01
C	00
D	01
H	00
L	42

Le test de la mémoire donne :

(0041) = 01
(0042) = 00

Les résultats du quatrième point d'arrêt sont :

REGISTRE	CONTENU
A	00
B	01
C	00
D	01
H	00
L	42

Ici, le contenu de C n'est pas bon : il devrait être 1 pour indiquer qu'une permutation a été opérée. En fait, un coup d'œil au programme montre qu'aucune instruction n'était chargée de modifier C pour marquer un échange. La correction consiste à introduire l'instruction LD C,1 juste après JR NC,CNT.
La procédure consiste maintenant à charger le registre C avec la bonne valeur et à poursuivre. Le second point d'arrêt de la deuxième itération donne :

REGISTRE	CONTENU
A	00
B	00
C	00
H	00
L	43
CARRY	1

Il est clair, à nouveau, qu'une erreur s'est produite car on n'a pas ré-initialisé les registres (en particulier HL). Le branchement conditionnel après le test de l'indicateur de permutation devrait renvoyer au début du programme, et non au label PASS1.
La version finale devient ainsi :

```
TRI :   LD      C,0         ; INDICATEUR PERMUTATION = 0
        LD      A,(40H)     ; COMPTE = LONGUEUR TABLEAU
        CP      2           ; COMPTEUR ≥2 ?
        JR      C,FIN       ; NON, FIN
        LD      B,A
        DEC     B           ; NOMBRE DE PAIRES = COMPTE – 1
        LD      HL,41H      ; DEPART DU TABLEAU
PASS1:  LD      A,(HL)      ; ACQUISITION ELEMENT DU TABLEAU
        INC     HL
        CP      (HL)        ; ELEMENT < ELEMENT SUIVANT ?
        JR      NC,CNT      ; NON, PAS DE PERMUTATION
        LD      C,1         ; OUI, INDICATEUR DE PERMUTATION = 1
        LD      D,(HL)      ; PERMUTATION ELEMENTS
        LD      (HL),A
        DEC     HL
        LD      (HL),D
        INC     HL
FIN:    DJNZ    PASS1
        DEC     C           ; INDICATEUR PERMUTATION = 1 ?
        JR      NZ,TRI      ; OUI, AUTRE PASSE
        HALT
```

Ici, on ne peut plus tester tous les cas possibles. Deux exemples de données pour le « debugging » sont :

1) Deux éléments égaux :

(0040) = 02
(0041) = 00
(0042) = 00

2) Deux éléments déjà dans l'ordre décroissant :

(0040) = 02
(0041) = 01
(0042) = 00

INTRODUCTION AU TEST

TEST ET « DEBUGGING »

Le test des programmes est intimement lié au « debugging ». Quelques cas de test pourront certainement servir d'exemples de données utilisées pour le « debugging » :

- **Cas triviaux où il n'y a pas de données, ou une seule.**
- **Cas spéciaux que le programme met en évidence pour certaines raisons.**
- **Exemples simples testant des parties spéciales du programme.**

Dans le cas du programme de conversion décimal en 7 segments, ces cas couvrent toutes les situations possibles. Le test des données consiste à essayer :

- Les nombres de 0 à 9.
- Le cas limite 10.
- Le cas aléatoire 6B.

Le programme ne distinguera pas d'autres cas. **Ici, le « debugging » et le test sont virtuellement identiques.**
Avec le programme de tri, le problème est plus difficile. Le nombre d'éléments peut varier de 0 à 255, chacun de ceux-ci pouvant se situer n'importe où dans cette gamme. Aussi, le nombre de cas possibles est-il immense. De plus, le programme est modérément complexe. Comment, alors, sélectionner les données de test qui pourront nous procurer un certain niveau de confiance dans le programme ? **Le test requiert donc ici, quelques décisions de conception.** Le problème du test est particulièrement difficile lorsque le programme dépend de séquences de données en temps réel. Comment sélectionner les données, les générer, et les présenter au micro-ordinateur de façon réaliste ?

AIDES AU TEST

AIDES AU TEST

La plupart des outils mentionnés précédemment pour le « debugging » serviront aussi au test. Les analyseurs logiques ou de microprocesseurs aideront à tester le matériel ; les simulateurs aideront à tester le logiciel. D'autres outils pourront encore intervenir, et par exemple :

1) Des simulateurs d'entrées-sorties, pour simuler une variété de périphériques à partir d'un dispositif à une seule entrée et une seule sortie.

2) L'émulateur en ligne (« in-circuit emulator », en anglais), qui vous permet de connecter votre prototype à un système de développement ou à un tableau de commande et de le tester.
3) Un simulateur de ROM qui offre la souplesse d'emploi des RAM avec le cadencement des ROM ou PROM particulières devant intervenir dans votre système final.
4) Des systèmes d'exploitation en temps réel fournissant des entrées ou des interruptions à des instants spécifiques (ou peut-être aléatoires) et marquant l'éventualité d'une sortie. Des points d'arrêt en temps réel et un dispositif de trace peuvent également en faire partie.
5) L'émulateur (souvent sur ordinateur micro-programmable) qui fournit une vitesse d'exécution en temps réel et des E/S programmables [5].
6) Des interfaces permettant à un autre ordinateur de piloter le système d'E/S et de tester le programme de micro-ordinateur.
7) Des programmes de test permettant de traquer les erreurs logiques dans chaque branche du programme.
8) Des programmes de génération de test servant à créer des données aléatoires ou des distributions diverses.

Des théories sur les tests de programmes existent mais ne sont généralement applicables que pour des programmes très courts.
On doit veiller à ce que l'équipement de test n'invalide pas le test en modifiant l'environnement. Souvent, l'équipement de test pourra mémoriser, amplifier ou modifier les signaux en entrée ou en sortie. Si le système réel ne le fait pas, on peut être confronté à un comportement différent.
De plus, le logiciel supplémentaire de l'environnement de test peut consommer de l'espace mémoire ou employer une partie du système d'interruptions. Il peut provoquer des erreurs de restitution ou autres qui n'existeront plus dans votre système final. Un ensemble de test logiciel doit ainsi être aussi réaliste qu'un ensemble de test matériel, dans la mesure où une défaillance du logiciel peut se révéler aussi critique qu'une défaillance du matériel.

L'émulation et la simulation ne sont jamais précises, naturellement. Elles conviennent généralement au test de la logique, mais peuvent rarement tester les interfaces ou le cadencement. Par ailleurs, l'équipement de test en temps réel ne pourra guère fournir davantage qu'une vue générale de la logique du programme, mais pourra affecter l'interfaçage et le cadencement.

SÉLECTION DES DONNÉES DE TEST

Très peu de programmes réels peuvent être testés pour tous les cas. Le concepteur doit choisir un jeu d'échantillons capable de décrire toute la gamme des possibilités.

TEST STRUCTURÉ

Le test devrait bien évidemment faire partie de la procédure complète de développement. La conception descendante et la programmation structurée incorporent le test dans la conception. C'est ce qu'on appelle le test structuré [6]. Chaque module d'un programme structuré devra être testé séparément. **Le test, tout comme le codage, doit être modulaire, structuré, et descendant.**

TEST DES CAS SPÉCIAUX

Il reste cependant à sélectionner les données servant au test d'un module. Le concepteur devra au préalable établir la liste de tous les cas spéciaux reconnus par un programme. Ils pourront comprendre :

- Des cas triviaux.
- Des égalités.
- Des situations spéciales.

CLASSES DE DONNÉES

Les données de test doivent tous les inclure. **On doit ensuite identifier chaque catégorie, ou classe de données que les instructions du programme distingueront.** Elles peuvent comprendre :

SÉLECTION DES DONNÉES DANS LES CLASSES

- Des nombres positifs ou négatifs.
- Des nombres supérieurs ou inférieurs à un seuil particulier.
- Des données comprenant ou non une séquence ou un caractère particulier.
- Des données présentes ou non à un instant donné.

Si les modules sont courts, le nombre total de classes pourra rester faible même si les options sont multiplicatives, (c'est-à-dire deux options par deux mènent à quatre classes de données).

Il faut ensuite distinguer deux cas selon que le programme produit un résultat différent pour chaque donnée de la classe (tout comme dans un tableau), ou un résultat identique pour chaque entrée(telle une alarme lorsqu'un paramètre est au-dessus d'un seuil). Dans les cas discrets, on peut inclure chaque élément si leur nombre total est faible, ou les échantillonner s'il est grand. Les échantillons doivent comprendre tous les cas limites et au moins un cas pris au hasard. Les tables de nombres aléatoires sont proposées dans certains livres, alors que les générateurs de nombres aléatoires font partie des possibilités de la plupart des ordinateurs.

Soyez attentifs à des distinctions qui peuvent ne pas être évidentes. Par exemple, un microprocesseur 8 bits considérera un nombre non signé sur un octet plus grand que 127 comme négatif ; vous devrez en tenir compte lorsque vous utiliserez les instructions de saut dépendant de l'indicateur de signe. Vous devrez aussi surveiller les instructions qui n'affectent pas les indicateurs, le dépassement en arithmétique signée, et les différences entre la longueur des adresses (16 bits) et des données (8 bits).

EXEMPLE DE TEST 1 : programme de tri

TEST D'UN PROGRAMME DE TRI

Les cas spéciaux sont ici évidents :

- Il n'y a pas d'éléments dans la liste.
- Il n'y a qu'un élément, de valeur quelconque.

Les autres cas spéciaux sont ceux dans lesquels des éléments sont égaux.

Des problèmes peuvent surgir, ici, avec les signes ou la longueur des données. Remarquez que la liste elle-même devra contenir moins de 256 éléments. L'emploi de l'instruction LD C,1 ou SET 1,C plutôt que DEC C pour mettre à zéro l'indicateur d'échange signifie qu'on ne rencontrera pas de difficulté si le nombre d'éléments ou d'échanges excède 128. On peut tester les effets du signe en choisissant la moitié des cas-test avec des nombres d'éléments compris entre 128 et 255, et 2 et 127. Toutes les grandeurs doivent être choisies aléatoirement de telle sorte à éviter une influence quelconque sur ces choix.

EXEMPLE DE TEST 2 : nombres d'auto-test

TEST D'UN PROGRAMME ARITHMÉTIQUE

On va supposer qu'un test de validité antérieur nous a garanti que le nombre est à la bonne longueur et qu'il se compose de chiffres valides. Puisque le programme n'établit pas d'autres distinctions, les données de test seront sélectionnées aléatoirement. On se servira d'un générateur ou d'une table de données aléatoires ; la gamme des nombres aléatoires va de 0 à 9.

PRÉCAUTIONS LORS DES TESTS

RÈGLES DE TEST

Le concepteur peut simplifier l'étape du test en suivant les règles suivantes lors de la conception de ses programmes :

1) Essayer d'éliminer chaque cas trivial aussitôt que possible sans introduire des distinctions inutiles.
2) Minimiser le nombre de cas spéciaux. Chaque cas spécial se traduit par du temps supplémentaire de test et de mise au point.
3) Considérer la validité de l'exécution ou les tests d'erreurs sur les données avant tout traitement.
4) Etre attentif aux distinctions par inadvertance, ou inutiles, en particulier lors du traitement de nombres signés ou avec des opérations se référant à des nombres signés.
5) Tester les cas limites manuellement. Ils sont souvent cause d'erreur. S'assurer que la définition du problème spécifie ce qui doit arriver dans ces cas-là.
6) Rendre le programme aussi général qu'il est raisonnablement possible de le faire. Chaque cas ou programme différent accroît le nombre de tests.
7) Diviser le programme et concevoir les modules de façon à ce que le test puisse procéder par étapes, en liaison avec les autres étapes du développement du logiciel [7].

CONCLUSIONS

La mise au point et le test sont la base du processus de développement du logiciel. La plupart des projets leur accordent bien trop peu de temps, et la plupart des ouvrages les négligent. Mais les concepteurs et les responsables s'aperçoivent souvent que ces étapes sont les plus onéreuses et les plus longues. Les progrès sont difficiles à mesurer ou à produire. Le « debugging » et le test des logiciels de microprocesseurs sont particulièrement difficiles en raison de la puissance limitée des outils disponibles.
Le concepteur devra soigneusement planifier le « debugging » et le test. Nous recommandons la procédure suivante :

1) Essayer de rédiger des programmes pouvant aisément être mis au point et testés. La programmation modulaire, structurée, et la conception descendante sont des techniques utiles.
2) Préparer un plan de « debugging » et de test en tant que partie de la conception. Décider très tôt des données à générer et de l'équipement nécessaire.
3) Mettre au point et tester chaque module au cours du processus de conception descendante.
4) Mettre au point la logique de chaque module systématiquement. Utilisez des listes de test, des points d'arrêt et le mode pas à pas. Si la logique du programme est complexe, utiliser un simulateur.
5) Tester les cadencements de chaque module systématiquement s'il y a une difficulté. Un oscilloscope pourra résoudre de nombreux problèmes si le test est bien plannifié. Si les cadencements sont complexes, employer un analyseur logique.
6) S'assurer que les données de test sont représentatives. Veillez à toutes les classes de données que le programme pourra distinguer. Inclure des cas spéciaux ou triviaux.
7) Si le programme traite chaque élément différemment, ou si le nombre de cas est élevé, choisir les données de test aléatoirement [8].
8) Enregistrer tous les résultats des tests en tant que partie intégrante de la documentation. Si une difficulté surgit, il est inutile de répéter des tests sur des cas déjà vérifiés.

RÉFÉRENCES

1. Pour plus d'information sur les analyseurs logiques, consulter :
   R.L. Down, « Understanding Logic Analyzers », *Computer Design,* juin 1977, pp. 188-191.
   W.A. Farnbach, « Bring up Your μP », *Electronic Design,* 10 juillet 1976, pp. 80-85.
   B. Farly, « Logic Analyzers Aren't All Alike », *Electronic Design,* 1er février 1978, pp. 70-75.
   K. Pines, « What Do Logic Analyzers Do ? », Digital Design, septembre 1977, pp. 55-77.
   N.A. Robin, « The Logic Analyzer : A Computer Troubleshooting Tool », *Computer Design,* mars 1976, pp. 89-96.
   S. Runyon, « Focus on Logic and μP Analyzers », *Electronic Design,* 1er février 1977, pp. 40-50.
   A. Santoni, « The Latest Logic Analyzers Offer More Functions and Less Cost », *Electronic Design,* 1er février 1978, pp. 26-32.
2. W.J. Weller, *Assembly Level Programming for Small Computers,* Lexington Books, Lexington, MA, 1975.
3. Quelques conseils sur la résolution de problèmes d'interruptions sont donnés par R.L. Baldridge, « Interrupts Add Power, Complexity to Microcomputer System Design », *EDN,* 5 août 1977, pp. 67-73.
4. Voir C. Bass, « PLZ : A Family of System Programming Languages for Microprocessors », *Computer,* mars 1978, pp. 34-39.
5. Voir par exemple, H.R. Burris, « Time-Scaled Emulations of the 8080 Microprocessor », *Proceedings of the 1977 National Computer Conference,* pp. 937-946.

6. Voir D.A. Walsh, « Structured Testing », *Datamation,* juillet 1977, pp. 111-118.
7. Le test (et la mise au point) sont aussi traités par R.A. DeMillo et al., « Hints on Test Data Selection : Help for the Practicing Programmer », *Computer,* avril 1978, pp. 34-41 et par W.F. Dalton, « Design Microcomputer Software », *Electronics,* 19 janvier 1978, pp. 97-101.
8. Les nombres aléatoires et leur génération sont traités par T.G. Lewins, *Distribution Sampling for Computer Simulation,* Lexington Books, Lexington, Mass, 1975, et par R.A. Mueller et al., « A Random Number Generator for Microprocessors », *Simulation,* avril 1977, pp. 123-127.

CHAPITRE 15

DOCUMENTATION ET OPTIMISATION

Le travail de programmation réalisé jusqu'à présent ne termine pas le développement du logiciel. Une documentation adéquate fait aussi partie intégrante d'un produit logiciel. Elle n'aide pas seulement le concepteur lors du test et de la mise au point mais est aussi essentielle lorsque, ultérieurement, on utilisera et développera le programme. Un programme mal documenté sera difficile à mettre au point, à ré-utiliser ou à étendre.
Parfois, la première version d'un programme consomme trop de mémoire ou s'exécute trop lentement. Le concepteur doit alors imaginer les façons de l'améliorer. Cette phase est appelée optimisation et demande une concentration sur les sections du programme pouvant supporter le plus d'améliorations.

PROGRAMMES AUTO-DOCUMENTÉS

RÈGLES POUR L'AUTO-DOCUMENTATION DES PROGRAMMES

Bien qu'aucun programme ne soit jamais totalement auto-documenté, les quelques règles que nous avons mentionnées antérieurement pourront y contribuer. Elles comprennent l'utilisation de :

- Structures simples, claires, avec aussi peu de transferts de contrôle (de branchements) que possible.
- Noms et labels significatifs.
- Noms pour les dispositifs d'E/S, les paramètres, les facteurs numériques, etc.

Privilégier la simplicité plutôt que des économies mineures de mémoire, de temps d'exécution ou de frappe.
Par exemple, le programme suivant envoie une chaîne de caractères à une imprimante :

```
        LD      A,(2000H)
        LD      B,A
        LD      HL,1000H
W:      LD      A,(HL)
        OUT     (6),A
        CALL    XXX
        INC     HL
        DJNZ    W
        HALT
```

Même sans commentaire on peut l'améliorer de la façon suivante :

```
MESSG   EQU     1000H
COMPTE  EQU     2000H
TTYSIO  EQU     6
        LD      A,(COMPTE)
        LD      B,A
        LD      HL,MESSG
SORTIE: LD      A,(HL)
        OUT     (TTYSIO),A
        CALL    BITDLY
        INC     HL
        DJNZ    SORTIE
        HALT
```

Cette rédaction est bien plus facile à comprendre que la précédente. Même sans autre documentation, on peut sans aucun doute deviner la fonction de ce programme et la signification de la plupart des variables. **Les autres techniques de documentation ne pourront se substituer à l'auto-documentation.**

LE CHOIX DES NOMS UTILES

Voici encore quelques notes sur le choix des noms :

1) Utiliser des noms évidents lorsqu'ils sont disponibles, par exemple COMPTE, ou DELAI, pour les anglicistes OUT pour sortie et IN pour entrée, TTY pour télé-imprimeur ou CRT pour écran cathodique, cela pour les données ou les adresses selon les cas.
2) Eviter les séries d'initiales telles que TL16B, pour « tri d'une liste sur 16 bits » qui ne signifieront pas grand chose pour d'autres utilisateurs.
3) Utiliser des mots complets si possible : FIN, RETOUR, ENVOI, etc.
4) Prendre des noms aussi distincts que possible.

COMMENTAIRES

LIGNES DE CONDUITE

La forme additionnelle la plus immédiate pour la documentation est constituée par les commentaires. Pourtant, bien peu de programmes (y compris ceux que propose la littérature) offrent réellement des commentaires efficaces. On doit respecter les indications suivantes pour les commentaires.

1) Ne pas répéter la signification du code d'instruction. Expliquer plutôt l'objet de l'instruction dans le programme. Ainsi, commenter :

```
DEC B                   ; B = B - 1
```

n'ajoute rien à la documentation. En revanche :

```
DEC B                   ; NOMBRE LIGNES = NOMBRE LIGNES - 1
```

précise l'objet de cette instruction. Rappelez-vous que vous comprenez les codes opératoires et que n'importe qui peut les retrouver dans le manuel d'utilisation. **L'important réside dans la tâche que le programme est censé accomplir.**

2) Rendre les commentaires aussi clairs que possible. Ne pas utiliser des abréviations ou des sigles, à moins qu'ils ne soient très courants (tels qu'ASCII, ou PIO, ou UART) ou standards (ms pour milliseconde).
Evitez ceci :

```
DEC B          ; NL = NL - 1
```

ou encore :

```
DEC B          ; DEC NL DE 1
```

car ce sera parfaitement incompréhensible ou inutile.
3) Commenter chaque point important ou obscur. Soyez particulièrement attentif à marquer les opérations dont le rôle n'est pas évident ; par exemple :

```
AND11011111B       ; SELECTION BIT DU LECTEUR BANDE
```

ou encore ;

```
ADD HL,DE          ; INDEX DE LA TABLE DU CODE GRAY
```

Les opérations d'entrées-sorties requièrent souvent de longs commentaires. Si vous n'êtes pas absolument sûr de ce que l'instruction fait, ou si vous devez vous y arrêter pour réfléchir, ajoutez un commentaire qui le clarifiera. Le commentaire vous fera économiser du temps ultérieurement et se révèlera d'une grande aide.
4) Ne pas commenter ce qui est évident. Si chaque ligne est commentée, vous trouverez difficilement ce qui est important. Des séquences standards telles que :

```
INC     HL
DJNZ    SEARCH
```

ne nécessitent aucune remarque particulière, à moins que vous ne leur ayez confié quelque tâche spéciale. Un unique commentaire pourra suffire pour plusieurs lignes :

```
RRCA                ; DEPLACER LES DIGITS
RRCA
RRCA
RRCA
LD      A,C         ; ECHANGE DES OCTETS
LD      C,B         ; DES PLUS FORT ET FAIBLE POIDS
LD      B,A
```

5) Placer les commentaires sur les lignes auxquelles ils se réfèrent, ou au début d'une séquence.
6) Mettre vos commentaires à jour. Si vous modifiez le programme, modifiez aussi les commentaires.
7) Adopter des formes standard de commentaires. Ne vous souciez pas des redites. Si vous utilisez plusieurs noms pour une même chose, vous créerez de la confusion : si vous employez indifféremment AFFICH ou LED, ou encore DEBUT et START, ou COMPTE et CMPTE, etc., il ne sera pas toujours évident qu'il s'agisse d'une seule et même chose.
Vous n'avez aucun bénéficie réel à attendre de telles inconstances. Ces noms peuvent vous paraître évident aujourd'hui, mais ne le seront plus ultérieurement. D'autres s'y perdront dès le début.

8) Faire des commentaires brefs. Réservez une explication complète à la documentation. Autrement, votre programme sera noyé sous les commentaires et vous-même aurez du mal à le retrouver.
9) Améliorer sans cesse les commentaires. Si vous en retrouvez un que vous ne compreniez plus, prenez le temps de le changer. Si votre listage est trop dense, ajoutez des lignes blanches. Les commentaires ne s'améliorent jamais d'eux-mêmes mais au contraire, ils se dégradent en compréhension au fur et à mesure que vous vous éloignez d'eux dans le programme, oubliant exactement ce que vous vouliez leur faire dire.
10) Avant chaque section, sous-section ou sous-programme important, introduire des commentaires décrivant les fonctions du codage qui suit. Décrivez soigneusement toutes les entrées et sorties, les effets de bords et l'algorithme employé.
11) Il est vivement conseillé lorsque vous modifiez des programmes qui fonctionnent déjà d'introduire des commentaires précisant la date, l'auteur et le type de modification.
Rappelez-vous que de bons commentaires vous épargneront du temps et des efforts. Investissez donc en commentaires et rendez-les aussi efficaces que possible.

EXEMPLES DE COMMENTAIRES 1 : additions de nombres de plusieurs octets

EXEMPLES DE COMMENTAIRES Le programme de base est :

```
        LD      A,(30H)
        LD      B,A
        LD      HL,41H
        LD      DE,51H
        AND     A
ADDM :  LD      A,(DE)
        ADC     A,(HL)
        LD      (HL),A
        INC     DE
        INC     HL
        DJNZ    ADDM
        HALT
```

Tout d'abord, commentez les points importants. Typiquement, il s'agira des initialisations, des recherches de données, et des opérations de traitement. Ne vous souciez pas des séquences standard, telles que la mise à jour des pointeurs et compteurs. Souvenez-vous que les noms sont plus évidents que les nombres, aussi utilisez-les largement.
La nouvelle version du programme est :

```
; ADDITION EN MULTIPRECISION
;
; CE PROGRAMME EFFECTUE UNE ADDITION SUR PLUSIEURS OCTETS
;
; MEMOIRE 30H = LONGUEUR DES NOMBRES (EN OCTETS)
;             MEMOIRES 41H – 50H = 1er OPERANDE ORDRE LSB→MSB
;             MEMOIRES 51H – 60H = 2e OPERANDE
; SORTIES : MEMOIRES 41H – 51H = SOMME
;
```

```
LONG    EQU   30H
NOMB1   EQU   41H
NOMB2   EQU   51H
        LDA   LENGTH     ; COMPTEUR = LONGUEUR DES NOMBRES (EN OCTETS)
        LD    B,A
        LD    HL,NUMB1   ; DEPART LSB DU 1er NOMBRE
        LD    DE,NUMB2   ; DEPART LSB DU 2e NOMBRE
        AND   A
ADDWD:  LD    A,(DE)     ; ACQUISITION OCTET DU 2e NOMBRE
        ADC   A,(HL)     ; AJOUTER OCTET DU 1er NOMBRE
        LD    (HL),A     ; RANGER RESULTAT DANS 1er NOMBRE
        INC   DE
        INC   HL
        DJNZ  ADDWD
        HALT
```

QUESTIONS SUR LES COMMENTAIRES

En second lieu, distinguez les instructions dont les fonctions ne sont pas évidentes et annotez-les. Ici, l'objet de AND A est de mettre le carry à zéro au premier passage.

En troisième lieu, demandez-vous si les commentaires vous disent bien ce que vous auriez besoin de savoir si vous aviez à utiliser le programme. Par exemple :

1) Où entre-t-on dans le programme ? Y a-t-il d'autres points d'entrée ?
2) Quels sont les paramètres requis ? Quand doivent-ils être fournis et sous quelle forme ?
3) Quelles sont les opérations exécutées par le programme ?
4) D'où obtient-il ses données ?
5) Où range-t-il les résultats ?
6) Quels sont les cas spéciaux à prendre en considération ?
7) Que fait le programme en cas d'erreur ?
8) Comment se conclut-il ?

Certaines de ces questions peuvent ne pas s'appliquer à un programme donné, et diverses réponses peuvent être évidentes. Assurez-vous cependant que vous n'aurez pas à vous prendre la tête entre les mains et à disséquer le programme pour trouver ces réponses. Rappelez-vous que trop d'explications est un poids mort dont il faudra se débarrasser en cours de route. Y a-t-il quelque chose que vous souhaiteriez ajouter ou soustraire du listage ? Si oui, faites-le — c'est à vous de sentir que les commentaires conviennent et sont rationnels.

```
; ADDITION EN MULTIPRECISION
;
; CE PROGRAMME EFFECTUE UNE ADDITION SUR PLUSIEURS OCTETS
;
; ENTREES : MEMOIRE 30H = LONGUEUR DES NOMBRES (EN OCTETS)
;           MEMOIRES 41H – 50H = 1re OPERANDE ORDRE LSB→MSB
;           MEMOIRES 51H – 60H = 2e OPERANDE
; SORTIES : MEMOIRES 41H – 51H = SOMME
```

NDT : LSB est utilisé pour « least significant bits » = bits de poids faible
MSB est utilisé pour « most significant bits » = bits de poids fort

```
;
LONG    EQU     30H         ; LONGUEUR DES NOMBRES
NOMB1   EQU     41H         ; LSB DU 1er NOMBRE ET RESULTAT
NOMB2   EQU     51H         ; LSB DU 2e NOMBRE
        LDA     LONG        ; COMPTEUR = LONGUEUR DES NOMBRES (EN OCTETS)
        LD      B,A
        LD      HL,NOMB1    ; DEPART LSB DU 1er NOMBRE
        LD      DE,NOMB2    ; DEPART LSB DU 2e NOMBRE
        AND     A           ; RAZ CARRY POUR COMMENCER
ADDM :  LD      A,(DE)      ; ACQUISITION OCTET DU 2e NOMBRE
        ADC     A,(HL)      ; AJOUTER OCTET DU 1er NOMBRE
        LD      (HL),A      ; RANGER RESULTAT DANS 1er NOMBRE
        INC     DE
        INC     HL
        DJNZ    ADDM
        HALT
```

Exemple de commentaires 2 : sortie de télé-imprimante

Le programme de base est le suivant (TTY est le sigle américain qu'on a conservé, dans les versions suivantes, pour « télé-imprimeur ») :

```
        LD      A,(60H)
        ADD     A,A
        LD      B,11
TBIT:   OUT     (PIODRB),A
        RRA
        SCF
        CALL    BITDLY
        DJNZ    TBIT
        HALT
```

L'ajout de noms et de commentaires aux points importants donne :

```
; PROGRAMME D'IMPRESSION SUR UN TELE-IMPRIMEUR (TTY)
;
; CE PROGRAMME IMPRIME LE CONTENU DE L'ADRESSE 60H
;     SUR LE TELE-IMPRIMEUR
;
; ENTREES : POSITION 60H = CODE DU CARACTERE
; SORTIES : NEANT
;
TTYPIO  EQU     PIODRB
NBITS   EQU     11          ; NOMBRE DE BITS PAR CARACTERE
TDATA   EQU     60H         ; ADRESSE DU CARACTERE A TRANSMETTRE

        LD      A,(TDATA)   ; ACQUISITION DE LA DONNEE
        ADD     A,A         ; DECALAGE A GAUCHE ET FORMATION BIT DE START
        LD      B,NBITS     ; COMPTEUR = NOMBRE DE BITS PAR CARACTERE
TBIT:   OUT     (TTYPIO),A  ; ENVOI DU BIT AU TTY
        RRA                 ; MISE A JOUR POUR BIT SUIVANT
        SCF                 ; FORMATION DU BIT DE STOP (1 LOGIQUE)
        CALL    BITDLY      ; TEMPORISATION DE 1 BIT
        DJNZ    TBIT
        HALT
```

On peut très facilement, maintenant, modifier ce programme pour lui faire transférer, par exemple, une chaîne complète de données, démarrant aux adresses contenues en DPTR et DPTR +1 et se terminant avec un caractère « 03 » (ETX en ASCII). Supposons, en outre, que le terminal soit à 30 caractères par seconde, et qu'on ajoute un bit de stop (il faudra changer BITDLY). Essayez d'introduire ces modifications avant d'examiner le listage suivant :

```
; PROGRAMME DE SORTIE D'UNE CHAINE DE CARACTERES
;
; CE PROGRAMME EFFECTUE LA SORTIE D'UNE CHAINE SUR LE TERMINAL.
;     LA TRANSMISSION CESSE QUAND ON RENCONTRE LE CARACTERE ASCII ETX (03H)
;
; ENTREES : MEMOIRES 60H-61H : ADRESSE DE LA CHAINE A SORTIR
; SORTIES : NEANT
;
DPTR    EQU     60H         ; EMPLACEMENT DE L'ADRESSE DE DEPART DU
                            ;     BUFFER DE SORTIE
FINC    EQU     03          ; CARACTERE DE TERMINAISON = CARACTERE ASCII ETX
NBITS   EQU     11          ; NOMBRE DE BITS PAR CARACTERE
TTYPIO  EQU     PIODRB
        LD      HL,(DPTR)   ; ACQUISITION ADRESSE DE DEPART DE LA CHAINE
TCAR :  LD      A,(HL)      ; LECTURE D'UN CARACTERE
        CP      FINC        : CARACTERE DE TERMINAISON ?
        JR      Z,FAIT      ; OUI, TERMINE
        ADD     A,A         ; DECALAGE DONNEE A GAUCHE ET FORMATION BIT DE START
        LD      B,NBITS     ; COMPTEUR = NOMBRE DE BITS PAR CARACTERE
TBIT:   OUT     (TTYPIO),A  ; ENVOI DU BIT VERS TTY
        RRA                 ; MISE A JOUR POUR BIT SUIVANT
        SCF                 ; FORMATION BIT DE STOP (1 LOGIQUE)
        CALL    BITDLY      ; TEMPORISATION PENDANT 1 BIT
        DJNZ    TBIT
        INC     HL
        JR      TCAR
FAIT:   HALT
```

De bons commentaires rendent aisées les modifications répondant à de nouvelles exigences. Pour vous en convaincre, tentez de modifier le programme précédent de façon à :

- Faire démarrer chaque message par le code ASCII STX (02 en hexadécimal), suivi par un code d'identification sur trois chiffres stocké en 0030, 0031 et 0032.
- N'ajoutez ni bit de start, ni bit de stop.
- Attendre 1 ms entre chaque bit.
- Transmettre 40 caractères, démarrant à l'adresse contenue en DPTR et DPTR +1.
- Terminer chaque message avec deux caractères ASCII ETX (03 en hexadécimal) consécutifs.

ORGANIGRAMME DOCUMENTAIRE

SUGGESTIONS POUR UTILISER LES ORGANIGRAMMES

Nous avons déjà décrit l'utilisation des organigrammes en tant qu'outils de conception dans le chapitre 13. Ils pourront aussi servir à la documentation, en particulier si :

- Ils ne sont pas détaillés au point de devenir illisibles.
- Les points de décisions sont clairement marqués et expliqués.
- Ils comprennent tous les branchements.
- Ils correspondent au listage réel du programme.

Les organigrammes vous seront utiles s'ils vous procurent un panorama du programme. Ils ne le seront pas s'ils deviennent aussi difficiles à lire que le listage.

PROGRAMMATION STRUCTURÉE DOCUMENTAIRE

La programmation structurée (traitée au chapitre 13) peut servir pour partie à la documentation si :

- On décrit l'objet de chaque section en commentaires.
- On fait nettement apparaître les instructions de chaque structure conditionnelle ou de boucle en usant de l'indentation et de marqueurs de fin de séquence.
- On rend la structure globale aussi simple que possible.
- On utilise un langage homogène et bien défini.

Le programme structuré vous aidera à tester la logique et à l'améliorer. De plus, et parce que le programme structuré est indépendant de la machine, vous pourrez le transposer aisément sur un autre calculateur.

CARTES DE LA MÉMOIRE (« MEMORY MAP »)

Une carte de la mémoire consiste simplement en une liste de toutes les affectations mémoires d'un programme. Elle vous servira à déterminer le volume mémoire nécessaire, la position des données, des sous-programmes, et les zones non attribuées. Une telle carte est une référence commode lorsqu'on recherche des positions de rangement et d'entrée ou lorsqu'on veut diviser la mémoire pour l'affecter à divers programmes ou divers programmeurs. Elle vous facilitera l'accès aux données et sous-programmes si vous en avez besoin ultérieurement à des fins d'extension ou de maintenance. Parfois, une carte graphique sera plus utile qu'un listage.

CARTE MÉMOIRE TYPE **Voici un exemple de « memory map » classique :**

Mémoire de programme :		
Adresse	Programme	Fonction
0000-0002	RESET	TRANSFERE CONTROLE AU PROGRAMME PRINCIPAL A L'ADRESSE 40 HEXA
0038-003A	INTRPT	TRANSFERE LE CONTROLE AU SERVICE INTERRUPTION A L'ADRESSE 300 HEXA
0040-0265	PRINCP	PROGRAMME PRINCIPAL
0270-027F	DELAI	TEMPORISATION
0280-0290	AFFICH	PROGRAMME D'AFFICHAGE
0300-0340	CLAVIN	PROGRAMME D'INTERRUPTION DU CLAVIER

Mémoire de données :		
Adresse	Programme	Fonction
1000	NCLES	NOMBRE DE CLES
1001-1002	CLAVR	POINTEUR DU BUFFER CLAVIER
1003-1041	BFCLAV	BUFFER CLAVIER
1042-1051	BFAFFI	BUFFER AFFICHAGE
1052-105F	TEMP	RANGEMENT TEMPORAIRE
10EO-10FF	STACK	PILE DE SAUVEGARDE

La carte mémoire peut aussi donner la liste des points d'entrée supplémentaires ainsi que la description spécifique des zones inutilisées de la mémoire.

LISTES DE PARAMÈTRES ET DE DÉFINITIONS

Au début du programme et de tous les sous-programmes, une liste des paramètres et des définitions employées facilitera leur compréhension et leurs modifications. Voici quelques conseils d'usage :

RÈGLES POUR LISTES DE DÉFINITIONS

1) Séparer les positions en RAM, les unités d'E/S, les paramètres, les définitions, et les constantes du système.

2) Organiser les listes alphabétiquement si possible, avec une description de chaque entrée.
3) Donner un nom à chaque paramètre pouvant être modifié et l'inclure dans les listes. Ces paramètres peuvent être des constantes de temps, des entrées ou des codes correspondants à des touches ou des fonctions particulières, des masques ou des structures de commandes, des caractères de début ou fin de séquence, etc.
4) Etablir séparément la liste des constantes du système. On trouvera là les « Reset », les adresses du service d'interruptions, l'adresse de départ du programme, les zones en RAM, l'emplacement des piles, etc.
5) Attribuer un nom à chaque port desservant un dispositif d'E/S, même si plusieurs périphériques se partagent des ports dans le système. Vous faciliterez les extensions et les reconfigurations.

LISTE TYPIQUE DE DÉFINITIONS

Voici une liste typique de définitions :

```
;
; CONSTANTES DU SYSTEME
;
RESET    EQU    0        ; ADRESSE DE RESET
INTRP    EQU    38H      : ENTREE D'INTERRUPTION
DEBUT    EQU    40H      ; DEBUT PROGRAMME PRINCIPAL
CLAVIN   EQU    300H     ; PROGRAMME D'INTERRUPTION DU CLAVIER
RAMST    EQU    1000H    ; DEBUT DE STOCKAGE DES DONNEES
PILEPR   EQU    1100H    ; DEBUT DE PILE
```

```
;
; UNITES D'E/S
;
AFFICH    EQU     0E0H        ; PORT DE SORTIE POUR AFFICHAGE
CLAVE     EQU     0E1H        ; PORT D'ENTREE POUR CLAVIER
CLAVS     EQU     0E0H        ; PORT DE SORTIE POUR CLAVIER
PORTTY    EQU     0F0H        ; PORT DE DONNEES TTY
;
; EMPLACEMENTS MEMOIRE
;
          ORG     RAMST
NCLES     DEFS    1           ; NOMBRE DE TOUCHES
CLAVPR    DEFS    2           ; POINTEUR DU BUFFER CLAVIER
BFCLAV    DEFS    40H         ; BUFFER D'ENTREE DU CLAVIER
BFAFFI    DEFS    10H         ; BUFFER D'AFFICHAGE DE DONNEES
TEMP      DEFS    14H         ; RANGEMENT TEMPORAIRE
;
; PARAMETRES
;
BONDS     EQU     2           ; TEMPS D'ELIMINATION DES REBONDS EN MILLISECONDES
CLEGO     EQU     10          ; IDENTIFICATION DE LA TOUCHE GO
CPTMS     EQU     133         ; COMPTEUR POUR TEMPORISATION 1MS
OPEN      EQU     0FH         ; SEQUENCE POUR TOUCHES OUVERTES
TPULS     EQU     1           ; LONGUEUR IMPULSION (EN MS) POUR AFFICHAGE
;
;DEFINITIONS
;
TOUS1     EQU     0FFH        ; SEQUENCE TOUS LES BITS A 1
DEBCNV    EQU     80H         ; IMPULSION DEBUT DE CONVERSION
```

Naturellement, les entrées dans la RAM ne seront généralement pas dans l'ordre alphabétique car le concepteur doit les ordonner de façon à minimiser le nombre de modifications d'adresses requis par le programme.

BIBLIOTHÈQUES DE PROGRAMMES

Une documentation de sous-programmes vous servira à monter une bibliothèque des programmes utiles. L'idée consiste à rendre de tels programmes accessibles à tous, autant que faire se peut. Une fiche normalisée vous permettra, à vous ou à quiconque, d'apprécier d'un coup d'œil ce que chaque programme accomplit. La meilleure des procédures consiste à adopter une présentation standardisée et à l'utiliser scrupuleusement. Rangez ces programmes de façon organisée (par exemple, en fonction du processeur, du langage, ou du type de programme) et vous disposerez vite d'un jeu de programmes utiles. **Mais rappelez-vous que sans ordre ni méthode et sans une bonne documentation, le recours à une bibliothèque de programmes peut être plus difficile que la ré-étude complète du programme voulu.** La mise au point d'un système requiert une compréhension parfaite de l'effet de chacun de ses sous-programmes.

FICHES DE PROGRAMMES EN BIBLIOTHÈQUE

Les informations à porter sur une fiche normalisée sont :

- Le rôle du programme.
- Le processeur utilisé.
- Le langage.
- Les paramètres requis et les méthodes de transmission au sous-programme.
- Les résultats obtenus et la façon de les transmettre au programme principal.
- Le nombre d'octets mémoire utilisés.
- Le nombre de cycles d'horloge demandés. Il peut constituer une valeur moyenne ou typique ou pourra varier largement. Le temps d'exécution réel dépendra, en fin de compte, de l'horloge du processeur employé.
- Les registres affectés.
- Les indicateurs affectés.
- Un exemple type.
- Le traitement des erreurs.
- Les cas spéciaux.
- Le listage documenté du programme.

Si le programme était complexe, la fiche standard de la bibliothèque peut aussi inclure un organigramme général ou un programme structuré. Ainsi que nous l'avons mentionné auparavant, un programme en bibliothèque est vraisemblablement plus utile s'il n'exécute qu'une fonction simple et distincte, de façon générale et rationnelle.

EXEMPLES DE FICHES DE PROGRAMMES

Exemple de fiches 1 : somme de données

But : Le programme SOM8 calcule la somme d'un ensemble de nombres binaires non signés sur un octet.
Langage : assembleur Z80.
Conditions initiales : l'adresse de départ de l'ensemble des nombres se trouve dans le double registre HL et la longueur de l'ensemble se trouve dans l'accumulateur.
Conditions finales : somme dans l'accumulateur A.

Besoins :

Mémoire	7 octets
Temps	13 + 26N cycles d'horloge, avec N = longueur de l'ensemble des nombres.
Registres	A, B, H, L

Tous les indicateurs sont modifiés.

Exemple : (données en hexadécimal)

```
Départ :
     HL = 0050
      A = 03
(0050) = 27
(0051) = 3E
(0052) = 26
Fin :
      A = 8B
```

Traitement des erreurs : le programme ignore les retenues ; le bit de carry traduit le résultat de la dernière opération. Le contenu initial de A doit être non nul.

Listage :

```
;
; SOMME DE DONNEES 8 BITS
;
SUM8:   LD      B,A         ; COMPTEUR = LONGUEUR DU BLOC
        SUB     A           ; SOMME = 0
ADD8:   ADD     A,(HL)      ; SOMME = SOMME + DONNEE
        INC     HL
        DJNZ    ADD8
        RET
```

Exemple de fiches 2 : conversion décimal à 7-segments

But : le programme SEPT convertit un nombre décimal en un code d'affichage sur sept segments.
Langage : assembleur Z80.
Conditions initiales : donnée dans l'accumulateur.
Conditions finales : code sept segments dans l'accumulateur.

Besoins :

Mémoire	26 octets y compris la table sept segments SSEG 10 entrées
Temps	74 cycles d'horloge si les données sont valides ; 40 sinon.
Registres	A, B, D, E, H, L

Tous les indicateurs sont affectés.
La donnée en entrée dans l'accumulateur est « écrasée ».

Exemple : (données en hexadécimal).

```
Départ :
        (A) = 05
Fin :
        (A) = 66
```

Traitement des erreurs : le programme met 00 dans l'accumulateur si la donnée n'est pas un chiffre décimal.

Listage :

```
;
; CONVERSION DECIMAL-7 SEGMENTS
;
SEPT    LD      B,0         ; ACQUISITION CODE ERREUR
        CP      10          ; DONNEE = CHIFFRE DECIMAL ?
        JR      NC,FAIT     ; NON, GARDER CODE ERREUR
        LD      L,A         ; OUI, DONNEE DANS UN INDEX 16 BITS
        LD      H,0
        LD      DE,SSEG     ; ACQUISITION ADRESSE DE BASE TABLE 7 SEGMENTS
        ADD     HL,DE       ; TROUVER ELEMENT PAR INDEXATION
        LD      B,(HL)      ; ACQUISITION CODE 7 SEGMENTS EN TABLE
FAIT :  LD      A,B         ; SAUVEGARDE CODE 7 SEGMENTS OU ERREUR
        RET
```

```
SSEG:   DEFB    3FH
        DEFB    06H
        DEFB    5BH
        DEFB    4FH
        DEFB    66H
        DEFB    6DH
        DEFB    7DH
        DEFB    07H
        DEFB    7FH
        DEFB    6FH
```

EXEMPLE DE FICHES 3 : Somme décimale

But : le programme SOMDEC ajoute deux nombres décimaux en multiprécision.

Langage : assembleur Z80.

Conditions initiales : • adresse de l'octet de poids faible d'un nombre dans le double registre HL.
• adresse de l'octet de poids faible de l'autre nombre dans le double registre DE.
• longueur des nombres (en octets) dans A.
• les nombres sont rangés à l'adresse la plus basse en commençant par l'octet de poids faible.

Conditions finales : la somme remplace le nombre dont l'adresse de départ se trouve dans la paire de registre HL.

Besoins :

Mémoire	11 octets
Temps	13 + 50N cycles d'horloge, où N est le nombre d'octets
Registres	A, B, D, E, H, L

Tous les indicateurs sont modifiés. L'indicateur de Carry indique si la somme a produit une retenue.

Exemple : (données en hexadécimal).

```
Départ :
      HL = 0060
      DE = 0050
       A = 2
  (0060) = 34
  (0061) = 55
  (0050) = 88
  (0051) = 15
Fin :
  (0060) = 22
  (0061) = 71
   CARRY = 0
```

Traitement des erreurs : le programme ne teste pas la validité des entrées décimales. L'accumulateur doit contenir un nombre supérieur ou égal à 1.

Listage :

```
SOMDEC :  LD      B,A       ; COMPTEUR = LONGUEUR DES NOMBRES (EN OCTETS)
          AND     A         ; RAZ RETENUE AU DEPART
DECADD:   LD      A,(DE)    ; ACQUISITION 2 CHIFFRES DECIMAUX DE CHAINE 2
          ADC     A,(HL)    ; ADDITION PAIRE DE CHIFFRES DE CHAINE 1
          DAA               ; ADDITION DECIMALE
          LD      (HL),A    ; RANGEMENT DU RESULTAT DANS CHAINE 1
          INC     DE
          INC     HL
          DJNZ    DECADD
          RET
```

SYNTHÈSE DE LA DOCUMENTATION

DOSSION DE DOCUMENTATION

La documentation complète d'un logiciel de microprocesseur comprend tous ou presque tous les éléments que l'on a traités.

Ainsi, un dossier de documentation pourra inclure :

- **Des organigrammes généraux.**
- **Une description écrite du programme.**
- **Une liste des paramètres et des définitions.**
- **Une carte de la mémoire.**
- **Un listage documenté du programme.**
- **Une description du plan de test et des résultats de tests.**

Cette documentation peut également inclure :

- **Les organigrammes des programmeurs.**
- **Les organigrammes des données.**
- **Des programmes structurés.**

Les procédures de documentation décrites ci-dessus constituent l'ensemble minimal des documents accompagnant un programme. Un logiciel de production demandera davantage d'efforts.

Il faut en effet y adjoindre les documents suivants :

- Le manuel technique de programmation
- Le guide de l'utilisateur
- Le manuel de maintenance.

Le manuel logique de programmation développe les explications écrites du logiciel. Il doit être rédigé pour un technicien compétent, qui peut d'ailleurs ne pas posséder la connaissance détaillée des commentaires du logiciel. Le manuel technique devra expliquer les objectifs du système, les algorithmes employés et les compromis intervenus.

Il devra ensuite expliquer avec force détails les structures des données employées et comment elles sont manipulées. Il devra fournir un guide pas à pas des opérations du programme. Finalement, il devra contenir toutes les tables spéciales ou graphiques contribuant à expliquer le programme. Les tables de conversion de codes, les diagrammes d'états, les matrices de translation et les organigrammes devront y être inclus.

Le guide de l'utilisateur est probablement l'élément de la documentation à la fois le plus important et le plus négligé. Quelle que soit la conception du système, celui-ci sera inexploitable si personne ne sait profiter de ses caractéristiques.
Le guide de l'utilisateur doit présenter le système à tous les utilisateurs, les plus avancés comme les autres. Il doit fournir des explications détaillées sur ses caractéristiques et son usage. Des exemples fréquents doivent aider à clarifier certains passages du texte. Des directives données pas à pas (et testées !) doivent être fournies. Les programmeurs possédant une connaissance détaillée d'un système prennent souvent des raccourcis que le lecteur moyen ne peut apprécier ou comprendre. Une discussion plus approfondie sur la rédaction du manuel de l'utilisateur dépasse le cadre de ce livre. Cependant, rappelez-vous que vous ne déploierez jamais trop d'efforts en préparant un guide de l'utilisateur, car il constituera le plus souvent la référence essentielle.
Le manuel de maintenance est destiné au programmeur ayant à modifier le système. Il doit définir, pas à pas, les procédures adoptées pour les reconfigurations du système. En outre, il doit décrire toutes les possibilités incluses dans le programme et prévues pour son extension future.
Il ne faudrait pas prendre à la légère ce problème de documentation, ni le remettre à la fin du développement du logiciel. Une bonne documentation, associée à de bonnes méthodes de programmation, ne constitue pas uniquement une composante importante du produit final mais peut aussi simplifier le développement, le rendre plus rapide, plus productif. Le concepteur devra rester homogène et complet dans sa documentation à chaque étape du développement du logiciel.

OPTIMISATION

Il arrive parfois que le concepteur ait à gagner quelques microsecondes dans l'exécution du programme, ou quelques octets qui débordent de la mémoire. Avec la baisse du prix des circuits intégrés, le problème de la mémoire devient moins critique. Le problème du temps, lui, ne sera impératif que dans les applications où les temps sont critiques ; dans bien des applications, le microprocesseur dépense la plus grande partie de son temps à attendre la réponse d'organes externes, ce qui fait que la vitesse d'exécution du programme ne constitue pas un facteur majeur.

LE PRIX DE L'OPTIMISATION

Tenter d'optimiser un programme jusqu'au dernier bit est rarement aussi important que ce que d'aucuns prétendent parfois. En premier lieu, une recherche d'optimisation parfaite peut coûter cher, et ce pour les raisons suivantes :

1) Il faut investir en temps supplémentaire. Or, ce temps représente la plus grosse dépense dans le développement du logiciel.
2) On sacrifie la structure et la simplicité : il en résulte un accroissement des temps de mise au point et de test.
3) Le programme exige un surcroît de documentation.
4) Le programme résultant sera difficile à étendre, maintenir ou utiliser.
En second lieu, un coût plus faible par unité ou de meilleures performances peuvent ne pas être vraiment si importants. Est-ce vraiment ce qui permettra de vendre plus ? Ne ferait-on pas mieux d'introduire des caractéristiques tenant mieux compte des besoins des utilisateurs ?

Les seules applications qui sembleraient justifier l'effort et le temps supplémentaires requis sont celles de très grand volume, les applications à bas prix et faibles performances, dans lesquelles le coût des mémoires supplémentaires contre-balancera le prix de développement du logiciel amélioré.

REORGANISATIONS MAJEURES OU MINEURES

Toutefois, si vous devez reconcevoir un programme, voici quelques conseils utiles. Tout d'abord, déterminez les performances supplémentaires nécessaires, et le gain de mémoire auquel il vous faut aboutir. Si les améliorations portent sur 25 % ou moins, vous pourrez les obtenir en ré-organisant le programme. S'il s'agit de plus de 25 %, vous avez commis une erreur fondamentale de conception ; vous devrez alors procéder à des modifications sensibles du matériel ou du logiciel. Nous allons d'abord traiter la ré-organisation. Consulter également les exemples du chapitre 5 du livre *Z80 Programming for Logic design* (Programmation du Z80 pour une conception logique).

Notez particulièrement que l'économie de mémoire peut être critique si elle permet à un programme d'entrer dans un volume limité de ROM ou de RAM, disponible dans un système micro-ordinateur à un ou deux circuits. Les coûts matériels des petits systèmes peuvent ainsi être substantiellement réduits si leurs besoins peuvent être limités, en mémoire et en E/S.

RE-ORGANISATION POUR CONSOMMER MOINS DE MEMOIRE

ECONOMIE DE MÉMOIRE

Les procédures suivantes réduiront les besoins en mémoire de vos programmes en assembleur pour Z80 :

1) Remplacer les séquences répétitives par des sous-programmes. Assurez-vous cependant que les instructions d'appel et de retour (CALL et RETURN) ne compromettent pas le bénéfice escompté. Notez que cette substitution se traduit généralement par des programmes plus lents, en raison du temps investi dans les appels et les retours.

2) Travailler avec les registres lorsque c'est possible. Mais rappelez-vous que c'est au prix d'une initialisation supplémentaire.

3) Utiliser la pile lorsque c'est possible. Le pointeur est automatiquement mis à jour après chaque usage de la pile, ce qui fait que vous économiserez les instructions explicites de mise à jour.

4) Eliminer les sauts (JUMP). Essayez de ré-organiser votre programme ou d'utiliser des sauts indirects (JP (HL) ou JP (IX ou IY) ; RST, ou les instructions RETURN.

5) Profiter des adresses qui peuvent tenir sur 8 bits. Elles comprennent les adresses en page zéro et des adresses multiples de 100_{16}. Par exemple, on peut placer toutes les tables de ROM dans une section de 100_{16} octets de la mémoire, et toutes les variables dans une autre section de 100_{16} octets.

6) Organiser les données et les tables de façon à pouvoir les adresser sans vous soucier des retenues de calculs d'adresses et sans indexation. Ceci permettra, à nouveau, de manipuler des adresses 16 bits comme des quantités sur 8 bits. (Voir pages 5-1 à 5-6 du livre *Z80 Programming for Logic Design.*)

7) Utiliser les instructions 16 bits au lieu de deux instructions sur 8. Cela pourra être très utile pour l'initialisation ou le rangement des résultats.

8) Utilisez les résultats obtenus précédemment dans votre programme.

9) Exploiter des instructions comme INC (HL), DEC (HL), LD (HL), RL (HL) et RR (HL) qui opèrent directement sur des positions mémoire sans utiliser des registres.
10) Utiliser les instructions INC ou DEC pour positionner les bits des indicateurs.
11) Préférer les sauts relatifs aux sauts avec adressage direct.
12) Exploiter les instructions de transfert de blocs, de recherche de blocs, d'entrée/sortie de blocs dès que l'on traite des blocs de données.
13) Faire attention à certaines formes courtes d'instructions telles que les décalages de l'accumulateur (RLCA, RLA, RRCA et RRA) et DJNZ.
14) Utiliser des algorithmes plutôt que des tables pour calculer des expressions arithmétiques ou logiques ou effectuer des conversions de code. On peut noter que ce choix se traduit par des programmes plus lents.
15) Réduire la taille des tables mathématiques par l'interpolation des entrées. Ici encore, on économise la mémoire aux dépens du temps d'exécution.
16) Exploiter l'intérêt de la banque de registres auxiliaires pour réduire l'utilisation de la mémoire. Ceci peut aussi gagner du temps.

RE-ORGANISATION POUR ÉCONOMISER LE TEMPS D'EXÉCUTION

ÉCONOMIE DE TEMPS D'EXÉCUTION

Bien que certaines des méthodes qui économisent la mémoire économisent aussi le temps d'exécution, vous pourrez généralement gagner un temps considérable en vous concentrant uniquement sur les boucles d'exécution fréquente. Même en éliminant une instruction qui n'est exécutée qu'une fois, vous pourrez gagner quelques microsecondes. Mais une économie dans une boucle exécutée fréquemment multipliera ce gain.
Ainsi, **si l'on désire réduire les temps d'exécution, il faut :**

1) Déterminer la fréquence d'exécution de chaque boucle. Vous pouvez le faire à la main, ou en utilisant le simulateur ou toute autre méthode de test.
2) Examiner les boucles dans l'ordre déterminé par leur fréquence d'exécution, en commençant par la plus fréquente. Poursuivre par les autres, jusqu'à ce que la réduction désirée soit obtenue.
3) Examiner d'abord si quelque opération peut être extraite de la boucle, par exemple, des calculs répétitifs, des données pouvant être rangées dans un registre ou dans la pile, des adresses pouvant être placées dans des paires de registres ou des registres d'index des cas spéciaux ou des cas d'erreurs pouvant être traités ailleurs, etc. Notez que cela vous demandera des initialisations supplémentaires et de la mémoire, mais vous gagnerez sur le temps.
4) Essayer d'éliminer les sauts (JUMP). Ils consomment un temps précieux. Ou bien, utiliser les sauts par adressage direct qui demandent plus de mémoire mais moins de temps que les sauts par adressage relatif.
5) Remplacer les sous-programmes par des codes en ligne. Vous économiserez au moins un CALL et un RETURN.
6) Employer la pile pour les sauvegardes temporaires.
7) Appliquer les conseils donnés pour économiser de la mémoire, lorsqu'ils décroissent aussi les temps d'exécution. Ceci inclut l'usage des instructions de traitement de blocs, des adresses 8 bits, des instructions 16 bits, de RST, et de certaines formes courtes d'instruction, etc.

8) Ne pas prêter attention aux instructions qui ne sont employées qu'une fois. Toute modification risque d'introduire des erreurs sans bénéfice appréciable.
9) Eviter les adressages indexés et relatifs quand c'est possible, car il prennent du temps.
10) Utiliser des tables plutôt que des algorithmes ; confiez-leur le maximum de tâches même si plusieurs entrées doivent être répétées.

RÉ-ORGANISATIONS MAJEURES

Si vous devez accroître la vitesse de 25 % ou plus, ou diminuer d'autant l'occupation mémoire, ne tentez pas de réorganiser le codage du programme. Vous aurez peu de chances d'obtenir un tel résultat, à moins de faire appel à un expert externe. Il vaudra mieux introduire des modifications majeures.

MEILLEURS ALGORITHMES

La modification la plus évidente consiste dans le choix d'un meilleur algorithme. En particulier si vous exécutez des tris, recherches, ou calculs mathématiques, vous trouverez peut-être des méthodes plus rapides ou plus courtes dans la littérature. On trouve des bibliothèques d'algorithmes dans les groupements professionnels et dans certaines revues. Reportez-vous, par exemple, aux références 1 à 10 à la fin de ce chapitre.
Du matériel supplémentaire pourra peut-être remplacer du logiciel. Des compteurs, registres à décalage, unités arithmétiques, circuits multiplicateurs, et autres circuits pourront tout à la fois vous faire économiser du temps et des mémoires. Des calculatrices, UART, claviers, codeurs, et autres dispositifs additionnels pourront, eux, vous faire économiser de la mémoire même s'ils opèrent lentement. Des interfaces compatibles, parallèles ou séries, ou d'autres éléments spécialement conçus pour le Z80 vous feront économiser du temps en déchargeant l'UCT de certaines tâches.

AUTRES MODIFICATIONS MAJEURES

D'autres modifications pourraient vous aider :

1) Une UC traitant des mots plus longs sera plus rapide si la donnée est suffisamment longue. Un tel microprocesseur utilisera aussi moins de mémoire. Un processeur 16 bits, par exemple, emploie plus efficacement la mémoire qu'un processeur 8 bits car la plupart de ses instructions sont codées sur un mot.
2) Des versions plus rapides de votre microprocesseur existant peut-être. Mais rappelez-vous qu'il vous faudra alors des mémoires ainsi que des ports d'E/S plus rapides, et que vous devrez recalculer toutes vos boucles de temporisation.
3) Deux UCT en parallèle ou non pourront peut-être résoudre votre problème si vous pouvez scinder la tâche et établir les liaisons nécessaires entre elles.
4) Un microprocesseur spécialement microprogrammé exécutera probablement le même programme bien plus vite. Le prix en sera, bien sûr, bien plus élevé même si vous employez un émulateur disponible.
5) Etablissez des compromis entre le temps et la mémoire. Des tables de correspondances et des ROM de fonctions seront plus rapides que des algorithmes mais occuperont davantage de mémoire.

POUR DÉCIDER DE CHANGEMENTS IMPORTANTS

Les problèmes qui demandent des modifications profondes, naissent souvent d'une planification déficiente lors des étapes de la définition et de la conception. C'est au cours de la définition du problème que l'on doit déterminer quel processeur et quelles méthodes répondent aux besoins du problème. Une erreur d'appréciation peut coûter cher par la suite. Une solution bon marché peut produire une dépense injustifiée en temps de mise au point. N'essayez pas de vous en accommoder ; la meilleure solution consiste généralement à élaborer la conception adéquate et à noter chaque échec jusqu'à acquisition de l'expérience nécessaire. Si vous avez adopté les méthodes que nous avons proposées, telles que les organigrammes, la programmation modulaire, la programmation structurée ou la conception descendante, et si vous avez bien documenté votre logiciel, vous économiserez bien des efforts même si vous devez procéder à des modifications majeures.

RÉFÉRENCES

1. Collected Algorithms from ACM, Inc., P.O. Box 12105. Church Street Station, New York 10249.
2. T.C. Chen, « Automatic Computation of Exponentials, Logarithms, Ratios, and Square-Roots », *IBM Journal of Research and Development,* Volume 18, pp. 380-388, juillet 1972.
3. H. Schmid, *Decimal Computation,* Wiley-Interscience, New York, 1974.
4. D.E. Knuth, *The Art of Computer Programming, Volume 1 : Fundamental Algorithms,* Addison-Wesley, Reading, Mass. 1967.
5. D.E. Knuth, *The Art of Computer Programming, Volume 2 : Semi-numerical Algorithms,* Addison-Wesley, Reading, Mass., 1969.
6. D.E. Knuth, *The Art of Computer Programming, Volume 3 : Sorting and Searching,* Addison-Wesley, Reading, Mass., 1973.
7. B. Carnahan et al., *Applied Numerical Methods,* Wiley, New York, 1969.
8. A.M. Despain, « Fourier Transform Computers Using CORDIC Iterations », *IEEE Transactions on Computers,* octobre 1974, pp. 993-1001.
9. Y.L. Luke, *Algorithms for the Computation of Mathematical Functions,* Academic Press, New York, 1977.
10. K. Hwang, *Computer Arithmetic,* Wiley, New York, 1978.
11. Dollhoff T., « Microprocessor Software : How to Optimize Timing and Memory Usage. Part Four : Techniques for the Zilog Z80 », *Digital Design,* février 1977, pp. 44-51.

CHAPITRE 16

EXEMPLES DE PROJETS

PROJET N° 1 : un minuteur

MINUTEUR : PROCÉDURE D'ENTRÉE

But : Voici un projet de minuteur. L'opérateur introduit deux chiffres (minutes et dixièmes de minutes) à partir d'un clavier de type calculatrice, puis presse sur la touche GO. Le système décompte le temps sur deux afficheurs à LED, à 7-segments (voir chapitre 11 : description de claviers non codés et d'afficheurs LED).

Matériel — Le projet utilise un port d'entrée et un port de sortie, deux afficheurs 7-segments, un clavier 12 touches, un inverseur 7404, et soit une porte NAND type 7400, soit une porte AND type 7408 selon la polarité des afficheurs. Les afficheurs peuvent nécessiter des amplificateurs (divers), inverseurs et résistances selon leur polarité et leur configuration.
Le matériel est organisé comme le montre la figure 16.1. Les lignes de sortie 0, 1 et 2 servent à scruter le clavier, les lignes d'entrée 0, 2 et 3 servent à déterminer si une touche a été pressée. Les lignes de sortie 0, 1, 2 et 3 sont employées pour envoyer des chiffres DCB aux décodeurs-drivers 7-segments. La ligne de sortie 4 active les afficheurs (si elle est à 1, les afficheurs sont en service). La ligne de sortie 5 sélectionne l'afficheur de droite ou de gauche : la sortie 5 est à 1 pour l'afficheur de gauche, à 0 pour celui de droite ; ainsi, la ligne commune de l'afficheur de gauche devrait être active lorsque la ligne 4 et la ligne 5 sont à 1, alors que la ligne commune de l'afficheur de droite devient active lorsque la ligne 4 est à 1 et la ligne 5 à zéro. La ligne de sortie 6 commande le point décimal de l'afficheur gauche. Il peut être piloté par un inverseur ou simplement laissé allumé.

Connexions du clavier — Le clavier est un simple clavier de calculatrice, très économique. Il comprend 12 touches non codées organisées en quatre rangées de trois colonnes. Parce que le câblage ne coïncide pas avec cettre structure, le programme recourt à une table pour identifier les clés pressées. Les tableaux 16.1 et 16.2 indiquent les connexions d'entrée et de sortie du clavier. La touche du point décimal n'est présente que pour le confort de l'utilisateur ou pour une extension future ; le programme actuel ne s'en servira pas.
Dans une application réelle, il faudra rajouter des résistances de rappel (« pullup ») au clavier pour être sûr que l'on lira effectivement des « 1 » logiques sur les entrées quand on n'appuie pas sur les touches. Il sera aussi nécessaire d'utiliser des résistances ou diodes sur le port de sortie pour éviter d'abîmer les drivers en cas de court-circuit de 2 sorties. Ceci peut arriver si l'on frappe simultanément sur 2 touches de la même rangée, provoquant ainsi la connexion de 2 sorties (colonnes) différentes.

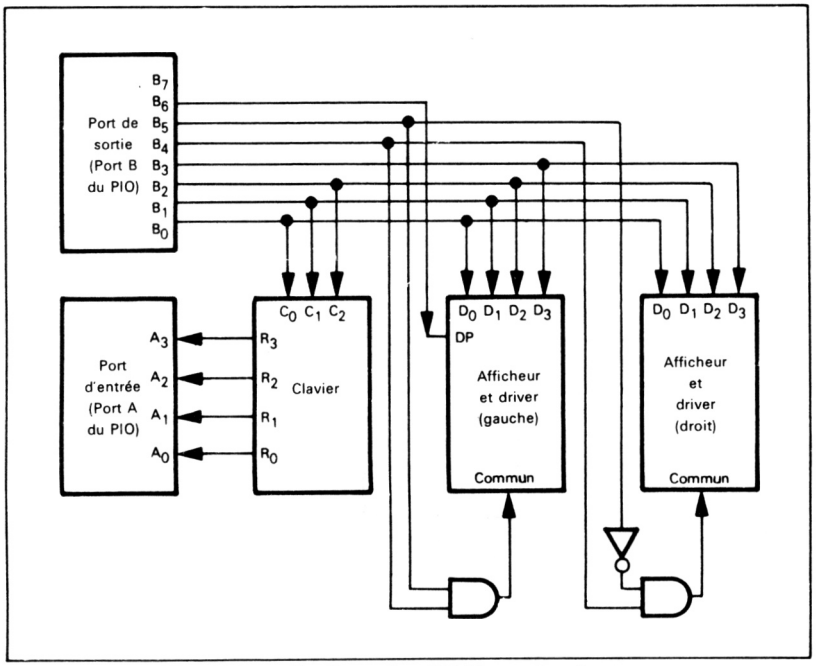


Fig. 16.1. — Configuration d'E/S pour le minuteur.

Tableau 16.1. — Connexions d'entrée pour le minuteur

Bit d'entrée	Touches connectées
0	'3', '5', '8'
1	'2', '6', '9'
2	'0', '1', '7'
3	'4', '.', 'GO'

Tableau 16.2. — Connexions en sortie pour le minuteur

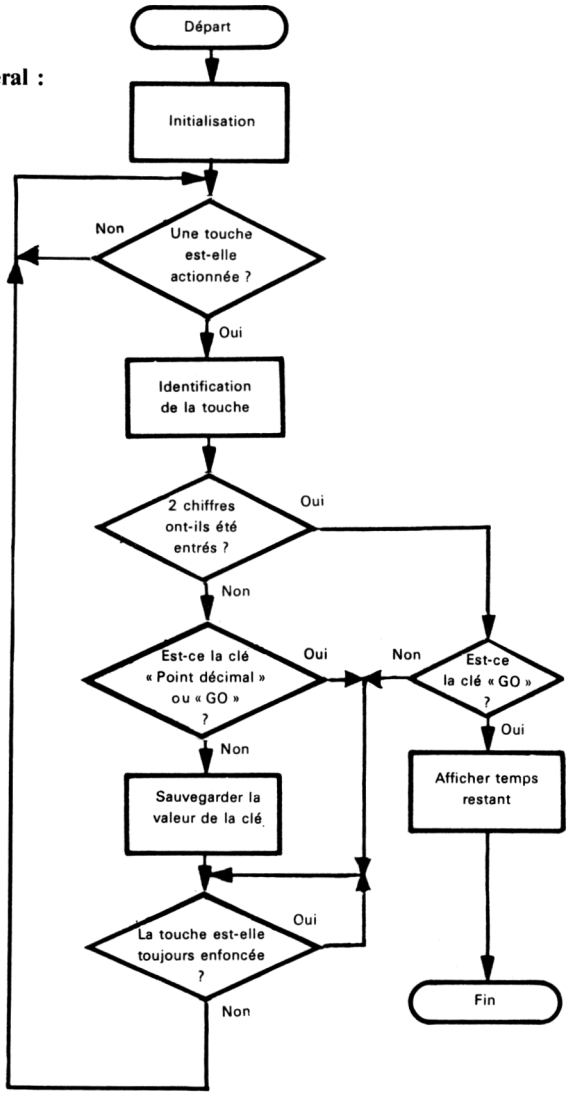
Bit de sortie	Touches connectées
0	'0', '2', '3', '4'
1	'1', '8', '9', 'GO'
2	'5', '6', '7', '.'

Connexions des afficheurs : l'affichage est constitué par des afficheurs 7 segments comprenant leurs propres décodeurs.

Le TIL 309 de Texas Instruments en est un exemple type ; il se compose d'un circuit interne TTL MSI avec mémoire, décodeur et amplificateur. Naturellement, les afficheurs standards 7-segments sont bien meilleur marché mais nécessitent un logiciel supplémentaire (le programme de conversion 7 segments est donné au chapitre 7). Les données sont affichées comme de simples chiffres décimaux codés en binaire ; les chiffres

sont représentés comme le montre la figure 11.15. Le point décimal est une simple LED qui s'allume lorsque l'entrée du point décimal est un « 1 » logique. D'autres informations concernant les affichages sont données aux références 10 et 11 à la fin de ce chapitre.

Organigramme général :



Description du programme :

Le programme est modulaire et comporte plusieurs séquences. L'accent a été mis sur sa clarté et sa généralisation plutôt que sur son efficacité ; à l'évidence, il n'utilise pas pleinement les capacités du processeur Z80. On va maintenant décrire en détail chaque section du listage.

1) Commentaires d'introduction
Les commentaires d'introduction décrivent complètement le programme ; ils constituent un document auquel les autres usagers se réfèreront pour l'appliquer aisément, le développer et le comprendre. Des formats standards, l'indentation, et des espaces accroissent sa lisibilité du programme.

2) Définition des variables
Toutes les définitions de variables sont placées au début du programme de façon à être facilement testées et changées. Chaque variable est placée dans une liste alphabétique avec les autres du même type ; des commentaires expliquent leur signification. Les catégories sont :

a) Constantes du système, pouvant varier d'un système à l'autre en fonction de l'espace mémoire attribué aux différents programmes ou aux divers types de mémoires.
b) Stockages temporaires (en RAM) pour les variables.
c) Adresses des ports d'E/S (PIO).
d) Définitions.
Les constantes du système mémoire sont placées dans les définitions de façon que l'utilisateur puisse translater le programme, les stockages temporaires et la pile sans avoir aucun autre changement à opérer. Ces constantes peuvent être changées pour s'adapter à d'autres programmes ou pour coïncider avec la distribution des adresses ROM et RAM d'un système particulier.
Les rangements temporaires sont réservés à l'aide de la pseudo-opération DEFS (« Define Storage », définition de rangements). Une pseudo-opération ORG (origine) les place en une zone particulière de la mémoire. Aucune valeur n'est chargée dans ces emplacements, afin que le programme puisse ensuite être logé en ROM ou PROM et que le système puisse être activé dès la mise sous tension sans aucun transfert.
Chaque adresse de port reçoit un nom, afin que les adresses ainsi affectées soient aisément modifiées selon les diverses configurations. Ces appellations serviront à distinguer clairement les registres de contrôle des registres de données.
Les définitions clarifient la signification de certaines constantes et permettent aux paramètres d'être facilement changés. Chaque définition est donnée dans la forme (binaire, hexadécimale, octale, ASCII ou décimale) pour laquelle sa signification est la plus évidente. Les paramètres (tels que le temps d'anti-rebonds) sont placés ici, afin qu'ils puissent être adaptés aux besoins des divers systèmes.

3) Initialisation
La mémoire d'adresse 0 (adresse de RESET du Z80) contient un JUMP à l'adresse de départ du programme principal. Ce dernier peut ainsi être logé n'importe où en mémoire et être lancé à l'aide d'un RESET.
L'initialisation consiste en quatre étapes :

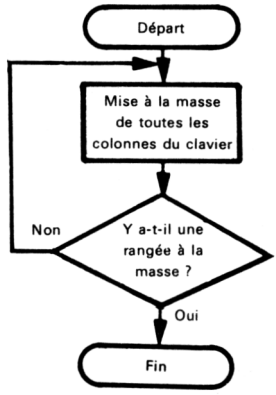
a) On place une valeur de départ dans le pointeur de pile. La pile servira uniquement à ranger les adresses de retour de sous-programmes.
b) On configure les registres de contrôle du PIO.
c) On met à zéro le nombre de touches frappées, au départ.
d) On initialise la position de rangement du prochain chiffre entré en y plaçant la valeur de départ de la table. Une procédure indirecte est utilisée dans laquelle la position

mémoire KEYAD (le mot KEY signifie « touche », en anglais) contient l'adresse à laquelle le chiffre suivant sera rangé. Chaque fois que le programme accepte une frappe, il incrémente KEYAD de façon à placer le chiffre suivant dans la cellule suivante.

4) Attente de la frappe d'une touche.

La fermeture d'une touche est reconnue en mettant à la masse toutes les colonnes du clavier, puis en recherchant si des rangées sont à la masse (puisque l'action sur une touche provoque un contact rangée-colonne). Notons que le programme ne suppose nullement que les entrées inutilisées soient à l'état haut ; au lieu de cela, les bits attachés au clavier sont masqués avec un ET logique (instruction AND).

Organigramme :



5) L'anti-rebonds

Le programme élimine les rebonds mécaniques par logiciel, en temporisant deux millisecondes. C'est généralement suffisant ; le sous-programme DÉLAI compte simplement pendant 1 milliseconde en utilisant le registre C. Le nombre de millisecondes est stocké dans l'accumulateur. Le programme DELAY doit être ajusté si une horloge ou des mémoires plus lentes sont employées. Il suffit alors de modifier simplement la constante MSCNT.

6) Identification d'une fermeture de clé

La touche pressée est identifiée par mise à la masse d'une seule colonne à la fois puis en observant si un contact est établi. Si oui (la colonne de la touche est donc trouvée), on détermine la rangée par décalage de l'entrée jusqu'à ce qu'un bit nul soit détecté.
Les séquences de sortie requises pour mettre à la masse les colonnes successives du clavier sont dans une table, PATT, rangée en mémoire. La séquence finale de ce tableau est un marqueur (ECODE) qui indique que toutes les colonnes ont été mises à la masse sans qu'une frappe ait été détectée. Il indique aussi au programme principal que le contact n'a pas pu être identifié (par exemple, une erreur matérielle est survenue avant sa détection).

Organigramme :

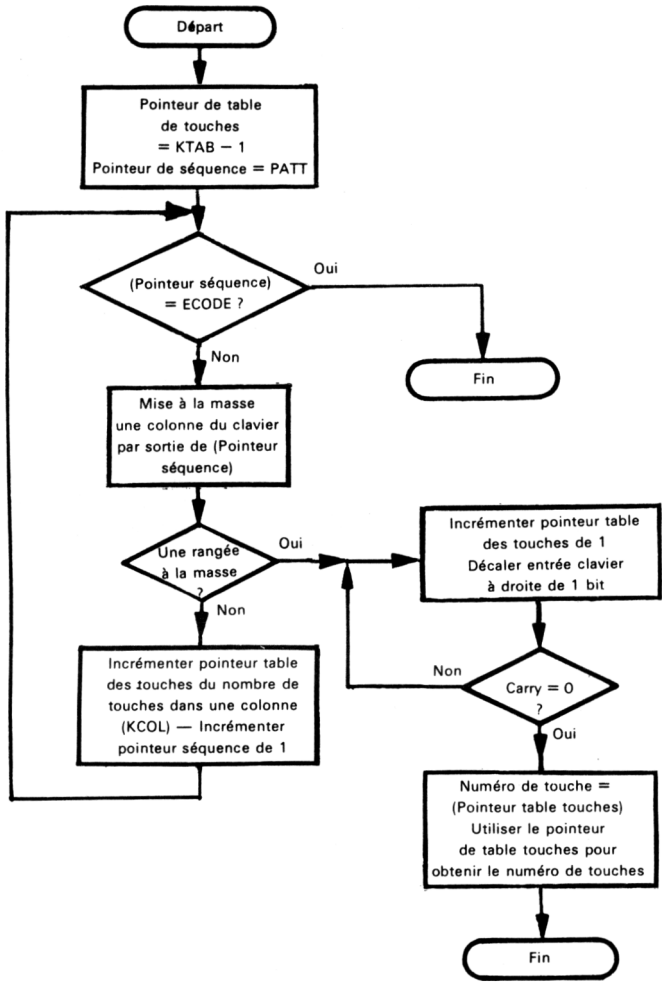


TABLE DES TOUCHES L'identification des touches s'exécute avec le tableau KTAB (« Keyboard Table » = table du clavier), rangé en mémoire. Les touches de la première colonne (reliées au bit de plus faible poids, en sortie) sont suivies par celles de la seconde colonne, etc. Dans chaque colonne, la touche de la rangée reliée au bit d'entrée de plus faible poids est la première ; etc. Ainsi, chaque fois qu'une colonne est scrutée sans qu'un contact soit établi, le nombre de touches dans une colonne (KCOL) doit être additionné à l'adresse de base de la table de touches afin de pointer la colonne suivante. Le pointeur de table de touches est également incrémenté de 1 avant l'examen de chaque bit de l'entrée rangée ; ce processus cesse dès qu'une entrée à zéro est lue. Remarquons que le pointeur de table de touches démarre une position avant le début de la table, car il est toujours incrémenté dès qu'on recherche la bonne rangée.

Si l'on ne peut identifier un contact de touche, on l'ignore et on poursuit la recherche.

7) Action sur identification de touche
Si le programme dispose de suffisamment de chiffres (deux, dans ce cas simple), il attend seulement une action sur la touche GO et ignorera toutes les autres. **S'il trouve une touche de chiffre, il sauvegarde sa valeur dans la table, incrémente le nombre de touches pressées, et incrémente le pointeur de la table.**
Si l'entrée est incomplète, le programme doit attendre que la fermeture d'un contact par action sur une touche s'achève, de façon à ne pas l'accepter à nouveau comme une seconde entrée. L'opérateur doit attendre entre deux frappes (c'est-à-dire, relâcher une touche avant d'en frapper une nouvelle). Notons que le programme identifiera l'action simultanée sur deux touches comme une seule, et ce sera la touche définie la première par le programme qui sera prise en compte. Une version améliorée de ce programme afficherait les chiffres au fur et à mesure de leur introduction et permettrait à l'opérateur d'omettre un zéro inutile de tête et de queue (de frapper, par exemple, « . », « 7 », « 60 » pour compter sept dixièmes d'une minute).

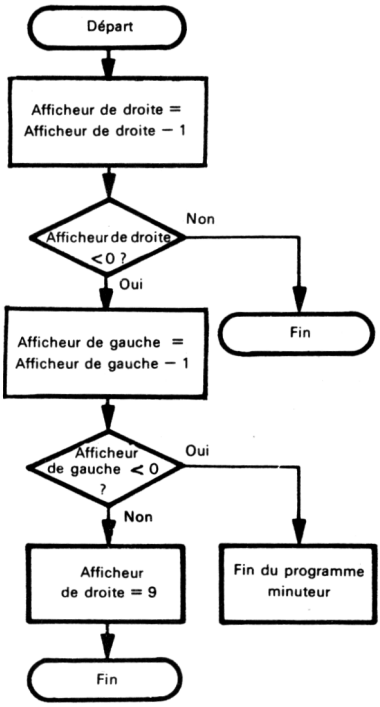
8) Etablissement de l'affichage
Les chiffres sont placés dans des registres ou des mémoires avec le bit 4 positionné à un, de façon que la sortie soit envoyée aux afficheurs. Les bits 5 et 6 sont à un pour que le chiffre le plus significatif s'affiche à gauche et allume le point décimal.

9) Activation des afficheurs LED
Chaque afficheur est mis en service pendant deux millisecondes. Ce processus est répété 1500 fois pour obtenir un temps total de 6 secondes. Les impulsions de mise en service sont suffisamment fréquentes pour que les afficheurs donnent l'impression d'être allumés en continu.

Organigramme : (voir page 591)

10) Décrémentation du compteur d'affichage
La valeur du chiffre de plus faible poids est décrémentée de 1. Si cette opération affecte le bit 4 (LEDON, utilisé pour mettre en service les afficheurs), c'est que le chiffre est devenu négatif. Une retenue (« borrow ») doit alors être obtenue à partir du chiffre de plus fort poids. Si la retenue du chiffre le plus significatif affecte le bit 4, le compteur a dépassé zéro et le décomptage est terminé. Autrement, le programme établit la valeur du chiffre le moins significatif à 9 et continue.
Notons que les commentaires décrivent à la fois les deux sections du programme et les instructions individuelles. Ils expliquent ce que le programme est en train de faire, et non ce que les codes d'instructions spécifiques veulent dire. Des espaces et l'indentation ont été utilisés pour rendre ce programme plus lisible.

```
; NOM DU PROGRAMME : MINUTEUR
; DATE DU PROGRAMME : 10/24/78
; PROGRAMMEUR : L.A. LEVENTHAL
; BESOINS EN MEMOIRE : D1 (209) OCTETS
; BESOINS EN RAM : 5 OCTETS
; BESOINS EN E/S : 1 PORT D'ENTREES, 1 PORT DE SORTIES (1 PIO Z80)
;
;
; PROGRAMME DE MINUTEUR LOGICIEL ACCEPTANT DES ENTREES D'UN CLAVIER
; DE TYPE CALCULATRICE. IL DECOMPTE LES TEMPS EN LES AFFICHANT SUR
; DEUX AFFICHEURS LEDS 7 SEGMENTS, EN MINUTES ET DIZIEMES DE MINUTE.
```



Organigramme

```
;
; CLAVIER
;
; ON SUPPOSE QUE LE CLAVIER EST A 12 TOUCHES
; LES TROIS COLONNES SONT CONNECTEES EN SORTIES DU PROCESSEUR ; UNE
;     COLONNE DE TOUCHES PEUT ETRE MISE A LA MASSE
; LES QUATRE RANGEES SONT CONNECTEES EN ENTREES DU PROCESSEUR,
;     DE FAÇON QUE LES TOUCHES PUISSENT ETRE IDENTIFIEES.
; L'ANTI-REBONDS DU CLAVIER EST ASSURE PAR UNE TEMPORISATION DE 2 MILLISECONDES
;     APRES DETECTION DE LA FERMETURE D'UNE TOUCHE
; UNE NOUVELLE FERMETURE N'EST IDENTIFIEE QU'APRES QUE LA TOUCHE
;     PRECEDEMMENT FERMEE A ETE RELACHEE, CAR AUCUN STROBE N'EST UTILISE
; LES COLONNES DU CLAVIER SONT CONNECTEES AUX BITS 0 A 2 DU PORT B DU PIO
; LES RANGEES DU CLAVIER SONT CONNECTEES AUX BITS 0 A 3 DU PORT A DU PIO
;
; AFFICHEURS
;
; ON UTILISE DEUX AFFICHEURS 7 SEGMENTS A LED AVEC DECODEUR SEPARE
;     (7447 OU 7448 SELON LE TYPE D'AFFICHEUR)
; LES DONNEES APPLIQUEES AUX DECODEURS PROVIENNENT DES BITS 0 A 3 DU PORT B
;     DU PIO
; LE BIT 4 DU PORT B DU PIO SERT A ACTIVER LES AFFICHEURS (IL EST A 1 LORS DE
;     L'ENVOI DE DONNEES AUX LED)
; LE BIT 5 DU PORT B DU PIO SERT A LA SELECTION D'UN AFFICHEUR (IL
;     EST A 1 POUR L'AFFICHEUR DE POIDS FORT, A 0 POUR CELUI DE POIDS FAIBLE)
; LE BIT 6 DU PORT B DU PIO COMMANDE LE POINT DECIMAL DE
;     L'AFFICHEUR DE POIDS FORT (IL EST A 1 POUR L'ALLUMAGE)
```

```
; METHODE
;
; PHASE 1 — INITIALISATION
;
; LE POINTEUR DE PILE (SERVANT AUX ADRESSES DE RETOUR DE SOUS-PROGRAMMES)
;     EST INITIALISEE. LE NOMBRE DE TOUCHES-CHIFFRES PRESSEES EST INITIALISE A ZERO
;     ET L'ADRESSE DE DEPART DE LA TABLE CONTENANT LES CHIFFRES ENTRES
;     EST INITIALISEE
;
; PHASE 2 — ATTENTE D'UNE FRAPPE DE TOUCHE
; TOUTES LES COLONNES DU CLAVIER SONT MISES A LA MASSE ; LES RANGEES SONT
;     EXAMINEES JUSQU'A LA DETECTION D'UNE FERMETURE DE CONTACT
; PHASE 3 — ANTI-REBONDS D'UNE TOUCHE
; UNE TEMPORISATION DE 2 MS ELIMINE LES REBONDS DES TOUCHES
; PHASE 4 — IDENTIFICATION D'UNE TOUCHE ACTIONNEE
; L'IDENTIFICATION SE FAIT PAR LA MISE A LA MASSE D'UNE UNIQUE COLONNE ;
;     ON DETERMINE LA RANGEE DANS LAQUELLE SE TROUVE LA TOUCHE FRAPPEÈ
; UNE TABLE SERT A CODER LA TOUCHE SELON SA RANGEE ET SA COLONNE
; LES CHIFFRES SONT IDENTIFIES PAR LEUR VALEUR DANS LA TABLE DES CLES. LE
;     POINT DECIMAL EST LE NUMERO 10 ET LA TOUCHE "GO" PORTE LE NUMERO 11.
; PHASE 5 — RANGEMENT DU CHIFFRE CORRESPONDANT A UNE TOUCHE ACTIONNEE
; LES CHIFFRES DES TOUCHES FRAPPEES SONT SAUVEGARDES DANS LA TABLE
;     DES CHIFFRES JUSQU'A CE QUE DEUX CHIFFRES AIENT ETE IDENTIFIES.
;     LE POINT DECIMAL, D'AUTRES CHIFFRES, OU L'ENFONCEMENT DE LA TOUCHE "GO"
;     RESTENT IGNORES TANT QUE DEUX CHIFFRES N'ONT PAS ETE ENTRES.
; APRES L'ACCEPTATON DE DEUX CHIFFRES, LA TOUCHE "GO" SERT A LANCER
;     LE DECOMPTAGE
; PHASE 6 — DECOMPTAGE DU TEMPS SUR LES AFFICHEURS
; POUR L'AFFICHAGE DU DECOMPTAGE SUR LES LEDS, LE CHIFFRE DE TETE REPRESENTE LE
;     NOMBRE DE MINUTES RESTANT ET LE CHIFFRE DE FAIBLE POIDS, LE NOMBRE
;     DE DIXIEMES DE MINUTE RESTANT A DECOMPTER
;
;
;
; DEFINITION DES VARIABLES DU MINUTEUR
;
; CONSTANTES DU SYSTEME MEMOIRE
;
BEGIN   EQU     50H             ; BEGIN EST L'ADRESSE DE DEPART
                                ; DU PROGRAMME
LASTM   EQU     1000H           ; LASTM L'ADRESSE DE DEPART DE LA PILE
TEMP    EQU     800H            ; TEMP L'ADRESSE DE DEPART DU RANGEMENT EN RAM

; RANGEMENT TEMPORAIRE EN RAM

        ORG     TEMP
KEYAD:  DEFS    2               ; KEYAD CONTIENT L'ADRESSE DANS LE TABLEAU DES CHIFFRES
                                ; ENTRES, DU PROCHAIN CHIFFRE
                                ;
KEYNO:  DEFS    2               ; KEYNO EST LE TABLEAU DES TOUCHES
                                ; IL CONTIENT L'IDENTIFICATION
                                ; DES TOUCHES FRAPPEES
NKEYS:  DEFS    1               ; NKEYS CONTIENT LE NOMBRE
                                ; DE CLES FRAPPEES
```

```
;
; UNITES D'E/S ET ADRESSES DU PIO
PIODRA  EQU     0E0H            ; PORT EN ENTREE POUR LE CLAVIER
PIOCRA  EQU     0E2H
PIODRB  EQU     0E1H            ; PORT EN SORTIE POUR LE CLAVIER ET
                                ; L'AFFICHAGE
PIOCRB  EQU     0E3H
;
;DEFINITIONS
;
DECPT   EQU     6               ; POSITION DU BIT QUI AFFICHE LE POINT DECIMAL
ECODE   EQU     0FFH            ; CODE ERREUR SI LE PROGRAMME NE TROUVE
                                ;     PAS LA TOUCHE
GOKEY   EQU     11              ; NOMBRE IDENTIFIANT LA TOUCHE « GO »
LEDON   EQU     4               ; POSITION DU BIT QUI ENVOIE LA SORTIE AUX LEDS
LEDSL   EQU     5               ; POSITION DU BIT QUI SELECTIONNE L'AFFICHAGE
                                ;     DE TETE
MSCNT   EQU     0F9H            ; COMPTE POUR TEMPORISER 1 MS
MXKEY   EQU     2               ; NOMBRE MAXIMAL DE FRAPPES
                                ;     UTILISEES
NROWS   EQU     4               ; NOMBRE DE RANGEES DU CLAVIER OU DE
                                ;     TOUCHES SUR UNE COLONNE
OPEN    EQU     00001111B       ; ENTREE CLAVIER EN L'ABSENCE
                                ;     DE FRAPPE
TPULS   EQU     2               ; NOMBRE DE MS ENTRE DEUX AFFICHAGES
TWAIT   EQU     2               ; NOMBRE DE MS POUR L'ANTI-REBONDS
;
;
;
        ORG     0
;
; PROGRAMME DE RESET POUR ACCEDER AU PROGRAMME TIMER (MINUTEUR)
;
        JP      BEGIN           ; LANCER PROGRAMME TIMER
;
; INITIALISATION DU PROGRAMME TIMER
;
        ORG     BEGIN
        LD      A,01001111B     ; ENTREE : PORT A DU PIO
        OUT     (PIOCRA),A
        LD      A,00001111B     ; SORTIE : PORT B DU PIO
        OUT     (PIOCRB),A
        LD      SP,LASTM        ; PLACER LA PILE EN FIN DE MEMOIRE
        SUB     A
        LD      (NKEYS),A       ; NOMBRE DE TOUCHES NUMERIQUES FRAPPEES = ZERO
        LD      HL,KEYNO        ; POSITION DE DEPART DES TOUCHES NUMERIQUES
        LD      (KEYAD),HL
;
; SURVEILLANCE DU CLAVIER POUR DETECTER UNE FRAPPE
;
START:  CALL    SCANC           ; ATTENTE FRAPPE
;
; ATTENTE DU TEMPS D'ANTI-REBONDS
;
        LD      A,TWAIT         ; CHARGEMENT DU TEMPS D'ANTI-REBONDS EN MS
        CALL    DELAY           ; ATTENDRE FIN DES REBONDS
```

```
;
; IDENTIFICATION DE LA TOUCHE
;
        CALL    IDKEY           ; IDENTIFIER UNE FRAPPE
        CP      ECODE           ; A-T-ELLE ETE IDENTIFIEE ?
        JR      Z,START         ; NON, ATTENDRE UN AUTRE CONTACT
;
; CAS DE L'IDENTIFICATION DE LA TOUCHE
;
        LD      B,A             ; RANGER LE NUMERO DE LA TOUCHE
        LD      HL,NKEYS        ; VERIFIER LE NOMBRE MAXIMAL
                                ;     DE FRAPPES
        LD      A,(HL)
        CP      MXKEY           ; MAXIMUM ATTEINT ?
        JR      Z,KEYF          ; OUI, SURVEILLER LA TOUCHE GO
        LD      A,B             ; NON, SURVEILLER SEULEMENT LES TOUCHES NUMERIQUES
        CP      10              ; TOUCHE NUMERIQUE ?
        JR      NC,WAITK        ; NON, L'IGNORER
        INC     (HL)            ; OUI, INCREMENTER LE COMPTEUR TOUCHES NUMERIQUES
        LD      HL,(KEYAD)      ; RANGEMENT NOMBRE DE TOUCHES DANS LE TABLEAU
        LD      (HL),A
        INC     HL
        LD      (KEYAD),HL
;
; ATTENTE DE LA FIN DU CONTACT
;
WAITK:  CALL    SCANO           ; ATTENTE FIN DE FRAPPE DE LA TOUCHE
        JR      START           ; ATTENTE FRAPPE SUIVANTE
;
; ATTENTE DE LA FRAPPE DE LA TOUCHE GO
; SI NOMBRE DE CHIFFRES SUFFISANT
KEYF:   LD      A,B             ; CHARGER NUMERO DE TOUCHE FRAPPEE
        CP      GOKEY           ; TOUCHE GO ?
        JR      NZ,WAITK        ; NON, L'IGNORER
;
; RANGEMENT DES CHIFFRES EN REGISTRE POUR AFFICHAGE
;
        LD      HL,KEYNO
        LD      D,(HL)          ; CHARGER CHIFFRE GAUCHE
        SET     DECPT,D         ; ALLUMAGE POINT DECIMAL
        SET     LEDON,D         ; ETABLISSEMENT SORTIE VERS LEDS
        SET     LEDSL,D         ; SELECTION AFFICHEUR GAUCHE
        INC     HL
        LD      E,(HL)          ; CHARGER CHIFFRE DROITE
        SET     LEDON,E         ; SORTIE VERS LEDS
;
; ENVOYER L'AFFICHAGE AUX LEDS
;
        LD      C,PIODRB        ; CHARGER ADRESSE DU PORT DE SORTIE
LEDLP:  LD      H,6             ; POSITIONNER LES COMPTEURS A 6 SECONDES
TLOOP:  LD      B,250
LDPUL:  OUT     (C),D           ; SORTIE DU CHIFFRE DE GAUCHE SUR LED1
        LD      A,TPULS         ; TEMPORISATION ENTRE CHIFFRES
        CALL    DELAY
        OUT     (C),E           ; SORTIE DU CHIFFRE DE DROITE SUR LED2
        LD      A,TPULS         ; TEMPORISATION ENTRE CHIFFRES
        CALL    DELAY
        DJNZ    LDPUL
        DEC     H
        JR      NZ,TLOOP
```

```
;
; DECREMENTATION DU COMPTE SUR L'AFFICHAGE
;
        DEC     E               ; DECOMPTE CHIFFRE DE DROITE
        BIT     LEDON,E         ; CHIFFRE DE DROITE < 0 ?
        JR      NZ,LEDLP        ; NON, CONTINUER
        DEC     D               ; DECOMPTE CHIFFRE GAUCHE
        BIT     LEDON,D         ; CHIFFRE DE GAUCHE < 0 ?
        JP      Z,BEGIN         ; OUI, ATTENDRE TACHE SUIVANTE
        LD      E,9             ; NON, METTRE A 9 LE CHIFFRE DE DROITE
        SET     LEDON,E         ; ETABLISSEMENT SORTIES VERS LED
        JR      LEDLP           ; RETOUR A LA SECTION D'AFFICHAGE
;
; LE SOUS-PROGRAMME SCANC SURVEILLE LE CLAVIER ATTENDANT UNE FRAPPE
; TOUTES LES ENTREES DU CLAVIER SONT MISES A LA MASSE
;
SCANC:  SUB     A               ; MISE A LA MASSE DE TOUTES LES COLONNES DU CLAVIER
        OUT     (PIODRB),A
        IN      A,(PIODRA)
        AND     OPEN            ; IGNORER LES ENTREES INUTILISEES
        CP      OPEN            ; Y-A-T-IL UNE FERMETURE DE TOUCHE ?
        JR      Z,SCANC         ; NON, CONTINUER LA RECHERCHE
        RET
;
; LE SOUS-PROGRAMME DELAY ATTEND LE NOMBRE DE
;     MILLISECONDES SPECIFIE DANS LE REGISTRE A
;
DELAY:  EXX                     ; SAUVEGARDE REGISTRES UTILISATEUR
DLY1:   LD      C,MSCNT         ; METTRE 1 MS DANS REGISTRE C
WTLP:   DEC     C               ; ATTENDRE 1 MS
        JR      NZ,WTLP
        DEC     A               ; DECOMPTE DU NOMBRE DE MS
        JR      NZ,DLY1
        EXX                     ; RESTAURER REGISTRES UTILISATEUR
        RET
;
; LE SOUS-PROGRAMME IDKEY DETERMINE LA RANGEE ET LA COLONNE DE LA
;     FERMETURE ET IDENTIFIE LA TOUCHE A L'AIDE D'UN TABLEAU
;
IDKEY:  LD      BC,PATT         ; POINTER LA SEQUENCE DE SCRUTATION
        LD      HL,KTAB-1       ; POINTEUR DU DEBUT DE TABLE
        LD      DE,NROWS        ; CHARGER NOMBRE TOUCHES DANS UNE COLONNE
;
; SCRUTATION COLONNES SUCCESSIVES POUR DETECTER UNE FRAPPE
;
FCOL:   LD      A,(BC)          ; CHARGER SEQUENCE POUR MISE A LA MASSE COLONNE
        CP      ECODE           ; TOUTES LES COLONNES SCRUTEES ?
        RET     Z               ; OUI, RETOUR CODE ERREUR
        OUT     (PIODRB),A      ; SCRUTER LES COLONNES
        IN      A,(PIODRA)
        AND     OPEN            ; IGNORER LES ENTREES INUTILISEES
        CP      OPEN            ; FERMETURE D'UNE TOUCHE DANS CETTE COLONNE ?
        JR      NZ,FROW         ; OUI, DETERMINER LA RANGEE
        ADD     HL,DE           ; NON, DEPLACER LE POINTEUR DU TABLEAU TOUCHES
                                ;     A LA COLONNE SUIVANTE
        INC     BC              ; POINTER LA SEQUENCE DE SCRUTATION SUIVANTE
        JR      FCOL
```

```
; DETERMINATION DE LA RANGEE
;
FROW:   INC     HL              ; METTRE LE POINTEUR DU TABLEAU SUR RANGEE SUIVANTE
        RRCA                    ; RANGEE SUIVANTE A LA MASSE ?
        JR      C,FROW          ; NON, CONTINUER A CHERCHER
;
; IDENTIFICATION DE LA TOUCHE A PARTIR DU TABLEAU
;
        LD      A,(HL)          ; LECTURE DU NUMERO DE CLE
        RET
;
; SEQUENCES DE SCRUTATION UTILISEES POUR METTRE A LA MASSE UNE COLONNE A LA FOIS.
; SEQUENCE D'ERREUR INDIQUANT QUE TOUTES LES COLONNES ONT ETE SCRUTEES
; LA COLONNE CORRESPONDANT AU BIT DE SORTIE 0 EST D'ABORD SCRUTEE, PUIS
;     CELLE CORRESPONDANT AU BIT DE SORTIE 1, ETC.
;
PATT:   DEFB    00000110B
        DEFB    00000101B
        DEFB    00000011B
        DEFB    ECODE
;
; TABLEAU DU CLAVIER
;
; LES COLONNES SONT LES INDICES PRIMAIRES, LES RANGEES SONT LES INDICES SECON-
DAIRES
; LES TOUCHES DE LA COLONNE CORRESPONDANT AU BIT DE SORTIE 0, SONT SUIVIES
;     DE CELLES DE LA COLONNE CORRESPONDANT AU BIT DE SORTIE 1, ETC.
;     POUR CHAQUE COLONNE, LA TOUCHE CORRESPONDANT AU BIT D'ENTREE 0 EST
;     SUIVIE DE CELLE CORRESPONDANT AU BIT D'ENTREE 1, ETC.
; LES TOUCHES NUMERIQUES VONT DE 0 A 9. LE POINT DECIMAL VAUT 10, LA TOUCHE GO 11
;
KTAB:   DEFB    3               ;C0,R0
        DEFB    2               ;C0,R1
        DEFB    0               ;C0,R2
        DEFB    4               ;C0,R3
        DEFB    8               ;C1,R0
        DEFB    9               ;C1,R1
        DEFB    1               ;C1,R2
        DEFB    11              ;C1,R3
        DEFB    5               ;C2,R0
        DEFB    6               ;C2,R1
        DEFB    7               ;C2,R2
        DEFB    10              ;C2,R3
;
; LE SOUS-PROGRAMME SCANO SCRUTE LE CLAVIER ET ATTEND LA FIN D'UNE FRAPPE
;     DE TOUCHE AVANT D'EN DETECTER UNE SUIVANTE
;
SCANO:  SUB     A               ; MISE A LA MASSE DE TOUTES LES COLONNES DU CLAVIER
        OUT     (PIODRB),A
        IN      A,(PIODRA)
        AND     OPEN            ; IGNORER LES ENTREES INUTILISEES
        CP      OPEN            ; TOUCHE ENCORE FERMEE ?
        JR      NZ,SCANO        ; OUI, CONTINUER LA SCRUTATION
        RET
        END
```

PROJET N° 2 : un thermomètre numérique

But : réaliser un thermomètre numérique donnant la température en degrés Celsius sur deux afficheurs à 7-segments.

Matériel : Le projet utilise un port d'entrée et un port de sortie, deux afficheurs à 7-segments, un inverseur 74LS04, une porte NAND 74LS00 ou une porte AND 74LS08 selon la polarité des afficheurs, un convertisseur monolithique A/N type AD7570J de *Analog Devices,* sur 8 bits, un comparateur LM311, et divers drivers, résistances et capacités requises par les afficheurs et le convertisseur. (Reportez-vous au chapitre 11 pour une discussion sur la conversion A/N. Voir également la référence 1 à la fin de ce chapitre.)

La figure 16.2 montre l'organisation matérielle. La ligne de sortie 7 du port B du PIO sert à l'émission d'un signal de « Début de conversion » vers le convertisseur A/D (analogique-digital, synonyme de A/N). Les lignes d'entrées 0 à 7 sont directement connectées aux huit lignes de données du convertisseur. Les lignes de sorties 0 à 3 servent à envoyer les chiffres DCB aux décodeurs-drivers des afficheurs. La ligne de sortie 4 active les afficheurs et la ligne de sortie 5 sélectionne celui de gauche ou de droite (un 1 pour celui de gauche).

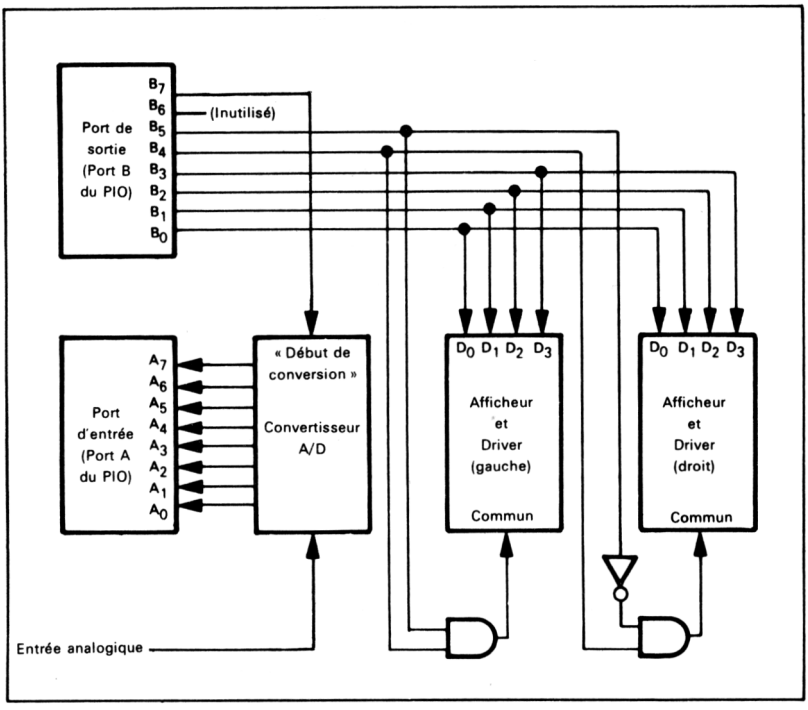
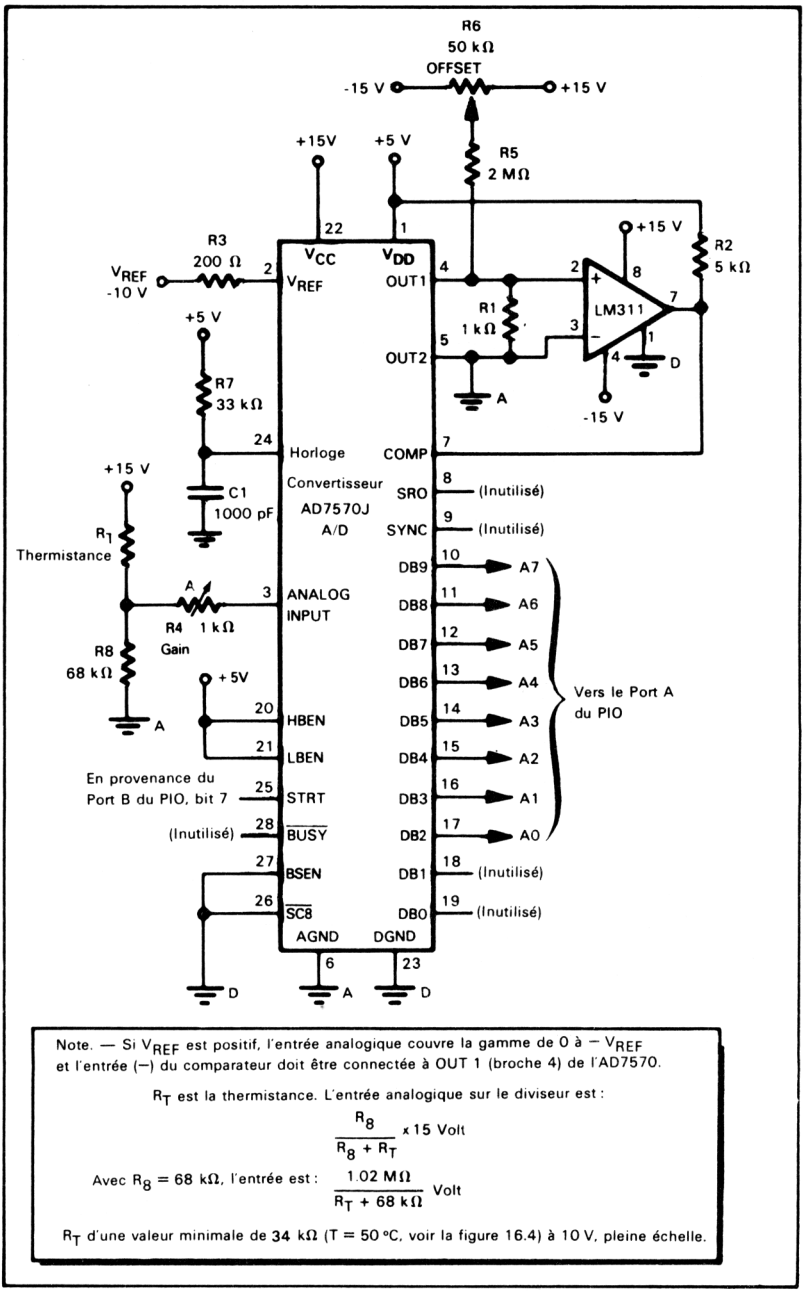


Fig. 16.2. — Configuration des E/S du thermomètre.



Note. — Si V_{REF} est positif, l'entrée analogique couvre la gamme de 0 à $-V_{REF}$ et l'entrée (−) du comparateur doit être connectée à OUT 1 (broche 4) de l'AD7570.

R_T est la thermistance. L'entrée analogique sur le diviseur est :

$$\frac{R_8}{R_8 + R_T} \times 15 \text{ Volt}$$

Avec $R_8 = 68$ kΩ, l'entrée est : $\dfrac{1.02 \text{ MΩ}}{R_T + 68 \text{ kΩ}}$ Volt

R_T d'une valeur minimale de 34 kΩ (T = 50 °C, voir la figure 16.4) à 10 V, pleine échelle.

Fig. 16.3. — Montage du convertisseur.

MATÉRIEL DU THERMOMÈTRE ANALOGIQUE

La section analogique du montage est donnée figure 16.3. La thermistance présente une résistance variable en fonction de la température. La figure 16.4 donne la courbe de variation de la résistance et la figure 16.5 montre la gamme des valeurs de l'intensité du courant au-delà de laquelle la résistance est linéaire. Le programme exécute la conversion en degrés Celsius à l'aide d'une table d'étalonnage. Les

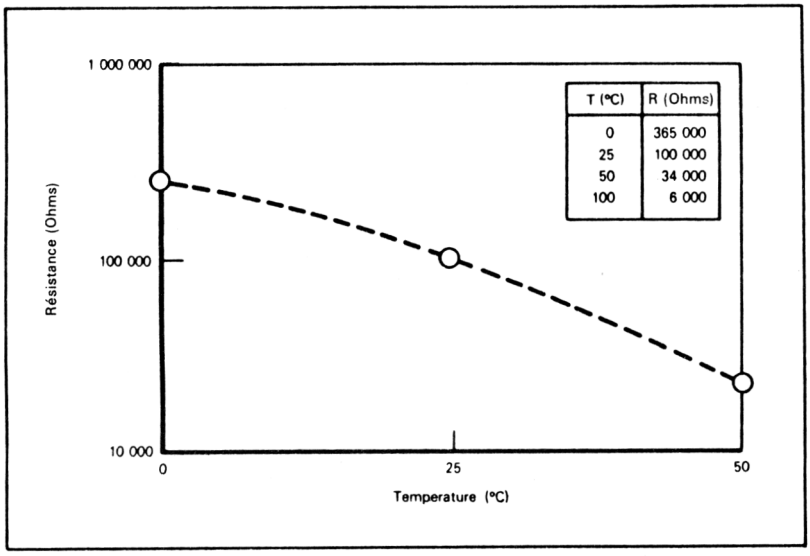


Fig. 16.4. — Caractéristique d'une thermistance Fenwal GA51J1.

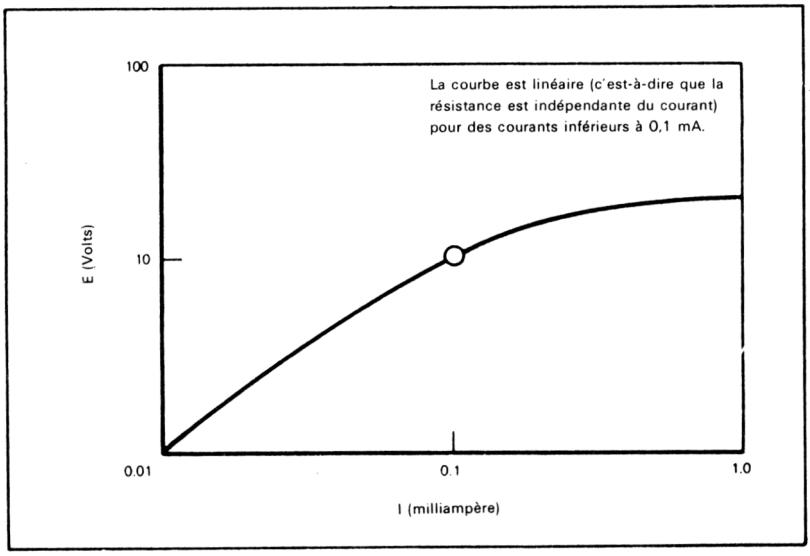


Fig. 16.5. — Courbe type E = f(I) d'une thermistance (à 25 °C).

deux potentiomètres permettent d'ajuster l'étalonnage. Un réseau RC fournit l'horloge au convertisseur, comme le montre la figure 16.3. Les valeurs sont R7 = 33 kΩ et C1 = 1000 pF pour une fréquence d'horloge d'environ 75 kHz. A cette fréquence, le temps maximal de conversion pour huit bits est d'environ 50 microsecondes. Un délai bien supérieur de conversion est attribué, de façon qu'on n'ait pas à tester le signal de fin de conversion.

La version 8 bits du convertisseur demande qu'on établisse les connexions spéciales suivantes. Les huit lignes de données sont DB2 à DB9 (DB1 est toujours à l'état haut durant la conversion, et DB0 à l'état bas). L'entrée cycle court (« Short Cycle »), broche 26 marquée SC8 est connectée à un niveau bas afin que seule une conversion 8 bits soit réalisée. Dans notre cas, « High Byte Enable » (broche 20, HBEN) et « Low Byte Enable » (broche 21, LBEN) ont toutes deux été connectées au niveau haut de façon que la sortie des données soit toujours autorisée.

Le convertisseur A/D utilise la méthode des approximations successives pour effectuer une conversion. Le registre de données du convertisseur est connecté aux entrées d'un convertisseur D/A interne dont les sorties (disponibles en OUT1 et OUT2) sont comparées à l'entrée analogique. Lorsqu'une conversion démarre, la logique met des zéros dans le registre de données, sauf en position du bit de plus fort poids, mis à 1. Si l'entrée analogique est moindre que la valeur interne ainsi créée, le MSB est remis à zéro et c'est le bit suivant qui est mis à 1.

Le processus se poursuit avec tous les bits, ainsi testés. Après huit cycles, la valeur de ce registre est la plus proche du résultat recherché.

La méthode est rapide mais exige que l'entrée reste stable pendant la conversion. Si elle change rapidement ou s'il y a du bruit superposé, il faudra mémoriser le signal. Les références à la fin de ce chapitre décrivent des méthodes plus précises pour gérer des E/S analogiques.

Organigramme général : (voir page 601)

Description du programme :

1) Initialisation

La mémoire d'adresse 0 (le RESET du Z80) contient un saut à l'adresse de départ du programme principal ; l'initialisation effectue la configuration des registres de contrôle du PIO et met le pointeur de pile à la plus haute adresse en RAM. La pile ne sert que pour les adresses de retour de sous-programmes.

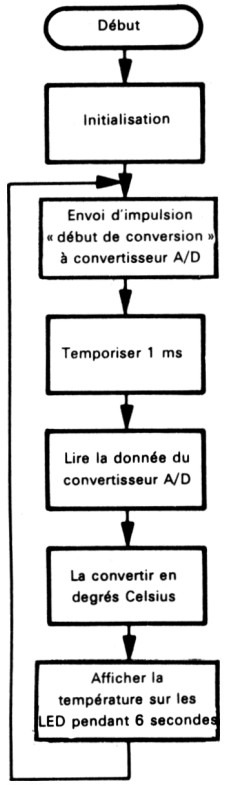
2) Envoi du signal « Début de conversion » au convertisseur A/D.

L'UC envoie une impulsion « Start conversion » en plaçant un 1 sur la ligne de sortie 7, puis en la remettant à zéro.

Chaque conversion demande une telle impulsion de démarrage.

3) Temporiser 1 ms pour la conversion.

Un délai d'une milliseconde après « début de conversion » est largement suffisant pour que cette conversion soit achevée. En fait, le convertisseur n'a besoin que de 100 microsecondes au maximum pour une conversion sur 8 bits. On pourrait réduire ce temps en testant l'état de la ligne BUSY sortant du convertisseur ; elle indique la fin de la conversion en passant à 1, le 0 signifiant une conversion en cours. Dans notre cas, rien ne nous oblige à aller plus vite. Naturellement, on pourrait utiliser les interruptions avec la ligne BUSY connectée à la ligne du STROBE du PIO.



Organigramme général

4) Lecture du convertisseur

La lecture de la donnée n'implique qu'une simple opération d'entrée. Remarquons que le AD7570J de Analog Devices dispose d'une entrée de validation et de sorties trois états, ce qui permet de le relier directement au bus de données du microprocesseur.

Le convertisseur 7570 est, bien sûr, sous employé dans cette application, d'autant plus qu'on l'interface avec le Z80, au moyen d'un PIO. Un convertisseur A/D sur 8 bits plus simple, tel que le National 5357, d'un coût moindre, conviendrait tout autant. Cet appareil se présente sous forme d'un ensemble 12 broches, dispose d'une entrée « START CONVERSION » et fournit des sorties 3 états. Il dispose également d'une mémorisation et d'un signal « FIN DE CONVERSION ».

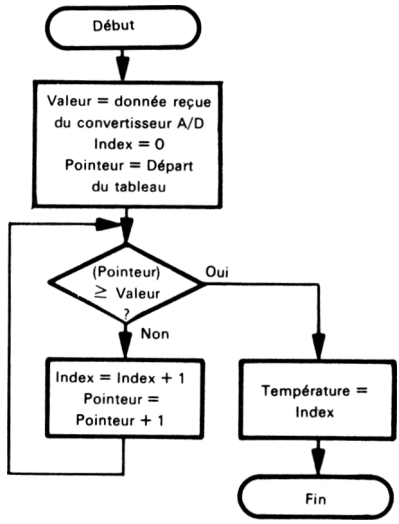
5) Conversion en degrés Celsius

UTILISATION D'UNE TABLE DE CONVERSION

La conversion fait appel à une table contenant les plus grandes valeurs d'entrées correspondant à une température donnée. Le programme appelle la table, recherchant une valeur supérieure ou égale à la valeur reçue du convertisseur. La première valeur répondant à cette condition qu'il trouvera correspond à la température voulue ; cela signifie que si la dixième entrée dans la table fournit la première valeur supérieure ou égale à la sortie du convertisseur, la température est de 10 °C. Une telle méthode est, certes, peu efficace, mais convient à la présente application.

Notons que l'entrée doit être traitée en décimal et non en binaire. La séquence ADD A,L ; DAA garde l'index sous forme de 2 chiffres décimaux et non binaire. Par exemple, l'entrée suivant 9 (en binaire, 00001001) donnera le décimal 10 (00010000 en DCB) et non le dix binaire (00001010). La raison de cette procédure tient au fait qu'on affiche la température sous forme de deux chiffres décimaux et qu'il faudrait exécuter une conversion autrement.
La table pourrait être obtenue par calibrage ou par une approximation mathématique. Le calibrage est plus simple, dans la mesure où il faut de toute façon étalonner le thermomètre. Cette table occupe une position mémoire par valeur de température [1]. Pour calibrer le thermomètre, vous devez d'abord ajuster les potentiomètres réglant la gamme, puis déterminer les valeurs de sorties du convertisseur correspondant à des températures spécifiques.

Organigramme :



6) Préparation des données pour l'affichage

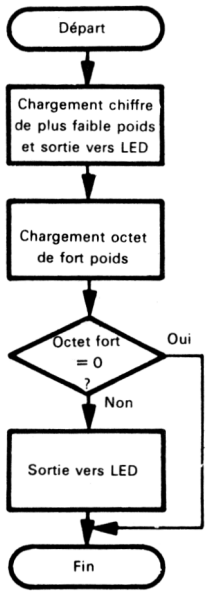
Organigramme : (voir page 603)

SUPPRESSION D'UN ZÉRO DE TÊTE

On supprime le chiffre de plus fort poids par masquage, on met à 1 le bit commandant l'affichage. Le résultat est rangé dans le registre E.

La seule différence, pour le chiffre le plus significatif, est le non-affichage de 0, (par exemple, l'affichage montrera 7 et non 07 pour 7 °C). Il suffit simplement de ne pas établir le bit commandant l'afficheur de gauche si le chiffre à afficher est zéro. Le résultat est rangé dans le registre D.

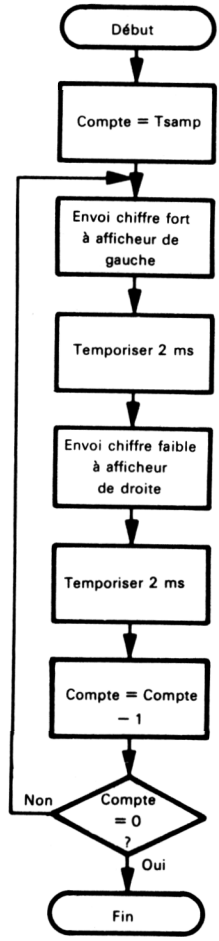
Organigramme



7) Affichage de la température pendant 6 secondes

Organigramme : (voir page 604)

Chaque affichage est envoyé suffisamment souvent pour sembler permanent. Si TPULS était plus long (par exemple, plus de 50 ms), l'affichage semblerait clignoter.
Le programme emploie un compteur sur 16 bits pour compter le temps s'écoulant entre deux mesures successives. Le Z80 dispose d'instructions permettant d'incrémenter ou de décrémenter des paires de registres ou des registres d'index sur 16 bits. Cependant, il n'existe aucun moyen de déceler directement le passage du compteur à zéro, car ces instructions ne modifient pas l'état des indicateurs. Aussi passe-t-on par un OU (OR) logique entre l'octet fort et l'octet faible du compteur. Si le résultat est nul, le compteur 16 bits est à zéro.



Organigramme

```
; NOM DU PROGRAMME : THERMOMETRE
; DATE DU PROGRAMME : 10/20/78
; PROGRAMMEUR : L.A. LEVENTHAL
; BESOINS EN MEMOIRE : 154 OCTETS
; BESOINS EN RAM : NEANT
; BESOINS EN E/S : 1 PORT D'ENTREE, 1 PORT DE SORTIE (1 PIO Z80)
; PROGRAMME DE THERMOMETRE NUMERIQUE. IL ACCEPTE LES ENTREES D'UN
; CONVERTISSEUR A/N PILOTE PAR UNE THERMISTANCE, CONVERTIT L'ENTREE
; EN DEGRES CELSIUS ET AFFICHE LE RESULTAT SUR DEUX AFFICHEURS LED A
;     7-SEGMENTS.
;
; CONVERTISSEUR A/D
;
; LE CONVERTISSEUR A/D EST UN 7570J DE ANALOG DEVICES QUI FOURNIT UNE SORTIE
;     SUR 8 BITS
; LA CONVERSION DEMARRE AVEC UNE IMPULSION SUR LA LIGNE DE DEPART
;     DE CONVERSION (BIT 7 DU PORT B DU PIO)
; ELLE EST EXECUTEE EN 50 MICROSECONDES ET LA DONNEE NUMERIQUE EST MEMORISEE
;
; AFFICHEURS
;
; DEUX AFFICHEURS LED A 7-SEGMENTS AVEC DES DECODEURS DISTINCTS SONT
;     UTILISES. LES DECODEURS SONT DES 7447 OU 7448 SELON LE TYPE D'AFFICHEUR.
; L'ENTREE DONNEES DES DECODEURS EST CONNECTEE AUX BITS 0 A 3
;     DU PORT B DU PIO
; LE BIT 4 DU PORT B DU PIO SERT A ACTIVER LES AFFICHEURS QUAND
;     IL EST A 1
; LE BIT 5 DU PORT B DU PIO SELECTIONNE L'AFFICHEUR EN SERVICE.
;     IL EST A 1 POUR L'AFFICHEUR DE TETE, A 0 POUR CELUI DE «PODS FAIBLE
;
; METHODE
;
; PHASE 1 — INITIALISATION
; ON INITIALISE LA PILE, QUI SERT A LA SAUVEGARDE DES ADRESSES DE RETOUR
; PHASE 2 — IMPULSION SUR LA LIGNE DE LANCEMENT DE CONVERSION
; LE CONVERTISSEUR A/D DEMARRE LA CONVERSION DES QU'UNE IMPULSION SUR
;     SA LIGNE DE "START" DE CONVERSION LUI PARVIENT (A PARTIR DU BIT 7
;     DU PORT B DU PIO)
; PHASE 3 — ATTENTE DE FIN DE CONVERSION
; UNE TEMPORISATION DE 1 MS LAISSE A LA CONVERSION LE TEMPS DE S'EXECUTER
; PHASE 4 — LECTURE DU CONVERTISSEUR ET CONVERSION EN DEGRES CELSIUS
; LA CONVERSION EN DEGRES CELSIUS SE FAIT A L'AIDE D'UNE TABLE DE
;     CORRESPONDANCES CONTENANT LES ENTREES MAXIMALES POSSIBLES POUR CHAQUE
;     TEMPERATURE LUE
; PHASE 5 — AFFICHAGE DE LA TEMPERATURE
; LA TEMPERATURE EST AFFICHEE SUR LES LED PENDANT 6 SECONDES AVANT LA
;     CONVERSION SUIVANTE
; DEFINITIONS DES VARIABLES DU THERMOMETRE
;
;
; CONSTANTES DU SYSTEMES
;
BEGIN   EQU     50H             ; ADRESSE DE DEPART DU PROGRAMME PRINCIPAL
LASTM   EQU     1000H           ; ADRESSE DE DEPART DE LA PILE
;
```

```
; UNITES D'E/S ET ADRESSES DU PIO
;
PIODRA  EQU     0E0H            ; PORT D'ENTREE POUR LE CONVERTISSEUR
PIOCRA  EQU     0E2H
PIODRB  EQU     0E1H            ; PORT DE SORTIE POUR L'AFFICHAGE
PIOCRB  EQU     0E3H
;
;DEFINITIONS
;
LEDON   EQU     4               ; POSITION DU BIT QUI ENVOIE LES DONNEES AUX LEDS
LEDSL   EQU     5               ; POSITION DU BIT QUI SELECTIONNE L'AFFICHAGE
MSCNT   EQU     0F9H            ; COMPTE POUR TEMPORISER 1 MS
STCON   EQU     10000000B       ; SORTIE METTANT A 1 LE DEBUT DE CONVERSION
TPULS   EQU     2               ; IMPULSION D'AFFICHAGE EN MS
TSAMP   EQU     1500            ; TSAMP EST LE NOMBRE D'IMPULSIONS
                                ; DES AFFICHEURS PENDANT UN ECHANTILLONNAGE
                                ; LA LONGUEUR D'UNE PERIODE D'ECHANTILLONNAGE
                                ; EST DE 2.TPULS.TSAMP MILLISECONDES
                                ; LE FACTEUR 2.TPULS VIENT DE L'IMPULSION
                                ; DE CHAQUE PAIRE D'AFFICHEURS PENDANT
                                ; TPULS MS


;
;
;
        ORG     0
;
; PROGRAMME DE RESET POUR ACCEDER AU PROGRAMME DE THERMOMETRE
;
        JP      BEGIN           ; APPEL PROGRAMME THERMOMETRE
;
; INITIALISATION DU PROGRAMME DE THERMOMETRE
;
        ORG     BEGIN
        LD      A,01001111B     ; ENTREE : PORT A DU PIO
        OUT     (PIOCRA),A
        LD      A,00001111B     ; SORTIE : PORT B DU PIO
        OUT     (PIOCRB),A
        LD      SP,LASTM        ; METTRE LA PILE EN FIN DE RAM
;
; IMPULSION DEBUT DE CONVERSION
;
START:  LD      A,STCON
        OUT     (PIODRB),A      ; MISE A 1 DEBUT DE CONVERSION
        SUB     A
        OUT     (PIODRB),A      ; RETOUR A 0 DEBUT CONVERSION
;
; TEMPORISE 1 MS POUR LA CONVERSION
;
        LD      A,1             ; CHARGEMENT DU DELAI DE TEMPORISATION EN MS
        CALL    DELAY           ; ATTENTE DE LA CONVERSION
;
; LECTURE DU CONVERTISSEUR
;
        IN      A,(PIODRA)      ; CHARGEMENT DONNEE DU CONVERTISSEUR
;
; CONVERSION DE LA LECTURE DU CONVERTISSEUR EN 2 CHIFFRES DCB
;
        CALL    CONVR           ; CONVERSION EN DCB
```

```
;
; CHARGEMENT DU CHIFFRE DE POIDS FAIBLE
;
        LD      B,A             ; RANGEMENT CHIFFRE DCB
        AND     0FH             ; EXTRACTION DU CHIFFRE FAIBLE (LSD) PAR MASQUAGE
        SET     LEDON,A         ; SORTIE VERS LEDS
        LD      E,A             ; RANGEMENT LSD DANS REGISTRE E
;
; CHARGEMENT CHIFFRE POIDS FORT EFFACEMENT 0 DE TETE
;
        LD      A,B             ; RECHARGEMENT CHIFFRES DCB
        RRCA                    ; DECALAGE MSD (CHIFFRE POIDS FORT)
        RRCA
        RRCA
        RRCA
        AND     0FH             ; EXTRACTION DU MSD PAR MASQUAGE
        JR      Z,SVMSD         ; NE PAS AFFICHER CHIFFRE TETE S'IL VAUT ZERO
        SET     LEDON,A         ; SORTIE VERS LEDS
        SET     LEDSL,A         ; SELECTION DE L'AFFICHEUR DE TETE
SVMSD:  LD      D,A             ; RANGEMENT MSD DANS REGISTRE D
;
; ENVOI DE L'IMPULSION AUX AFFICHEURS
;
        LD      C,PIODRB        ; LECTURE ADRESSE DU PORT DE SORTIE
        LD      HL,TSAMP        ; LECTURE COMPTEUR D'IMPULSION 16 BITS
DSPLY:  OUT     (C),D           ; ENVOI DU CHIFFRE DE TETE A L'AFFICHEUR
        LD      A,TPULS         ; TEMPORISATION D'AFFICHAGE
        CALL    DELAY
        OUT     (C),E           ; ENVOI DU CHIFFRE FAIBLE A L'AFFICHEUR
        LD      A,TPULS         ; TEMPORISATION D'AFFICHAGE
        CALL    DELAY
        DEC     HL              ; DECOMPTE DU COMPTEUR 16 BITS
        LD      A,H             ; ATTENTION, DEC HL NE MODIFIE PAS L'INDICATEUR Z
        OR      L
        JR      NZ,DSPLY        ; CONTINUER A ENVOYER DES IMPULSIONS AUX AFFICHEURS
        JP      START           ; ECHANTILLONNER A NOUVEAU LA TEMPERATURE
;
; LE SOUS-PROGRAMME DELAY ATTEND LE NOMBRE DE MILLISECONDES
;    SPECIFIE DANS LE REGISTRE A
;
DELAY:  EXX                     ; RANGEMENT DES REGISTRES UTILISATEUR
DLY1:   LD      C,MSCNT         ; CHARGEMENT DU REGISTRE C POUR TEMPORISATION D'1 MS
WTLP:   DEC     C               ; ATTENDRE 1 MS
        JR      NZ,WTLP
        DEC     A               ; DECOMPTE DU NOMBRE DE MS
        JR      NZ,DLY1
        EXX                     ; RESTAURATION DES REGISTRES UTILISATEUR
        RET
;
; LE SOUS-PROGRAMME CONVR CONVERTIT L'ENTREE DU CONVERTISSEUR EN
;    DEGRES CELSIUS A L'AIDE D'UNE TABLE. LES DONNEES EN ENTREE SONT
;    ADMIS DANS L'ACCUMULATEUR. LE RESULTAT (2 CHIFFRES DCB) EST PLACE
;    DANS L'ACCUMULATEUR
```

```
;
; REGISTRES UTILISES : A,B,C,H,L
;
;
CONVR:  LD      HL,DEGTB        ; LECTURE ADRESSE DE BASE DE LA TABLE
                                ;      DE CONVERSION
        LD      B,A             ; RANGEMENT ENTREE A/D
        LD      C,0             ; INITIALISATION DU NOMBRE DE DEGRES A ZERO
CHVAL:  LD      A,(HL)          ; CHARGEMENT ENTREE A PARTIR DE TABLE
        CP      B               ; ENTREE DU CONVERTISSEUR < OU = ENTREE ?
        LD      A,C             ; LECTURE VALEUR EN DEGRES CELSIUS
        RET     NC              ; OUI, VALEUR TROUVEE
        ADD     A,1             ; NON, AJOUTER 1 AU NOMBRE DE DEGRES
        DAA                     ; GARDER LES DEGRES EN DCB
        LD      C,A
        INC     HL
        JR      CHVAL
;
; LE TABLEAU DEGTB A ETE OBTENU PAR ETALONNAGE AVEC UNE REFERENCE CONNUE.
; DEGTB CONTIENT LES PLUS GRANDES VALEURS D'ENTREE CORRESPONDANT A LA
;     LECTURE D'UNE TEMPERATURE PARTICULIERE (C'EST-A-DIRE LA PREMIERE ENTREE
;     EST 58 EN DECIMAL, CE QUI FAIT QUE 58 EST LA PLUS GRANDE VALEUR DONNANT
;     UNE LECTURE DE ZERO DEGRES). LES VALEURS INFERIEURES A ZERO SONT
;     AFFICHEES A ZERO.
```

```
DEGTB:  DEFB  58
        DEFB  61
        DEFB  63
        DEFB  66
        DEFB  69
        DEFB  71
        DEFB  74
        DEFB  77
        DEFB  80
        DEFB  84
        DEFB  87
        DEFB  90
        DEFB  93
        DEFB  97
        DEFB 101
        DEFB 104
        DEFB 108
        DEFB 112
        DEFB 116
        DEFB 120
        DEFB 124
        DEFB 128
        DEFB 132
        DEFB 136
        DEFB 141
        DEFB 145
        DEFB 149
        DEFB 154
        DEFB 158
        DEFB 163
        DEFB 167
        DEFB 172
        DEFB 177
        DEFB 181
        DEFB 186
        DEFB 191
        DEFB 195
        DEFB 200
        DEFB 204
        DEFB 209
        DEFB 214
        DEFB 218
        DEFB 223
        DEFB 227
        DEFB 232
        DEFB 236
        DEFB 241
        DEFB 245
        DEFB 249
        DEFB 253
        DEFB 255
        END
```

RÉFÉRENCES

1. Une méthode qui utilise bien moins de mémoire est décrite par T.A. Seim, « Numerical Interpolation for Microprocessor-based Systems », *Computer Design,* février 1978, pp. 111-116.
2. Auslander D.M. et al., « Direct Digital Process Control : Practice and Algorithms for Microprocessor Applications », *Proceedings of the IEEE,* février 1978, pp. 199-208.
3. Bernstein N., « What to Look for in Analog Input/Output Boards », *Electronics,* 19 janvier 1978, pp. 13-119.
4. Bibbero R.J., *Microprocessors in Instruments and Control,* Wiley, New York, 1977.
5. Burton D.P. et A.L. Dexter, *Microprocessors Systems Handbook,* Analog Devices, Inc., P.O. Box 796, Norwood, MA. 02062, 1977.
6. Finkel J., *Computer-Aided Experimentation,* Wiley, New York, 1975.
7. Garrett P.H., *Analog Systems for Microprocessors and Minicomputers,* Reston Publishing Co., Reston, VA., 1978.
8. Hnatek E.R., *A User's Handbook of D/A and A/D Converters,* Wiley, New York, 1976.
9. Mrozowski A., « Analog Output Chips Shrink A-D Conversion Software », *Electronics,* 23 juin 1977, pp. 130-133.
10. *The Optoelectronics Data Book,* Texas Instruments, Inc., P.O. Box 5012, Dallas, TX., 1978.
11. *The Optoelectronic Designer's Catalog,* Hewlett-Packard Inc., 1820 Embarcadero Road, Palo Alto, CA. 94303, 1978.
12. Peatman J.B., *Microcomputer-based Design,* McGraw-Hill, New York, 1977.
13. Rony P.R. et al., « Microcomputer Interfacing : Sample and Hold Devices », *Computer Design,* décembre 1977, pp. 106-108.
14. Sheingold D.H. ed., *Analog-Digital Conversion Notes,* Analog Devices, Inc., P.O. Box 796, Norwood, MA. 02062, 1977.

TABLE DES MATIÈRES

Chapitre 1. — **Introduction au langage de programmation assembleur** **5**
Comment lire ce livre 5
Signification des instructions 5
Un programme informatique 6
Le problème de la programmation 7
Utilisation de l'octal ou de l'hexadécimal 7
Mnémoniques des codes d'instructions 9
Le programme assembleur 10
Autres caractéristiques des assembleurs 11
Inconvénients du langage assembleur 12
Langage haut niveau 13
Avantages des langages haut niveau 13
Inconvénients des langages haut niveau 14
Langages haut niveau pour microprocesseurs 16
Quel niveau utiliser ? 17
Et pour le futur ? 18
Pourquoi ce livre ? 18
Références 19

Chapitre 2. — **Les assembleurs** **20**
Caractéristiques des assembleurs 20
Instructions de l'assembleur 20
Labels 21
Codes opérations (mnémoniques) de l'assembleur 23
Pseudo-opérations 24
Pseudo-opérations sur données (DATA) 24
Pseudo-opérations Equate (ou Define) 25
Pseudo-opération origine 27
Pseudo-opération réserve 28
Pseudo-opérations de référence externe 29
Pseudo-opérations d'intendance 29
Labels avec pseudo-opérations 29
Champs opérandes et adresses 30
Assemblage conditionnel 32
Macros 32
Commentaires 34
Types d'assembleurs 35
Erreurs 36
Chargeurs 37
Références 38

Chapitre 3. — **Jeu d'instructions du Z80 langage assembleur** **39**
Les registres et les indicateurs (flags) 40
Les modes d'adressage du Z80 42
Les instructions en assembleur (mnémoniques) 57
Les instructions en code objet 57
Les temps d'exécution 57
Les indicateurs 58

ADC A,data — Addition de la retenue et d'une valeur immédiate à l'accumulateur 78
ADC A,reg — Addition de la retenue et d'un registre à accumulateur 79
ADC A,(HL) — Addition de la retenue et d'une donnée en mémoire à l'accumulateur 80
ADC A,(IX + disp) 80
ADC A,(IY + disp) 80
ADC HL,rp — Addition d'une paire de registres et de la retenue AH et L 82
ADD A,data — Addition d'une valeur immédiate à l'accumulateur 83
ADD A,reg — Addition du contenu d'un registre à l'accumulateur 84
ADD A,(HL) — Addition d'une donnée en mémoire à l'accumulateur 85
ADD HL,rp — Addition d'une paire de registres A H et L 86
ADD xy,rp — Addition d'une paire de registres à un registre d'index 87
AND data — ET logique d'une valeur immédiate avec l'accumulateur 88
AND reg — ET logique d'un registre avec l'accumulateur 89
AND (HL) — ET logique d'une donnée en mémoire avec l'accumulateur 90
AND (IX + disp) 90
BIT b,reg — Test du bit b dans le registre reg 91
BIT b,(HL) — Test du bit de la position mémoire indiquée 92
BIT b,(IX + disp) 92
BIT b,(IY + disp) 92
CALL label — Appel du sous-programme identifié par l'étiquette 93
CALL condition, label — Appel du sous-programme identifié par l'étiquette si la condition est satisfaite 94
CCF — Complémentation de l'indicateur de retenue (« Complement carry flag ») 95
CP data — Comparaison d'une valeur immédiate avec l'accumulateur 96
CP reg — Comparaison d'un registre avec l'accumulateur 97
CP (HL) — Comparaison d'une donnée en mémoire avec l'accumulateur 98
CP (IX + disp) 98
CP (IY + disp) 98
CPD — Comparaison de l'accumulateur avec la mémoire décrémentation de l'adresse et du compteur 99
CPDR — Comparaison de l'accumulateur avec la mémoire décrémentation de l'adresse et du compteur répétition jusqu'à égalité ou compteur nul 100
CPI — Comparaison de l'accumulateur avec la mémoire décrémentation du compteur incrémentation de l'adresse 101
CPIR — Comparaison de l'accumulateur avec la mémoire décrémentation du compteur incrémentation de l'adresse répétition jusqu'à égalité ou compteur nul 102
CPL — Complémentation de l'accumulateur 102
DAA — Ajustement décimal de l'accumulateur (BCD) 103
DEC reg — Décrémentation du contenu d'un registre 104
DEC rp — Décrémentation d'une paire de registre spécifiée 105
DEC IX 105
DEC IY 105
DEC (HL) — Décrémentation d'un contenu mémoire 106
DEC (IX + disp) 106
DEC (IY + disp) 106
DI — Masquage des interruptions (« disable interrupts ») 107
DJNZ disp — Décrémentation du registre et saut relatif si résultat non nul 108
EI — Autorisation des interruptions (« Enable interrupts ») 108
EX AF,AF' — Echange de l'accumulateur et du registre d'état avec leurs équivalents auxiliaires 110

EX DE,HL — Echange des contenus des doubles registres DE et HL 110
EX (SP),HL — Echange du contenu d'un registre avec le sommet de pile 111
EX (SP),IX 111
EX (SP),IY 111
EXX — Echange des paires de registres avec les paires des registres auxiliaires 112
HALT 113
IM 0 — Mode d'interruption 0 114
IM 1 — Mode d'interruption 1 114
IM 2 — Mode d'interruption 2 114
IN A,(port) — Entrée vers l'accumulateur 115
INC reg — Incrémentation du contenu d'un registre 116
INC rp — Incrémentation du contenu de la paire de registres spécifiée 117
INC IX 117
INC IY 117
INC (HL) — Incrémentation d'un contenu mémoire 118
INC (IX + disp) 118
INC (IY + disp) 118
IND — Entrée en mémoire et décrémentation du pointeur 119
INDR — Entrée en mémoire et décrémentation du pointeur tant que le compteur est non nul 120
INI — Entrée en mémoire et incrémentation du pointeur 121
INIR — Entrée en mémoire et incrémentation du pointeur tant que le compteur est non nul 121
IN reg,(C) — Entrée vers un registre 122
JP label — Saut à l'instruction identifiée par l'étiquette 123
JP condition, label — Saut à l'adresse identifiée par l'étiquette si la condition est satisfaite 124
JP (HL) — Saut à l'adresse spécifiée par le contenu d'un registre 16 bits 125
JP (IX) 125
JP (IY) 125
JR C,disp — Saut relatif si l'indicateur de retenue = 1 126
JR disp — Saut relatif au contenu du compteur ordinal 127
JR NC,disp — Saut relatif si l'indicateur de retenue est nul 128
JR NZ,disp — Saut relatif si l'indicateur de zéro est nul 128
JR Z,disp — Saut relatif si l'indicateur de zéro = 1 129
LD A,I — Transfert du contenu du registre d'interruptions ou du registre de rafraîchissement vers l'accumulateur 129
LD A,R 129
LD A,(addr) — Chargement de l'accumulateur à partir de la mémoire à l'aide de l'adressage direct 130
LD A,(rp) — Chargement de l'accumulateur à partir de l'emplacement mémoire adresse par un double registre 131
LD dst,src — Transfert du contenu d'un registre source vers un registre destination 132
LD HL,(addr) — Chargement d'une paire de registres ou d'un registre d'index à partir de la mémoire à l'aide de l'adressage direct 133
LD rp,(addr) 133
LD IX,(addr) 133
LD IY,(addr) 133
LD I,A — Chargement du registre d'interruption ou du registre de rafraîchissement à partir de l'accumulateur 134
LD R,A 134
LD reg,data — Chargement immédiat dans un registre 135
LD rp,data — Chargement immédiat d'une donnée 16 bits dans un registre 136

LD IX,data 136
LD IY,data 136
LD reg,(HL) — Chargement d'un registre à partir de la mémoire 137
LD reg,(IX + disp) 137
LD reg,(IY + disp) 137
LD SP,HL — Transfert du contenu de HL ou d'un registre d'index vers le pointeur de pile 138
LD SP,IX 138
LD SP,N 138
LD (addr),A — Rangement de l'accumulateur en mémoire par adressage direct 139
LD (addr),HL — Rangement en mémoire du contenu d'un double registre ou d'un registre d'index par adressage direct 140
LD (addr),rp 140
LD (addr),xy 140
LD (HL),data — Chargement d'une valeur immédiate en mémoire 142
LD (IX + disp),data 142
LD (IY + disp),data 142
LD (HL),reg — Chargement en mémoire à partir d'un registre 143
LD (IX + disp),reg 143
LD (IY + disp),reg 143
LD (rp),A — Chargement de l'accumulateur à l'emplacement mémoire adressé par un double registre 144
LDD — Transfert de données mémoire à mémoire décrémentation des adresses source et destination 145
LDDR — Transfert de données mémoire à mémoire tant que le compteur est non nul décrémentation des adresses source et destination 146
LDI — Transfert de données mémoire à mémoire incrémentation des adresses source et destination 147
LDIR — Transfert de données mémoire à mémoire tant que le compteur est non nul, incrémentation des adresses source et destination 147
NEG — Changement de signe du contenu de l'accumulateur (« negate ») 148
NOP — Pas d'opération (« No operation ») 149
OR data — OU logique avec l'accumulateur 149
OR reg — OU logique d'un registre avec l'accumulateur 150
OR (HL) — OU logique de la mémoire avec l'accumulateur 151
OR (IX + disp) 151
OR (IY + disp) 151
OUT (C),reg — Sortie à partir d'un registre 152
OUTD — Sortie à partir de la mémoire décrémentation de l'adresse 153
OTDR — Sortie à partir de la mémoire, décrémentation de l'adresse tant que le registre B est non nul 154
OUTI — Sortie à partir de la mémoire, incrémentation de l'adresse 155
OTIR — Sortie à partir de la mémoire, incrémentation de l'adresse tant que le registre B est non nul 156
OUT (port),A — Sortie à partir de l'accumulateur 156
POP rp — Lecture du sommet de pile 157
POP IX 157
POP IY 157
PUSH rp — Ecriture en sommet de pile 158
PUSH IX 158
PUSH IY 158
RES b,reg — Mise à zéro de registres (« Reset ») 159
RES b,(HL) — Mise à zéro d'un bit en mémoire (« Reset ») 160

RES b,(IX + disp) 160
RES b,(IY + disp) 160
RET — Retour de sous-programme 162
RET cond — Retour de sous-programme si la condition est satisfaite 162
RETI — Retour après interruption 163
RETN — Retour d'une interruption non masquable 164
RL reg — Rotation à gauche du contenu d'un registre à travers la retenue 165
RL (HL) — Rotation à gauche à travers la retenue du contenu d'une adresse 166
RL (IX + disp) 166
RL (IY + disp) 166
RLA — Rotation à gauche à travers la retenue du contenu de l'accumulateur 167
RLC reg — Rotation circulaire à gauche du contenu d'un registre 168
RLC (HL) — Rotation circulaire à gauche du contenu d'une adresse 169
RLC (IX + disp) 169
RLC (IY + disp) 169
RLCA — Rotation circulaire à gauche de l'accumulateur 170
RLD — Rotation à gauche d'un chiffre BCD entre l'accumulateur et un emplacement mémoire 171
RR reg — Rotation à droite d'un registre, à travers la retenue 172
RR (HL) — Rotation à droite d'un octet mémoire à travers la retenue 173
RR (IX + disp) 173
RR (IY + disp) 173
RRA — Rotation à droite de l'accumulateur à travers la retenue 174
RRC — Rotation circulaire à droite du contenu d'un registre 175
RRC (HL) — Rotation circulaire à droite du contenu d'un octet mémoire 176
RRC (IX + disp) 176
RRC (IY + disp) 176
RRCA — Rotation circulaire à droite de l'accumulateur 177
RRCA — Rotation à droite d'un chiffre BCD entre l'accumulateur et un emplacement mémoire 178
RST n — Instruction de relance (« Restart ») 179
SBC A,data — Soustraction avec retenue d'une donnée à partir de l'accumulateur 180
SBC A,reg — Soustraction avec retenue du contenu d'un registre à partir de l'accumulateur 181
SBC A,(HL) — Soustraction avec retenue d'un octet mémoire à partir de l'accumulateur 182
SBC A,(IX + disp) 182
SBC A,(IY + disp) 182
SBC HL,rp — Soustraction avec retenue du contenu d'une paire de registres à partir de H et de L 183
SCF — Positionnement à 1 de l'indicateur de retenue (« Set carry flag ») 184
SET b,reg — Positionnement à 1 d'un bit de registre 185
SET b,(HL) — Positionnement à 1 du bit b de l'octet mémoire indiqué 186
SET b,(IX + disp) 186
SET b,(IY + disp) 186
SLA reg — Décalage arithmétique à gauche du contenu d'un registre 187
SLA (HL) — Décalage arithmétique à gauche d'un octet mémoire 188
SLA (IX + disp) 188
SLA (IY + disp) 188
SRA reg — Décalage arithmétique à droite du contenu d'un registre 190
SRA (HL) — Décalage arithmétique à droite d'un octet mémoire 191

SRA (IX + disp) 191
SRA (IY + disp) 191
SRL reg — Décalage logique à droite du contenu d'un registre 192
SRL (HL) — Décalage logique à droite d'un octet mémoire 193
SRL (IX + disp) 193
SRL (IY + disp) 193
SUB data — Soustraction d'une donnée à partir de l'accumulateur 194
SUB reg — Soustraction du contenu d'un registre à partir de l'accumulateur 195
SUB (HL) — Soustraction du contenu d'un octet mémoire à partir de l'accumulateur 196
SUB (IX + disp) 196
SYB (IY + disp) 196
XOR data — OU exclusif d'une donnée avec l'accumulateur 197
XOR reg — OU exclusif entre un registre et l'accumulateur 198
XOR (HL) — OU exclusif entre la mémoire et l'accumulateur 199
XOR (IX + disp) 199
XOR (IY + disp) 199
Compatibilité 8080A/Z80 200
Conventions de l'assembleur du Z80 ZILOG 205
Structure de l'assembleur 206
Etiquettes 207
Noms réservés 207
Pseudo-opérations 207
Etiquettes des pseudo-opérations 209
Adresses 209
Assemblage conditionnel 211

Chapitre 4. — **Programmes simples** **212**
Format général des exemples 212
Directives pour les exercices 213
Exemples de programmes 214
Complémentation à un 214
Addition 8 bits 216
Décalage de 1 bit à gauche 218
Masquage de bits 218
Remise à zéro d'un emplacement mémoire 219
Déassemblage d'un octet 220
Recherche du plus grand de deux nombres 222
Addition sur 16 bits 223
Table des carrés 225
Complémentation à un sur 16 bits 228
Exercices 229
Complémentation à deux 229
Soustraction 8 bits 229
Décalage à gauche de deux bits 229
Masquage des quatre bits de poids faible 229
Mise à un d'un emplacement mémoire 230
Assemblage d'un octet 230
Addition sur 24 bits 230
Somme de carrés 231
Complémentation à deux sur 16 bits 213

Chapitre 5. — **Les boucles simples** **232**
Exemples 234
Somme de données 234
Somme de données sur 16 bits 237
Nombre d'éléments négatifs 239
Recherche du maximum 243
Cadrage binaire 245
Exercices 248
Octet de contrôle 248
Somme de données sur 16 bits 248
Nombre d'éléments nuls, positifs et négatifs 249
Recherche du minimum 249
Comptage des bits à 1 250

Chapitre 6. — **Traitement des caractères** **251**
Exemples 252
Longueur d'une chaîne de caractères 252
Remplacement des zéros de début de chaîne par des blancs 261
Addition d'une parité paire aux caractères ASCII 263
Egalité de séquences 266
Exercices 270
Longueur d'un message d'une téléimprimante 270
Recherche du dernier caractère non blanc 270
Troncature d'une chaîne décimale sous forme entière 270
Contrôle de la parité paire dans les caractères ASCII 271
Comparaison de chaînes 272

Chapitre 7. — **Conversion de code** **273**
Exemples 270
Hexa-ASCII 274
Décimal-Affichage 7-segments 276
ASCII-Décimal 281
DCB (décimal codé binaire) - Binaire 283
Conversion d'un nombre binaire en une chaîne ASCII 284
Exercices 286
Conversion d'ASCII en hexa 286
Conversion du code 7-segments en décimal 287
Conversion de décimal en ASCII 287
Conversion de binaire en DCB (décimal codé binaire) 287
Conversion d'une chaîne ASCII en un nombre binaire 288
Références 288

Chapitre 8. — **Problèmes arithmétiques** **289**
Exemples 289
Addition en multiple précision 289
Transfert de blocs 292
Addition décimale 293
Multiplication binaire 8 bits 296
Division binaire sur 8 bits 301
Chiffres de contrôle 305
Méthode de doublage Modulo 10 305
Exercices 313
Soustraction en multiple précision 313
Soustraction décimale 314
Multiplication binaire 8 bits par 16 bits 314

Division binaire signée 315
Chiffres de contrôle - Méthode de l'alignement 1, 3, 7 Modulo 10 315
Références 316

Chapitre 9. — **Tables et listes** **317**
Exemples 317
Ajout d'un élément à une liste 317
Vérification d'une liste ordonnée 321
Retrait d'un élément d'une file 324
Tri sur 8 bits 326
Emploi d'une table de saut ordonnée 329
Retrait d'un élément d'une liste 331
Ajout d'un élément à une file 332
Tri sur 16 bits 333
Emploi d'une table de saut avec clé 333

Chapitre 10. — **Sous-programmes** **335**
Documentation des sous-programmes 337
Exemples 337
Conversion hexa-ASCII 337
Longueur d'une chaîne de caractères 341
Calcul de parité sur les caractères ASCII 344
Egalité de chaînes 347
Addition à multiples précision 350
Exercices 353
Conversion ASCII-Hexa 354
Longueur d'un message ASCII 354
Contrôle de la parité paire dans les caractères ASCII 354
Comparaison de chaînes 355
Soustraction décimale 356
Références 356

Chapitre 11. **Entrées/Sorties** **357**
Temporisations 357
Programme de temporisation utilisant l'accumulateur 365
Evaluation du temps 366
Circuits d'entrée/sortie parallèle du Z80 (PIO) 367
Contrôle du mode de fonctionnement du PIO 370
Configuration d'un PIO 373
Instructions d'entrées/sortie du Z80 374
Exemples 377
Exemple de programmation 378
Un commutateur (interrupteur) 383
Exemple de programmation 384
Un commutateur à multiple position (zone codeuse, sélecteur) 387
Exemple de programmation 388
Une simple diode électroluminescente (LED) 393
Affichage sept-segments à diode 395
Exercices 404
Un bouton-poussoir (marche-arrêt) 404
Gestion des antirebonds par logiciel 405
Contrôle d'un commutateur à multiples positions 405
Affichage de la position des interrupteurs 405
Comptage sur un afficheur sept-segments 405
Circuits d'E/S plus complexes 406

Exemples ... 409
Un clavier non encodé ... 409
Un clavier encodé ... 417
Convertisseur digital-analogique ... 420
Convertisseur analogique-digital ... 424
Un télé-imprimeur (TTY) ... 429
Le circuit d'entrée-sortie série du Z80 (S10) ... 436
Exemples ... 443
E/S d'un télé-imprimeur via un UART ... 443
Interfaces standard ... 446
Exercices ... 447
Séparation des touches sur un clavier non encodé ... 447
Lecture d'une phrase à partir d'un clavier encodé ... 447
Un générateur de signal carré d'amplitude variable ... 447
Moyenne de lectures analogiques ... 448
Un terminal à 30 caractères par seconde ... 448
Bibliographie et références ... 449

Chapitre 12. — **Interruptions** ... **450**
Système d'interruptions du Z80 ... 452
Interruption non masquable ... 453
Modes d'interruption du Z80 ... 454
Mode 1 ... 454
Mode 2 ... 455
Compatibilité d'interruption Z80/8080 ... 455
Interruptions d'un SIO ... 460
Exemples d'interruptions ... 462
Interruption de démarrage ... 462
Interruption du clavier ... 465
Interruption d'imprimante ... 468
Interruption d'horloge temps réel ... 471
Attente de l'interruption d'une horloge temps réel ... 472
Attente de 10 interruptions de l'horloge temps réel ... 473
Interruption d'un télé-imprimeur ... 477
Avec un PIO ... 479
Programmes de traitements plus généraux ... 481
Exercices ... 482
Interruption de test ... 482
Interruption de clavier ... 482
Interruption d'imprimante ... 482
Interruption d'horloge temps réel ... 483
Interruption d'un télé-imprimeur ... 483
Références ... 483

Chapitre 13. — **Définition des problèmes et conception des programmes** ... **484**
Fonctions du logiciel de développement ... 484
Définition des étapes ... 486
Définition des problèmes ... 487
Définition des entrées ... 487
Définition des sorties ... 488
Section de traitement ... 488
Traitement des erreurs ... 489
Facteurs humains ... 489
Exemples ... 490
Réponse à un interrupteur ... 490

Chargeur mémoire à interrupteurs ... 492
Terminal de vérification ... 495
En résumé ... 499
Conception du programme ... 499
Les organigrammes ... 500
Exemples ... 502
Réponse à un interrupteur ... 502
Le chargeur de mémoire à interrupteurs ... 503
Terminal de vérification de cartes de crédit ... 504
Programmation modulaire ... 508
Exemples ... 509
Réponse à un interrupteur ... 509
Le chargeur mémoire à interrupteurs ... 510
Le terminal de vérification ... 510
Résumé de la programmation modulaire ... 511
Programmation structurée ... 512
Une séquence ordinaire ... 512
Une structure conditionnelle ... 513
Une structure de boucle ... 514
Une structure de cas ... 515
Exemples ... 518
Réponse à un interrupteur ... 518
Le chargeur à interrupteurs ... 519
Terminal de vérification des cartes de crédit ... 520
Résumé de la programmation structurée ... 525
Conception descendante ... 526
Exemples ... 527
Réponse à un interrupteur ... 527
Le chargeur à interrupteurs ... 528
Le terminal de transaction ... 529
Résumé de la conception descendante ... 531
Résumé de la définition des problèmes et de la conception des programmes ... 532
Références ... 533

Chapitre 14. — **Mise au point et test** ... **534**
Outils simples de mise au point ... 534
Outils de mise au point plus évolués ... 541
Débugage avec listes de test ... 544
Recherche des erreurs ... 545
Exemple de débugage : Conversion décimale à 7-segments ... 549
Exemple de débugage : Tri par ordre décroissant ... 553
Introduction au test ... 559
Aides au test ... 559
Sélection des données de test ... 560
Exemple de test : programme de tri ... 561
Exemple de test : nombres d'auto-test ... 562
Précautions lors des tests ... 562
Conclusions ... 562
Références ... 563

Chapitre 15. — **Documentation et optimisation** ... **565**
Programmes auto-documentés ... 565
Commentaires ... 566
Exemples de commentaires : addition de nombres sur plusieurs octets ... 568

Exemple de commentaires : sortie de télé-imprimante 570
Organigramme documentaire 571
Programmation structurée documentaire 572
Carte de la mémoire (« MEMORY MAP ») 572
Listes de paramètres et de définitions 573
Bibliothèque de programmes 574
Exemple de fiches de programmes 575
Exemple de fiches : sommes décimales 576
Synthèse de la documentation 578
Optimisation 579
Ré-organisation pour consommer moins de mémoire 580
Ré-organisation économiser le temps d'exécution 581
Ré-organisation majeures 582
Références 583

Chapitre 16. — **Exemples de projets** **584**
Projet n° 1 : un minuteur 584
Description du programme 586
Commentaires d'introduction 587
Définition des variables 587
Initialisation 587
Attente de la frappe d'une touche 588
L'anti-rebonds 588
Identification d'une fermeture de clé 588
Action sur identification de touche 590
Etablissement d'affichage 590
Activation des afficheurs LED 590
Décrémentation du compteur d'affichage 590
Projet n° 2 : un thermomètre numérique 597
Description du programme 600
Initialisation 600
Lecture du convertisseur 601
Conversion en degrés Celsius 601
Préparation des données pour l'affichage 602
Affichage de la température pendant 6 secondes 603
Références 610

Table des matières 611

Service lecteurs

(à retourner à S.E.C.F.-Éditions Radio, 9, rue Jacob, 75006 Paris)

Pour nous permettre de vous proposer des ouvrages toujours meilleurs, nous souhaiterions recevoir vos critiques, appréciations et suggestions sur le présent livre :

_____

_____

_____

_____

Quels sont les ouvrages (thème, sujet, niveau) que vous souhaiteriez voir publier par notre société ?

_____

_____

_____

_____

Nous vous remercions de votre confiance et de votre coopération.

S.E.C.F.-Éditions Radio

Je désire recevoir gratuitement et sans engagement (mettre une croix dans la case) :

☐ Votre catalogue général (Electronique professionnelle et grand public, Informatique, Hi-Fi, Vidéo)

☐ Votre catalogue spécial informatique.

Nom : _____ Prénom : _____

Adresse : _____

_____

Secteur d'activité et fonction : _____

_____

CENTRES D'INTÉRÊTS

☐ Electronique professionnelle

☐ Electronique de loisirs

☐ Hifi, Vidéo, CB

☐ Micro-informatique

Matériels : _____

_____

_____

EXTRAIT DU CATALOGUE SECF ÉDITIONS RADIO

INITIATION AUX MICRO-ORDINATEURS NIVEAU 1 par **A. Osborne.** — Pour apprendre à exploiter les micro-ordinateurs sans connaissance particulière préalable.
304 pages, format 16 × 20.

INITIATION AUX MICRO-ORDINATEURS NIVEAU 2 par **A. Osborne.** — Un ouvrage complet pour une connaissance approfondie des microprocesseurs et des micro-ordinateurs.
488 pages, format 16 × 24.

6800 PROGRAMMATION EN LANGAGE ASSEMBLEUR par **L.A. Leventhal.** — Un cours complet de programmation en langage assembleur du microprocesseur 6800.
480 pages, format 16 × 24.

6502 PROGRAMMATION EN LANGAGE ASSEMBLEUR par **L.A. Leventhal.** — Un cours complet de programmation, en langage assembleur du microprocesseur 6502.
480 pages, format 16 × 24.

Z80 PROGRAMMATION EN LANGAGE ASSEMBLEUR par **L.A. Leventhal.** — Un cours complet de programmation en langage assembleur du microprocesseur Z80.
480 pages, format 16 × 24.

8080, 8085 PROGRAMMATION EN LANGAGE ASSEMBLEUR par **L.A. Leventhal.** — Un cours complet de programmation, en langage assembleur des microprocesseurs 8080 et 8085.
480 pages, format 16 × 24.

DU MICROPROCESSEUR AU MICRO-ORDINATEUR par **H. Lilen.** — Le point des connaissances actuelles sur les microprocesseurs.
352 pages, format 16 × 24.

PRATIQUE DU CP/M par **H. Lilen.** — Didactique et très progressif, ce livre met le CP/M à la portée de tous.
192 pages, format 21 × 29,7.

70 PROGRAMMES BASIC pour... — Une série d'ouvrages adaptés chacun à un micro-ordinateur de grande diffusion.

S. E. C. F.

Éditions Radio

9, RUE JACOB - 75006 PARIS
TÉL. 329.63.70

Z 80
PROGRAMMATION EN LANGAGE ASSEMBLEUR

Ce livre constitue un cours complet de programmation, en langage assembleur, du microprocesseur Z80. Chacune des instructions est présentée séparément et commentée dans tous ses détails. De nombreux programmes rédigés à la fois en assembleur et en langage machine, développent les applications.

Extrait de la table des matières : Introduction en langage de programmation en assembleur · Les assembleurs · Le jeu d'instructions du Z80 en langage assembleur · Programmes simples · Programmes simples avec boucles · Le codage des caractères · Conversion de codes · Problèmes arithmétiques · Tables et listes · Sous-programmes · Entrées-sorties · Interruptions · Définition des problèmes et conception des programmes · Débugage et test · Documentation et reconception · Exemples de projets.

9 782709 109093
ISBN 2 7091 0909 3
42
09 0265 0

S. E. C. F.
ÉDITIONS RADIO

Prix : 215 F

Z 80 PROGRAMMATION EN LANGAGE ASSEMBLEUR

LANCE A. LEVENTHAL