J.-D. Nicoud

COMPRENDRE LES MICROPROCESSEURS

TABLE DES MATIERES

Juin 1977, deuxième édition Janvier 1978

COMPRENDRE LES MICROPROCESSEURS

Première partie: INTRODUCTION AU SYSTÈME DAUPHIN

1. INTRODUCTION

L'objectif de ce fascicule est d'apporter des explications simples et correctes sur tous les aspects des microprocesseurs, afin de permettre un démarrage à tous ceux qui trouvent la documentation originale des fabricants indigeste.

Les exemples illustreront la programmation du système DAUPHIN et de son processeur Signetics 2650, mais grâce aux notations employées, différentes de celles de Signetics, ce sont en fait les concepts fondamentaux des microprocesseurs qui apparaîtront dans ces exemples. Nous nous efforcerons d'utiliser les termes techniques français, en mentionnant leur équivalent anglais entre deux barres obliques. La connaissance de l'anglais est indispensable dans ce domaine si l'on veut avoir accès aux notices des fabricants.

2. SYSTEME MICROPROCESSEUR

Il faut bien distinguer un microprocesseur et un système à microprocesseur. Le microprocesseur se réduit généralement à une unité arithmétique et de contrôle; sa structure sera étudiée plus loin. Un système microprocesseur comporte, en plus du processeur, de la mémoire et des interfaces d'entrée/sortie, selon l'application visée.

La mémoire contient le programme et les données; elle peut être à lecture et écriture(mémoire vive) */RAM: Random Access Memory/* ou à lecture seulement (mémoire morte) */ROM: Read Only Memory/*. Rappelons que la mémoire est un ensemble de cases numérotées. Le numéro est appelé adresse /address/ et le contenu est appelé mot, donnée, contenu, information, code */data/*. Le contenu est un mot binaire de 8 bits en général.

Il est agréable d'avoir le programme dans une mémoire morte (nous dirons souvent: en RDM) pour éviter de la recharger à chaque enclenchement de l'appareil.Les données sur lesquelles le programme agit doivent par contre être en RAM pour que le programme puisse les modifier.

Le système simple de la Fig. 1 contrôle un haut-parleur ou une lampe. La mémoire est de type ROM ou RAM; elle contient le programme. L'interface qui contrôle le haut-parleur est équivalent à une mémoire n'ayant qu'une adresse et dans laquelle on peut écrire seulement un bit. Si ce bit vaut Ø, le haut-parleur est au repos (lampe éteinte). Si ce bit vaut 1, la membrane du haut-parleur est attirée (lampe allumée). Pour le haut-parleur, c'est donc un changement de l'état Ø à 1 ou de 1 à Ø qui va créer le bruit.

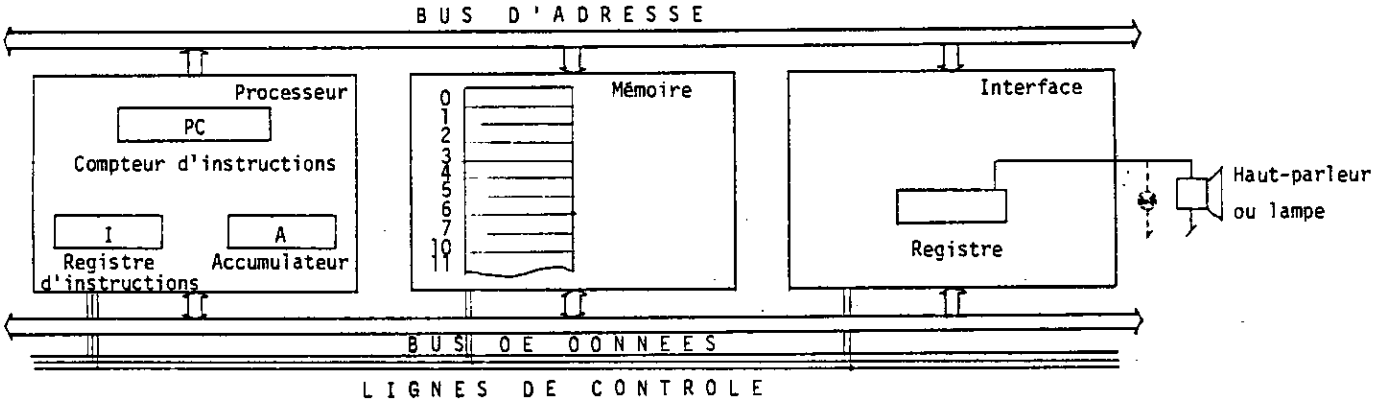


Fig. 1. Système microprocesseur contrôlant un haut-parleur.

Le processeur contient au moins trois registres: le compteur de programme */PC: Program Counter/* pointe l'adresse de la position mémoire contenant l'instruction à exécuter. Le registre d'instruction */Instruction Register/* mémorise l'instruction pendant son exécution. Le registre accumulateur */A: Accumulator/* est utilisé pour les transferts et opérations arithmétiques. Il y a en général d'autres registres dans l'unité arithmétique; ils seront expliqués plus tard.

Le processeur, la mémoire et l'interface sont reliés par des lignes de contrôle appelées bus */bus/*. On distingue en général le bus d'adresse, qui transporte les adresses mémoire et adresses de périphériques, le bus de données, qui transporte les contenus des positions mémoire et registres périphériques, et les lignes de contrôle qui synchronisent les transferts. Le détail des signaux de contrôle sera vu plus loin.

3. PROGRAMME

Le programme stocké en mémoire correspond à la succession d'ordres à donner au processeur pour qu'il effectue correctement les transferts et opérations entre ses registres, la mémoire et les périphériques.

Le programme pour faire osciller la membrane de notre haut-parleur (ou clignoter la lampe) est le suivant:

```
        HP=     74      ;valeur des constantes utilisées par le programme
        HPOF=   Ø
        HPON=   1

        .LOC    Ø       ;adresse de début du programme

OSCIL:  LOAD    A,# HPOF
        LOAD    $HP,A   ;membrane relâchée
        LOAD    A,# HPON
        LOAD    $HP,A   ;membrane attirée
        JUMP    OSCIL
```

Les deux premières instructions transfèrent un Ø dans le registre du périphérique HP. Ce Ø doit tout d'abord être chargé dans l'accumulateur (l'instruction qui permet d'écrire directement une valeur quelconque en mémoire ou dans un interface n'existe pas, car le nombre d'instructions est limité dans un microprocesseur).

En analysant de plus près ces deux premières lignes du programme, on voit d'abord l'étiquette OSCIL: c'est un nom donné à la première instruction du programme. Ce nom correspond à une adresse, donc à un nombre, mais l'emplacement exact du programme dans la mémoire ne nous intéresse pas pour l'instant. Il est plus clair de se référer à des noms, à des symboles abrégés qui évoquent la fonction effectuée; on parle d'adressage symbolique.

L'instruction qui suit cette première étiquette symbolique est LOAD A,# HPOF . Ce qui signifie que le registre A du processeur est chargé */load/* par la valeur HPOF donnée dans l'instruction (on parle de valeur immédiate). Le signe # se prononce "valeur" et précise que l'on veut charger une valeur numérique figurant dans l'instruction (valeur immédiate), et non pas le contenu d'une position mémoire dont l'adresse est donnée dans l'instruction. HPOF vaut zéro, comme déclaré initialement. Il pourrait sembler plus simple d'écrire LOAD A,# Ø , mais ceci ne montre pas aussi clairement l'intention du programmeur. De plus, si l'interface change (adjonction d'un amplificateur inverseur, par exemple), c'est une valeur différente qu'il faut mettre dans l'instruction; il est plus simple de corriger la valeur une fois au début du programme plutôt que d'aller chercher dans le programme toutes les instructions qui agissent sur le haut-parleur.

L'instruction suivante LOAD $HP,A charge le périphérique d'adresse HP par le contenu de A. Le signe $ (dollar) se prononce "périphérique" et précise qu'il s'agit d'une adresse de périphérique et non pas d'une adresse mémoire. De nouveau, un nom a été donné à l'adresse du périphérique (adresse symbolique). La valeur

numérique associée à ce nom est déclarée au début du programme. Les deux instructions suivantes sont du même type et mettent la sortie de l'interface à l'état 1 (membrane attirée, lampe allumée).

Le programme se termine par une instruction de saut */jump/* à l'instruction dont l'adresse est donnée sous forme symbolique: OSCIL, dont la valeur est Ø. Lorsque ce programme est exécuté, la fréquence d'oscillation du haut-parleur dépend de la vitesse du processeur. Nous verrons plus loin comment perfectionner ce programme pour rendre la fréquence d'oscillation réglable indépendamment de celle du processeur.

4. EXECUTION DU PROGRAMME

Lorsque le programme est chargé en mémoire (par des interrupteurs de contrôle ou grâce à un programme spécial de chargement), l'exécution peut commencer. Un signal de remise à zéro est nécessaire sur chaque processeur pour l'initialiser et lui faire chercher la première instruction, en général à l'adresse Ø.

Dans ce cas, si notre petit programme est seul en mémoire, il doit commencer à l'adresse Ø, ce qui est précisé par la pseudo-instruction .LOC Ø (instruction pour le programme ou la personne effectuant la traduction en binaire du programme).

La première phase de l'instruction est la recherche */fetch/* de l'instruction proprement dite. Le compteur d'adresse PC place son contenu sur le bus d'adresse et sélectionne le mot correspondant en mémoire. Le contenu est transféré par le bus de donnée dans le registre d'instruction (Fig. 2a). Dans la 2e phase de cette première instruction, les circuits de décodage du processeur reconnaissent qu'il s'agit d'une instruction de charge d'une valeur immédiate dans l'accumulateur et exécutent */execute/* le transfert de la valeur dans A (Fig. 2b). Simultanément, le compteur d'instructions PC est augmenté de 1 (incrémenté). Un nouveau cycle d'instructions ira donc chercher le code de l'instruction à l'adresse 1 (Fig. 2c). Le décodage de cette instruction indique au processeur qu'il s'agit d'un transfert vers un périphérique. L'adresse du périphérique, qui fait partie de l'instruction, est placée sur le bus d'adresse, un signal de contrôle adéquat sélectionne le périphérique et non pas la mémoire, et la valeur contenue dans A est transférée dans le registre du périphérique sélectionné (Fig. 2d). A nouveau le compteur d'adresse de programme est augmenté pour passer à l'instruction suivante.

Les deux instructions qui suivent étant d'un type connu, passons à la 5e instruction, qui est tout d'abord transférée (Fig. 2e) avant d'être reconnue comme un saut inconditionnel. Dans ce cas, l'adresse du saut est transférée dans le compteur d'adresse (PC), au lieu de l'incrémentation habituelle (Fig. 2f). L'instruction suivante sera recherchée à cette adresse.

Les explications ci-dessus ne tiennent pas compte des contraintes imposées par le codage en binaire des instructions, valeurs immédiates, adresses de périphériques et adresses de saut en mémoire. Il a été supposé que chaque instruction correspond à un seul mot mémoire. Dans les microprocesseurs actuels, des mots de 8 bits sont utilisés, cette valeur étant un compromis courant entre les performances, d'autant meilleures que le mot est long, et le prix qui dépend des possibilités d'intégration dans un nombre minimum de circuits intégrés.

Il faut dans ce cas coder les instructions en utilisant un, deux ou trois (quatre dans le Z80) mots mémoire consécutifs. Le programme ci-dessus, codé pour le 2650, a l'allure suivante:

```
                    HP=      74
                    HPOF=    Ø
                    HPON=    1
                    .LOC=    Ø
Adresse  Contenu
   Ø        4       OSCIL:   LOAO     A,# HPOF
   1        Ø
   2       324              LOAO     $HP,A
   3        74
```

4	4	LOAD	A,# HPON
5	1		
6	324	LDAD	$HP,A
7	74		
10	37	JUMP	OSCIL
11	0		
12	0		

On remarque que les adresses sont numérotées en octal et non pas en décimal. De même les contenus sont en octal. Le système utilisé est en fait le binaire. L'octal et l'héxadécimal sont utilisés comme notations condensées du binaire. L'un des avantages de l'octal est que la conversion en binaire est plus rapide, et que les opérations sont plus faciles (10 110 011 = 263 ; 127 + 34 = 163). Les nombres décimaux sont suivis d'un point pour éviter toute confusion (26.=32).

L'exécution du programme pour 2650 se fait selon la même séquence, mais plusieurs transferts (cycles) */cycles/* sont nécessaires pour chaque instruction. Chaque cycle peut lui-même nécessiter plusieurs impulsions d'horloge, le processeur devant passer par une succession d'états */states/* pour effectuer un transfert (sortir l'adresse, sélectionner la mémoire ou le périphérique, effectuer le transfert, incrémenter le PC).

Lorsque le programme est effectué en pas à pas */step/* lors de la mise au point, il peut dont s'agir de trois types de pas à pas: état par état */state step/*, cycle par cycle */cycle step/* ou instruction par instruction */instruction step/*. On peut aussi imaginer des pas plus grands si des circuits de décodage appropriés sont implémentés.

5. SIGNAUX PROCESSEUR

Comme déjà mentionné, le processeur doit recevoir un signal de contrôle pour être initialisé, remis à zéro */reset/*. Il doit ensuite sélectionner la mémoire ou les périphériques en lecture */read/* ou en écriture */write/*. Chaque processeur utilise pour cela des signaux différents, qui se ramènent avec une logique simple aux signaux suivants.

ADMEM (M): Sélection d'une adresse mémoire
ADPER (P): Sélection d'une adresse périphérique
WRITE (W): Indique qu'une écriture est effectuée.

Pour différentes raisons, les signaux de contrôle sont inversés sur le bus et s'appellent ADMEMLOW, WRITELOW et RESETLOW pour montrer qu'ils sont actifs à l'état zéro, bas */low/*. Par exemple, pour une lecture en mémoire, il faut WRITE = Ø, la lampe correspondante est éteinte, mais sur le bus WRITELOW = 1. La notion de lecture et d'écriture est toujours relative au processeur. Les autres signaux et la structure de l'interface avec le 2650 seront vus plus tard.

6. SIGNAUX MEMOIRE

Une mémoire est sélectionnée par le signal ADMEM; si WRITE = Ø, le contenu de la mémoire d'adresse sélectionnée est lu */read/* et apparaît sur le bus de données. Si WRITE = 1, l'état du bus de données est transféré dans la position mémoire dont l'adresse est sur le bus d'adresse. L'adresse doit naturellement correspondre à une adresse existante. La dimension de la mémoire est limitée et n'occupe qu'une partie des adresses possibles. Le schéma détaillé d'une mémoire de 128. mots de 8. bits compatible avec le DAUPHIN est donné dans la Fig. 3. Cette mémoire répond aux adresses Ø à 177 (octal): il est facile de vérifier que lorsque les adresses 2^7 à 2^{11} sont à Ø, la porte ET de sélection de circuit */CS: Chip Select/* est à l'état 1.

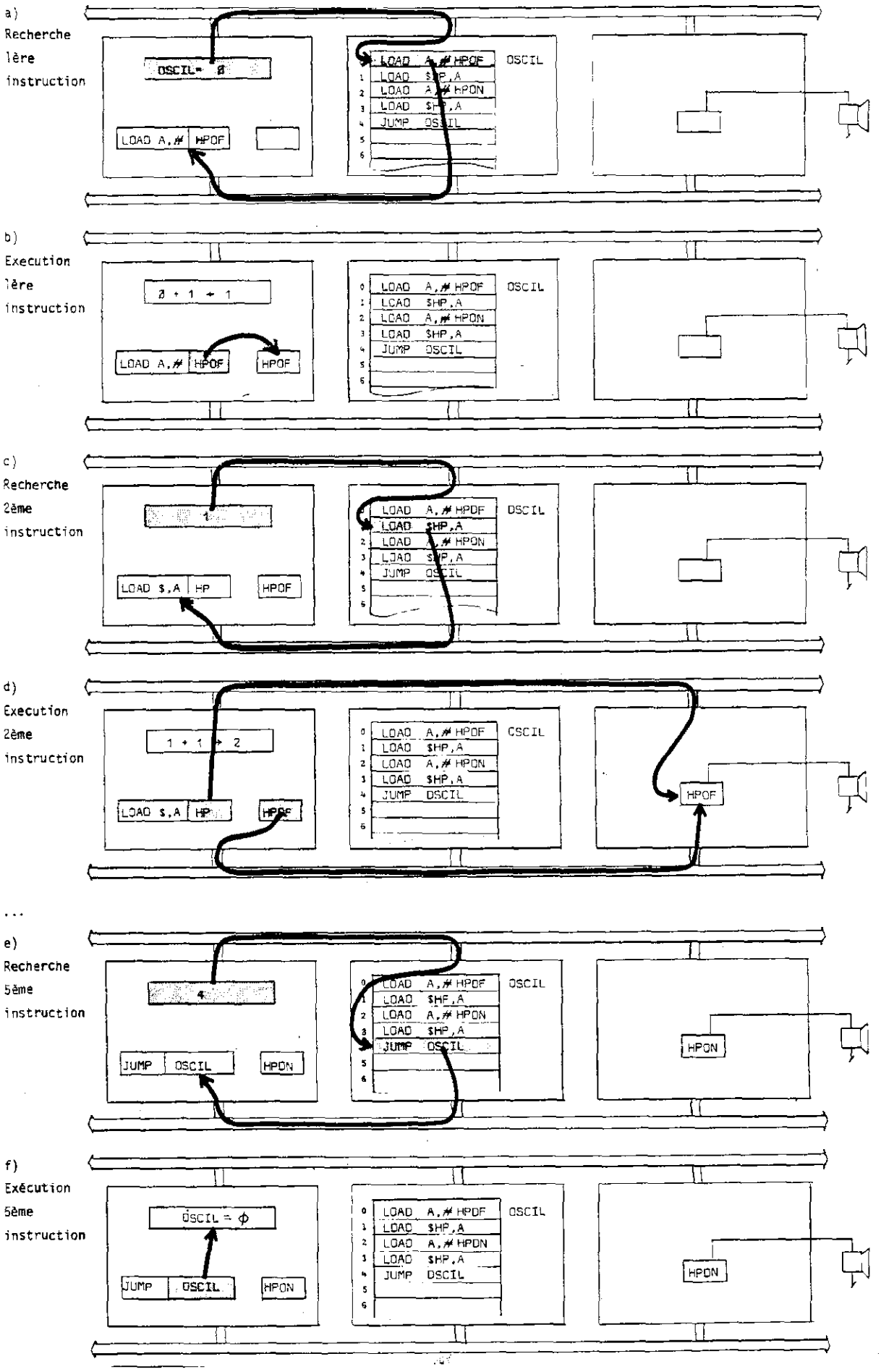


Fig. 2. Séquences de l'exécution du programme du paragraphe 4.

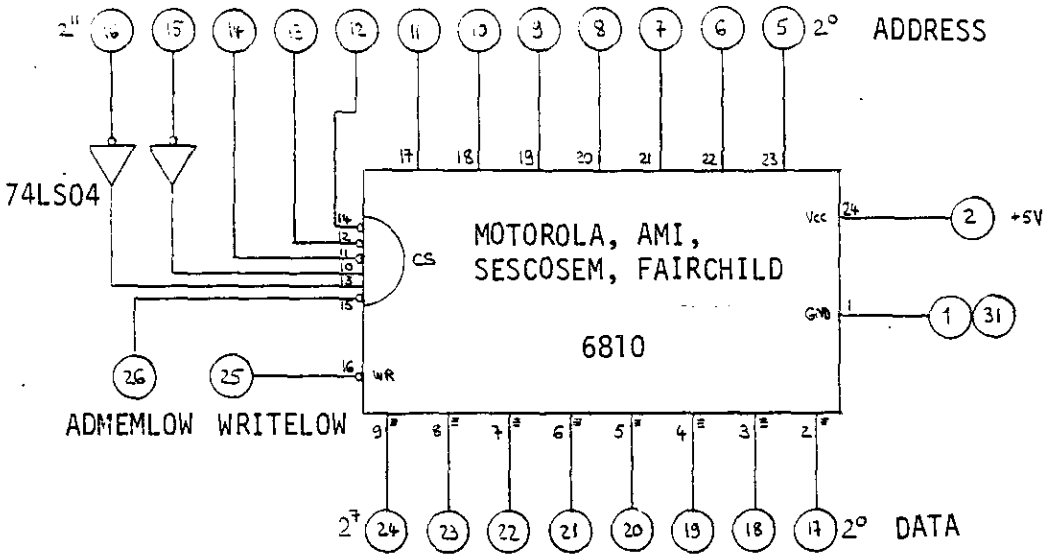


Fig. 3. Schéma détaillé d'une mémoire 6810.

7. SIGNAUX ET INTERFACES

Les périphériques sont sélectionnés par ADPER, en lecture ou en écriture selon la valeur de WRITE. Il existe des circuit spéciaux interface périphérique, mais pour les applications simples, quelques circuits TTL de la série LS */Low Power Schottky/* résolvent le problème à meilleures conditions. Par exemple l'interface haut-parleur compatible avec le programme précédent est donné dans la Fig. 4. Le décodeur d'adresse utilise un circuit 74LS138 et décode les adresses 64, 65 66, 67 (octal naturellement) en lecture et en écriture. Seuls 6 bits d'adresse sont décodés, permettant ainsi un maximum de 64. périphériques, d'adresses 0 à 77 . Si le périphérique 64 est sélectionné en écriture, le flip-flop reçoit à la fin de l'impulsion ADPER un flanc montant actif qui mémorise dans la bascule l'état de la ligne du bus correspondante (ligne de poids faible, donc valeur 1 ou Ø du mot binaire transféré par le bus).

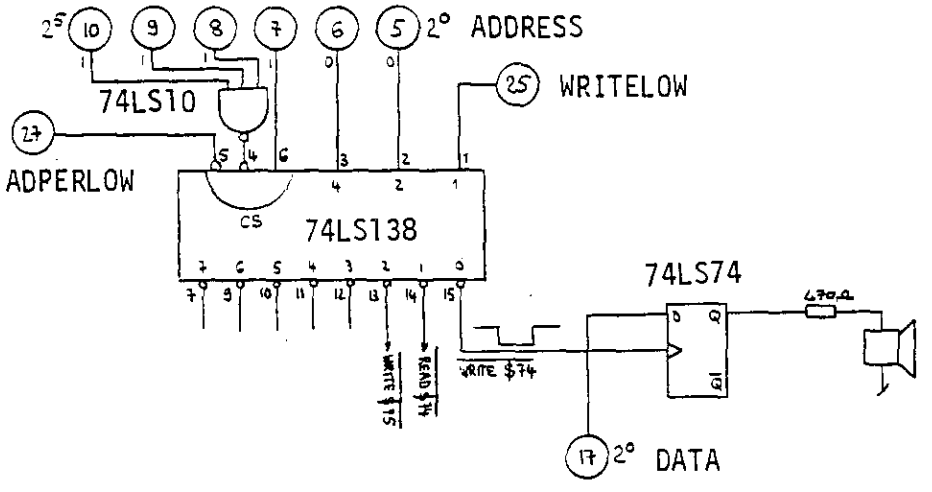


Fig. 4. Interface haut-parleur compatible avec le programme du § 4.

8. PROGRAMME ET INTERFACES SIMPLIFIES

Le même problème peut se résoudre d'innombrables façons, tant du point de vue programmation */software/* que du point de vue matériel */hardware/*. Par exemple, le programme initialement donné peut s'écrire

```
OSCIL:  CLR     A       ;met à zéro /clear/ A
        LOAD    $HP,A
        INC     A       ;ajoute 1 /increment/ à A
        LOAD    $HP,A
        JUMP    OSCIL
```

Le programme prend moins de place en mémoire car les instructions CLR et INC n'ont pas de valeur numérique associée et sont plus courtes (8 bits). Ces deux instructions n'existent toutefois pas sous cette forme sur le 2650.

L'inconvénient de ce nouveau programme est que le flip-flop de l'interface haut-parleur ne peut être branché que sur le bit de poids faible.

Une simplification plus grande du programme peut être obtenue avec l'interface de la Fig. 5, utilisé dans le DAUPHIN.

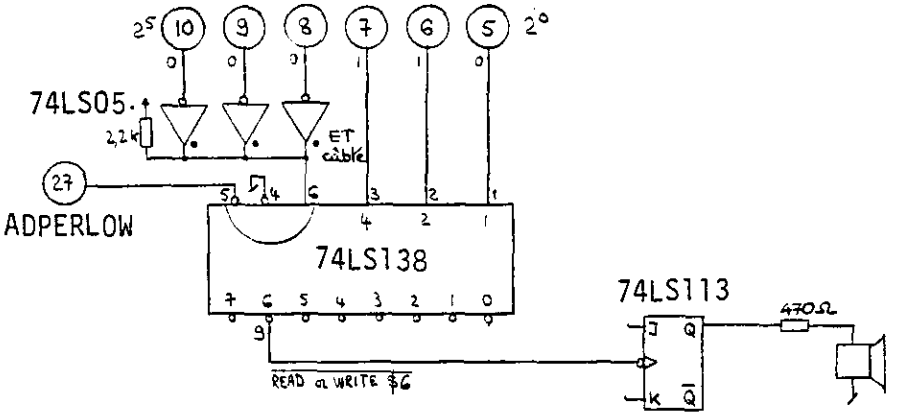


Fig. 5. Interface haut-parleur implémenté dans le DAUPHIN.

La sélection du périphérique 6 fait basculer un diviseur par deux, indépendamment du mot sur le bus de données, et aussi bien en lecture qu'en écriture.

Le programme s'écrit alors:

```
                HP=     6
                .LOC    Ø
0       324     OSCIL:  LOAD    $HP,A   ;7,2 µs
1         6
2       37              JUMP    OSCIL   ;7,2 µs
3        0
4        0
```

9. BOUCLES D'ATTENTE

Les programmes OSCIL précédents fournissent des ultra-sons à vitesse nominale du processeur. Une instruction s'exécute en quelques microsecondes, et le programme ci-dessus donne une fréquence de 34. kHz. Pour réduire la fréquence, il faut insérer une boucle d'attente, faite avec un compteur qui décompte . Un 2e registre du processeur peut être utilisé pour cela. Une instruction de saut conditionnel revient en arrière pour décompter le registre tant que le résultat n'est pas égal */NE: non equal/* à zéro.

```
HP=     6
DELAY=  57.     ;71 octal
.LOC    Ø

OSCIL:  LOAD    $HP,A           ;5 µs
        LOAD    B,# DELAY       ;5 µs
OSC2:   DEC     B               ;5 µs
        JUMP,NE OSC2            ;5 µs
        JUMP    OSCIL           ;5 µs
```

Si l'on suppose, pour simplifier, que toutes les instructions précédentes durent 5 µs, une demi-période dure 5 + 5 + DELAY·(5+5) + 5 , soit 15 + DELAY·10 µs. Pour obtenir un LA normal à 432 Hz, il faut donc fixer DELAY= 115. (décimal). La valeur octale correspondante est 161 car 113. = 1.x64. + 6.x8. + 1. Tant que DELAY est inférieur à 377 = 255., il n'y a pas de problème. Pour des retards plus grands, il faut utiliser par exemple deux boucles de comptage imbriquées utilisant deux compteurs.

Le programme ci-dessus est valable pour l'Intel 8080 et pour le Motorola 6800. Avec le Signetics 2650, une instruction unique remplace la décrémentation et la saut conditionnel. On peut donc écrire le programme suivant, donné avec son équivalent octal, prêt à être chargé et exécuté dans un DAUPHIN:

```
                HP=     6
                DELAY=  157.    ;demi-période = 7,2+4,8+(DELAY.7,2)+7,2
                .LOC    Ø       ;             = DELAY·7,2 (autres termes négligeables
                                                           pour des délais longs)
0   324     OSCIL:  LOAD    $HP,A       ;7,2 µs
1     6
2     5             LOAD    B,#DELAY    ;4,8 µs
3   235                                 ;235 = 157.
4   375     OSC2:   DECJ,NE B,OSC2      ;7,2 µs
5     0
6     4
7    37             JUMP    OSCIL       ;7,2 µs
10    0
11    0
```

Remarquons tout de suite que ce programme peut être codé différemment, en utilisant le mode d'adressage relatif disponible dans le 2650 et expliqué dans la deuxième partie.

Deuxième partie:
PROGRAMMATION DU PROCESSEUR SIGNETICS/PHILIPS 2650

1. INTRODUCTION

Dans la première partie de ce fascicule, les notions de système microprocesseur, de processeur et de programme ont été introduites. Le but de cette partie est de montrer dans le détail la structure et les instructions d'un processeur, le Signetics 2650. Un certain nombre d'instructions non indispensables au débutant seront vues plus loin; sans ces instructions, le 2650 ressemble du reste beaucoup plus à des processeurs très courants comme le 8080 ou le 6800.

2. REGISTRES INTERNES DU 2650

Un seul registre pour les transferts et les opérations arithmétiques n'est pas suffisant. Le 6800 a deux registres, le 8080 sept registres et le 2650 quatre registres (plus trois que nous verrons ultérieurement), que nous appellerons A, B, C, D (et non pas R0, R1, R2, R3 comme le fabricant). Ces registres sont reliés entre eux et avec le bus de données par des aiguillages commandés par le décodeur d'instructions. Les possibilités de transfert dépendent des instructions prévues par Signetics.

3. INSTRUCTIONS DE TRANSFERT ENTRE REGISTRES ET MEMOIRE

Les instructions de transfert seront toutes caractérisées par le code mnémonique LOAD. Il y a sept instructions de transfert entre les 4 registres; à chacune de ces instructions se trouve associé un équivalent binaire qui est un mot de 8 bits. Dans nos tableaux, nous placerons devant ce code la notation utilisée par Signetics.

LODZ,RØ	0	LOAD	A,A				
LODZ,R1	1	LOAD	A,B	STRZ,R1	301	LOAD	B,A
LODZ,R2	2	LOAD	A,C	STRZ,R2	302	LOAD	C,A
LODZ,R3	3	LOAD	A,D	STRZ,R3	303	LOAD	D,A

L'instruction LOAD copie dans le registre destination, donné en premier, le contenu du registre source, donné en second. Ce registre source n'est pas modifié. L'instruction LOAD A,A peut sembler inutile. Elle modifie en fait le registre d'état, comme nous le verrons plus loin.

Les transferts avec la mémoire peuvent se faire avec chacun des 4 registres. Pour le moment, nous considérerons que la mémoire du système est plus petite que 8k (8192. bytes exactement). Les particularités de l'organisation de la mémoire du 2650 seront vues ultérieurement. Les instructions de transfert sont:

LODA,RØ m	14 / m	LOAD	A,m	STRA,RØ m	314 / m	LOAD	m,A
LODA,R1 m	15 / m	LOAD	B,m	STRA,R1 m	315 / m	LOAD	m,B
LODA,R2 m	16 / m	LDAD	C,m	STRA,R2 m	316 / m	LOAD	m,C
LODA,R3 m	17 / m	LOAD	D,m	STRA,R3 m	317 / m	LOAD	m,D

Trois mots de 8 bits sont nécessaires pour coder une instruction de transfert entre un registre et une position mémoire. Le premier byte caractérise l'opération et le registre concerné, le deuxième byte contient les bits de poids fort de l'adresse, et le troisième byte les bits de poids faible. Par exemple, l'instruction LOAD A,PREMNB , destinée à transférer dans A le premier nombre d'une série de valeurs préalablement mises en mémoire (PREMNB est par exemple à l'adresse valant 2057 octal) se code

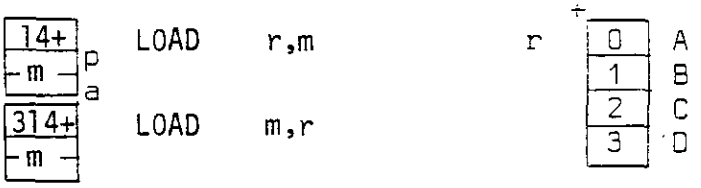
14		
4	page 4	adresse complète 2057 = 4·400 + 57
57	adresse 57	

Le fractionnement de l'adresse complète en deux mots de 8 bits revient à considérer des pages de 256. positions, avec dans chaque page une adresse 8 bits valant de 0 à 377. L'adresse dans la page se trouve dans le 3e byte, l'adresse de la page se trouve dans le 2e byte, et le tableau de correspondance suivant facilite le fractionnement:

adresse	0 - 377	page Ø	(ROM ou RAM DAUPHIN)
	40D - 777	1	
	1000 - 1377	2	(RAM DAUPHIN)
	1400 - 1777	3	
	200D - 2377	4	(Extension
	2400 - 2777	5	mémoire
	3000 - 3377	6	DAUPHIN)
	3400 - 3777	7	
	4000 - 4377	10	
	etc.		

On remarque une relation simple entre le nombre de milliers, multiplié par 2 en octal, et la page: par exemple, l'adresse 6750 se code 6400 + 350
page 2x6+1 = 15

Le tableau des instructions de transfert avec la mémoire, donné au haut de la page, prend beaucoup de place; on peut écrire de façon plus condensée



à condition d'effectuer un calcul mental simple pour assembler l'instruction finale. Le + dans le code de base de l'instruction indique qu'il faut ajouter une certaine valeur qui dépend des opérandes. Le tableau des valeurs à ajouter se trouve à proximité.

EXEMPLE DE PROGRAMME

Imaginons que l'on doive décaler circulairement le contenu des 4 registres ABCD du 2650. A doit recevoir B, qui doit recevoir C, qui doit recevoir D, qui doit recevoir A. Aucune instruction n'a été prévue par le fabricant pour faire cette opération en une seule instruction, et il faut se débrouiller avec les instructions de transfert vues ci-dessus, en faisant attention de ne pas perdre le contenu de l'un des registres. On ne peut pas écrire:

```
LOAD    A,B
LOAD    B,C
LOAD    C,D
LOAD    D,A
```

pour deux raisons: d'abord la première instruction détruit le contenu du registre A, et lorsque la 4e instruction s'exécutera, c'est en fait le contenu de B qui va passer en D; ensuite, les instructions LOAD B,C et LOAD C,D ne font pas partie du répertoire d'instructions du 2650.

Une façon simple et sûre d'écrire le programme revient à définir 4 positions mémoire dans lesquelles on va sauver le contenu des registres, avant de rétablir les registres dans l'ordre voulu.

```
20                      SAVA=   20      ;la position 20 a pour nom SAVA
21                      SAVB=   SAVA+1  ;les positions suivantes sont repérées
22                      SAVC=   SAVA+2  ;par rapport à SAVA pour permettre un
23                      SAVD=   SAVA+3  ;déplacement plus facile de l'ensemble

                        .LOC    100     ;le début du programme est en 100

100   314       PERMUT: LOAD    SAVA,A  ;sauvons A,B,C,D en mémoire
101   0
102   20
103   315               LOAD    SAVB,B
104   0
105   21
106   316               LOAD    SAVC,C
107   0
110   22
111   317               LOAD    SAVD,D
112   0
113   23
114   14                LOAD    A,SAVB  ;rétablissons dans l'ordre voulu
115   0
116   21
117   15                LOAD    B,SAVC
120   0
121   22
122   16                LOAD    C,SAVD
123   0
124   23
125   17                LOAD    D,SAVA
126   0
127   20
                        ...             ;suite du programme
```

Ce programme a pour seul défaut d utiliser beaucoup de place en mémoire, et on peut l'optimiser en écrivant:

```
                        SAVA=   20
                        SAVB=   SAVA+1

                        .LOC    100
100     314     PERMUT: LOAD    SAVA,A
101     0
102     20
103     315             LOAD    SAVB,B
104     0
105     21
106     2               LOAD    A,C
107     301             LOAD    B,A
110     3               LOAD    A,D
111     302             LOAD    C,A
112     17              LOAD    D,SAVA
113     0
114     20
115     14              LOAD    A,SAVB
116     0
117     21
                        ...
```

Chaque programme peut être optimisé de plusieurs façons, et on peut parfois passer à côté d'une solution simple sans s'en rendre compte.

4. INSTRUCTIONS D'INITIALISATION DES REGISTRES

Un registre peut être initialisé à une certaine valeur prévue dans le programme, comme nous l'avons vu à la page 6. Quatre instructions sont prévues dans ce but:

LODI,R n	4+ / n	LOAD	r,# n

+		
r	0	A
	1	B
	2	C
	3	D

Ces instructions ont deux bytes.
n est une valeur numérique de 8 bits (0 à 377) représentée dans l'instruction par un symbole évoquant sa signification.

5. INSTRUCTIONS DE TRANSFERT ENTRE REGISTRES ET PERIPHERIQUES

Le 2650 permet de définir jusqu'à 256. adresses de périphériques, numérotées de 0 à 377, dans lesquelles on peut lire ou écrire comme en mémoire avec les instructions suivantes:

REDE,Ri n	124+ / n	LOAD	r,$n
			lecture */read, input/*
WRTE,Ri n	324+ / n	LOAD	$n,r
			écriture */write, output/*

+		
r	0	A
	1	B
	2	C
	3	D

Les adresses de périphériques se distinguent des adresses mémoire grâce au signe $. Le transfert peut donc se faire directement avec chaque registre. Souvent, seule une partie des huit bits d'adresse est décodée, parce que l'on a rarement besoin de 256. périphériques. Le DAUPHIN, par exemple, ne décode que 6 bits d'adresse de périphériques. Les instructions LOAD B,$1, LOAD B,$101, LOAD B,$201 et LOAD B,$301 sont alors tout à fait équivalentes, mais il n'y a pas lieu d'utiliser les trois dernières.

6. COMPLEMENT A 1 ET A 2

Sans vouloir redéfinir de façon détaillée les notions de complément et de représentation des nombres négatifs, rappelons que le complément à 1 d'un mot binaire s'obtient en inversant tous ses bits, ou en effectuant la différence avec un mot de même longueur qui ne comporte que des 1.

Exemples:	mot de 8 bits:	10 110 101	=	265 octal
	complément à 1:	01 001 010	=	112 octal
calcul du complément à 1		11 111 111		377
par soustraction:		- 10 110 101		- 265
		01 001 010		112

mot de 16 bits:	1 101 011 100 111 110	=	153 476 octal
complément à 1:	0 010 100 011 000 001	=	024 301
calcul du complément à 1 par soustraction:	1 111 111 111 111 111		177 777
	- 1 101 011 100 111 110		- 153 476
	0 010 100 011 000 001		024 301

Le calcul en octal directement évite de devoir écrire des lignes de 0 et de 1: il est très facile, car les règles sont ici identiques à celles du système décimal.

Le complément à 2 s'obtient en faisant la différence avec un nombre égal à 2^n, n étant le nombre de bits. Ce nombre a partout des 0, avec un 1 pour le poids immédiatement supérieur.

Exemples: mot de 8 bits:	10 110 100	=	264 octal
calcul du complément à 2:	$\overline{1}\overline{0}\overline{0}\ \overline{0}\overline{0}\overline{0}\ 000$		400
	- 10 110 100		- 264
	$\overline{0}1\ 001\ \overline{1}0\overline{0}$		114

En octal 8 bits, le calcul se ramène à des différences à 8, puis à 7, avec pour le chiffre de poids le plus fort une différence à 4 ou à 3, car ce chiifre ne code que deux bits.

Si on a des mots de 16 bits, il faut faire la différence à 200 000. Pour des mots de 7 bits, tels qu'on va en trouver dans le 2650, il faut faire la différence à 200.

Le complément à 2 d'un nombre est généralement utilisé pour représenter la valeur négative de ce nombre, mais il est important de connaître la longueur du mot binaire et être sûr que les nombres positifs ou négatifs à représenter ne dépassent pas la grandeur permise.

Exemples: mots de 8 bits + 45 représenté 0 0 1 0 0 1 0 1
- 45 représenté par son complément à 2^8, égal à

400 - 45 = 333 1 1 0 1 1 0 1 1
bit de signe

+ 243 ne peut pas être représenté (val. max. + 177)
- 303 ne peut pas être représenté (val. min. - 200 représenté par 200)

mots de 7 bits + 45 représenté 045 0 1 0 0 1 0 1
- 45 représenté par son complément à 2^7, égal à

200 - 45 = 133 1 0 1 1 0 1 1
bit de signe

+ 147 ne peut pas être représenté (val. max. + 77)
- 101 ne peut pas être représenté (val. min. - 100 représenté par 100)

Les nombres représentés ainsi sont appelés nombres arithmétiques. Le bit de poids le plus fort est le bit de signe. Lorsqu'il vaut 1, le nombre est négatif et représenté sous forme de complément à 2. Pour connaître sa valeur absolue, il faut en prendre le complément à 2.

Exemple: Quel est l'équivalent décimal du nombre arithmétique octal 7 bits 123 ?

- Ce nombre est négatif car le 7e bit vaut 1 ($\geq 100_8$)
- La valeur absolue égale au complément à 2 vaut 200 - 123 = 055
  (calcul mental: 0 - 3, c'est à dire 8 - 3 = 5 avec emprunt
  0 - 2 - emprunt, càd 8 - 3 = 5 avec emprunt
  2 - 1 - emprunt = 0)
- La valeur décimale équivalente est 55 = 5·8. + 5 = 45.
- Le nombre arithmétique 7 bits 123 donné représente le nombre décimal - 45.

Il faut remarquer dans la donnée de cet exemple que les 3 termes arithmétique octal 7 bits sont importants. Le nombre pourrait être un nombre positif octal 7 bits (Signetics et quelques autres parlent alors de nombre logique). Dans ce cas, les 7 bits sont utilisés pour les nombres de 0 à 177, et il n'est plus possible de représenter les nombres négatifs.

Le nombre pourrait être un nombre arithmétique octal 8 bits. Dans ce cas, 123 serait un nombre positif, car le 8e bit vaut 0.

Si le nombre était un nombre arithmétique décimal, il ne faudrait pas préciser le nombre de bits, mais le nombre de digits, et définir clairement comment le signe est représenté.

La représentation des nombres négatifs sous forme de nombres arithmétiques en complément à 2 facilite considérablement les opérations arithmétiques. Il suffit d'ajouter les compléments au lieu de soustraire. Par exemple, si l'on doit effectuer l'opération 8 bits 124 + 31 - 173 (octal), et que l'on s'est assuré que chaque nombre et chaque résultat partiel ne dépasse pas la capacité de la machine (7 bits plus un 8e bit de signe), il suffit de convertir - 173 en son complément à deux 8 bits (400 - 173 = 205) et d'additionner

```
  124        155
+  31      + 205
  155        362
```

Le résultat est négatif ($\geqslant$200) et peut être conservé tel quel pour les calculs ultérieurs.

Ces quelques exemples montrent bien la diversité des représentations des nombres, et les problèmes et erreurs qui peuvent apparaître lorsque la représentation utilisée n'a pas été bien définie et comprise.

7. ADRESSAGE RELATIF

Dans un programme, la plupart des adresses qui figurent dans une instruction ont une valeur proche de l'adresse de l'instruction. L'adressage relatif permet de définir une adresse par rapport à l'adresse de l'instruction qui s'y réfère, de même que l'on dit dans le langage courant "il habite deux maisons plus loin que moi". L'avantage est que les nombres sont plus petits et prennent moins de place, donc les programmes sont plus courts et moins coûteux en mémoire.

Les instructions de transfert entre registres et mémoires utilisant l'adressage relatif sont les suivantes:

LODR,Ri ℓ'	10+ / ℓ'	LOAD	r,.+ℓ'
STRR,Ri ℓ'	310+ / ℓ'	LOAD	.+ℓ',r

r	
0	A
1	B
2	C
3	D

Le déplacement ℓ' est un nombre arithmétique 7 bits. (Le 8e bit a une signification qui sera vue plus tard). Il est important de bien comprendre comment s'exécute une telle instruction. La séquence des opérations est montrée dans la Fig. 6 qui montre comment s'exécute l'instruction LOAD A, DATA avec DATA = .+4 (charge A avec le contenu de la position DATA, qui se trouve 4 positions mémoire en dessous).

Un additionneur d'adresses calcule l'adresse effective à partir du contenu du compteur d'adresses PC et du contenu du déplacement dans le registre d'instructions. Comme le compteur d'adresses est automatiquement incrémenté lors de la recherche de l'instruction, c'est l'adresse de l'instruction suivante qui intervient dans le calcul de l'adresse effective.

Lorsque le déplacement est négatif, le calcul est identique. L'additionneur d'adresses tient compte du fait que le déplacement est un nombre arithmétique 7 bits et convertit ce nombre en un nombre arithmétique 15 bits pour que l'opération soit consistante (les bits 8 à 15 ont la même valeur que le bit 7).

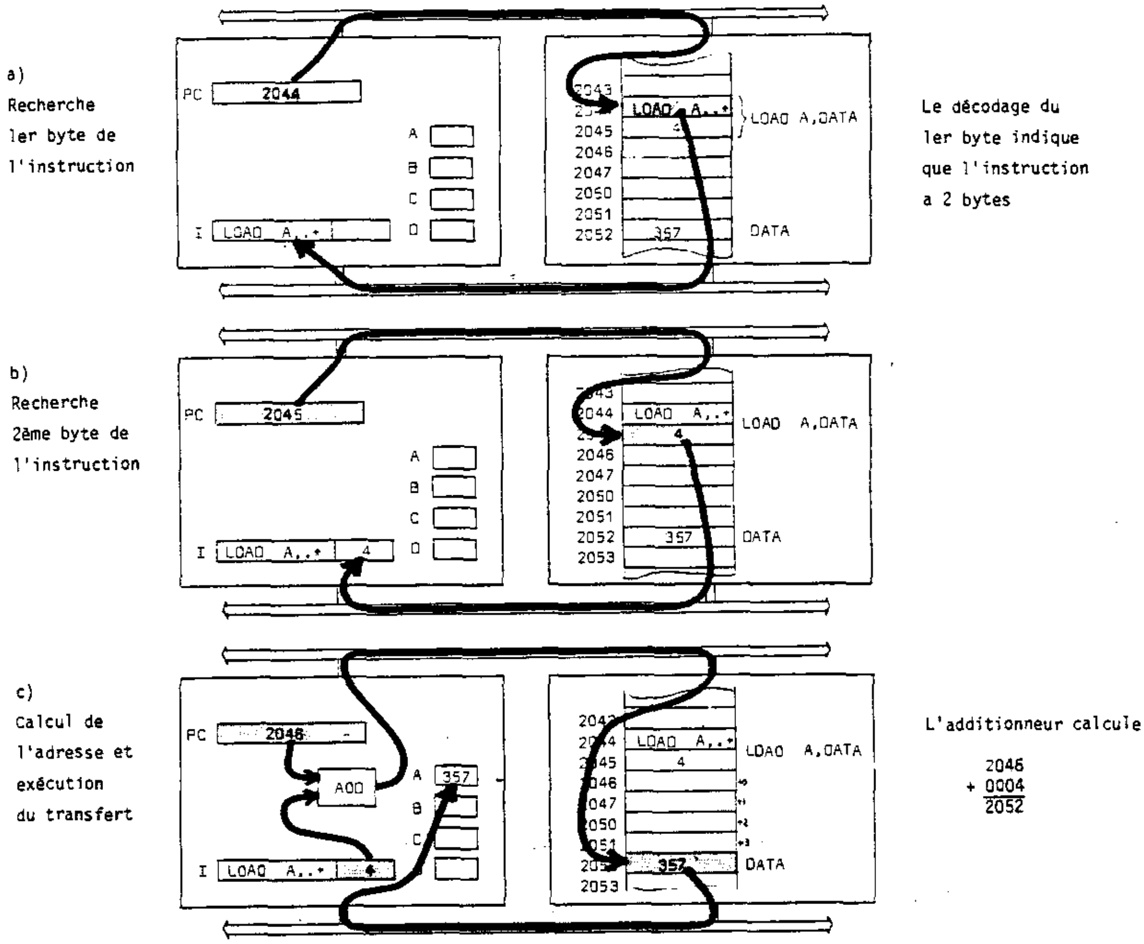


Fig. 6. Séquence d'exécution d'une instruction avec adressage relatif.

EXEMPLE DE PROGRAMME AVEC ADRESSAGE RELATIF

Reprenons l'exemple de permutation de registres précédent, avec deux codages différents du programme selon que les positions mémoire SAVA et SAVB se trouvent avant ou après le programme (à distance faible).

```
 20          SAVA=  20
 21          SAVB=  SAVA+1
             .LOC   100                                      20  vérification binaire   000 000 010 000
100  | 310 | LOAD   SAVA,A  ;calcul du déplacement       - 102                        - 000 001 000 010
101  | 116 |                                               116   021                   111 111 001 110
102  | 311 | LOAD   SAVB,B  ;calcul du déplacement              -104                       7 bits
103  | 115 |                                                     115
104  |  2  | LOAD   A,C
105  | 301 | LOAD   B,A
106  |  3  | LOAD   A,D
107  | 302 | LOAD   C,A                       20
110  |  13 | LOAD   D,SAVA  ;déplacement     -112
111  | 106 |                                  106     21
112  |  10 | LOAD   A,SAVB  ;déplacement             -114
113  | 105 |                                           105
             ...
```

Le programmeur, s'il fait l'assemblage à la main, doit calculer le déplacement à mettre dans le champ de l'instruction en faisant la différence entre l'adresse du paramètre et l'adresse de l'instruction suivante. Pour le 2650, le résultat doit être un nombre arithmétique 7 bits. On voit dans cet exemple qu'il s'en suffit de peu pour que l'adressage relatif ne soit plus possible. Dans ce cas, il faut soit déplacer les paramètres ou le programme, soit utiliser l'adressage absolu qui prend plus de place en mémoire.

Le début de ce même programme, avec SAVA et SAVB situés après le programme donne:

```
140          SAVA=   140
141          SAVB=   SAVA+1
                     .LOC    100
                                                              140
100  | 310 |  LOAD   SAVA,A  ;calcul du déplacement          -102
101  |  36 |                                                   36   141
102  | 311 |  LOAD   SAVB,B  ;calcul du déplacement                -104
103  |  35 |                                                        35
104  |   2 |  LOAD   A,C
     |     |  ...
```

Le processeur 2650 dispose encore des modes d'adressage indirect et indexé. Ils seront expliqués aux pages 32 et 35.

8. REGISTRE D'ETAT

En plus des 4 registres A B C D se trouve un registre d'état */status register/* lié à l'unité arithmétique et à un circuit qui teste le signe et la valeur des nombres qui sont transférés. Nous désignons ce registre par la lettre L (Signetics utilise les lettres PSL */Processor Status word Lower/*). Les valeurs de 5 des 8 bits de ce registre donnent une indication sur l'opération qui vient de s'effectuer.

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	200	100	40	20	10	4	2	1
L	LT	GT	X			V		C

CC1	200	LT	Lower than
CCØ	100	GT	Greater than
IDC	40	X	Auxiliary carry
OVF	4	V	Overflow bit
C	1	C	Carry

Les instructions LOAD ne modifient que les deux premiers bits de L.
Le nombre transféré peut être positif différent de zéro /greater than zero/, négatif /lower than zero/ (dans ce cas, le bit de signe est à 1) ou égal à zéro. Ces trois cas chargent les bits de L de différentes façons.

Nombre transféré nul	0 0 x x x x x x	LT = 0 GT = 0
Nombre transféré positif	0 1 x x x x x x	LT = 0 GT = 1
Nombre transféré négatif	1 0	LT = 1 GT = 0

Les trois autres bits d'état seront étudiés plus loin.

Des instructions de transfert simples existent entre L et A:

```
SPSL  | 23 |   LOAD   A,L
LPSL  | 223|   LOAD   L,A
```

Le 2650 ne permet pas de transfert direct entre L et la mémoire. Ces instructions seront toutefois rajoutées dans le futur 2650B, pour accélérer les routines d'interruption.

La compréhension des bits du registre d'état (que nous appellerons souvent flags) est très importante. Il est bon de vérifier sur de petits exemples que l'on a bien compris. Avec le DAUPHIN sans moniteur en ROM, on peut insérer 3 instructions pour faire apparaître en instruction pas à pas l'état des flags sur les lampes du panneau de test.

Exemple:

```
                NBPOS=  54
  4             LOAD    A,# NBPOS ; exemple d'instruction modifiant L
  54
 23             LOAD    A,L
324             LOAD    $4,A        } Instructions pour
  4                                 l'affichage de L
223             LOAD    L,A
```

La première instruction effectue le transfert d'un nombre positif et modifie L:
GT = 1, LT = 0.

La seconde instruction transfère L dans A et modifie les flags LT et GT de L. L'ancienne valeur de L qui nous intéresse est toutefois sauvée dans A. La troisième instruction fait apparaître la valeur sur les lampes du DAUPHIN. L'adresse 4, choisie arbitrairement, apparaît sur le bus d'adresses. La dernière instruction rétablit l'état de L et permet de continuer l'exécution comme si les trois instructions "espion" n'avaient pas été insérées. Ceci est très important car il peut y avoir tout de suite après une instruction qui teste LT ou GT.

9. REGISTRE DE MODE

Certains bits du registre L agissent sur des fonctions opérées dans l'unité arithmétique.

	2^7							2^0
L	200	100	40	20 B	10 W	4	2 L	1 C

BS	20	B	Bank 1
WC	10	W	Withcarry
COM	2	L	Logical compare
C	1	C	Carry - no borrow

Par exemple le bit W de poids 10 (2^3) décide si certaines opérations se font avec ou sans la retenue */carry/*. Si ce bit est à 1, les opérations tiennent compte de la retenue */with carry/*. La valeur de la retenue elle-même (flag C) se trouve dans un autre bit et peut être modifiée, présélectionnée en même temps (ou indépendamment) que le bit W.

L'instruction LOAD L,A permet de mettre ces bits à toute valeur qui a été préalablement préparée dans A. Des instructions spéciales de 16 bits permettent d'agir directement sur chaque bit.

PPSL H'1'	167 1	SETC SET	CARRY	CPSL H'1'	165 1	CLRC CLR	CARRY
PPSL H'8'	167 10	SET	WITHCARRY	CPSL H'8'	165 10	CLR	WITHCARRY
PPSL H'9'	167 11	SET	CARRY,WITHCARRY	CPSL H'9'	165 11	CLR	CARRY,WITHCARRY
				CPSL H'FF'	165 377	CLR	L

On peut remarquer que deux écritures différentes sont proposées pour certaines instructions. SETC correspond à la notation utilisée dans le 6800 et 8080. SET CARRY est plus explicite et mieux adapté au 265D, où chacun des flags peut être mis à un ou à zéro indépendamment. Il est évident que si certains bits doivent être mis à un, et d'autres à zéro, deux instructions doivent être utilisées.

Les instructions SET et CLR sont des cas particuliers des instructions 167 / n OR L,# n et 165 / n BIC L,# n. La première force à 1 tous les bits de L qui correspondent à des bits de n valant 1. La seconde, appelée BIC pour "bit clear", force à zéro tous les bits de L qui correspondent à des bits de n valant 1. Ainsi BIC L,# 377 remet à zéro tous les bits de L et s'appelle aussi CLR L.

Le flag BANK1 sélectionne le groupe de registres supplémentaires B'C'D' s'il vaut 1. Le flag LOGICOMP sera vu à la page 23.

10. OPERATIONS ARITHMETIQUES D'ADDITION

Toutes les notions qui précèdent sont nécessaires à la compréhension des opérations arithmétiques du 2650. Les opérations à deux opérandes permettent de combiner deux registres, un registre et une valeur immédiate, ou un registre et une position mémoire, et de mettre le résultat dans le registre donné en premier (registre destination).

Par exemple, pour l'addition, on a les possibilités principales suivantes.

ADDZ	200+	ADD	A,r
ADDI	204+ / n	ADD	r,# n
ADDA	214+ / m	ADD	r,m
ADDR	210+ / ℓ'	ADD	r,.+ ℓ'

r	+
0	A
1	B
2	C
3	D

Des nombres de 8 bits, arithmétiques ou logiques, sont additionnés. La différence entre nombre arithmétique avec bit de signe en 7e ou 8e position et nombre positif 8 bits n'intervient pas au moment de l'addition, mais au moment de l'analyse du résultat, qui dépend des flags C (carry), V (overflow), GT (greater than) et LT (lower than).

Comme exemple de programme simple à vérifier sur un DAUPHIN et destiné à se familiariser avec les opérations arithmétiques, considérons le programme suivant, qui peut, par exemple, être chargé depuis la position Ø.

```
0   165   TEST:  CLR    L
1   377
2   4            LOAD   A,# 123 ;1er nombre
3   123
4   5            LOAD   B,# 345 ;2e nombre
5   345
6   201          ADD    A,B
7   324          LOAD   $4,A    ;affichage résultat
10  4
11  23           LOAD   A,L
12  324          LOAD   $4,A    ;affichage des flags
13  4
14  223          LOAD   L,A
```

Le résultat est égal à 123 et le registre d'états a pour valeur 101 (octal)
+345 indiquant qu'une retenue a été produite (CARRY = 1)
070 et que le résultat est positif (GT = 1).

Si l'on voulait additionner deux nombres logiques égaux à 123 et 345, CARRY = 1 signifierait qu'il y a eu un dépassement de capacité. Si les deux nombres donnés sont des nombres arithmétiques 8 bits (opération 123 - 33), alors le fait que le bit V de poids 4 soit à zéro indique qu'il n'y a pas eu de dépassement de capacité. Le CARRY n'a pas de signification et le résultat est positif car GT = 1.

Des additions sur des mots de plus de 8 bits sont possibles en répétant l'opération sur autant de tranches de 8 bits que nécessaire.

Exemple:

```
                  ①          ①
    10010011 11000110 01111011
+   00100110 10001101 11101001
  ----------------------------
    10111010 01010100 01100100
```

On remarque que la retenue produite par la première opération 8 bits ne doit pas être considérée comme un dépassement de capacité, mais doit être additionnée comme une unité simple avec l'opération d'addition suivante. Si le flag WITHCARRY est à 1, ceci est fait automatiquement, et nous changerons le nom de l'instruction pour bien montrer que l'opération effectuée est différente: ADDC sera utilisée toutes les fois que l'instruction SET WITHCARRY , ou une instruction ayant un effet équivalent, précède. Dans les processeurs 8080 et 6800, il n'y a pas d'instruction de ce type, dont l'effet est modifié selon une instruction précédente.

La programmation de l'exemple numérique ci-dessus peut se faire en supposant que l'un des mots de 24 bits est dans les registres A, B, C et l'autre dans les positions mémoire DATA, DATB, DATC (poids faibles dans A et DATA). La partie de programme qui nous intéresse peut s'écrire:

```
20          DATA=   20
21          DATB=   DATA+1
22          DATC=   DATA+2
            .LOC    0
0           CLR     WITHCARRY
1
2           ADD     A,DATA
3
4           SET     WITHCARRY
5
6           ADDC    B,DATB
7
10          ADDC    C,DATC
11
```

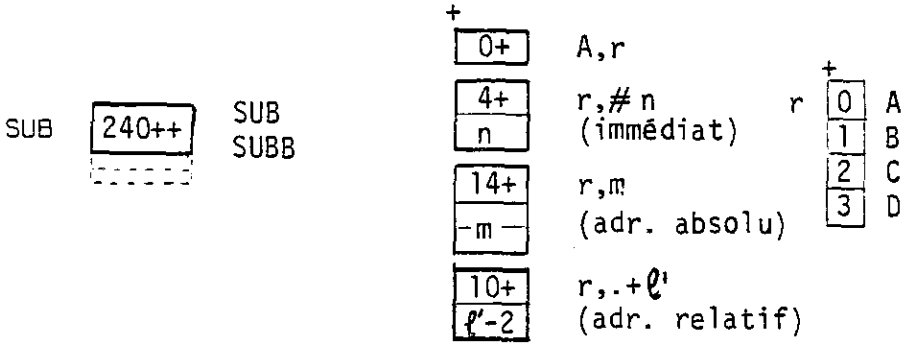
```
20          DATA=   20
21          DATB=   DATA+1
22          DATC=   DATA+2
            .LOC    0
0           SET     WITHCARRY
1
2           CLR     CARRY
3
4           ADDC    A,DATA
5
6           ADDC    B,DATB
7
10          ADDC    C,DATC
11
```

EXERCICE: coder ces 2 programmes, puis faire la somme en octal des codes des instructions. (Résultat: 1264 et 1253)

Dans le premier cas, la première addition est faite dans le mode qui ne tient pas compte de la retenue. Le second programme utilise partout le même mode d'addition, mais il faut s'assurer que la retenue est à zéro au moment de la première addition.

11. OPERATION DE SOUSTRACTION

L'instruction de soustraction du 2650 a les même possibilités d'adressage que celle d'addition. On peut écrire de façon un peu plus condensée



Si WITHCARRY = 0, l'opération de soustraction est une addition du complément à 2, indépendante de la valeur préalable du CARRY. La valeur finale du CARRY dépend du fait que l'addition du complément à 2 a ou non produit un CARRY.

Par exemple, les opérations 133 - 47 et 0 - 0 produisent un CARRY, alors que dans d'autres processeurs (6800, 8080), un soustracteur est utilisé et le CARRY reste à zéro dans ces cas.

			①	②			
Exemples:	1)	133	133	01 011 011	2)	34	34
		-47	+331	11 011 001		-357	+021
			064	00 110 100			055

La première opération correspond à la soustraction de deux nombres positifs 8 bits, ou arithmétiques 7 bits positifs tous les deux.

Si l'on examine le registre L après cette opération, on trouve L = 141 que l'on décompose facilement en 3 bits actifs (cf. p.14).

```
100  :  GT = 1  Le  résultat est un nombre arithmétique positif
 40  :   X = 1  L'opération a produit une retenue au 4e bit
                (peut être utile dans certaines opérations BCD)
  1  :   C = 1  L'opération a produit un CARRY.
```

La seconde soustraction peut être soit considérée comme une soustraction de 2 entiers 8 bits, avec un résultat négatif représenté en complément à 2 puisque CARRY = 0, soit comme une soustraction entre un nombre arithmétique positif, +34, et un nombre arithmétique négatif 357 = -21; le résultat est positif.

Il faut être très prudent avec l'arithmétique signée ou non. Les tests de dépassement de capacité sont délicats et nécessitent une analyse attentive.

Si WITHCARRY = 0, soustraire l'accumulateur à lui-même est un moyen simple de le mettre à zéro, avec le CARRY mis à 1 à coup sûr à la fin de l'opération.

Exemple:	Contenu de A: 123	SUB A,A	123	① 123
			-123	+255
				000

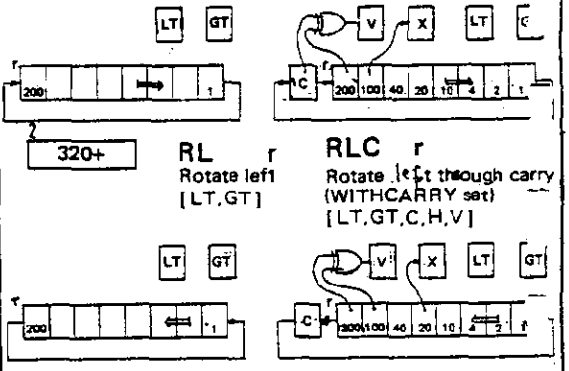
[Flags modified]

x y

0++	**LOAD** Load x with y [LT,GT]	2 (1) 0+	A,r Register
		2 4+ n	r,#n Immediate
300++	**LOAD** Load y with x (store). LOAD #n,r does not exist LOAD r,A modified LOAD memory does not modify flags	3 10+ ℓ'-2	r, · + ℓ' Relative
		5 10+ 200+ℓ'-2	r,@ · + ℓ' Relative indirect
200++	**ADD** Add x and y, result in x **ADDC** Add carry if WITHCARRY set [LT,GT,C,H,V]	4 14+ p' a	r,m Absolute m'
240++	**SUB** Subtract with borrow if **SUBB** WITHCARRY set [LT,GT,C,H,V] (add 2-s complement)	6 14+ 200+p' a	r,@ m Absolute indirect 100000+m'
100++	**AND** Logical AND [LT,GT] ! AND A,A = WAIT	4 14+ 140+p' a	A,(r)+m Indexed 60000+m'
140++	**OR** Logical OR [LT,GT]	6 14+ 340+p' a	A,(r)+@m Indirect indexed 160000+m'
40++	**XDR** Exclusive OR [LT,GT]	4 14+ 40+p' a	A,(+r)+m Pre-incrementing indexed 20000+m'
340++	**COMP** Compare arithmetic, logic if LOGICOMP set) [LT,GT]	6 14+ 240+p' a	A,(+r)+@m Pre-incrementing indirect indexed 120000+m'
		4 14+ 100+p' a	A,(−r)+m Pre-decrementing indexed 40000+m'
		6 14+ 300+p' a	A,(−r)+@m Pre-decrementing indirect indexed 140000+m'

2 (1) 300	**NDP** No operation [none]	
100	**WAIT** Wait for interrupt	

2 120+	**RR** r Rotate right [LT,GT]	**RRC** r Rotate right though carry (WITHCARRY set) [LT,GT,C,H,V]
2 320+	**RL** r Rotate left [LT,GT]	**RLC** r Rotate left though carry (WITHCARRY set) [LT,GT,C,H,V]



3 364+ n	**TEST** r,#n Test bit [LT=NO,GT←o[	r: 0 A, 1 B, 2 C, 3 D
3 224+	**DA** r [LT,GT] Decimal adjust Flag value does not corresoond to final result	
3 124+ n	**LOAD** r,$n [LT,GT]	Read peripheral r data read in r
3 324+ n	**LOAD** $n,r [none]	Write in peripher: data in r
2 160 +	**LOAD** r,$DATA [LT,GT]	Read specially decoded peripheral
2 360 +	**LOAD** $DATA,r [none]	Write in specially decoded peripheral
2 60 +	**LOAD** r,$CTRL [LT,GT]	Read specially decoded peripheral
2 260 +	**LOAD** $CTRL,r [none[	Write in specially decoded peripheral

3 166 100	**SET** [O]	**OUTPUT**
164 100	**CLR** [O]	**OUTPUT**
164 40	**ION**	CLR IOF
166 40	**IOF**	SET IOF
164 7	**CLR** [SP]	**STACK**
167 20	**SET** [B]	**BANK1**
165 20	**CLR** [B]	**BANK1**

3 167 10	**SET** [W]	**WITHCARRY**
165 10	**CLR** [W]	**WITHCARRY**
167 2	**SET** [L]	**LOGICOMP**
165 2	**CLR** [L]	**LOGICOMP**
167 1	**SETC** [C]	SET CARRY
165 1	**CLRC** [C]	CLR CARRY
167 4	**SETV** [V]	SET OVERFLOW
165 4	**CLRV** [V]	CLR OVERFLOW

3 264 100	**TEST** [LT=NO,GT←o]	**OUTPUT**
264 200	**TEST**	**INPUT**
264 40	**TEST**	IOF
265 20	**TEST**	BANK1
265 10	**TEST**	WITHCARRY
265 2	**TEST**	LOGICOMP
265 1	**TEST**	CARRY
265 4	**TEST**	OVERFLOW

166 / n	OR	U,#n — Bit set
167 / n	OR	L,#n — Bit set
164 / n	BIC	U,#n — Bit clear
165 / n	BIC	L,#n — Bit clear
264 / n	TEST	U,#n — Test bit [LT=NO, GT←o]
265 / n	TEST	L,#n — Test bit [LT=NO, GT←o]
22	LOAD	A,U [LT,GT]
23	LOAD	A,L [LT,GT]
222	LOAD	U,A [O, IOF, SP]
223	LOAD	L,A [LT,GT,H,B,W,V,L,C]

Only on 2650-B (may never be produced)

20 / p / a	LOAD	L,m — m [LT,GT,H,B,W,V,L,C]
20 / 200+p / a	LOAD	L,@ m — 100000+m' [all]
21 / p / a	LOAD	m,L — m [none]
21 / 200+p / a	LOAD	@m,L — 100000+m [none]

30++	JUMP,t	Jump if t true [none]
70++	CALL,t	Call if t true [SP]

0+ / ℓ'−2	·+ℓ'
0+ / 200+ℓ'−2	@·+ℓ'
4+ / p / a	m
4+ / 200+p / a	@ m — 100000+m

t		
3		Unconditional
0	EQ	AO — Equal or equal to zero; All selected bits one after TEST instruction
1	GT	Greater than or strictly positive — GT=1
2	LT	NO — Lower than or negative; Not all selected bits one — LT=1
200	NE	Non equal or non equal to zero
201	LE	Lower or equal — GT=0
202	GE	Greater or equal or positive — LT=0
100	ANE	Register A non equal to zero
101	BNE	Register B non equal to zero
102	CNE	Register C non equal to zero
103	DNE	Register D non equal to zero

233+	JUMP	Jump always [none]
273+	CALL	Call always [SP]

0 / ℓ	0+ℓ — Relative to location 0
0 / 200+ℓ	@0+ℓ
4 / p / a	(D)+m — Indexed
4 / 200+p / a	(D)+@m — Indirect indexed — 100000+m

24+	RET,t	Return if t true [SP]
64+	RETION,t	Return and clear interrupt mask [SP,IOF]

t		
3		Unconditional
0	EQ	AO
1	GT	
2	LT	NO

330++	INCJ,NE	Increment register r and jump if non equal to zero [none]
370++	DECJ,NE	Decrement register r and jump if non equal to zero [none]

0+ / ℓ'−2	r,·+ℓ'
0+ / 200+ℓ'−2	r,@·+ℓ'
4+ / p / a	r,m
4+ / 200+p / a	r,@ m — 100000+m

r: 0 A, 1 B, 2 C, 3 D

Common Assembly Language for Microprocessors REFERENCE CARD

SIGNETICS 2650

REGISTERS

A

BANKØ	BANK1
B	B'
C	C'
D	D'

U Flag and mode register

200	I	INPUT (Sense)
100	O	OUTPUT (Flag)
40	IOF	IOF (Interrupt mask bit)
0-7	SP	STACK pointer

Return address stack

L Flag and mode register

200	LT	Lower than
100	NO	Not all one
40	GT	Greater than
20	H	Half carry
10	B	BANK1 (second register bank)
4	W	WITHCARRY (with carry mode bit)
2	V	OVERFLOW
1	C	CARRY (carry and no borrow bit)

Number of cycles per instruction

330+ / 0	INC	r [none]
370+ / 0	DEC	r [none]
40	CLR	A (LT←o, GT←o)

r: 0 A, 1 B, 2 C, 3 D

Minimum cycle duration

2650	2.4 µs (1.25 Mhz clock)
2650-A	
(2650-B)	
2650 AI	1.5 µs

2650-B will have additional instructions and 1 cycle register to register transfer.

Addresses and data conventions

m	15 bit address
m'	13 bit address in same chapter as PC
p	7 bit page address (1 page = 256. locations)
p'	5 bit page address in same chapter
a	8 bit address in a page
ℓ'	7 bit displacement in 2-s complement (6 bit + sign bit) 0 to 101 positive, 177 (−1) to 102 (−76) negative
n	8 bit value (positive or signed)

MEMORY ORGANISATION

0	CHAPTER 0 — 32. pages
20000	CHAPTER 1 — $CTRL
40000	CHAPTER 2
60000	CHAPTER 3 — $DATA
77777	

IB instruction add 1-s complement plus one
SUBB instruction add 1-s complement plus CARRY
Carry is set if result is positive (8-bit logical numbers)

A instruction add a value depending on C(carry) and X(auxiliary carry)

X	C	Added value Binary	Hexa	Octal
0	0	1010 1010	AA	252
0	1	1010 0000	A0	240
1	0	0000 1010	0A	12
1	1	0000 0000	00	0

BCD addition:
```
ADD   r,#146 (octal)
ADD   r,ADDEND
DA    r
```

BCD subtraction:
```
SUB   r,SUBTRAHEND
DA    r
```

Interrupt sequence:
IOF (set interrupt mask bit)
First byte of CALL 0+ℓ' is forced internally
Intack reads second byte of instruction
(Jump to location 0—77 and 177700—17777 with possible indirection)

Reset : Start at 0 with interrupt on (IOF=Ø)

For information on compatible cross-assemblers on PDP-11 minicomputers and on large computers, write to LCD EPFL, Bellerive 16, CH 1007 Lausanne Switzerland

REV4, Sept 1977.

Que se passe-t-il si A vaut déjà zéro ?
Pour répondre à cette question, il faut savoir que le complément à 2 utilisé lors de la soustraction est en fait un complément à 1 (inversion de tous les bits) et une addition de 1. Le soustracteur calcule donc 0 - 0 de la façon suivante:

```
 0      ① 0
-0      +377    complément à 1 de 0
        + 1     correction pour obtenir le complément à 2
         000
```

On voit donc que la règle précédente est tout à fait générale.

Si WITHCARRY = 1, l'opération de soustraction est identique à l'addition du complément à 1 et du CARRY. Le mnémonique SUBB signifie *subtract with borrow* et il est différent du code SUBC *(subtract with carry)* que l'on trouve dans le 6800 et le 8080, car l'opération effectuée est différente. Pour que l'opération de soustraction donne bien la soustraction binaire usuelle lorsque WITHCARRY = 1, il faut que CARRY = 1 (BORROW = 0) et non pas CARRY = 0.

Pour la soustraction de deux nombres de 24 bits, on a le choix entre les programmes suivants, qui appellent les mêmes définitions que l'exemple du bas de la page 16.

```
CLR     WITHCARRY       ou      SET     WITHCARRY,CARRY
SUB     A,DATA                  SUBB    A,DATA
SET     WITHCARRY               SUBB    B,DATB
SUBB    B,DATB                  SUBB    C,DATC
SUBB    C,DATC
```

C	B	A
DATC	DATB	DATA

Le premier programme soustrait les poids faibles dans le mode normal. Le CARRY doit intervenir pour les deux bytes suivants. On remarque que l'instruction s'appelle SUB, puis SUBB: c'est en fait le même code binaire et la différence de nom est destinée uniquement à attirer l'attention du programmeur sur la différence de fonction effectuée.

Les opérations en décimal (codé BCD) sont facilitées par une instruction "Decimal Adjust" DA assez subtile.

```
DAR  [224+]  DA   r
```

r	+	
	0	A
	1	B
	2	C
	3	D

Bornons-nous ici à donner la "règle de cuisine" qui peut se vérifier en écrivant les mots binaires correspondants, groupés par tranches de 4 bits.

Pour l'addition BCD, il faut ajouter 01 100 110 = 146 octal au résultat, puis appliquer l'instruction DA, qui corrige le résultat binaire en un résultat BCD valable, avec le CARRY à 1 si le résultat est supérieur à 99. L'instruction DA ajoute 1010 (10 en binaire) toutes les fois que le résultat (4 bits + retenue) est inférieur ou égal à 1010, mais il n'y a pas de report plus loin que les 4 bits.

Exemples:

```
 33.  ⟶  0011 0011
+18.      0001 1000   Addition binaire
          ① ⓪ ① ①
          0100 1011
          0110 0110   Correction binaire
          1011 0001
          1010        Decimal adjust
 51.  ⟵  0101 0001
```

```
                          Valeur mémorisée
                          dans le bit d'état X
 47  ⟶  0100 0111
+88      1000 1000   Addition binaire
①①  ⓪  ①  ①
         1100 1111
         0110 0110
         0011 0101
         ————————   Decimal adjust
 ①35 ⟵  0011 0101
```

La partie de programme ajoutant deux nombres BCD de 4 digits, situés dans les registre A (poids faibles) et B, et dans les positions mémoires DATA et DATB est donc:

```
CLR     WITHCARRY
ADD     A,DATA
ADD     A,#146
DA      A
SET     CARRY,WITHCARRY
ADDC    B,DATB
ADDC    B,#146
DA      B
```

Pour la soustraction, la correction binaire de 146 n'est pas nécessaire.
Il suffit de soustraire en binaire et d'exécuter l'instruction DA.

```
Exemples:   33  ⟶  0011 0011  ⟶  ①0011 0011
           -18     - 0001 1000     + 1110 0111
                                   +        1   Complément à 2
                                   ------------
                                    0001 1011
                                         1010   Decimal adjust
                                   ------------
           ①15  ⟵                  0001 0101
```

Comme pour la soustraction binaire, le CARRY est à 1 s'il n'y a pas de retenue décimale.

Les programmes se simplifient légèrement si l'on ne doit qu'ajouter ou soustraire 1. Par exemple, pour un compteur/décompteur 4 digits occupant les registres C et D (poids faibles dans C), on a les routines de base suivantes:

```
COUNT:  SET     CARRY,WITHCARRY      DECNT:  SET     CARRY,WITHCARRY
        ADDC    C,#146                       SUBB    C,#1
        DA      C                            DA      C
        ADDC    D,#146                       SUBB    D,#0
        DA      D                            DA      D
```

Pour faire des compteurs BCD, il peut parfois être avantageux d'utiliser une position mémoire par digit, et de comparer le résultat obtenu à chaque incrément pour décider de la correction à faire.

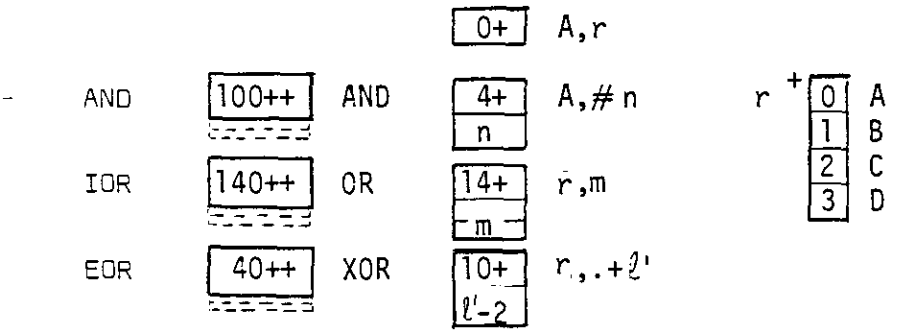
Par exemple, si CNTØ, CNT1, CNT2, CNT3 sont les positions mémoire occupées par le compteur (CNTØ correspond au digit de poids faible), le programme qui ajoute 1 dans ce compteur s'écrit:

```
COUNT:  LOAD    A,#10. ;10.=12 octal
        LOAD    B,#0
        LOAD    C,CNTØ  ;transfert du 1er digit (poids faible)
        INC     C       ;ajouter 1
        COMP    A,C     ;la valeur 10. est-elle atteinte ?
        JUMP,NE END     ;si non, c'est fini
        LOAD    CNTØ,B  ;si oui, il faut mettre Ø en CNTØ
        LOAD    C, CNT1 ;et ajouter 1 au digit suivant
        INC     C       ;et continuer les tests et reports
        COMP    A,C
        JUMP,NE END
        LOAD    CNT1,B
        LOAD    C,CNT2
        INC     C               ;les instructions COMPare et JUMP,NE
        COMP    A,C             ;(saut si la comparaison a donné une
        JUMP,NE END             ;inégalité) sont expliquées plus loin
        LOAD    CNT2,B
        LOAD    C,CNT3
        INC     C
        COMP    A,C
        JUMP,NE END
        LOAD    CNT3,B
END:    ...             ;suite du programme
```

Ces programmes se prêtent à mille variantes et il faut les optimiser en tenant compte des autres opérations, en particulier de l'affichage des résultats, qui peuvent conduire à préférer certaines représentations.

12. OPERATIONS LOGIQUES.

Les trois opérations logiques agissant sur les registres A,B,C et D sont le "et logique" AND , le "ou logique" OR et le "ou exclusif" XOR . Ces instructions ne modifient que les bits GT et LT du registre d'état L. Le codage de ces instructions est le suivant.



L'instruction AND est souvent utilisée pour masquer, enlever, c'est-à-dire mettre à zéro une partie d'un mot. Par exemple, si seuls les 4 bits de poids faible d'un nombre dans A doivent être additionnés à un autre nombre dans B, on définira un masque égal à $00001111_2 = 17$, et on écrira

```
MASK=   17
...
AND     A,# MASK        A     10101010
ADD     A,B             MASK  00001111
...                           00001010    à ajouter à B
```

Le AND est une espèce de passoire qui ne laisse des bits à l'état 1 que là où se trouvent des 1 dans le second opérande.
Souvent on ne laisse passer qu'un seul bit, et les flags GT, LT permettent de savoir si ce bit (en fait le mot binaire complet) est égal à zéro ou non.

L'instruction OR permet de faire le OU de deux mots binaires. Elle est rarement utilisée pour forcer à 1 certains bits, et s'appelle BIS */bit set/* sur certains processeurs (PDP 11, etc.).

```
OR      A,# SBIT        A       01011010
                        SBIT    11000000
                        RESULT  11011010
```

L'instruction XOR est utile dans certaines opérations de contrôle (elle inverse les bits sélectionnés par le masque) et pour remettre à zéro le contenu du registre A.

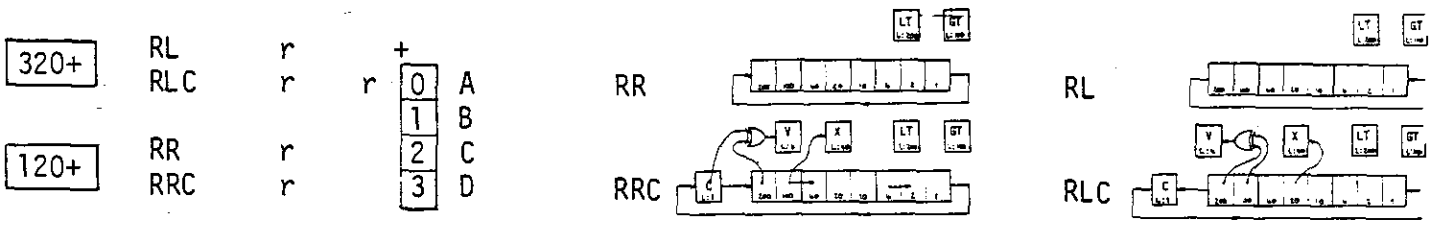
```
        XOR     A,A     A   01100101
40      CLR     A       A   01100101
                    RESULT  00000000
```

La notation CLR A peut être utilisée à la place de XOR A,A pour augmenter la lisibilité du programme.

On a déjà remarqué que l'instruction SUB A,A (si WITHCARRY = Ø) permet aussi de mettre A à zéro, mais SUB A,A modifie les flags C (CARRY), V (OVERFLOW) et X (AUXILIARY CARRY).

13. OPERATIONS DE DECALAGE

Deux instructions de décalage seulement sont disponibles dans le 2650, mais elles sont dédoublées par l'effet du bit de mode WITHCARRY.



Si WITHCARRY = 0. l'instruction ne modifie pas le CARRY, mais charge LT et GT selon la valeur du contenu décalé.

Si WITHCARRY = 1, la rotation inclut le CARRY, qui reçoit le bit de poids fort (RLC) ou de poids faible (RRC).

Les flags V (oVerflow) et X (auXiliary carry) sont aussi modifiés. V prend la valeur 1 si le nombre arithmétique situé dans le registre r change de signe au moment du décalage. X prend la valeur finale du bit de poids $2^5 = 40$.

Pour décaler un mot de 16 bits de 2 positions à droite, situé par exemple dans A et B, on peut écrire le programme suivant:

```
SET     WITHCARRY
RRC     A
RRC     B
RRC     A
RRC     B
```

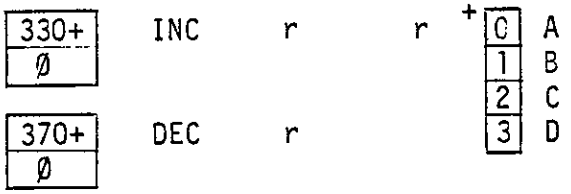
Si l'on veut que les bits introduits par le registre soient nuls, on peut soit masquer ces bits à la fin de l'opération, soit mettre le CARRY à zéro avant chaque décalage.

```
SET     WITHCARRY
RRC     A
RRC     B
RRC     A
RRC     B
AND     A,# 77
```

```
SET     WITHCARRY
CLR     CARRY
RRC     A
RRC     B
CLR     CARRY
RRC     A
RRC     B
```

14. OPERATIONS D'INCREMENTATION ET DE DECREMENTATION

Ces instructions ont deux bytes car elles sont des cas particuliers d'instructions plus générales (étudiées plus loin). Elles ne modifient aucun des flags, contrairement aux instructions équivalentes d'autres processeurs.

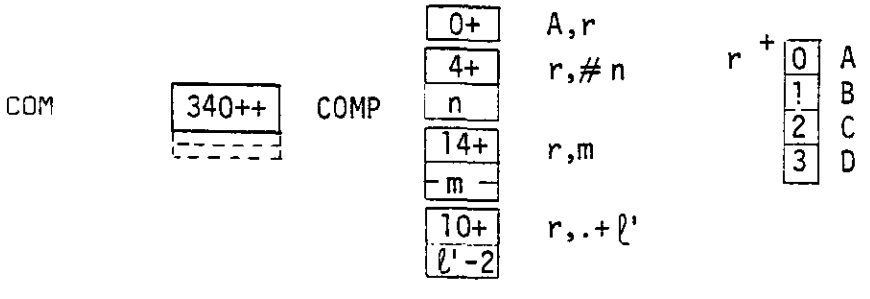


Pour modifier les flags, on a la possibilité d'ajouter ou soustraire 1, ce qui prend la même place.

15. OPERATIONS DE COMPARAISON ET DE TEST

Comparer deux nombres est une opération utile et fréquente. Le Signetics 2650 dispose comme la plupart des processeurs de l'instruction COMPare, avec la différence importante qu'un bit du registre de mode, appelé LOGICOMP (bit de poids $2^1 = 2$) permet de présélectionner si la comparaison concerne des nombres arithmétiques (7 bits + signe), dans le cas où LOGICOMP = 0, ou des nombres "logiques" 8 bits, si LOGICOMP = 1.

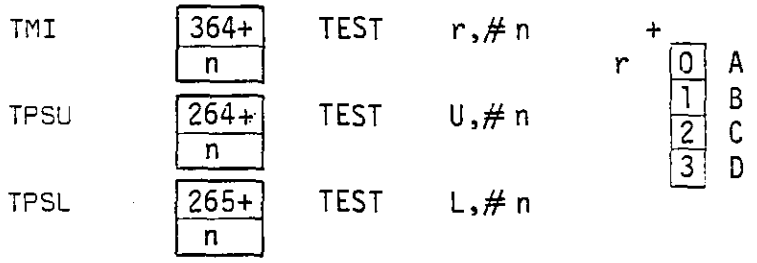
Par exemple si les nombres à comparer soit 342 et 57:
si LOGICOMP = 0, le premier nombre, négatif, est plus petit que le second
si LOGICOMP = 1, le premier nombre est plus grand que le second.



Cette instruction ne modifie que les flags GT (Greater Than) et LT (Lower Than) du registre L.

Les exemples d'illustration de l'instruction COMPare seront suffisamment nombreux pour qu'il ne soit pas nécessaire d'en mettre ici.

Quelques instructions de test existent, mais elles n'agissent que sur les registres A,B,C,D et sur les registres U et L.

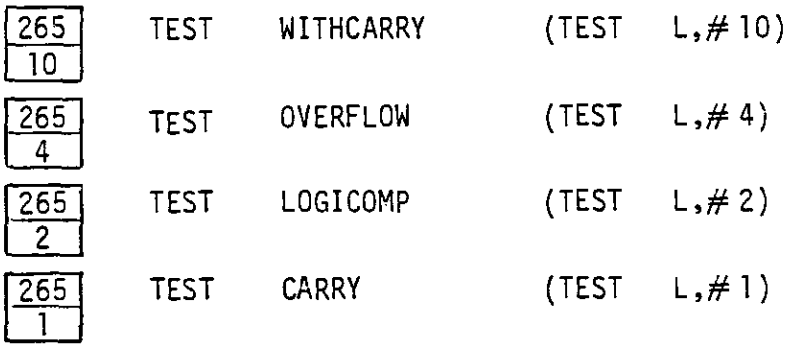


Cette instruction compare le contenu du registre avec le mot binaire n donné dans le 2ème byte de l'instruction et permet de savoir si le registre contient des 1 en face de tous les 1 de n. Le flag GT n'est pas modifié, et le flag LT prend la valeur Ø si tous les bits "sélectionnés" par le nombre n valent 1 */all one AO/*. Par contre LT vaut 1 dès qu'un des bits sélectionnés n'a pas cette valeur */not all one NO/*.

Par exemple, l'instruction TEST A,# 201 permet de savoir si les 2 bits extrêmes du mot (poids $2^0 = 1$ et poids $2^7 = 200$) sont à 1, ce qui veut par exemple dire que l'on a un nombre négatif impair. Si A contient 323 au moment de l'instruction, on aura après exécution LT=0 (All One). Si A contient 324, LT = 1 (Not All One).

Des instructions de saut conditionnel, vues dans le prochain paragraphe, permettent d'agir différemment selon ce résultat.

Les instructions de test des registres de flag sont souvent utilisées pour savoir si l'un des flags est à 1 ou à Ø. La valeur n correspond alors au poids du bit de flag, et l'instruction TEST met LT = 0 (AO) si le flag correspondant est à 1. Un nom particulier a été donné aux instructions de test de flags, pour pouvoir plus facilement les reconnaître dans les listings: elles montrent en effet clairement l'intention du programmeur et évitent un commentaire.



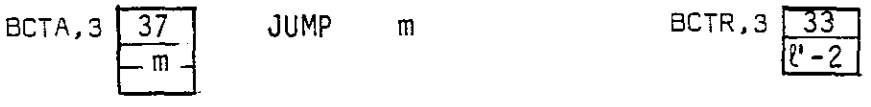
Si l'on veut tester plusieurs de ces variables simultanément, on peut écrire en une seule instruction:

TEST OVERFLOW,CARRY

On aura LT = Ø si les deux flags sont à 1 et LT = 1 (NO) si l'un des deux, ou les deux, est à Ø.

16. INSTRUCTIONS DE SAUT

Les instructions de saut simple permettent de revenir en arrière, passer par-dessus une zone réservée, etc. Les sauts peuvent être absolus ou relatifs.



Comme exemple d'application d'un saut, citons le racommodage */patch/* d'un programme dans lequel on doit insérer une ou deux instructions oubliées.

```
PROG:  100   LOAD   0,#0
       101
       102   LDAD   B,#15
       103                ←—— OUBLI:  CLR     CARRY
       104   ADD    A,#-7  ;-7 est codé par son complément à 2, 371
       105
       106   LDAD   TRUC,A
       107
       110
       111   SUB    A,#15
       112
             ...
```

Plutôt que de réassembler et recharger toute la suite du programme, on peut supprimer quelques instructions et les remettre avec l'instruction oubliée dans la zone de racommodage.

```
PROG:  100   LOAD   0,#0
       101
       102   JUMP   PATCH1
       103
       104
       105   (plus utilisé)
NEXT1: 106   LOAD   TRUC,A
       107
       110
       111   SUB    A,#15
       112
             ...
PATCH1:200   LOAD   B,#15
       201
       202   CLR    CARRY
       203
       204   ADD    A,#-7
       205
       206   JUMP   NEXT1
       207
       210
```

Les instructions de saut conditionnel donnent toute la puissance à la programmation. Selon l'état des flags LT et GT, le saut s'effectue ou non (si la condition n'est pas satisfaite, on continue à l'instruction suivante), avec les possibilités suivantes:

BCTA,0	34 / m	JUMP,EQ m	BCTR,0	30 / ℓ'-2	JUMP,EQ .+ℓ'

Saut si LT=0 et GT=0, c'est-à-dire si l'opération précédente a donné un résultat nul (après un LOAD, une opération, un décalage), si la COMParaison qui précédait a mis en évidence une égalité, ou si le TEST qui précédait a montré que tous les bits sélectionnés étaient à 1 (All One).

BCTA,1	35 / m	JUMP,GT m	BCTR,1	31 / ℓ'-2	JUMP,GT .+ℓ'

Saut si GT = 1 (résultat positif, 1er nombre supérieur au second) */Greater Than/*.

BCTA,2	36 / m	JUMP,LT m JUMP,NO m	BCTR,2	32 / ℓ'-2	JUMP,LT .+ℓ' JUMP,NO .+ℓ'

Saut si LT=1 (résultat négatif, 1er nombre inférieur au second */Lower Than/*, certains bits sélectionnés pas égaux à 1).

BCFA,0	234 / m	JUMP,NE m	BCFR,0	230 / ℓ'-2	JUMP,NE .+ℓ'

Saut si LT=1 ou GT=1 (résultat non nul, différence dans une comparaison */Non Equal/*.

BCFA,1 | 235 | -m- | JUMP,LE m

BCFR,1 | 231 | ℓ'-2 | JUMP,LE .+ℓ'

Saut si GT=0 (résultat négatif ou nul, comparaison inférieure ou supérieure */Lower or Equal/.*

BCFA,2 | 236 | -m- | JUMP,GE m

BCFR,2 | 232 | ℓ'-2 | JUMP,GE .+ℓ'

Saut si LT=0 (résultat positif ou nul, comparaison supérieure ou égale */Greater or Equal/.*

BRNA | 134+ | -m- | JUMP,rNE m

BRNR | 130+ | ℓ'-2 | JUMP,rNE .+ℓ'

r	+
0	A
1	B
2	C
3	D

Saut si le registre r (A,B,C ou D) n'est pas égal à zéro.

L'instruction de saut si l'un de ces registres est égal à zéro n'existe pas. On peut la remplacer par deux instructions:

```
      JUMP,ANE SUITE
      JUMP     NUL    ;Saut en NUL si A=0
SUITE: ...
```

Toutes ces instructions peuvent s'écrire de façon condensée:

BCT
BCF | 30++ | JUMP,t
BRN

0+	.+ℓ'
ℓ-2	(relatif)
4+	m
p	(absolu)
m	

t		
3		Unconditional
0	EQ AO	Equal or equal to zero All selected bits one after TEST instructions
1	GT	Greater than or strictly positive GT=1
2	LT NO	Lower than or negative Not all selected bits one LT=1
200	NE	Non equal or non equal to zero
201	LE	Lower or equal GT=0
202	GE	Greater or equal or positive LT=0
100	ANE	Register A non equal to zero
101	BNE	Register B non equal to zero
102	CNE	Register C non equal to zero
103	DNE	Register D non equal to zero

Comme exemple de programme, considérons d'abord la multiplication de deux nombres entiers binaires de 8 bits, avec test de dépassement de capacité. Le multiplicateur est décompté jusqu'à zéro, avec chaque fois addition du multiplicande dans un registre accumulateur. Le test de dépassement de capacité se fait à chaque opération, et il ne faut pas oublier de tester certaines conditions particulières telles que la multiplication par zéro.

Exemples numériques (en octal, mots de 8 bits)

15 x 5

5	4	3	2	1	0
→ 0	15	32	47	64	=0
+15	+15	+15	+15	+15	opération terminée
15	32	47	64	101	

133x17

17	16	15
→ 0	133	266
+133	+133	+133
133	266	421 ⩾ 400 dépassement de capacité

25 x 0

0 → =0 Résultat = 0

Si les deux nombres à multiplier sont dans les registres B et C, et si le produit cherché doit se trouver dans le registre A, le programme correspondant s'écrit simplement:

```
TITLE   MULT8           ;programme de multiplication 8 bits par additions répétées,
                        ;avec test de dépassement de capacité
;multiplicande:  Registre B
;multiplicateur: Registre C
;résultat:       Registre A, carry bit clear
;en cas de dépassement de capacité, Carry Bit Set à la fin du programme

MULTI:  CLR     CARRY,WITHCARRY ;initialise le bon état
        CLR     A
        JUMP,CNE MUL2           ;saut si C≠0
;résultat nul, effectuer les cycles d'additions répétées
MUL2:   ADD     A,B
        TEST    CARRY
        JUMP,AO NEXT            ;arrêt du calcul si dépassement de capacité
        DEC     C               ;décompte C
        JUMP,CNE MUL2           ;nouveau cycle si C≠0

NEXT:   ---             ;opération à faire ensuite. Pour le test de ce programme isolé,
                        ;mettre par exemple un  LOAD  $4,A   avec la liaison bloquant le
                        ;processeur lorsque cette instruction est exécutée.
```

Le temps d'exécution de ce programme est très variable puisque de 0 à 255 cycles d'addition et test peuvent être effectués. Il y a naturellement avantage à ce que le plus petit des deux nombres soit le multiplicateur. Les instructions suivantes permutent multiplicande et multiplicateur dans le cas où le multiplicande donné dans B est inférieur au multiplicateur dans C.

```
MULT8:  SET     LOGICOMP
        LOAD    A,B
        COMP    A,C
        JUMP,GE MULTI   ;saut si A (donc B) > C
;A < C, permutons
        LOAD    SAVC,C  ;C doit être sauvé en mémoire pour ne pas mo-
        LOAD    B,SAVC  ;difier A qui contient l'ancienne valeur de B
        LOAD    C,A     ;ancienne valeur de B dans C
;suite de la multiplication comme avant
MULTI:
```

Un programme comme celui-là admet encore d'innombrables variantes. Avec des nombres arithmétiques en complément à 2, il faut tester le flag d'"overflow" et non pas le carry après l'addition. En décimal (BCD), la correction s'impose après chaque addition et le décomptage doit naturellement se faire en décimal.

Donnons encore un autre exemple: le programme attend un ordre du clavier, mais n'accepte que les touches 3, 4 et 5. Un = apparaît sur l'affichage pendant l'exécution du programme. La touche acceptée est affichée sur les lampes.

```
.TITLE  ATTEND  ;attend que l'une des 3 touches consécutives soit pressée
FIRST=  3       ;première touche de la série active
LAST=   5       ;dernière touche
EGAL=   110     ;segments à allumer pour le signe "égal"
DIGØ=   Ø       ;adresse du premier digit
CLA=    7       ;adresse du clavier
MCLA=   37      ;masque des touches actives du clavier
FULL=   200     ;bit de nouvelle action sur une touche

ATTEND: LOAD    A,#EGAL         ;attend en affichant le signe =
        LOAD    $DIGØ,A
        LOAD    A,$CLA
        JUMP,GE ATTEND          ;retour tant que le bit FULL est égal à zéro
;une touche 0 - 7 a été pressée, est-ce la bonne ?
        AND     A,#MCLA         ;seuls les 5 bits de droite sont significatifs
        COMP    A,#FIRST
        JUMP,LT ATTEND          ;touches 0, 1, 2 pas acceptées
        COMP    A,#LAST
        JUMP,GT ATTEND          ;touches > 5 pas acceptées
;la touche 3, 4 ou 5 a été pressée. Pour le test, par exemple:
AFF:    LOAD    $4,A            ;affiche le code de la touche sur les lampes
        JUMP    AFF
```

EXERCICES PROPOSES:

- *Calcul de la moyenne de n nombres en mémoire*
- *Calcul de la somme des entiers de 1 à n, la valeur n étant donnée dans un registre*
- *Multiplication de 2 nombres par addition répétée, avec test de dépassement de capacité.*

17. DECOMPTAGE AVEC SAUT

Le Signetics 2650 dispose d'instructions de comptage et décomptage incluant une adresse de saut conditionnel. C'est en fait l'exécution simultanée de deux instructions de base

INC r	DEC r
JUMP,NE adresse	JUMP,NE adresse

que l'on écrit INCJ,NE r,adresse DECJ,NE r,adresse.
L'adresse peut être relative ou absolue.

BIRA	334	INCJ,NE A,m	BIRR	330	INCJ,NE A,.+ℓ'
	m			ℓ'-2	
	335	INCJ,NE B,m		331	INCJ,NE B,.+ℓ'
	m			ℓ'-2	
	336	INCJ,NE C,m		332	INCJ,NE C,.+ℓ'
	m			ℓ'-2	
	337	INCJ,NE D,m		333	INCJ,NE D,.+ℓ'
	m			ℓ'-2	
BDRA	374	DECJ,NE A,m	BDRR	370	DECJ,NE A,.+ℓ'
	m			ℓ'-2	
	375	DECJ,NE B,m		371	DECJ,NE B,.+ℓ'
	m			ℓ'-2	
	376	DECJ,NE C,m		372	DECJ,NE C,.+ℓ'
	m			ℓ'-2	
	377	DECJ,NE D,m		373	DECJ,NE D,.+ℓ'
	m			ℓ'-2	

L'instruction DECJ,NE C,MUL2 aurait pu être utilisée dans le programme de multiplication, avec économie de deux bytes en mémoire.

Les instructions INC r et DEC r (voir p. 23) ne sont en fait que des cas particuliers des instructions ci-dessus, avec saut à l'adresse suivante (.+2) lorsque le registre est différent de zéro. On continue donc de toutes façons à l'instruction suivante.

Les instructions de décomptage avec saut sont très appréciées dans les boucles, de façon générale, et dans les boucles d'attente en particulier. La boucle suivante attend 1 milliseconde, mais la précision dépend de la fréquence de l'impulsion d'horloge du processeur. Avec un réglage à la fréquence de 1,2 MHz (le maximum du processeur est 2,4 MHz), on a la partie de programme suivante

```
                MILLISEC=   133.        ;paramètre de la boucle d'attente
                                        ;133. = 2·64. + 5 = 205 octal
 7
205             LOAD    0,#MILLISEC     ;5 µs
373
                BCLE:   DECJ,NE 0,BCLE  ;7,5 µs
```

Pour obtenir une seconde, on peut exécuter 1000 fois le programme ci-dessus, ce qui nécessite deux boucles de comptage, car un compteur 8 bits est limité à 256. On a alors le programme suivant qui utilise trois boucles imbriquées.

```
MILLISEC=       133.
SEC1=           100.
SEC2=           10.          100. x 10. = 1000.

        LOAD    B,#SEC2
BC1:    LOAD    C,#SEC1
BC2:    LOAD    0,#MILLISEC
BC3:    DECJ,NE 0,BC3
        DECJ,NE C,BC2
        DECJ,NE B,BC1
        ...
```

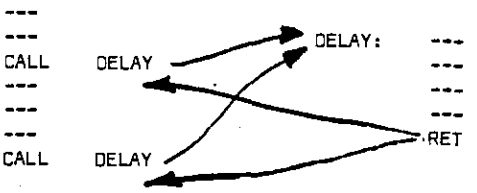
Pour économiser un registre, une routine de 4 ms doit être construite. Des instructions neutres ralentissent la boucle pour parvenir à ce but.

```
QUATREMILLI=    229.
SECONDE=        250.
        LOAD    C,#SECONDE
BC1:    LOAD    0,#QUATREMILLI
BC2:    NOP             ;5 µs
        NOP             ;5 µs
        DECJ,NE 0,BC2   ;7,5 µs
        DECJ,NE C,BC1
        ...
```

18. APPEL DE SOUS PROGRAMMES

Lorsque la même partie de programme doit apparaître à plusieurs endroits du programme (par exemple une attente de quelques millisecondes), il est stupide de la recopier.

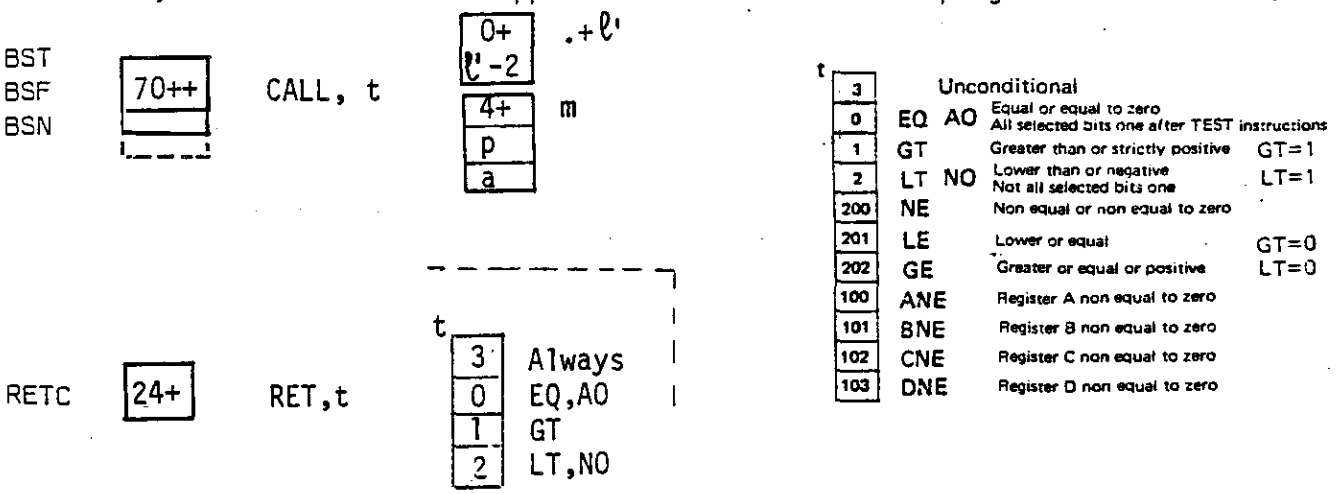
L'instruction CALL permet de sauter à la copie unique de cette partie de programme appelée alors sous-programme */sub-routine/*, avec retour automatique à la suite du programme à la fin du sous-programme, lorsque l'instruction RET */return/* est exécutée.



On ne peut pas mettre une instruction JUMP à la fin du sous-programme, car on retournerait toujours à la même adresse. L'instruction RET va rechercher une adresse, mémorisée dans une registre supplémentaire du processeur lors du CALL qui est l'adresse de l'instruction suivante dans le programme principal.

L'instruction CALL est identique à un JUMP, avec sauvetage de l'adresse de l'instruction suivante (qui se trouve dans le PC */Program Counter/*). On peut avoir des CALL conditionnels et des RETours conditionnels; si la condition n'est pas remplie le programme continue en séquence.

En résumé, les instructions d'appel et de retour de sous-programmes sont les suivantes:



Pour permettre des appels de sous-programmes dans un appel de sous-programme, 8 registres d'adresse de retour existent dans le processeur 2650, formant ce que l'on appelle une pile */stack/* d'adresses de retour.

Un compteur-décompteur de 3 bits pointe le registre contenant la dernière adresse de retour: ce compteur occupe les trois bits de poids faible d'un deuxième registre d'état, appelé U, et le programme suivant permet de vérifier le mécanisme du stack (pile) en fonctionnement pas à pas avec le DAUPHIN 2650.

```
TEST CALL:  0  164     CLR     STACK   ;remet à zéro le compteur de pile
            1    7
            2   22     LOAD    A,U
            3  324     LOAD    $4,A    ;affiche sur les lampes l'état de U.
            4    4                     ;les trois bits de poids fort ont une
                                       ;signification qui sera expliquée plus loin
            5   77     CALL    ROUT1   ;appel routine (position 40). L'adresse de
            6    0                     ;retour (10) est sauvée sur la pile (dans le
            7   40                     ;registre pointé par le compteur
           10   22     LOAD    A,U
           11  324     LOAD    $4,A    ;le compteur de pile est dans le même
           12    4                     ;état qu'avant
           13   77     CALL    ROUT2   ;appel direct de la 2ème routine (position 100)
           14    0
           15  100
           ...
```

```
;premier sous-programme à l'adresse 40
ROUT1:    40   22     LOAD    A,U
          41   324    LOAD    $4,A     ;contrôle visuel de l'état de la pile
          42   4
          43   77     CALL    ROUT2    ;appel imbriqué d'une deuxième routine à
          44   0                       ;adresse 100. L'adresse de retour (46) est
          45   100                     ;mémorisée en 2ème position mémoire sur la pile
          46   22     LOAD    A,U
          47   324    LOAD    $4,A
          50   4
          51   27     RET             ;retour au programme principal
          ...
ROUT2:    100  22     LOAD    A,U      ;contrôle visuel de l'état de la pile
          101  324    LOAD    $4,U
          102  4
          103  27     RET
```

Cette manipulation montre que le compteur de pile compte à chaque CALL et décompte à chaque RET. Jusqu'à 8 appels de sous-programmes peuvent avoir lieu. Il n'est pas nécessaire de mettre le compteur de pile à zéro au début du programme, car la pile est en fait un barillet: le registre de pile 0 suit le registre 7, et la valeur affective n'a pas d'importance, tant que l'on quitte bien toujours un sous-programme par une instruction de RETour.

Comme exemple d'appel de sous-programme, considérons un programme qui lit une ligne genre Telex. Toutes les 20 ms, un "un" ou un "zéro" est transmis sur la ligne. Des "zéro" remplissent les silences, et un premier "un" */start bit/* marque la fin d'un silence et le début d'un mot de 5 bits correspondant à un caractère. Chaque mot est suivi d'un silence (zéro) de 1 bit au moins.

EXEMPLE DE MESSAGE

0 0 0 0 0 0 [1] 1 0 1 0 1 0 [1] 1 1 1 1 1 0 0 0 0 0 [1] 0 0 1 0 0 0 [1] 0 0 0 0 1 0 0 0 0 0

1er caractère 2e car. 3e car. 4e car.

Une routine BIT, non écrite ici car elle dépend de l'interface avec le télex, lit l'état de la ligne et met le CARRY à un si la ligne est à "un", et à zéro si la ligne est à "zéro". Une routine CARACTERE forme un mot de 5 bits une fois que le premier bit (start) a été reconnu. Le programme principal donné en exemple passe par-dessus les silences et attend un caractère spécial de début, par exemple 10101 = 25 octal.

```
.TITLE  TELEX   ;attend un caractère de début
LONG=   5       ;nombre de bits par caractère
CARDEB= 25      ;caractère de début de message
;---    sous-programmes
BIT:    ---     ;selon l'interface
                ;cette routine ne peut modifier que A et B
        RET     ;retour avec CS (carry set) si le bit lu vaut 1
CARACTERE:
        LOAD    C,#0        ;registre caractère
        LOAD    0,#LONG     ;compteur de bits par caractère
CAR2:   CALL    BIT
        RLC     C           ;décalage de Carry dans C
        DECJ,NE 0,CAR2
        RET                 ;retour avec le caractère dans C
;---    Programme
TELEX:  SET     WITHCARRY   ;à cause de l'instruction RLC
TEL2:   CALL    BIT
        TEST    CARRY
        JUMP,ND TEL2        ;saut si Carry à Ø (Not all selected bits One)
; le bit vaut 1; c'est le début d'un caractère
        CALL    CARACTERE
        CALL    BIT
        TEST    CARRY
        JUMP,AD ERROR       ;erreur si le bit lu vaut 1
        LOAD    A,C
        COMP    A,#CARDEB   ;le mot n'ayant que 5 bits, il n'y a pas eu besoin
                            ;de préciser la valeur du bit LOGICOMP
        JUMP,NE TEL2        ;retour pour attente du caractère suivant
;suite du programme lorsque le caractère de début est trouvé
        ---
;action en cas d'erreur (pas de silance à la fin d'un caractère)
ERROR:  ---
        ---
```

Dans cet exemple, les sous-programmes ont été placés devant le programme principal, mais ils peuvent être mis n'importe où. Pour pouvoir utiliser l'adressage relatif, il y a avantage à mettre les sous-programmes à proximité de l'endroit où ils sont appelés le plus souvent. Au moment de l'écriture du programme, lorsque l'analyse de celui-ci est terminée, il faut commencer par écrire les routines pour savoir exactement quels sont les registres et flags modifiés ou utilisés pour le transfert des paramètres.

19. INSTRUCTIONS DIVERSES

L'instruction NOP (code 30D) n'a pas d'effet. Elle est utilisée pour perdre du temps ou réserver de la place pour des instructions supplémentaires ultérieures. En prévoyant 3 NOP successifs, on peut insérer un saut à une adresse quelconque et faire facilement un "patch", c'est-à-dire insérer des instructions manquantes dans une partie inoccupée de la mémoire.

L'instruction WAIT (code 100) sera étudiée ultérieurement avec l'interruption, de même que le RETION,t (retour de routine d'interruption).

L'instruction TRAP de retour au moniteur de contrôle, n'existe pas avec le Signetics. L'insertion de points d'arrêt est un peu plus difficile et nécessite 2 bytes.

Avec le DAUPHIN, on utilise soit l'instruction LOAD $4, A pour bloquer le processeur sur cette instruction et continuer en step, soit l'instruction spéciale

ZBRR | 233 / 0 | JUMP Ø, non encore vue, pour retourner au moniteur en ROM.

Les deux lignes d'entrée et de sortie du processeur sont liées au registre U.
Les instructions suivantes permettent de contrôler ces lignes.

166 / 100	SET	OUTPUT	Agissent sur la pin 40 du processeur (douille 0 de la plaque DAUPHIN).
164 / 100	CLR	OUTPUT	
264 / 100	TEST	OUTPUT	Permet de tester la valeur précédemment assignée à la sortie Condition A0 vraie sortie à 1 Condition N0 vraie sortie à 0
264 / 200	TEST	INPUT	Permet de tester l'entrée, pin 1 du processeur. Cette entrée est inversée (douille $\overline{IN}$) sur la plaque processeur 2650 du DAUPHIN. Condition A0 vraie entrée extérieure à 0 Condition N0 vraie entrée extérieure à 1

EXEMPLE: programme faisant clignoter la lampe de la plaque processeur DAUPHIN à une fréquence visible (grâce à l'insertion de 2 boucles d'attente).

DEBUT:	0	166	SET	OUTPUT
	1	100		
DEB2:	2	370	DECJ,NE	A,DEB2
	3	176		
	4	371	DECJ,NE	B,DEB2
	5	174		
	6	164	CLR	OUTPUT
	7	100		
DEB4:	10	370	DECJ,NE	A,DEB4
	11	176		
	12	371	DECJ,NE	B,DEB4
	13	174		
	14	33	JUMP	DEBUT
	15	162		

La durée du premier clignotement est indéterminée

Variante préférable, car plus claire et lisible:

DEBUT:		SET	OUTPUT
		CALL	DELAY
		CLR	OUTPUT
		CALL	DELAY
		JUMP	DEBUT
DELAY:		DECJ,NE	A,DELAY
		DECJ,NE	B,DELAY
		RET	

Ce programme permet de vérifier si la fréquence du processeur est correcte. Si la fréquence de clignotement est de 1 Hz (60 flashs par minute) la vitesse est la vitesse maximum permise. Il y a avantage à la réduire pour être moins sensible aux variations de tension.

20. ADRESSAGE INDEXE

Quatre modes d'adressage, c'est-à-dire façons de spécifier où se trouve l'information considérée, ont été rencontrées dans les précédentes parties. Dans le mode registre, le No du registre qui contient l'information (0 pour A, 1 pour B, ...) est spécifié dans le code même de l'instruction. Dans le mode immédiat, l'information se trouve dans le 2e byte de l'instruction. Dans le mode absolu, l'adresse complète de la position mémoire contenant l'information figure dans l'instruction. Dans le mode relatif, c'est la différence entre l'adresse courante (adresse contenue dans le compteur d'adresse PC) et l'adresse de la position mémoire contenant l'information qui est donnée dans l'instruction.

Dans l'adressage indexé, l'adresse de l'information se trouve dans un registre du processeur, appelé registre d'index, et l'on écrit LOAD A,(XX) pour dire que le contenu de la position mémoire dont l'adresse est dans XX est transféré dans A. La parenthèse autour de XX indique que le contenu de XX est une adresse; LOAD A,XX voudrait dire que le contenu de XX est transféré dans A, sans passer par une référence mémoire.

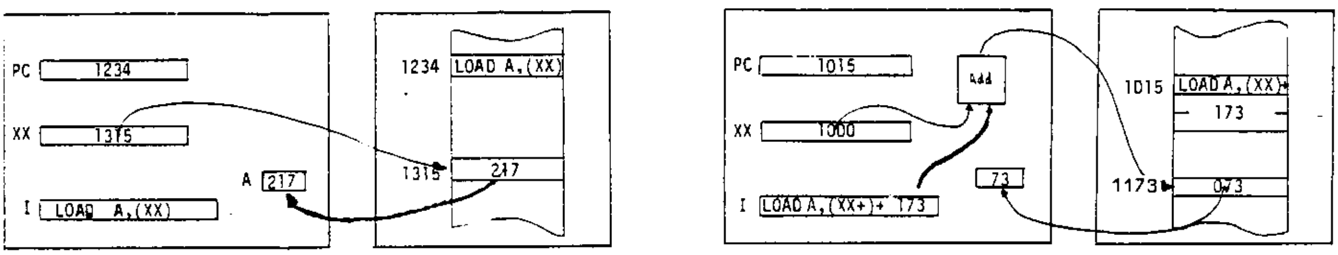


Fig. 7. a) Adressage indexé simple. L'adresse du mot à transférer se trouve dans le registre d'index XX.
b) Adressage indexé avec déplacement. Le déplacement situé dans l'instruction s'ajoute au contenu de XX pour former l'adresse du mot à transférer.

Dans la plupart des processeurs (8080 et dérivés non compris), un déplacement peut être ajouté à l'index, et le déplacement */offset or displacement/* est donné dans l'instruction, permettant l'exécution d'une instruction de type LOAD A,(XX)+VALEUR. Selon le processeur, VALEUR est un nombre de 8 ou 16 bits, arithmétique ou logique, qui est ajouté au contenu de XX au moment de la sélection mémoire.

Avec le Signetics 2650, les registres ABCD peuvent jouer le rôle de registre d'index et un déplacement de 13 bits peut être ajouté. Ce déplacement peut être considéré comme un déplacement arithmétique dans le même chapitre que le compteur d'adresses PC. L'un des 4 registres peut être choisi comme registre d'index, mais les transferts ne peuvent se faire que dans A. On a par exemple les instructions

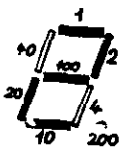
15	LOAD A,(B)+1000	Charge A avec le contenu de la position "contenu de B+1000", soit une position entre 1000 et 1377
142	140+p.2	
0		

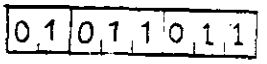
217	ADD A,(D)+TABLE
140	
137	avec TABLE = 137

Le codage de ce mode d'adressage se fait dans le 2e byte de l'instruction et limite les adresses à 13 bits (adresse dans le même chapitre). La valeur 140 (bits 2^6 et 2^5) caractérise ce mode d'adressage, et il faut ajouter l'adresse de la page pour avoir le second byte de l'instruction.

Une application courante de l'adressage indexé est la recherche d'un correspondant à un nombre défini par une table de correspondance. Par exemple avec le DAUPHIN, si le registre A contient 2, le transfert de cette valeur sur l'un des digits de l'affichage ne fait pas apparaître le chiffre 2. A cause du codage des segments, c'est la valeur DEUX = 133 qui doit être affichée, et l'on a la correspondance suivante entre le contenu initial du registre A et le mot à afficher sur les segments:

Valeur	segments	
0	77	ZERO
1	6	UN
2	133	DEUX
3	117	TROIS
4	146	QUATRE
5	155	CINQ
6	175	SIX
7	7	SEPT
8	177	HUIT
9	157	NEUF





La suite des valeurs des segments peut être mise en mémoire, à partir d'une position appelée TABLE, dans l'ordre des valeurs correspondantes. L'instruction LOAD A,(A)+TABLE permet alors de remplacer la valeur octale ou BCD dans A par le code des segments correspondants. Dans le programme donné ci-dessous, on remarque l'instruction AND A,#MCLA (MCLA = 7) qui garantit que le contenu de A est inférieur ou égal à 7. Si cette instruction est oubliée, A contient une valeur supérieure à 7 et le programme va chercher en dehors de la table définie une équivalence non prévue.

```
                        OI=     0       ;adresse du 1er digit
                        CLA=    7       ;adresse du clavier
                        MCLA=   37      ;masque pour ne garder que les 5 bits de droite
DEB:    0       124     LOAD    A,$CLA  ;lecture du clavier
        1       7
        2       104     AND     A,#MCLA;suppression (mise à zéro) des 3 bits
        3       37                      ;de poids fort
        4       14      LOAD    A,(A)+TABLE  ;appel indexé des segments correspondants
        5       140                     ;le rang dans la table est donné par le
        6       14                      ;contenu de A
        7       324     LOAD    $OI,A   ;sortie sur le 1er digit
       10       0
       11       37      JUMP    DEB     ;retour au début
       12       0
       13       0
TABLE: 14       77      .BYTE   77      ;0
       15       6                       ;1
       16       133                     ;2
       17       117                     ;3
       20       146                     ;4
       21       155                     ;5
       22       175                     ;6
       23       7                       ;7
       24       167                     ;A    ;touche 10 seule
       25       174                     ;B    ;touche 10 et touche 1
```

Un autre exemple d'application est la somme d'une liste de nombres, préalablement mémorisés par un autre programme. Le registre d'index D pointe successivement chaque nombre, et le résultat est accumulé dans le registre A. Un compteur C définit le nombre d'additions à effectuer (longueur de la table de nombres).

```
        1124            CLR     WITHCARRY
        1125
SOMTA:  1126            CLR     A       ;somme partielle nulle au début
        1127            LOAD    D,A     ;pointeur au début de la liste
        1130            LOAD    C,#LONGLISTE  ;initialisation du compteur
        1131
SOM2:   1132            ADD     A,(D)+ADLISTE  ;addition d'un nouveau terme
        1133
        1134
        1135            INC     D       ;pointe la position suivante
        1136
        1137            DECJ,NE C,SOM2  ;décompte pour le test de fin
        1140
        1141            JUMP    0       ;retour au moniteur ou suite du programme
        1142

ADLISTE:1233            .BLKB   LONGLISTE       ;réserve une zone en mémoire.
                                                ;pour le test du programme,
                                                ;charger directement des valeurs
```

Il faut remarquer que ce programme ne prévoit pas de test de dépassement de capacité. Si les nombres sont logiques (positifs 8 bits), les deux instructions

```
TEST     CARRY
JUMP,AO  ERROR
```

suivant l'instruction d'addition détectent un dépassement de capacité. Si les nombres sont arithmétiques en complément à 2, il faut écrire d'abord un TEST OVERFLOW.

On remarque dans ce programme la succession des instructions

```
ADD      A,(D)+ADLISTE
INC      D
```

Cette succession étant fréquente, Signetics a prévu deux modes d'adressage supplémentaires, dans lesquels le registre d'index est incrémenté ou décrémenté automatiquement à chaque exécution. En fait, le comptage ou décomptage est effectué avant le transfert, et on a les définitions suivantes:

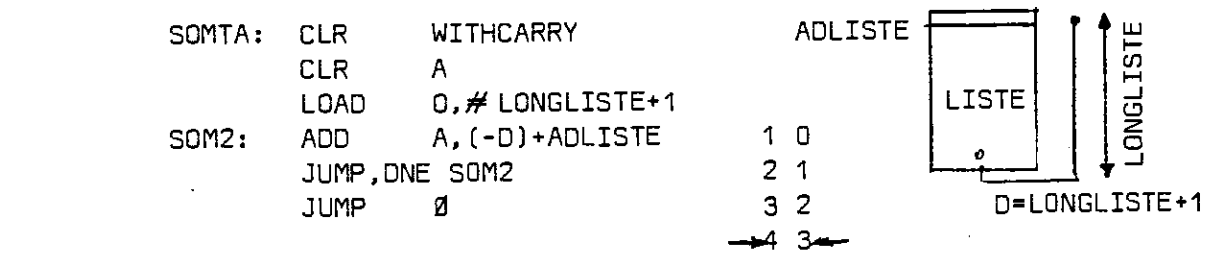
217 42 233	valeur 40 à ajouter à la page	ADD A,(+D)+ADLISTE ADLISTE=1233 (p. 2, adresse 233)	équivalent à	INC D ADD A,(D)+ADLISTE
217 102 233	valeur 100 à ajouter à la page	ADD A,(-D)+ADLISTE veut dire que D est diminué de 1 avant le transfert	équivalent à	DEC D ADD A,(D)+ADLISTE

Avec cette instruction, le programme précédent s'écrit:

```
SOMTA:  CLR      WITHCARRY
        CLR      A
        LOAD     D,A
        LOAD     C,# LONGLISTE
SOM2:   ADD      A,(+D)+ADLISTE-1
        DECJ,NE  C,SOM2
        JUMP     Ø
```

Il peut encore se simplifier si la liste est parcourue en sens inverse

```
SOMTA:  CLR      WITHCARRY
        CLR      A
        LOAD     D,# LONGLISTE+1
SOM2:   ADD      A,(-D)+ADLISTE
        JUMP,DNE SOM2
        JUMP     Ø
```



En écrivant ce genre de programme, une très grande attention doit être donnée à l'initialisation et au test de fin pour éviter des erreurs.

L'adressage indexé existe aussi pour les instructions de saut et d'appel de sous-programme, mais seul le registre D peut être utilisé comme regsitre d'index. Comme exemple d'application, le programme suivant montre comment on peut attendre que l'utilisateur tape sur une touche du clavier, et sauter à 4 adresses différentes (dans un même groupe de 256 positions successives) selon que l'on a pesé sur 0, 1, 2 ou 3.

```
            CLA=    7
            MORD=   3       ;masque pour les touches 0,1,2,3
ATT:        LOAD    A,$CLA  ;lecture du clavier
            JUMP,GE ATT     ;retour tant que le bit de poids fort (FULL=200),
            AND     A,# MORD;qui est aussi le bit de signe, est à zéro
            LOAD    A,(A)+TADRORES
            LOAD    D,A
            JUMP    (D)+ORORE
TADRORES:   .BYTE   Ø
            .BYTE   ORORE1 - ORORE
            .BYTE   ORORE2 - ORORE
            .BYTE   ORORE3 - ORORE
ORORE:      ---
ORORE1:     - -
```

21. ADRESSAGE INDIRECT

L'adressage indirect permet de placer en mémoire plutôt que dans le programme l'adresse d'une position dont le contenu doit être transféré.

Le programme contient l'adresse d'une position (en fait deux positions consécutives) qui contient l'adresse de la position concernée, d'où le terme d'adressage indirect. Le symbole @ (prononcer "indirect" ou "ät") caractérise l'adressage indirect.

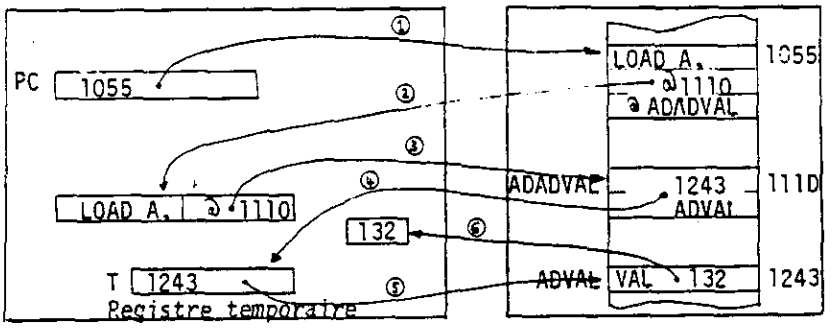


Fig. 8. Exécution de l'instruction LOAD A,@ADADVAL. *Les transferts sont numérotés dans l'ordre de leur exécution.*

L'adressage indirect peut se combiner avec l'adressage relatif et absolu. Dans les deux cas, le bit de poids fort du 2ème byte de l'instruction (valeur octale 200) caractérise l'indirection.

L'adressage indirect est surtout intéressant dans les programmes complexes, lorsque le même programme doit agir sur des groupes de données dont l'emplacement varie en mémoire. Par exemple, si une partie de programme doit additionner un nombre dont l'emplacement dépend d'un autre programme exécuté auparavant, on ne peut pas écrire ADD A,ADNBRE car ADNBRE est une adresse fixe. Par contre, on peut écrire ADD A,@ADADNBRE où ADADNBRE est l'adresse, préparée par le programme précédent, qui contient l'adresse du nombre à additionner.

Le programme précédemment vu d'addition d'une liste de nombres peut être écrit en utilisant l'adressage indirect plutôt que l'adressage indexé, en mettant dans une position mémoire le registre d'index.

```
SOMTA:  CLR      WITHCARRY
        CLR      A
       *LOAD     INDEX,# ADLISTE
        LOAD     C,# LONGLISTE

SOM2:   ADD      A,@INDEX
       *INC      INDEX
        DECJ,NE  C,SOM2
        JUMP     Ø              ;retour au moniteur
INDEX:  .WORD    Ø
ADLISTE:.BLKB    LONGLISTE
```

La position INDEX contient l'adresse du nombre pointé dans la liste. Elle est initialisée ici par la macroinstruction LOAD INDEX,# ADLISTE, qui n'existe pas dans le 2650, car elle transfère la valeur ADLISTE (adresse 16 bits) dans la position mémoire INDEX et la position suivante; il faut 4 instructions du 2650 pour faire ce transfert, en modifiant l'un des regsitres du processeur.

```
LOAD    B,#ADLISTE/400  ;transfère les 8 bits de poids fort (400 octal=2^8)
LOAD    INDEX,B
LOAD    B,# ADLISTE     ;transfère les 8 bits de poids faible
LOAD    INDEX+1,B
```

De même, l'instruction INC INDEX n'existe pas et il faut ajouter 1 dans ce mot mémoire de 16 bits par le groupe d'instructions

```
        LOAD    B,INDEX+1       ;ajoute 1 dans les poids faibles
        ADD     B,# 1
        LOAD    INDEX+1,B
        JUMP,NE SDM4            ;si le résultat des poids faibles est Ø, il faut
        LOAD    B,INDEX         ;ajouter 1 dans les poids forts
        ADD     B,# 1
        LOAD    INDEX,B
SDM4:   instruction suivante
```

Le programme ainsi écrit est plus long que le programme utilisant l'adressage indexé, mais il permet d'ajouter des listes de plus de 256. nombres si le compteur est dédoublé.

22.ADRESSAGE INDEXE INDIRECTEMENT

L'adressage indexé réserve une place pour le bit d'indirection, et l'on peut trouver les instructions

```
LOAD    A,(D)+ @ADTABLE
```

17
355
142

340+15

ADTABLE= 6 542
page 15 adresse 142
(2x6+1)

```
LOAD    A,(+D)+@ADTABLE
```

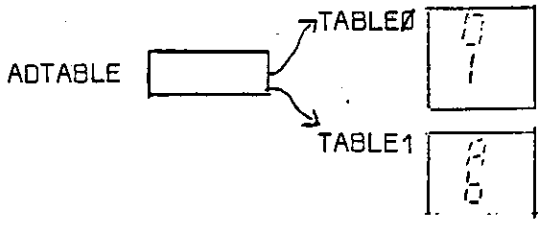
17
257
142

240+15

dont l'effet est d'ajouter le contenu de D (éventuellement pré-incrémenté ou pré-décrémenté) au contenu de la position dont l'adresse est dans la position ADTABLE.

Comme exemple d'application, on peut imaginer que le programme d'affichage de segments doit afficher tantôt des chiffres en correspondance aux valeurs 0,1,2,3,... tantôt des lettres ou signes spéciaux. Le sous-programme d'affichage suivant permet ces affichages variés, le programme principal modifiant ADTABLE chaque fois qu'un autre ensemble de caractères doit être affiché.

```
DISPLAY:        AND     A,# MCLA
                LOAD    A,(A)+ @ADTABLE
                LOAD    $DIGØ,A
                RET

                ...     programme

                Instructions équivalentes à  LOAD ADTABLE,# TABLEØ

                CALL    DISPLAY         ;affiche des chiffres
                ...

                Instructions équivalentes à  LOAD ADTABLE,# TABLE1

                CALL    DISPLAY         ;affiche des lettres
                ...

ADTABLE:        .WORDØ                  ;sera initialisé par le programme
TABLEØ:         .BYTE   ZERO,UN,DEUX,TROIS,QUATRE,CINQ,...
TABLE1:         .BYTE   LETA,LETB,LETC,LETD,...
```



23. INSTRUCTIONS SPECIALES D'ENTREE-SORTIE

Quatre instructions ont été prévues dans le 2650 pour la sélection de 4 périphériques privilégiés, décodés à partir des bits d'adresse 2^{13} et 2^{14}. Ces instructions n'ont pas d'application dans le DAUPHIN, qui n'incorpore pas les circuits de décodage permettant de tirer parti de ces instructions.

Elles permettent toutefois de réaliser le programme de démonstration le plus simple du DAUPHIN: enlever la plaque mémoire, mettre sur ADFLOT (adresses libres), WRITE avec le mot 60 et sur RUN. Le processeur exécute l'instruction 60 (LOAD A,$CTRL), c'est-à-dire lit le périphérique dont l'adresse est dans le compteur d'adresses. Tous les périphériques sont sélectionnés et les lampes d'adresse clignotent.

24. INTERRUPTION

Lors d'une demande d'interruption, par la ligne $\overline{\text{INTREQ}}$ du processeur (actif à l'état Ø), si le processeur a été mis dans l'état ION (interrupt on), il exécute à la fin de l'instruction en cours non pas l'instruction suivante, mais une instruction d'appel de sous-programme, le sous-programme étant appelé dans ce cas routine d'interruption.

De façon interne, le code 273 (CALL Ø+ℓ') est forcé dans le registre d'instruction et le processeur lit sur le bus le second byte de cette instruction en générant une impulsion négative $\overline{\text{INTACK}}$ */interrupt acknowledge/*. Le périphérique demandant l'interruption est sensé reconnaître l'impulsion INTACK et placer sur le bus l'adresse de la routine d'interruption (ou l'adresse de la position qui contient l'adresse de cette routine si l'adrewsage indirect est utilisé), cette adresse se trouvant entre 0 et 77 ou 0 et -100 (1777700).

La plaque 2650 du DAUPHIN amène sur 2 douilles les signaux $\overline{\text{INTREQ}}$ (IR) et $\overline{\text{INTACK}}$ (IA) Pour des expériences on peut lier $\overline{\text{INTREQ}}$ au 0V pour faire une demande d'interruption, et lier $\overline{\text{INTACK}}$ à la douille $\overline{\text{WRITE}}$ se trouvant vers le commutateur WRITE (laisser ce commutateur sur DAFLOT). Lorsque $\overline{\text{INTACK}}$ est actif, l'état des interrupteurs DATA est lu et il suffit d'y placer le vecteur d'interruption pour vérifier comment le processeur l'interprète.

Au moment de l'interruption, le processeur se met automatiquement en IOF *interrupt off/*. Lors du retour de la routine d'interruption, il faut en général resensibiliser le processeur aux interruptions. L'instruction RETION permet de simultanément revenir sur l'interruption et continuer le programme au point interrompu et se remettre en ION.

La gestion correcte des interruptions pose plusieurs problèmes délicats. En particulier, il ne faut pas oublier de sauver l'état de tous les registres utilisés par la routine d'interruption, y compris U et L, si l'on veut pouvoir continuer l'exécution correctement après le service d'interruption. Pour pallier certains points faibles de cette routine de sauvetage, Signetics a prévu d'ajouter de nouvelles instructions dans le 2650-B, qui sera à utiliser de préférence dans les applications avec interruption.

L'expérience en programmation s'acquiert en grande partie en étudiant les programmes écrits par d'autres personnes. Ce ne sont pas toujours des modèles, car les personnes ayant beaucoup d'expérience sont rares, mais il y a toujours quelque bonne idée à reprendre.

ELEclub, le journal des clubs d'électronique du GESO et de MICROCLUB (club des adeptes du microprocesseur), publie régulièrement des exemples de programmes pour DAUPHIN, Pour s'abonner, s'adresser à Rédaction ELEclub, Rte de Prilly 3, 1008 Lausanne.

Pour profiter également des exemples de programmes fournis par SIGNETICS (AS52: General Delay Routines; AS50 Input/Output; AS53: Binary Arithmetic routines; AS54: Conversion routines; Interrupt Save and Restore Routine using the 2650A; AS55 Fixed Point Decimal Arithmetic), la liste donnée en dernière page facilite la conversion.

ANNEXE : CONVERSION DES PROGRAMMES

Le tableau ci-dessous permet de convertir le code d'une instruction originale Signetics

Signetics	CALM										
ADDA ADDI ADDR ADDZ	ADD ou AODC	BST-	CALL,(EQ,GT,LT,un)	LPSU	LOAD U,A	SPSL	LOAD A,L
AND-	AND	BSX-	CALL (D)+	NOP	NOP	SPSU	LOAD A,U
BCF-	JUMP,(NE,LE,GE,un.)	BX-	JUMP (D)+	PPSL	OR L,#	STR-	LOAD m,r ou LOAD (r)+m,A
BCT-	JUMP,(EQ,GT,LT,un)	COM-	COMP	PPSU	OR U,#	SUB-	SUB ou SUBB
BDR-	DECJ,NE	CPSL	BIC L,#	REDC	LOAD r,SCTRL	TMI	TEST r,#n
BIR-	INCJ,NE	CPSU	BIC U,#	REDD	LOAD r,SDATA	TPSL	TEST L,#n
BRN-	JUMP,rNE	DAR	DA	REDE	LOAD r,Sn	TPSU	TEST U,#n
BSF-	CALL,(NE,LE,GE,un)	EDR-	XOR	REDC	RET,(EQ,GT,LT,un)	WRTC	LOAD SCTRL,r
BSN-	CALL,rNE	HALT	WAIT	RETE	RETION,(EQ,GT,LT,un)	WRTD	LOAD SDATA,r
		IDR-	DR	RRL	RL ou RLC	WRTE	LOAD Sn,r
		LDD-	LOAD	RRR	RR ou RRC	ZBRR	JUMP Ø+ℓ
		LPSL	LOAD L,A			ZBSR	CALL Ø+ℓ

Les symboles suivants sont équivalents

	Signetics			Mnémo-nic			
CONDITIONS:	0	Z	ON	AO	EQ	NE	ANE
	1	P			GT	LE	BNE
	2	N		NO	LT	GE	CNE
	3	UN					DNE

REGISTRES:	0	RØ	A
	1	R1	B
	2	R2	C
	3	R3	D

Conversion Hexadecimal-octal

Hexaines	octal	Unités	octal
0-	0+	0	0
1-	20+	1	1
2-	40+	2	2
3-	60+	3	3
4-	100+	4	4
5-	120+	5	5
6-	140+	6	6
7-	160+	7	7
8-	200+	8	10
9-	220+	9	11
A-	240+	A	12
B-	260+	B	13
C-	300+	C	14
D-	320+	D	15
E-	340+	E	16
F-	360+	F	17

La syntaxe générale des instructions du Signetics apparaît dans les instructions suivantes:

```
EORZ    RØ                          XOR     A,A     (CLR A)
   ↑
   register zero addressing

ANDI,3  H'4A'                       AND     D,# 112
   ↑ ↑    ↑
   | |    valeur hexadécimale 4A
   | registre No 3  c'est-à-dire D
   immédiat

BCTR,Ø  LABEL                       JUMP,EQ LABEL
   ↑ ↑
   | condition Ø, c'est-à-dire EQ
   relatif

SUBA,RØ  BINN,R2,-                  SUB     A, (-C)+ ω BINN
   ↑ ↑      ↑
   | |      indexé pré-décrémenté
   | registre A
   absolu
```