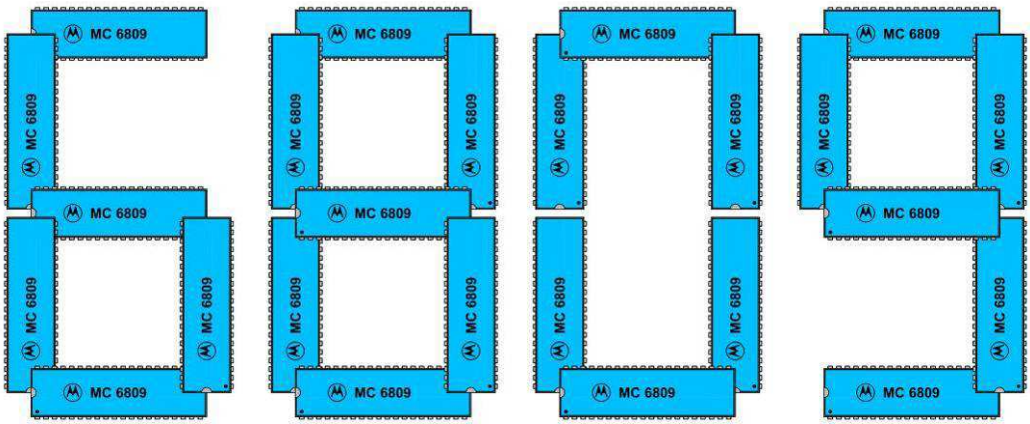
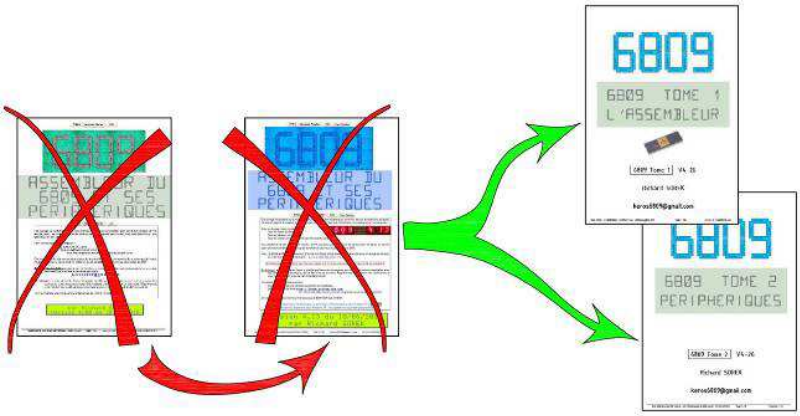


6809 TOME 1
L'ASSEMBLEUR



6809 Tome 1 V4-26

Richard SOREK

keros6809@gmail.com

PRÉFACE

Cet ouvrage de synthèse sur le microprocesseur 6809 (tome 1) et de ses périphériques (tome 2), est le fruit de nombreuses années de travail, de modifications, de mise en page et de création durant mes soirées, mes nuits d'insomnies, mes week-ends et mes vacances.

Dans cet ouvrage (sans exception) : Tous **les textes** ont été **saisis**
Tous **les tableaux** ont été **créés**
Tous **les croquis** ont été **dessinés** par mes propres soins.

Après avoir recherché de la documentation sur le 6809 sur Internet, je me suis vite rendu compte que les documents que j'ai eu l'occasion de voir étaient trop succincts, incomplets et souvent avec quelques petites erreurs.

C'est pour cela que j'ai créé ces documents, ils sont destinés à la compréhension pragmatique et didactique de l'assembleur du 6809 (tome 1) et de ses périphériques (tome 2). Ce travail fût guidé par l'idée d'avoir une documentation précise, pragmatique, détaillée et vivante et surtout de faire partager ces informations gratuitement pour un usage uniquement personnel.

Dans la version précédente de cet ouvrage, j'avais mis une multitude de liens hypertextes. Suite à un bug de Word, j'ai eu énormément de difficulté a récupérer le texte. J'ai donc été contrait de revoir complètement la mise en page et de supprimer à mon grand regret les liens hypertextes. J'ai également scinder l'ancien ouvrage en deux tomes (**Tome 1** l'assembleur, **Tome 2** les périphériques).

Mes propres cours datant de mon passage à l'Ecoles des Mines de DOUAI 59500 ont été complémentés et comparés aux informations trouvées dans les livres suivants :

- Ouvrage 01 : L'ASSEMBLEUR FACILE DU 6809 de François BERNARD de 1984
- Ouvrage 02 : LE MICROPROCESSEUR 6809 – SES PERIPHERIQUES ET LE PROCESSEUR GRAPHIQUE 9355-66 de Claude DARDANNE de 1991 neuvième édition
- Ouvrage 03 : PROGRAMMATION DU 6809 de Rodnay ZAKS de 1983 (édition SYBEX)
- Ouvrage 04 : MICROPROCESSEURS : DU 6800 AU 6809 MODES D'INTERFACE de Gérard REVELLIN
- Ouvrage 05 : ETUDES AUTOUR DU 6809 (CONSTRUCTIONS ET LOGICIELS) de Claude VICIDOMINI (2ième édition du hors série de la revue LED)
- Ouvrage 06 : PROGRAMMATION EN ASSEMBLEUR 6809 de BUI MINH DUC édition EYROLLES
- Ouvrage 07 : LE MICROPROCESSEUR 6809 DE MOTOROLA, partie 3 chapitre 4.2.1
- Ouvrage 08 : ASSEMBLEUR ET PÉRIPHÉRIQUE DES MO5 ET TO7/70 de Frédéric Blanc et de François Normand
- Ouvrage 09 : MANUEL DE L'ASSEMBLEUR 6809 DU TO7 de Michel Weissgerber
- Ouvrage 10 : MANUEL DE L'ASSEMBLEUR 6809 DU MO5 de Michel Weissgerber

Je précise que ce document n'a aucun et n'aura jamais de caractère commercial.
Ce document est à l'attention de quelques de personnes et uniquement pour un usage personnel.

Si vous découvrez des erreurs, des fautes d'orthographes ou encore mieux si vous souhaitez m'apporter vos améliorations, merci de me les envoyer par mail. Je me ferai un devoir et un plaisir d'en apporter la correction ou d'en faire les ajouts.

Je vous demande d'avoir la grande gentillesse de me rapporter par mail vos critiques, vos appréciations, vos corrections éventuelles et/ou vos compléments d'informations, afin de faire vivre ce document.
En fonction des modifications, je vous transmettrai en retour de mail, mes dernières versions.

Un petit message me ferai également un grand plaisir.

Je souhaite remercier chaleureusement :
- **Jacques BRIGAUD** pour ces corrections sur l'assembleur.
- Les modérateurs du site http://forum.system-cfg.com

Un grand merci également aux sites suivants pour avoir mis mes documents en ligne.
colorcomputerrchive.com
studocu.com
pdfcoffee.com
asm.developpez.com
kswichit.net

N'hésitez pas à me contacter par mail pour savoir si il n'y a pas une version plus récente de mes documents.
Je souhaite une bonne lecture aux amoureux du 6809 et de ses périphériques. **R i c h a r d S O R E K**

Pour votre information, j'ai développé un Assembleur-Désassembleur pour le 6809 c'est le **P30RS09**
Il fonctionne pour **l'instant** uniquement sous Windows XP. Il est également disponible gratuitement.

INDEX

A

B

C

D

E

F

G

. . . . . .

H

I

J

K

. . . . . .

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

SOMMAIRE

CONVENTION D'ECRITURE

FIRQ | ≡ $\overline{\textbf{FIRQ}}$

≡ équivalent)

{ A **}** Contenu du registre A.

{ $F083 **}** Contenu de la case mémoire ayant pour adresse $F083.

ABREVIATIONS

ASCII	American Code For Information Interchange
EBCDIC	Extended Binary Coded Decinal Interchange Code (code utilisé par les machines IBM 360/370)
FIFO	First In First Out (Pile Fifo, premier entré dans la pile, premier sorti)
LIFO	Last In First Out (Pile Fifo, dernier entré dans la pile, premier sorti)
LSB	(Least Significant Byte) Bit le moins significatif
MSB	(Most Significant Byte) Bit le plus significatif

Exemple d'écriture

Préfixe	Nombre	Suffixe	Base	Plage	Caractères admis
..... ou **&**.....	Décimaux	.....T	10	[0 .... à .... 65 535]	0,1,2,3,4,5,6,7,8,9
@ ..... ****.....	Octal	.....Q	8	[0 .... à .... 0017 7777]	0,1,2,3,4,5,6,7
$	Hexadécimal	.....H	16	[0 .... à .... FFFF]	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
%	Binaire	.....B	2		0,1

	Déci	Octal	Hexa	Binaire
8 bits	0	@ 0	$ 00	% 0000 0000
	255	@ 377	$ FF	% 1111 1111
16 bits	256	@ 0000 0400	$ 0100	% 0000 0001 0000 0000
	65 535	@ 0017 7777	$ FFFF	% 1111 1111 1111 1111
32 bits	65 536	@ 0000 0020 0000	$ 0001 0000	
	4 294 967 295	@ 0377 7777 7777	$ FFFF FFFF	

Système de numération OCTAL

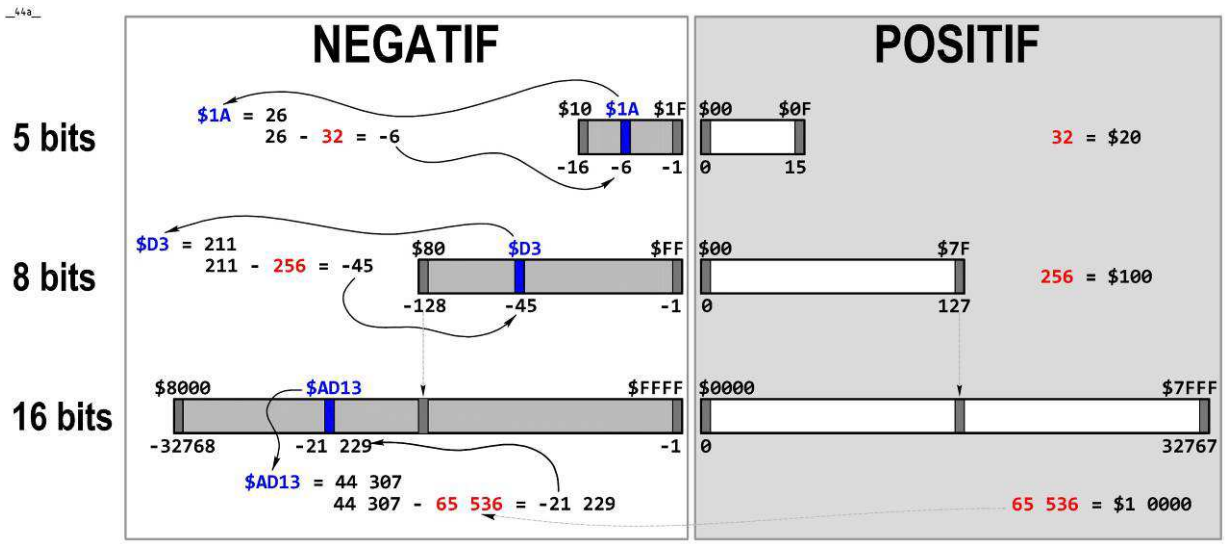
Le système de numération octal est le système de numération de base 8, et utilise les chiffres de 0 à 7.
Le système octal est quelquefois utilisé en calcul à la place de l'hexadécimal.
Il possède le double avantage de ne pas requérir de symbole supplémentaire pour ses chiffres et d'être une puissance de deux pour pouvoir grouper les chiffres.

Système de numération HEXADÉCIMAL

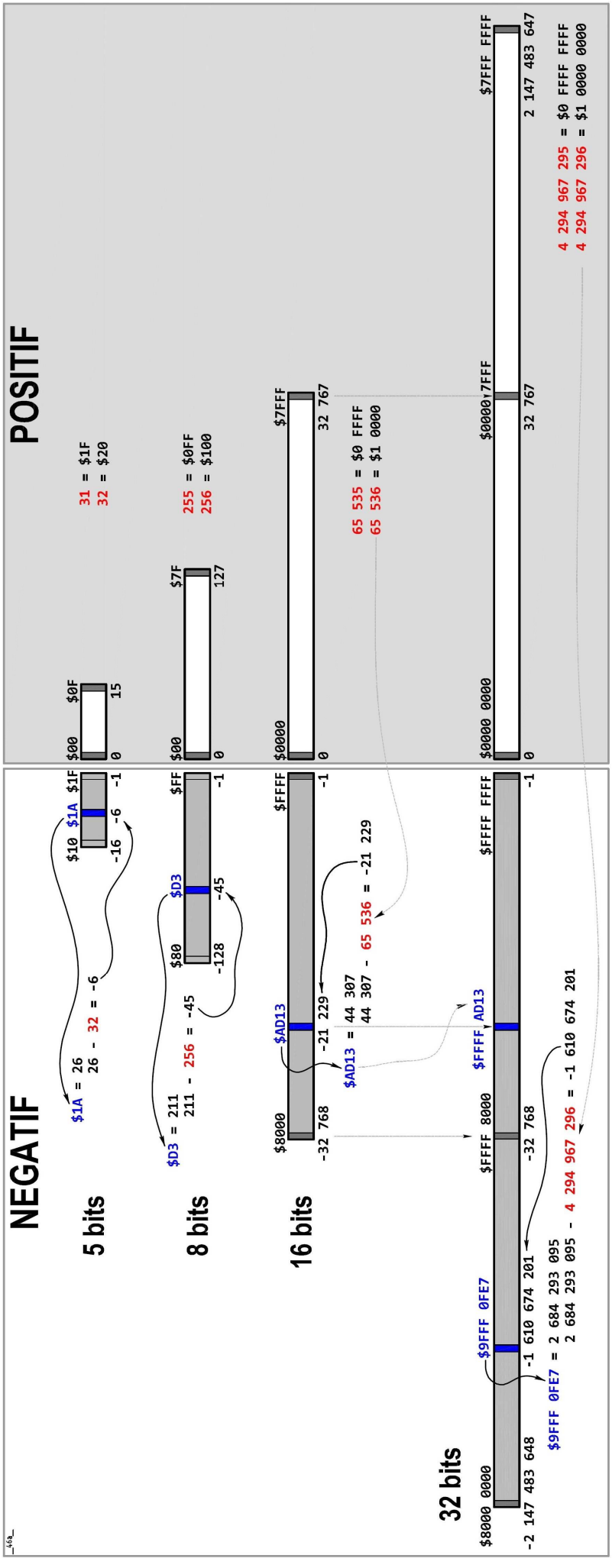
Le système hexadécimal est un système en base 16. Il utilise ainsi 16 symboles, les chiffres de 0 à 9 puis les lettres de A à F. Chaque chiffre hexadécimal correspond exactement à quatre chiffres binaires (ou bits).
Utilisé en électronique numérique et en informatique.
Il permet un compromis entre le code binaire et une base de numération pratique à utiliser

Nombres Signés sur 5, 8 ou 16 Bits

Nombres signés ou encore appelés Codage en Complément à 2



Nombres Signés sur 5, 8, 16 ou 32 Bits



Notion d'Addition sur les nombres Binaires

Exemple : Addition de $05 et $17 **Rappel en binaire** 0+0=0 0+1=1 1+0=1 1+1=10

```
    $05          05          0000 0101
  + $17        + 23        + 0001 0111
---------    ------      -------------
  = $1C        = 28        = 0001 1100    = $1C
```

Notion de Soustraction sur les nombres Binaires

Un microprocesseur ne sait pas faire de soustraction, il ne fait que des additions.
En arithmétique décimal : on additionne au premier l'inverse du second. `13 – 10 ≡ 13 + ( -10 )`

En arithmétique binaire on additionne avec le complément à deux `$20 - $15 ≡ $20 + ($-15)`
(≡ équivalent)

Représentation des Nombres Négatifs

Considérons l'opération 0 – 1 = -1

```
    0      0000  0000
  - 1      0000  0001
-----      ----------
= - 1      1111  1111    =  $FF   ( -1 sera représenté par $FF )
```

Considérons l'opération -1 – 1 = -2

```
   -1      1111  1111
   -1      0000  0001
-----      ----------
= - 2      1111  1110    =  $FE   ( -2 sera représenté par $FE )
```

Nombres NON Signés

Les nombres non signés vont de :

[0..... à .....+31]	pour le 5 bits	**[$ 00 .....à .....$ 1F]**
[0..... à .....+255]	pour le 8 bits	**[$ 00 .....à .....$ FF]**
[0..... à .....+65 535]	pour le 16 bits	**[$ 0000 .....à .....$ FFFF]**
[0..... à .....+4 294 967 265]	pour le 16 bits	**[$ 0000 0000 .....à .....$ FFFF FFFF]**

N'ont pas de rôle dans une représentation non signée :
- Le bit 7 (en 8 bits)
- Le bit 15 (en 16 bits)
- Le bit 31 (en 32 bits)

Nombres Signés sur 5 Bits (notation en complément à 2)

Les nombres signés 8 bits vont de **[-16..... à .....+15]**

Le bit de poids fort (bit 4) sert de signe :
- **-- Bit 4 = 0** Le nombre est positif. Valeurs entre $00 et $0F
- **-- Bit 4 = 1** Le nombre est négatif. Valeurs entre $10 et $1F

```
            0 0000      0  $00
16          0 0001      1  $01
octets      0 ....      .  .
Positifs    0 ....      .  .
            0 1111     15  $0F
-------------------------------------------------------------
            1 0000    -16  $10        -16+32=16     16=$10
16          1 ....      .  .             .           .
octets      1 ....      .  .             .           .
Négatifs    1 ....      .  .             .           .
            1 1111     -1  $1F         -1+32=31     31=$1F
```

Nombres Signés sur 8 Bits (notation en complément à 2)

Les nombres signés 8 bits vont de **[-128..... à .....+127]**

Le bit de poids fort (bit 7) sert de signe :
- **-- Bit 7 = 0** Le nombre est positif. Valeurs entre $00 et $7F
- **-- Bit 7 = 1** Le nombre est négatif. Valeurs entre $80 et $FF

```
            0000 0000      0  $00
128         0000 0001      1  $01
octets      0... ....      .  .
Positifs    0... ....      .  .
            0111 1111   +127  $7F
-------------------------------------------------------------
            1000 0000   -128  $80    -128+256=128   128=$80
128         1... ....      .  .           .           .
octets      1... ....      .  .           .           .            -3 = 253-256
Négatifs    1... ....     -2  $FE         .           .            -2 = 254-256
            1111 1111     -1  $FF     -1+256=255    255=$FF        -1 = 255-256
```

Nombres Signés sur 16 Bits (notation en complément à 2)

Les nombres signés 16 bits vont de **[-32 768..... à .....+32 767]**

Le bit de poids fort (bit 15) sert de signe :
- **-- Bit 15 = 0** Le nombre est positif. Valeurs entre $0000 et $7FFF
- **-- Bit 15 = 1** Le nombre est négatif. Valeurs entre $8000 et $FFFF

```
            00000000 00000000       0  $0000
32 Ko       00000001 00000001       1  $0001
octets      0....... ........       .  .
Positifs    0....... ........       .  .
            01111111 11111111  +32767  $7FFF
-------------------------------------------------------------
            10000000 00000000  -32768  $8000   -32768+65536=32767 = $8000
32 Ko       1....... ........       .  .                  .
octets      1....... ........       .  .                  .
Négatifs    1....... ........       .  .                  .
            11111111 11111110      -2  $FFFE
            11111111 11111111      -1  $FFFF       -1+65536=65535 = $FFFF
```

Exemple en partant d'un nombre en HEXA

```
$FF46 = 65350        65350 – 65536 = -186     $FF46 = -186
```

Tableau des Nombres Signés sur 8 Bits

-1 $FF	-17 $EF	-33 $DF	-49 $CF	-65 $BF	-81 $AF	-97 $9F	-113 $8F
-2 $FE	-18 $EE	-34 $DE	-50 $CE	-66 $BE	-82 $AE	-98 $9E	-114 $8E
-3 $FD	-19 $ED	-35 $DD	-51 $CD	-67 $BD	-83 $AD	-99 $9D	-115 $8D
-4 $FC	-20 $EC	-36 $DC	-52 $CC	-68 $BC	-84 $AC	-100 $9C	-116 $8C
-5 $FB	-21 $EB	-37 $DB	-53 $CB	-69 $BB	-85 $AB	-101 $9B	-117 $8B
-6 $FA	-22 $EA	-38 $DA	-54 $CA	-70 $BA	-86 $AA	-102 $9A	-118 $8A
-7 $F9	-23 $E9	-39 $D9	-55 $C9	-71 $B9	-87 $A9	-103 $99	-119 $89
-8 $F8	-24 $E8	-40 $D8	-56 $C8	-72 $B8	-88 $A8	-104 $98	-120 $88
-9 $F7	-25 $E7	-41 $D7	-57 $C7	-73 $B7	-89 $A7	-105 $97	-121 $87
-10 $F6	-26 $E6	-42 $D6	-58 $C6	-74 $B6	-90 $A6	-106 $96	-122 $86
-11 $F5	-27 $E5	-43 $D5	-59 $C5	-75 $B5	-91 $A5	-107 $95	-123 $85
-12 $F4	-28 $E4	-44 $D4	-60 $C4	-76 $B4	-92 $A4	-108 $94	-124 $84
-13 $F3	-29 $E3	-45 $D3	-61 $C3	-77 $B3	-93 $A3	-109 $93	-125 $83
-14 $F2	-30 $E2	-46 $D2	-62 $C2	-78 $B2	-94 $A2	-110 $92	-126 $82
-15 $F1	-31 $E1	-47 $D1	-63 $C1	-79 $B1	-95 $A1	-111 $91	-127 $81
-16 $F0	-32 $E0	-48 $D0	-64 $C0	-80 $B0	-96 $A0	-112 $90	-128 $80

Notion d'Arithmétique Signée (Appelé aussi notation en Complément à 2)

Respecter les 3 étapes, exemple le nombre 17

1°) Mise en format : Mettre le nombre dans un format binaire 8 bits ou 16 bits.

Exemple : 17

`%10001` va donc s'écrire sous le format `%0001 0001`

2°) Complément à 1 : Ecrire le complément en inversant chaque bit (un 0 devient un 1 et inversement)

Ex : 17

`%0001 0001 = 17 = $11`

En inversant chaque bit 17 devient :

`%1110 1110 = 238 = $EE`

3°) Complément à 2 : On ajoute +1 au nombre de la deuxième étape

```
  %1110 1110      238      $EE
+ %0000 0001    +   1    + $01
------------    -----    -----
= %1110 1111    = 239    = $EF = -17     pour info 239 - 256 = -17
```

L'ordinateur considéra que ce résultat est l'opposé de 17 donc -17

Exemple : on recherche la représentation binaire de **-2**

```
2  =  %0000  0010

  %1111  1101 <- - - - inversion de tous les bits
+          %1
-------------
= %1111  1110 = $FE = 254
```

et `254 - 256 = -2` (pour info -1 sera représenté par $FF)

Bit de poids fort :

	En 8 bits	En 16 bits	En 32 bits
Nombre est **POSITIF**	bit 7 = 0	bit 15 = 0	bit 31 = 0
Nombre est **NÉGATIF**	bit 7 = 1	bit 15 = 1	bit 31 = 1

Cette notation en complément à deux n'est pas obligatoire, c'est au programmeur de décider si les nombres qu'il utilise sont compris entre :

	Non Signé	Signé
8 bits	0 et 255	-127 et +127
16 bits	0 et 65 535	-32 768 et + 32 767
32 bits	0 et 4 294 967 295	- 2 147 483 648 et + 2 147 483 647

Les Registres du 6809 en Arithmétique signé (appelé aussi notation en Complément à Deux)

Bit **C** (Carry) indique une retenue

```
   128          1000 0000
+  129       +  1000 0001
= +257       = 1 0000 0001
                 |
                 └──────────mise à 1 du bit C
```

En arithmétique signé on ignore la retenue (bit C)

C = 0 pas eu de retenue
C = 1 retenue

Bit **V** (oVer flow) représente un débordement du complément à deux

Changement accidentel du signe du résultat parce que les nombre sont trop grands.

```
    64          0100 0000
+   65       +  0100 0001
= +129       =  1000 0001      1000 0001 = $81 = -127 en Arithmétique Signé
                 |                                     car le bit 7 est à 1
                 |
                 └──────────une retenue interne a été produite du bit 6 vers le bit 7
                                  Ici la retenue s'appelle un débordement
```

V = 1 signifie qu'il y a un débordement. Il signifie que le résultat d'une addition ou d'une soustraction requiert plus de bits d'un registre de 8 bits.

Le débordement se produit dans les cas suivants :
- Addition de grands nombres positifs
- Addition de grands nombre négatifs
- Soustraction d'un grand nombre positif à un nombre négatif
- Soustraction d'un grand nombre négatif à un nombre positif

V = 0 pas de débordement
V = 1débordement

Autres exemples pour les bits C et V

Exemple 01 : sans débordement

```
     6          0000 0110       avec    V = 0
+    8     +    0000 1000              C = 0
=  +14     =    0000 1110                      Résultat exact
```

Exemple 02 : avec débordement

```
   127          0111 1111       avec    V = 1
+    1     +    0000 0001              C = 0
= +128     =    1000 1110                      Résultat FAUX car il y a eu débordement

                1000 1110 = $80      et  $80 en arithmétique signé =  -128
```

Exemple 03 :

```
    +4          0000 0100       avec    V = 1
+   -2     +    1111 1110              C = 1
=   +2     =  1 0000 0010                      Résultat exact  le bit C est ignoré
```

Exemple 04 :

```
    +2          0000 0010       avec    V = 0
+   -4     +    1111 1100              C = 0
=   -2     =    1111 1110                      Résultat exact  car le bit C = 0

                1111 1110       est bien égal à  -2
```

Exemple 05 :

```
    -2          1111 1110       avec    V = 0
+   -4     +    1111 1100              C = 1
=   -6     =  1 1111 1010                      Résultat exact  car le bit V = 0

                1111 1010       est bien égal à  -6
```

Exemple 06 :

```
  -127          1000 0001       avec    V = 0
+  -62     +    1100 0010              C = 1
= -189     =  1 0100 0011                      Résultat FAUX  car le bit V = 1

  -189   ≠      0100 0011
                0100 0011 = 67       et non pas  -189
```

Code BCD (Binary Coded Decimal)

Chaque chiffre compris entre 0 et 9 est codé par un groupement de 4 bits.

Binaire	BCD	
0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	8	
1001	9	
1010	.	Inutilisés
1011	.	
1100	.	
1101	.	
1110	.	
1111	.	

Exemple : le chiffre décimal **16** sera représenté sous la forme :

0001 0110 en BCD
1 6

au lieu de

0001 0000 en hexadécimal

Code Baudot

(Jean Marie BAUDOT 1845-1903)

Code à 5 bits conçu et employé par le réseau de télégraphie TELEX. (TELégraph Exchange) 1920-1988

Pour les 60 caractères deux suites binaires de 5 bits sont nécessaires **xxxxx yyyyy**.

-- Pour les lettres le code **xxxxx** = 11 111
-- Pour les chiffres et symboles le code **xxxxx** = 11 011

Opérations Sur Nombres Hexa

Si on divise un nombre Hexa par $100 on décale l'octet de poids fort vers l'octet de poids faible

```
$78BC / $100 = $0078
```

DIVERS RENSEIGNEMENTS SUR LE 6809

Le 6809 fut introduit vers 1977-1978, il est le plus puissant microprocesseur 8 bits de l'époque.
Le créateur est la Sté MOTOROLA.
Le circuit EF6809 était fabriqué par Thomson-EFCIS, il est conçu en technologie N-MOS.

Il existe deux versions de ce processeur, le 6809 (avec horloge interne) et le 6809E (avec horloge externe).
A noter que la société HITACHI a sortie le 6309 une version améliorée.
La puissance dissipée est de 1W max sous 5V

Le 6809 est compatible avec le 6800 sur le plan du code source, même si le 6800 a 78 instructions alors que le 6809 n'en a que 59. Certaines instructions sont remplacées par d'autres qui sont plus générales, et d'autres furent même remplacées par des modes d'adressage.

Ce microprocesseur comporte :
- **59 instructions** de bases différentes,
- Combinées aux **9 modes d'adressage**,
- On arrive à **1464 instructions différentes**.
- Des instructions spécialisées dans le traitement d'information 16 bits

Fabriqué en technologie MOS canal N, il intègre 3 bus indépendants
- Le bus des Données sur 8 bits, Bidirectionnel à 3 états, chaque broche peut piloter l'équivalent de 8 charges TTL
- Le bus des Adresses sur 16 bits, Unidirectionnel, en sortie à 3 états, chaque broche peut piloter 8 charges TTL
- Le bus Contrôle, Sur 10 bits pour le 6809, sur 12 bits pour le 6809 E

Le **6809** a un générateur de cadence interne qui n'a besoin que d'un quartz extérieur relié aux broches XTAL et EXTAL.
Le **6809E** a besoin d'un générateur de cadence externe.

Il y a des variantes où la lettre du milieu indiquait la vitesse d'horloge nominale du processeur.
La fréquence d'horloge est égale au 1/4 de la fréquence du quartz
Voir également les broches E et Q.

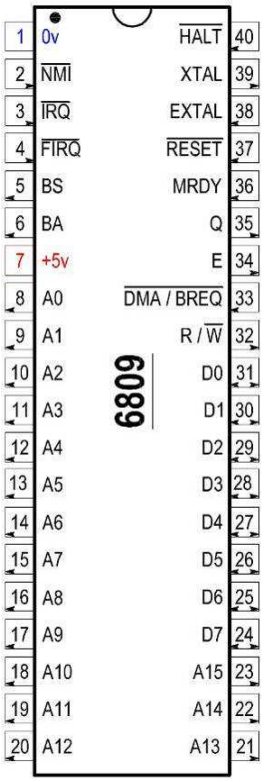
Version	Fréquence d'entrée	Délivre donc une fréquence de
EF 6809	4 MHz	1 MHz
EF 68 A 09	6 MHz	1,5 MHz
EF 68 B 09	8 MHz	2 MHz

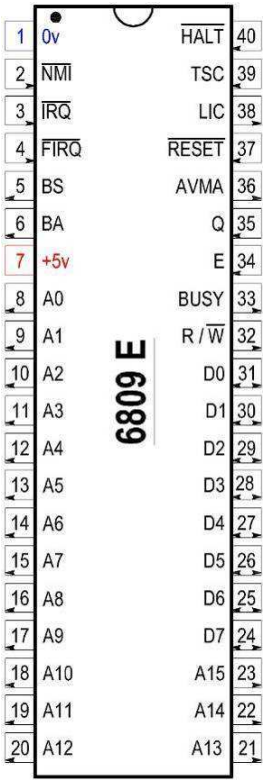
Package Boitier

6809 L	Céramique
6809 C	Céramique Thomson
6809 EP	Plastique
6809 P	Plastique
6809 J	Cerdip Thomson
6809 S	Cerdic Packtage DIL
6809 FN	PLCC

Le 6809 permet la programmation moderne, tel que la programmation :
-- Indépendante de la position mémoire
-- Indépendante du système utilisé
-- De programme réentrant

Brochages du 6809 et du 6809 E





Signification des Diverses Broches

__47a__

Broches		6809		6809E	Broches	
1		0v			1	0v (ou Vss) relié à la masse
2	E	NMI	E		2	Interruption non masquable
3	E	IRQ	E		3	Interruption la moins prioritaire
4	E	FIRQ	E		4	Interruption rapide
5	S	BA	S		5	(Bus Available)
6	S	BS	S		6	(Bus State)
7		+5V			7	relié au +5v (+/- 5%)
8 à 23	S	A0 à A15	S		8 à 23	Bus d'adresses
24 à 31	E/S	D0 à D7	E/S		24 à 31	Bus de données
32	S	R/W	S		32	(Read / Write) Sens des données
33				BUSY S	33	(Occupé)
	E	DMA / BREQ				(Direct Memory Access / Bus REQuest)
34	S	E	S		34	Signal horloge système
35	S	Q	S		35	Signal horloge système
36				AVMA S	36	(Advanced Valid Memory Adress)
	E	MRDY				(Memory Read)
37	E	RESET	E		37	Initialisation complète du 6809
38				LIC S	38	(Last Instruction Cycle)
	E	EXTAL				Connection de l'oscillateur quartz
39				TSC E	39	(Three State Control)
	S	XTAL				Connection de l'oscillateur quartz
40	E	HALT	E		40	Bloque le fonctionnement du 6809

Différence entre 6809 et 6809 E

La différence est dans la nécésité d'avoir un circuit d'horloge extérieur pour le 6809 E.
Ce qui permet d'avoir une souplesse du circuit d'horloge et sert aussi aux cartes multiprocesseurs.

-- Pour le 6809 E c'est donc un circuit externe qui produit les signaux Q et E
(leurs définitions sont les même que pour le 6809)

-- Le signal DMA / BREQ | (pour le 6809) est remplacé par le signal TSC (pour le 6809 E)

Fonctionnement de la Broche NMI | (**N**on **M**asquable **I**nterrupt) Interruption non masquable

(Broche n°2) Interruption Matérielle. Une interruption matérielle provient d'un signal externe.

Cette ligne d'entrée est active au niveau bas.

C'est une interruption mon masquable sensible à un front descendant ↓ entraîne une séquence d'interruption NON MASQUABLE. Cette interruption perrmet d'exécuter une routine d'interruption dont l'adresse est contenue dans le vecteur **$FFFC - $FFFD**

Cette interruption ne peut être interdite d'où son nom. Le programmeur ne peut donc pas interdire son fonctionnement par programme et possède une priorité supérieure à FIRQ| ou à IRQ| ou aux interruptions logicielles SWI, SWI2, SWI3. L'interruption NMI| est la plus prioritaire après la séquence RESET.

Cette interruption est destinée à configurer des séquences de sauvegarde de l'ensemble du système en présence de défaut d'alimentation électrique.

Eventuellement, on peut se servir de cette entrée pour installer un arrêt manuel du type "abort". Dans la plupart des moniteurs, ce bouton joue à peu près le même rôle que SWI. Cependant l'intervention demeure parfaitement aléatoire.

Décrivons ici la réponse à NMI :

1) L'indicateur bit E du registre d'état, est mis à 1 pour mémoriser l'étendue de l'opération d'empilement.

2) Le 6809 empile tous les registres dans la pile système S dans l'ordre suivant : PCL, PCH, UL, UH, YL, YH, XL, XH, DP, B, A, CC. Le pointeur S décroît de 12 cases mémoire.

3) Les interruptions IRQ et FIRQ sont inhibées par masquage des bits I et F du registre d'état CC.

4) Le 6809 vient chercher l'adresse de la séquence d'exception dans les mémoires $FFFC et $FFFD. L'exécution devrait se terminer par RTI pour rendre le contrôle au programme initial.

Le 6809 sauvegarde tous ses registres internes et se branche automatiquement à un sous-programme dont l'adresse est stockée dans les adresses :

{$FFFC} + {$FFFD} {.....} = contenu de
LSB (Least Signifiant Bit) poids Faible
MSB (Most Signifiant Bit) poids Fort

Cette entrée NMI| est souvent réservée aux traitements devant résulter d'une défaillance d'alimentation, sauvegarde dans une mémoire CMOS alimentée par une pile.

Cette broche est dévalidée par la broche RESET| et n'est revalidée qu'après un chargement du registre S (système) en mode d'adressage immédiat.

Le contenu de la totalité des registres du 6809 est sauvegardé dans la pile système.

Le bit E de CC est mis à 1 avant son chargement dans la pile système, pour indiquer que l'état complet du processeur a été sauvegardé (dans l'ordre PC bas, PC haut, U bas, U haut, Y bas, Y haut, X bas, X haut, DP, B, A puis CC).

Fonctionnement de la Broche IRQ | (**I**nterrupt **R**e**Q**uest), interruption masquable

(Broche n°3, broche en entrée) Interruption Matérielle. Une interruption matérielle provient d'un signal externe.

Dès que cette broche passe au niveau bas ⊥, elle provoque une demande d'interruption. Cette interruption a son vecteur en **$FFF8 - $FFF9** (A condition qu'un fonctionnement en DMA ne soit pas en cours)

Cette interruption peut ou non être autorisée, elle est la moins prioritaire des interruptions matérielles. IRQ et un mode d'interruption très populaire. IRQ| à donc une priorité plus basse que les broches FIRQ| ou NMI|.

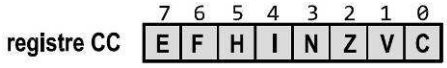
C'est-à-dire que le déroulement de la routine $FFF8-$FFF9 peut être interrompu par FIRQ | ou par NMI |

Le déroulement de la routine engendré par IRQ| peut-être interrompue par les broches FIRQ| ou NMI|
Tous les registres sont sauvegardés (empilés).

Sur un niveau bas sur la broche IRQ| et après la fin de l'instruction en cours, le 6809 sauvegarde tous ses registres internes (l'état complet du 6809) et se branche automatiquement à un sous-programme dont l'adresse est stockée dans les adresses $FFF8 + $FFF9 à condition que le bit I du registre CC soit à 0.

{$FFF8} + {$FFF9} {.....} = contenu de
LSB (Least Signifiant Bit) poids Faible
MSB (Most Signifiant Bit) poids Fort

Le sous-programme de traitement de l'interruption doit libérer la source de l'interruption avant de d'exécuter l'instruction RTI
IRQ| est masquable par l'intermédiaire du bit I de CC.
-- Lorsque le **bit I = 1** l'interruption est interdite.
-- Lorsque le **bit I = 0** l'interruption est autorisé.



La mise à 1 de ce bit I, part une instruction du type **ORCC #%00010000** rend le 6809 insensible à l'état électrique de la broche IRQ.
Le bit I peut-être remis à zéro par une opération du type **ANDCC #%11101111**.

Examinons la réponse du 6809 à une interruption IRQ lorsque cette dernière n'est pas masquée :

1) l'indicateur E est mis à 1, d'où empilement de l'ensemble des registres.

2) Le 6809 effectue une sauvegarde de l'ensemble des registres dans l'ordre suivant : PCL, PCH, UL, UH, YL, YH, XL, XH, DP, B, A, CC. Le pointeur S décroît de 12 cases mémoire.

3) L'indicateur I est mis à 1 inhibant ainsi toutes les autres instructions de type IRQ survenues ultérieurement durant le traitement de la séquence d'exception.

4) L'indicateur F n'est pas touché, donc pendant le traitement du son programme d'interruption IRQ, le 6809 accepte les interruptions du type FIRQ.
Il est tout à fait possible d'autoriser les interruptions multiples par IRQ réparties en niveaux.
Pour ce faire à l'entrée de chaque séquence d'exception il faut mettre le bit I à 0 par ANDCC #%11101111 et manipuler avec soin les contenus des adresses $FFF8 et $FFF9.

5) Le 6809 charge le contenu des adresses $FFF8 et $FFF9 dans le compteur programme PC, puis exécute la séquence d'exception qui doit se terminer par une instruction RTI.

De façon générale, dans un système simple, tous les périphériques pouvant émettre une interruption sont câblés à IRQ à travers une porte OU à plusieurs Entrées. Le plus souvent les circuits ont une sortie à collecteur ouvert, ce qui évite l'ajout de logique supplémentaire.

Donc, chaque fois que le 6809 reçoit un niveau bas sur cette entrée, il ignore totalement quel est le périphérique qui a émis cette demande.

Donc dans la conception des logiciels le début de la séquence d'exception relative à une interruption IRQ contient une identification de la source émettrice. Ceci explique l'utilité des bits d'état des registres d'interface 6821 et 6850.

Il peut arriver que plusieurs périphériques émettent des demandes d'interruption dans le laps de temps écoulé entre la première demande et le début du traitement d'exception.

Dans une telle situation, l'ordre avec lequel le programmeur teste les différents périphériques détermine l'ordre de priorité.

Il existe une façon matérielle autorisant une identification rapide de la source interruption. Le procédé s'appelle une vectorisation des interruptions, qui nécessite l'implantation d'un certain nombre de circuits spécialisés. La méthode d'identification de la source par logiciel s'appelle "identification logicielle" "polling".

Fonctionnement de la Broche FIRQ |

(**F**ast **I**nterrupt **R**e**Q**uest), interruption Rapide

(Broche n°4) Interruption Matérielle. Une interruption matérielle provient d'un signal externe.

Semblable au signal IRQ mais avec une exécution plus rapide.

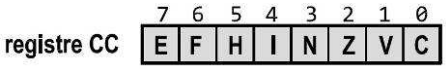
Dans certains cas, le 6809 doit traiter les interruptions très rapidement de qui pose un problème avec les interruptions classiques où la sauvegarde totale du contexte microprocesseur prend un certain temps (1 cycle d'horloge par octet sauvegardé).

Les registres du 6809 ne sont sauvegardé que partiellement.

Cette interruption FIRQ est plus rapide puisqu'il n'y a que le compteur programme et le registre de condition CC qui sont sauvegardés, ce qui fait 3 octets au lieu de 12 en temps normal.

Ici le bit E est positionné à 0 de manière à indiquer que seuls les registres PC bas, PC haut et le registre CC sont sauvegardés dans la pile système.

Cette interruption est masquée ou non suivant l'état du bit F du registre CC. Un niveau bas sur cette broche FIRQ| entraîne la séquence FIRQ à condition que le bit F du registre CC soit à 0.



Cette interruption a priorité par rapport à une demande d'interruption standard la broche IRQ|.
L'adresse de départ du sous-programme de traitement des FIRQ est donnée par le contenu des adresses :

{$FFF6} + {$FFF7} {.....} = contenu de

LSB (Least Signifiant Bit) poids Faible
MSB (Most Signifiant Bit) poids Fort

Le sous-programme de traitement des interruptions doit libérer la source de l'interruption avant l'exécution de l'instruction RTI.

L'opération de sauvegarde effectuée lors d'une interruption IRQ est relativement longue et retarde le traitement de la séquence d'exception.

FIRQ n'effectue qu'une sauvegarde partielle (PC et CC) et permet d'accélérer la procédure en économisant 9 cycles machine en comparaison du temps mis par le 6809 pour répondre à IRQ. C'est à l'utilisateur d'empiler les autres registres qui va utiliser.

FIRQ est également masquable par le bit F, bit b6 du registre d'état CC. Sous l'angle matériel, son implantation est identique à IRQ.

La prise en compte de FIRQ par le 6809 quand le bit F=0 se déroule de la manière suivante :

1) Le 6809 met le bit E à 0 indiquant qu'il va empiler seulement le compteur programme PC et le registre d'état CC.

2) La sauvegarde partielle s'effectue dans l'ordre PCL, PCH puis CC. Le pointeur S décroît de 3 cases mémoire à la fin de l'opération.

3) Le masquage simultané des bits I et F interdit toute interruption ultérieure sur les lignes correspondantes.
   On remarquera que la prise en compte de FIRQ désactive à la fois FIRQ et IRQ, alors que la prise en compte d'IRQ n'exclut pas une interruption ultérieure par FIRQ.
   On dit parfois que FIRQ est prioritaire par rapport à IRQ.

4) Le 6809 vient chercher l'adresse du sous-programme d'exception à l'adresse $FFF6 et $FFF7.
   Le contrôle est rendu au programme initial également par une instruction RTI. Dans la séquence d'exception au besoin on peut effectuer une sauvegarde partielle des registres utile par l'instruction PSHS. Dans un tel cas, ne pas oublier de les dépiler juste avant la rencontre de l'instruction RTI, car cette dernière ne dépilera que les registres PC et CC.

Fonctionnement des broches BA et BS

La broche de sortie BS

Broche n°5 (**B**us **S**tatus) Etat du Bus BS est conjointe à BA

Lorsqu'elle est décodée avec l'état de la sortie BA, représente l'état du 6809 (validé sur le front montant du signal Q)

Les 4 combinaisons possibles des broches BA et BS permettent de connaître à chaque instant, l'état du 6809. Ces indications sont validées sur le front montant de la broche Q (pin 35).

La broche de sortie BA

Broche n°6 (**B**us **A**vailable) Bus accordé ou Bus libre BA est conjointe à BS

Cette ligne signale que les bus trois états sont passés en "haute impédance". Un signal de commande interne fait passer les bus d'adresses et de données à l'état "Haute impédance".

Le signal de disponibilité du Bus (BA) indique la présence d'un signal de contrôle.
BA = 1 ⇒ A0-A15, D0-D7 et R/W| dans l'état de haute impédance.

Ce signal est très utile pour les applications possédant un périphérique capable de gérer les bus adresses et données à la place du 6809 (un DMA par exemple).
Il n'implique pas que le bus soit disponible pendant plus d'un cycle.

Quand BA passe à l'état Bas, il s'écoule un cycle supplémentaire avant que le microprocesseur ne prenne le contrôle des bus.

BA	BS	Etat	Fonctionnement
0	**0**	Fonctionnement Normal	Le 6809 est en fonctionnement **normal**, il gère les bus adresses et données.
0	**1**	Acquittement d'interruption ou de réinitialisation RESET	Le 6809 est en phase de **reconnaissance d'une remise à 0** pendant deux cycles. Cet état correspond à la recherche matérielle du vecteur interruption : RESET, NMI, IRQ, FIRQ, SWI, SWI2, SWI3 Le 6809 attend une entrée d'une interruption, il recherche le vecteur correspondant. Les 4 lignes d'adresses de poids Faibles indiquent quel est le niveau d'interruption pris en compte et permet une vectorisation par périphériques.
1	**0**	Acquittement de synchronisation externe	**Reconnaissance de synchro externe**, cet état est activé lorsque le 6809 rencontre l'instruction de synchronisation externe SYNC. Le 6809 attend cette synchronisation sur une des lignes d'interruption. Les bus sont en haute impédance pendant ce temps.
1	**1**	Acquittement d'autorisation arrêt / bus (arrêt ou bus Libéré)	**Le 6809 est à l'arrêt (ou bus accordé)**. Le 6809 a été arrêté par le signal HALT| Cet état est vrai BA = BS = 1 lorsque le 6809 est : Dans l'état HALT **Ou** Bus accordé (broche HALT| = 0) Reconnaissance du signal DMA / BREQ Correspond à l'arrêt du 6809 ou à l'autorisation venant du 6809 de faire gérer les bus de données et d'adresses par un circuit annexe (ex : DMA). Les bus du 6809 sont en haute impédance, les bus sont disponibles.

Fonctionnement du Bus d'Adresses A0 à A15

Broches n°8 à n°23

Spécifie l'adresse d'origine ou à destination des données qui transitent par le bus de données.

Lorsque le bus d'adresse n'est pas utilisé par le 6809 pour un transfert de données, les broches d'adresse sortent l'adresse $FFFF avec la broche R/W| = 1 et la broche BS = 0.

Les adresses sont validées sur le front montant de la broche Q.
Il est nécessaire de renforcer ce bus par un amplificateur du type 74 LS 241.
Tous les amplificateurs du bus d'adresses sont mis à l'état haute impédance lorsque la broche de sortie BA est à l'état haut.

Fonctionnement du Bus de données D0 à D7

Broche n°31 à n°24
Bus bidirectionnel, il sert à la transmission de données 8 bits
Il est nécessaire de renforcer ce bus par un amplificateur du type 74 LS 245.

Fonctionnement de la broche R/W | (**R**ead / **W**rite |) Lecture / Ecriture

Broche n°32 (c'est une sortie)

Le seul signal pour la commande de la mémoire indique si le transfert fait par le 6809 sur le bus de données est en lecture ou en écriture.

Cette broche détermine le sens du transfert des données.
Elle indique à la périphérie si le 6809 est en lecture ou en écriture de données
Cette broche est une sortie, et est en logique 3 états. Cette broche devra être connectée à chacun des boîtiers

R/W | = 0 (Etat Bas) 0v
Le 6809 est en **écriture** sur le bus
Les broches D0 à D7 sont en sorties, Les données Sortent du 6809

R/W | = 1 (Etat Haut) +5v
Le 6809 est en **lecture** sur le bus
Les broches D0 à D7 sont en entrées, Les données Arrivent au 6809

Fonctionnement de la broche DMA / BREQ |

(Direct Memory Access / Bus REQuest) Accès direct à la mémoire / Demande de Bus

Broche n°33 (pour un 6809 c'est une entrée).
(Uniquement pour un 6809)

Sert avec BA et BS à la déconnexion du 6809 de ses bus. Ces broches servent à indiquer que les bus ont été mis en haute impédance.
.
Signal de demande de bus envoyé au 6809 et en réponse le 6809 met son bus d'adresses, son bus de données et quelques autres signaux de commandes dans l'état haute impédance à la fin de l'instruction en cours.

DMA et rafraîchissement mémoire : Si cette broche est au niveau bas, cela permet de suspendre l'utilisation des bus par le 6809 (déconnexion des bus), pour faire de l'accès direct ou un rafraîchissement mémoire (mémoires dynamiques).

La Mémoire est lue pendant le front descendant de la broche Q, le 6809 termine l'instruction en cours et répond en mettant BA et BS au niveau 1.

Les broches BA et BS passent à l'état 1. Indique la prise en compte de la demande faite par DMA| / BREQ| le circuit demandeur aura alors jusqu'à 15 cycle bus avant que le 6809 ne récupère le bus pour autorafraîchissement.

Lorsque le 6809 répond BA = BS = 1, ce cycle est un cycle perdu utilisé pour transférer le contrôle au système de DMA.

Les bus de données et adresses ainsi que la broche R/W|, sont à l'état haute impédance, l'horloge interne est bloquée, E et Q continuent à osciller.

Indique que le 6809 E est en train d'exécuter une instruction qui nécessite plus d'un cycle d'horloge pour stabiliser la donnée en mémoire.

Le 6809 peut permettre un DMA pendant seulement 15 cycles d'horloge à la fois. DMA (Direct Memory Access, accès direct à la mémoire)

Le 6809 reprend ensuite le contrôle du bus au moins un cycle pour effectuer un rafraîchissement interne.

Autrement dit : Cette broche permet une utilisation des bus du 6809 par un circuit extérieur. Les bus sont mis en haute impédance. Exemple d'utilisation de DMA ou du Rafraîchissement mémoire.

Cette entrée offre une méthode de suspension d'exécution et acquisition du bus 6809 pour une autre utilisation. La transition de la broche DMA| / BREQ| doit se produire pendant le signal Q, un niveau bas sur cette broche arrêtera l'exécution de l'instruction à la fin du cycle en cours.

L'autorafraîchissement nécessite un cycle bus comportant un cycle perdu de début et de fin.

En général le contrôleur DMA fait une demande d'accès au bus en mettant au niveau bas la broche DMA| / BREQ| sur le front montant du signal E.

Les faux accès mémoires doivent être évités pendant tout cycle perdu.

Lorsque la broche BA est remis à 0 (Soit comme résultat de DMA| / BREQ| = 1 ou soit par autorafraîchissement du 6809), le circuit DMA doit être déconnecté du bus.

Un autre cycle perdu s'écoule avant que le 6809 ne se voit alloué un accès mémoire pour transférer le contrôle sans litige.

Fonctionnement des broches E et Q

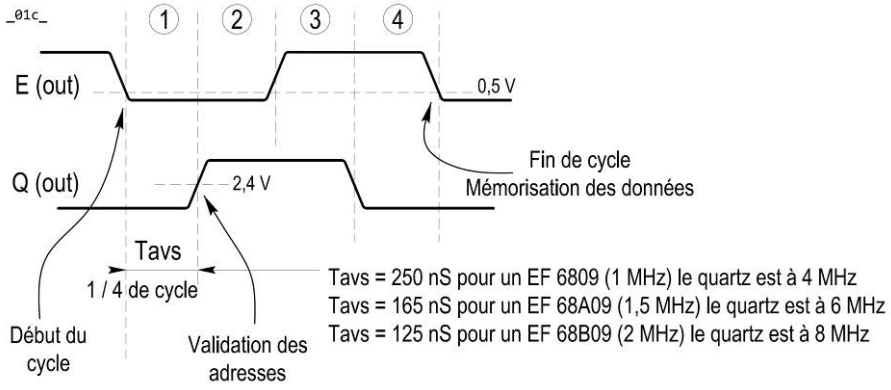
E	**E**nable	Broche n°34	Pour un 6809	(E et Q sont des sorties)
Q	**Q**uadrature	Broche n°35	Pour un 6809E	(E et Q sont des entrées)

Les adresses du 6809 sont correctes à partir d'un front montant de la broche Q. Les données sont mémorisées sur un front descendant de la broche E.

E (out) : Signal horloge système (synchronisation avec la périphérie). C'est une sortie horloge pour le timing des bus (synchronisation avec la périphérie) dont la fréquence est celle de base du 6809.
La broche E est identique au signal d'horloge **Φ2** (phase d'horloge) du processeur EF6800

Q (out) : Signal horloge, en quadrature avec la broche E.
Les adresses sur le bus seront validées sur le **front montant** de la broche Q,
Les données seront mémorisées sur le **front descendant** de la broche E.

La broche Q n'a pas d'équivalence avec le processeur EF6800



Fonctionnement de la broche MRDY (**M**emory **R**ea**DY**) Mémoire Prête

Broche n°36 (pour le 6809 c'est une entrée)

Cette entrée de commande permet l'allongement de l'horloge E pour utiliser des mémoires lentes (temps d'accès aux données augmenté) ou aux dispositifs d'entrée-sorties lents.

Si MRDY est actif, il signal que la mémoire n'est pas prête pour le transfert de données.
Le 6809 attend jusqu'à ce que MRDY redevienne inactif.
MRDY peut être actif pendant un maximun de 10 cycle d'horloge.

L'allongement de E est un multiple de 1/4 de cycle bus, sa valeur maximum est de 10µs.

Dans les phases VMA, la broche MRDY n'a pas d'effet sur E.

Si la broche MRDY est à l'état :
- -- Haut le signal E est fonctionnement normal.
- -- Bas le signal E peut être allongé de multiples (entier de 1/4 de cycles bus) permettant ainsi l'utilisation de mémoires lentes.

L'allongement maximum est donc de 10 microsecondes.
Pendant les accès mémoires non utiles la broche MRDY n'a pas d'effet sur l'allongement du signal E.

Ceci évite le ralentissement de la vitesse du processeur pendant les accès bus non utiles.

Fonctionnement de la broche d'entrée RESET (Remise à Zéro)

Broche n°37 (c'est une entrée)

Le 6809 se relance en lecture du vecteur d'initialisation aux adresses **$FFFE - $FFFF** et dès lors que la condition logique BA = 0 et BS = 1 (reconnaissance d'une remise à zéro).

Le 6809 charge ce vecteur dans le compteur programme PC et se place en exécution.

Un niveau Bas  (pendant un temps supérieur à un cycle du 6809) sur cette broche entraîne l'initialisation complète du 6809. Le travail en cours est totalement perdu.

- -- L'instruction en cours est arrêtée.
- -- Le registre de page DP est mis à zéro.
- -- Les interruptions IRQ| et FIRQ| sont masquées.
- -- L'interruption non masquable NMI| est désarmée.

Cette séquence d'initialisation dure environ une dizaine de cycles processeurs.
L'adresse de départ du programme de traitement du RESET est donnée par le contenu des adresses :

{$FFFE} + {$FFFF} = adresse du programme qui sera lancé après un RESET
LSB_____ (Least Signifiant Bit) poids Faible
MSB_____(Most Signifiant Bit) poids Fort {.....} = contenu de

L'adresse constituée par ces deux octets mémoire est chargée dans le compteur programme PC puis le 6809 exécute le programme à partir de cette adresse.

Un simple réseau RC peut-être utilisé pour initialiser l'ensemble du système puisque l'entrée RESET| possède un trigger de Schmidt dont la tension de seuil est supérieure à celle des périphériques 6800.
L'entrée RESET| sera donc active plus rapidement sur les périphériques que sur le processeur.

De cette façon, lorsque le 6809 commence le programme d'initialisation, on est assuré que tous les périphériques ont terminé leur phase de mise sous tension.

Toute mise sous tension du 6809 :
- -- Commence automatiquement par l'adresse du vecteur RESET.
- -- Le registre DP est mis à 0.
- -- Les interruptions (normales et rapides) sont masquées.

Initialisation générale par RESET

Un niveau bas appliqué sur cette broche provoque une initialisation complète du microprocesseur, les différentes opérations suivantes se déroulent dans l'ordre :

1) Remise à zéro du registre de page directe DP
2) Masquage des interruptions matérielles IRQ et FIRQ
3) Désactivation de NMI
4) Le 6809 ira chercher l'adresse du programme RESET dans le contenue des mémoires **$FFFE** et **$FFFF**.

Le programme RESET est contenu dans une mémoire ROM non volatile. Il sert à démarrer l'ensemble système. D'une façon générale les tâches accomplies dans ce programme sont :

-- Définition de l'emplacement de la pile système. Cette instruction portant sur la pile S dégage l'interdiction sur NMI qui rentre en activité après le chargement du pointeur S.

-- Chargement des adresses d'exécution des sous-programmes d'exception dans les mémoires spécialement réservées à cet usage, dont la plage va de $FFFE à $FFFF.

-- Initialisation et configuration du mode de fonctionnement des interfaces disponibles sur le système.

-- Envoi d'une information l'utilisateur sur la nature du système sous tension.

-- Exécution des tâches particulières propre à chaque système, par exemple connexion automatique vers un programme d'application.

Fonctionnement des broches XTAL et EXTAL

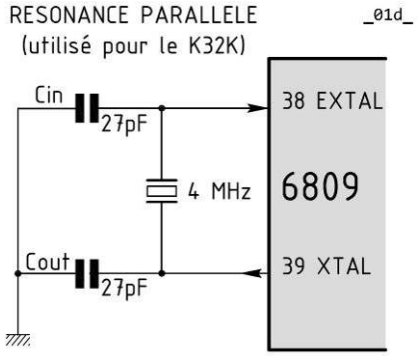
Broche n°38 pour EXTAL (pour un 6809 c'est une entrée)
Broche n°39 pour XTAL (pour un 6809 c'est une sortie)

Elles sont utilisées pour connecter l'oscillateur externe à quartz.
Connexion d'un quartz externe de 4, 6 ou 8 Mhz, Le quartz ou la fréquence externe est 4 fois la fréquence du bus.

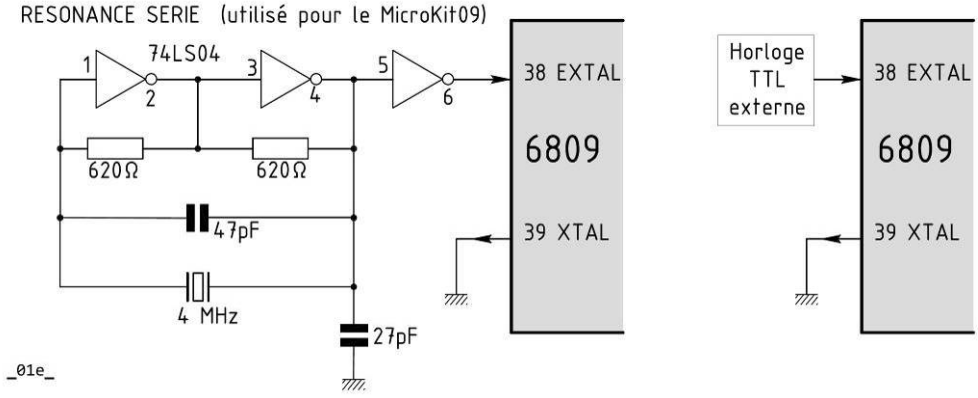
Deux modes de fonctionnement sont possibles :

-- L'oscillateur interne est connecté par ces deux broches à un quartz externe et 2 condensateurs (résonance parallèle).

27 pF dans le cas du kit K32K de DATA RD
22 pF dans le cas du Microkit09



-- Une horloge TTL est connectée aux broches EXTAL, la broche XTAL étant reliée à la masse.



Fonctionnement de la broche HALT |

Broche n°40 (c'est une entrée)

Broche HALT| (arrêt du 6809 ou du 6809E). Sert a arrêter le 6809, le fonctionnement est donc bloqué quand elle est à l'état bas (le 0v).

Cette entrée permet d'interrompre le déroulement d'un programme de façon matérielle (Hardware).
Le 6809 termine l'instruction en cours, puis positionne les broches BA et BS à 1.

Les broches IRQ| et FIRQ| sont invalidées.

Les broches RESET| et NMI| sont valides mais leur traitement ne se fera qu'à la libération du 6809

Le 6809 termine l'instruction en cours et reste arrêté indéfiniment sans perte de données. Le 6809 reprend la suite du programme dès que la broche HALT | est de nouveau au niveau Haut et sans perte d'information
-- La broche BS = 1 indiquant que le 6809 est arrêté ou à l'état bus accordé (bus libre).
-- La broche BA = 1 indiquant que les bus sont à l'état haute impédance.

Tant que le 6809 est à l'arrêt :
-- Les demandes d'interruption IRQ| et FIRQ| sont inhibées.
-- Les demandes d'accès direct mémoire sont autorisées.
-- Les demandes d'interruption prioritaires RESET| et NMI| sont prise en compte mais leur traitement est différé.
-- Les horloges fonctionnent normalement.

Arrêt temporaire par HALT

Sur un niveau bas de cette broche, le 6809 se met au repos. Cette suspension d'activité n'efface pas les contenus des registres.

Le rétablissement ultérieur d'un niveau Haut sur la broche HALT autorise le 6809 à continuer dans sa tâche sans perdre d'informations.

Cette broche est configurée par un interrupteur ou par l'intermédiaire d'un système de logique électronique.

Cette interruption peut donc être employée à tout instant sans aucun risque de perte d'information à condition que le système soit sous le contrôle de l'unique 6809 interrompu.

D'autres détails intéressants à connaître sont :
-- Le 6809 n'examine une demande HALT qu'à la fin de l'exécution d'une instruction.
-- A l'arrêt le 6809 ignore les demandes IRQ et FIRQ. Les entrées RESET et NMI sont en revanche mémorisées pour une réponse ultérieure.
-- Un sous-programme d'interruption quelconque peut être arrêté par HALT.
-- Par sa définition, HALT n'a pas de traitement d'exception, donc ignore la pile.

BROCHES UNIQUEMENT POUR LE 6809 E

Fonctionnement de la broche BUSY (occupé)

Broche n°33 (pour un 6809E c'est une sortie). Facilite les applications multiprocesseur (comme la broche AVMA 6809E).

Indique que le 6809 E est en train d'exécuter une instruction qui nécessite plus d'un cycle d'horloge pour stabiliser la donnée en mémoire.

La broche BUSY a le même fonctionnement que la broche AVMA, mais la broche BUSY ne passe à l'état Haut que pendant l'exécution d'une instruction de type lecture, modification, écriture d'une donnée (ASL par exemple). De cette façon, les zones mémoires ne sont pas adressées simultanément par deux processeurs.

On ne doit pas effectuer de TSC quand BUSY est actif. C'est très important pour les systèmes multiprocesseurs.

Servent à indiquer les instants de validités des bus de données et d'adresses.

Fonctionnement de la broche AVMA (**A**dvanced **V**alid **M**emory **A**dress) "Adresse mémoire valide avancée"

Broche n°36 (pour un 6809E c'est une sortie) remplace l'entrée MRDY du 6809
Contrôle des ressources communes en multiprocesseur.

AVMA anonce que le 6809E s'apprête à faire un accès valide en mémoire pendant le cycle d'horloge suivant.

AVMA indique au circuit d'horloge, que si l'on utilise une mémoire lente ou un dispositif entrée-sortie lent, les temps d'horloge doivent être prologés.

Le 6809E ne peut pas commander l'horloge par lui même.

C'est une sorte de validation d'adresse mémoire perfectionnée qui passe à l'état 1 au cours du cycle précédant un accès bus par le 6809.

Le signal permet d'optimiser l'échange et effectue un contrôle efficace des ressources communes d'un dispositif multiprocesseur.

Si le 6809 est en HALT| ou SYNC| ou en calcul interne alors la broche AVMA est au niveau Bas.

Fonctionnement de la broche LIC (**L**ast **I**nstruction **C**ycle) Dernier cycle d'une instruction.

Broche n°38 (pour un 6809E c'est une sortie)

Cette sortie est à l'état haut pendant le dernier cycle de chacune des instructions exécutées par le 6809E.

Quand la broche LIC passe à l'état bas, cela indique que la premier octet d'une instruction va être recherché à la fin du cycle en cours.

Le cycle qui suit ce signal est donc toujours un cycle de recherche de code opératoire.

La broche LIC est à l'état Haut quand le 6809 est en attente d'une synchronisation externe (broche SYNC), en phase d'empilage au cours d'une gestion d'interruption, ou dans l'état HALT|

Fonctionnement de la broche TSC (**T**hree **S**tate **C**ontrol) Contrôle trois états

Broche n°39 (pour un 6809E c'est une entrée)

Contrôle d'état haute impédance des bus.

Cette entrée joue le même rôle que l'entrée DMA| / BREQ| du 6809 normal.

Quand TSC est à l'état 1, le bus adresse, le bus de donnée et la ligne R/W| sont en haute impédance.

On de doit pas effectuer de TSC quand BUSY est actif.
C'est très important pour les systèmes multiprocesseurs.

Il est à ce moment possible de faire de l'accès direct ou du rafraîchissement mémoire ou encore de faire la gestion des bus avec un autre 6809.

GENERALITES SUR L'ASSEMBLEUR DU 6809

Il est à préciser ici qu'il y a plusieurs **Assembleurs**, les explications qui suivent sont donc d'ordre générale.

Inconvénients du langage machine

Impossibilité de portabilité sur d'autres machines.
Programmes difficiles à lire et à corriger.
Calculs arithmétiques difficiles, la gestion de nombre en virgule flottante est compliquée.
Apprentissage plus important, le langage machine demande beaucoup de rigueur et d'attention.
Structuration difficile d'un programme.

Avantages du langage machine

Vitesse d'exécution maximale
Utilisation plus rationnelle de la taille mémoire, un programme en langage machine occupe moins de place en mémoire.
Possibilité de création de fonctions irréalisables en BASIC.
Utilisation de toutes les ressources de l'ordinateur.
Possibilité de mêler BASIC et langage machine

L'ASSEMBLAGE

Assemblage : Généralités

Un programme assembleur permet de traduire les mnémoniques d'un programme SOURCE écrit en ASCII en un programme OBJET en HEXA directement exécutable.
L'assembleur accomplit deux tâches principales :

-- Il traduit les codes opérations mnémoniques en équivalent binaire.
-- Il traduit les symboles utilisés pour les constantes et les adresses en équivalent binaire.

Programme SOURCE

Écrit en mnémonique, listing du programme lisible (n'est pas exécutable directement)

Le programme OBJET

Génère le code OBJET à partir du programme source
Le code OBJET est directement exécutable par le 6809.

Lors de l'assemblage il se produit deux choses :

Production d'un listing du programme avec :

-- A droite le programme source inchangé avec en plus une numérotation des lignes.
-- A gauche se trouvent deux colonnes avec l'implantation en mémoire des codes opérations suivi du code machine correspondant au programme.
-- Une détection des erreurs éventuelles.

Génération d'un code objet destiné à être stocké, avec :

-- Le code machine
-- Son adresse d'implantation en mémoire
-- Son adresse de lancement.

L'assemblage s'effectue généralement en passes (deux lectures) :

Première passe :

-- Lecture de chaque nom de constante (ou symbole) afin d'établir une table des symboles
-- A chaque étiquette l'assembleur assigne l'adresse qu'occupera le code opération (mnémonique) présent sur la même ligne.
-- Le pointeur d'adresse est fixé à une valeur origine au début du programme
-- A chaque instruction ce pointeur est incrémenté selon la longueur de l'instruction
-- Pendant cette première passe l'assembleur détecte les erreurs de syntaxe.

Durant la seconde passe :

-- L'assembleur fait l'assemblage proprement dit afin de générer le code objet.
-- Chaque fois que l'assembleur rencontre un nom de variable ou une étiquette, il recherche dans la table et affecte l'adresse ou la donnée stockée lors de la première passe.

STRUCTURE D'UN LISTING ASSEMBLEUR

Listing Assembleur, Généralité

L'assembleur standard reconnaît la majorité des caractères ASCII (American Standard Code for International Interchange), l'alphabet principal comprend :

Les majuscules de A à Z (les minuscules sont pour le champ commentaires)
Les chiffres de 0 à 9
Le séparateur de champ : . . . . . l'espace sodé $20 en ASCII
Le retour chariot ou retour de ligne : codé $0D en ASCII
Les signes principaux : . . . . . + - * / # $ % & @ ' , < > [ ]

Les minuscules et les signes secondaires peuvent être introduits dans un opérande à condition d'être précédés par une apostrophe **'** dans ce cas le caractère derrière l'apostrophe sera remplacée lors de l'assemblage par le code ASCII en hexa.

'! Remplacé dans le code objet par $21
'a " " " " " $61

L'alphabet du champ commentaire est plus vaste, il contient tous les caractères visualisables (signes secondaires, ainsi que les minuscules.

Deux exceptions sont à retenir :

-- Les minuscules et les signes secondaires peuvent être introduits dans un opérande à condition d'être précédés par des apostrophes, dans ce cas ils seront remplacés à l'assemblage par leur code équivalent ASCII
exemples : **'!** remplacé par $21 **'a** remplacé par $61)
-- La directive FCC, qui manipule les chaînes de caractères, accepte évidemment l'alphabet élargi.

Une instruction écrite en assembleur standard doit comporter au moins 4 composantes appelée champs :

-- Champ Etiquette (de 6 à 20 caractères)
-- Champ Mnémonique (appelé aussi Instruction)
-- Champ Opérande (de 10 à 70 caractères)
-- Champ Commentaire (dont le premier caractère est obligatoirement le point-virgule)

A la fin du listing, l'assembleur signale le nombre total d'erreurs et d'avertissements relevés durant la phase d'assemblage.

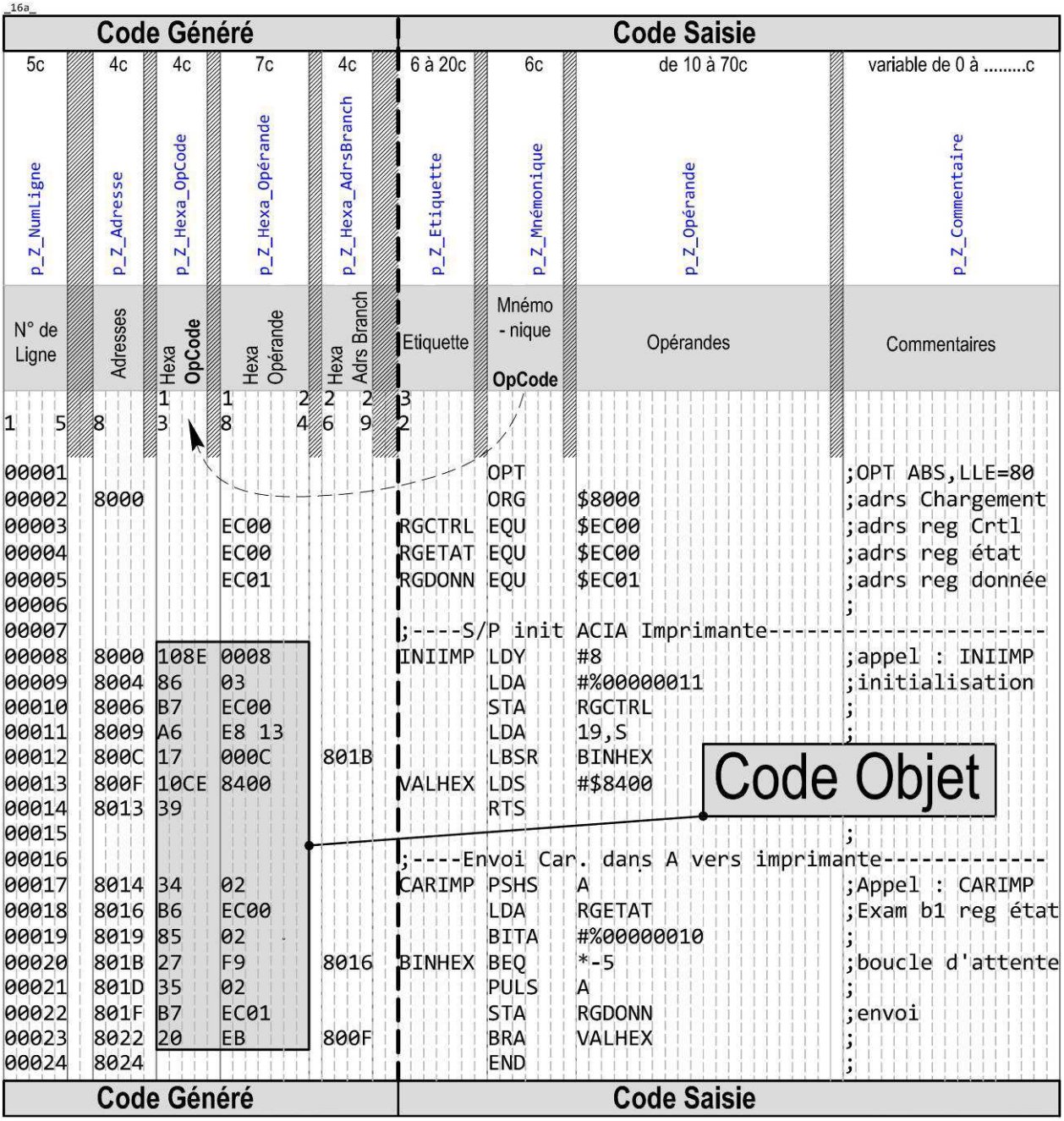
Table des références croisées

A la fin des listings d'Assemblage on édite une table des références croisées.

Dans ces tables on y retrouvera tous les symboles définis pas le programmeur, le plus souvent classés selon leur ordre alphabétique.

-- Etiquettes
-- Noms de constante

Exemple de programme Assemblée (Organisation des Colonnes)



Numéro de Ligne

L'assembleur re-numérote toutes les lignes existantes dans le listing source (sur 5 positions).

Section Absolue

La lettre A signifie "section Absolue". Elle n'a de signification que pour les programmes compilés en code translatable puis reconfigurés par l'éditeur de liens.

Adresses

On y trouve les adresses absolues d'implantation des instructions où se trouvera éventuellement le code binaire exécutable.
Ces colonnes sont vides pour des lignes de commentaires ou pour des lignes contenant des directives n'invoquant aucun emplacement mémoire du type OPT, END, FCB, FDB, FCC etc ...

Op-Code en Hexa (appelé Op-code, Instruction ou Mnémonique)

Fait partie du code machine exécutable.

Contiennent le code opération ou Op-Code du 6809, résultat de la traduction des mnémoniques standards, c'est le code généré par l'assembleur.
Certains Op-Codes requièrent 2 octets.

Opérande en Hexa

Fait partie du code machine exécutable.

Contiennent le Code opérande du 6809 qui peut être de 0, 1 2 ou 3 octets.

Adresse de Branchement en Hexa

Quand il s'agit d'un branchement, cette colonne fournisse l'adresse absolue de destination, ce qui évite tout calcul de la part de l'opérateur. L'affichage de cette adresse dépend de l'option sélectionnée dans l'assembleur.
Exemple : la ligne 20 du listing ci-dessus, $8011 représente l'adresse du branchement BEQ *-5.
Ce type de rappel est de grande utilité pour la **M**ise **au P**oint des programmes (M a u P).

Champ Etiquette

(de 6 à 20 caractères pour **P30RS09**) *(le P30RS09 est un Assembleur Désassembleur que j'ai développé et disponible gratuitement)*

Utilisé pour les instructions de saut conditionnel ou inconditionnel et pour les sous-programmes.
Essayer de mettre des noms de label compréhensifs.

Pour l'Assembleur Désassembleur **P30RS09** (que j'ai développé et disponible gratuitement), les caractères acceptés pour ce champ sont :

A…..Z	Les majuscules
a…..z	Les minuscules
0…..9	Les chiffres
é è à	Certaines lettres accentuées
.	Le point
_	Le soulignement, code ASCII $5F
|	Le code ASCII $7C
:	Les deux points
=	Le signe égal

ATTENTION : Le premier caractère ne doit pas être numérique.

Les noms d'étiquettes ou de constantes suivant sont interdits, (ils sont réservés aux noms de registres) :

```
PC, PCR, Y, X, S, U, DP, A, B, D, CC
```

L'étiquette doit apparaître qu'une seule fois dans tout le programme.

Les noms des directives d'assemblage sont à éviter comme étiquette pour une raison de clarté.

Dans un format libre, le format de saisie est celui de MOTOROLA. C'est le premier caractère qui détermine la nature de la ligne, 4 cas :

1. Le premier caractère est un point virgule	C'est un commentaire de BLOC
2. Le premier caractère est un caractère autorisé pour le champ Etiquette	C'est une ligne au format étiquette **ÉTIQUETTE MNÉMONIQUE OPÉRANDE ;COMMENTAIRE**
3. Le premier caractère est un ESPACE	C'est une ligne au format sans étiquette **MNÉMONIQUE OPÉRANDE ;COMMENTAIRE**
4. Le premier caractère est un ESPACE suivi d'un Point-virgule	C'est un commentaire de LIGNE

Eviter le vocabulaire peu suggestif du type (TOTO, TATA, ESSAI, CHOSE, TRUC, XXX, ...)

Faire attention à des ressemblances formelles du type : O et 0 1 et I Z et 2 B et 8 A et 4 S et 5

Lors d'un chiffrage, une bonne habitude consiste à utiliser un format avec des zéros devant les chiffres, exemples :
- Pour les nombres de 0 à 99 on utilisera deux chiffres 00, 01, 02, …
- Pour les nombres de 0 à 999 on utilisera trois chiffres 001, 002, 003 …

Champ Mnémonique (appelé aussi instructions)

Contient le code mnémonique de l'opération qui peut être :

-- Une vraie opération c'est à dire celle qui possède un code binaire (LDA, STA,...).

-- Une pseudo opération, encore appelée directive d'assemblage (OPT, ORG, EQU, END,...) qui ne produit en général aucun code objet, mais agit en revanche sur le processus d'assemblage lui-même.

Champ Opérande

(de 10 à 70 caractères pour **P30RS09**) *(le P30RS09 est un Assembleur Désassembleur que j'ai développé et disponible gratuitement)*

Complète le champ opérande appelé aussi champ opération, sert à définir la donnée sur laquelle s'effectue l'instruction.
L'opérande doit donner à l'assembleur le mode d'adressage.
Le champ opérande peut contenir tous les caractères de l'alphabet principal (sauf le point virgule qui est réservé au début du champ commentaire.
Il peut contenir des symboles définis dans le champ étiquette.

Dans ce champ Opérande on peut trouver :

Des nombres

Décimaux	Préfixe **&**.est facultatif		75 ou &75
Hexadécimaux	Préfixe **$**	ou suffixe .H	**$**75 ou 75**H**
Binaire	Préfixe **%**	ou suffixe .B	**%**01000110 ou 01000110**B**
Octal	Préfixe **@**	ou suffixe .Q	**@**12 ou 12**Q**

Caractères ASCII : le caractère sera précédé par une apostrophe, ex pour le caractère A : `'A`
Déplacement par rapport au registre PC : `*+5` déplacement de 5 octets, déplacement peut être + ou –
L'astérisque `*` est là pour l'adresse courante

Des noms de variable

Des noms de variable pour définir une adresse ou une valeur numérique.
Ces noms :

-- 20 caractères maxi.

-- Les caractères admis sont les mêmes que le champs Etiquettes

`a…z A….Z 0….9 é è à . _ | : =`

-- Ne pas choisir des noms identiques

Aux codes opérations
Aux registres internes
Aux directives d'assemblages

Des noms d'étiquettes

Exemple :

```
JMP  BOUCLE
```

Des expressions arithmétiques ou logiques

Certains Assembleurs autorisent l'écriture d'expressions arithmétique et/ou logiques dans le champ opérande. Quelques exemples :

```
INDEX+OFFSET
INDEX+10
-(ADRFIN-ADRINI)*2
*+5
*-4
```

Ces expressions peuvent aussi utiliser les opérations **+ - * /**

```
ADCA  VALEUR + 1
JMP   BOUCLE - 5
```

Champ Commentaire

Le seul caractère pour délimiter ce champ est obligatoirement le caractère point virgule **;**
Le caractère astérisque trouvé dans certains vieux programmes assembleur Motorola est réservé ici pour les opérations arithmétiques.

Ce champ est optionnel. Il ne produit aucun code exécutable.
Les commentaires bien souvent négligés, sont très utiles pour documenter le programme et de ce fait le rendre plus lisible. Un programme bien commenté facilite sa maintenance et sa transmission.

On distingue les commentaires :

-- Commentaires de **BLOC** Le **;** est placé au début du champ Etiquette

-- Commentaires de **LIGNE** Le **;** est placé au début du champ Commentaire

Conseils et règles à observer pour les commentaires :

-- Ils doivent servir à éclairer un programme.

-- Peuvent servir à expliquer l'utilité d'une instruction ou d'un sous-programme.

-- Mettre des commentaires à des endroits clefs. Toutes les lignes ne doivent pas être commentées.

-- Utiliser de préférence les minuscules.

-- Etre explicite pour les commentaires de blocs,

-- Etre très concis pour les commentaires de lignes, supprimer les articles, les ponctuations (si possible) et les liaisons grammaticales.

-- Eviter de commenter la nature de l'instruction comme

```
LDX #$8000    ; chargement adresse $8000 dans X
```

Mais utiliser plutôt un commentaire du style

```
LDX #$8000    ; adrs Début tab. Constantes
```

-- Définir toutes les abréviations personnelles en début de programme
exemple : `AttCarCla` pour Attente caractère Clavier

DIRECTIVES D'ASSEMBLAGE

On doit placer les directives d'assemblage dans le champ Mnémonique.

Elles agissent plutôt sur le processus d'assemblage. Tel que par exemple le positionnement du programme et des différents tableaux en mémoire.

Les directives d'Assemblage gérées par le **P30RS09** sont :
`END, EQU, FCB, FDB, FCC, ORG, RMB, SETDP`

Les directives d'Assemblage **NON** gérées par le **P30RS09** sont :
`BSZ, FAIL, FILL, NAM, OPT, PAGE, REG, SET, SPC, TTL, ZMB`

*(le **P30RS09** est un Assembleur Désassembleur que j'ai développé et disponible gratuitement)*

Directive ORG (Origine)

Permet de définir l'origine de départ d'un programme ou d'un sous-programme. Elle permet de définir l'adresse absolue d'implantation du programme. Elle est utile uniquement si on a des contraintes d'implantation de programme (saut absolue par JSR, JMP)

Mais n'est pas obligatoire si le programme est complètement écrit en relatif (BSR, LBSR, BRA ….).

Les instructions suivantes seront assemblées dans les emplacements mémoire situés à partir de la nouvelle adresse contenue dans le compteur de programme.

Cette directive définit donc l'adresse du premier octet du programme objet (programme objet = ensemble du code machine en hexa directement exécutable).

```
ORG   $0400           ; opérande du type constante
```

Après assemblage, le premier octet de la première instruction occupera l'adresse $0400.

```
ORG   RESET           ; opérande du type symbole
ORG   *+100           ; opérande du type expression
ORG   SBR1+LGTAB*8    ; opérande du type expression
```

Etant donné leur rôle, cette directive n'a pas d'étiquette.

La gestion de l'espace mémoire est une tâche importante en assembleur, un programme écrit dans ce langage comporte plusieurs sections ou blocs, dans ces derniers on peut trouver :

-- Le programme principal
-- Les sous-programmes
-- Les adresses des interfaces
-- Les adresses des vecteurs d'interruption
-- L'adresse de la zone de stockage de paramètres
-- L'adresse de la zone des données temporaires
-- L'adresse des piles

ORG initialise les débuts des sections et se trouve donc obligatoirement en entête de chaque section. Aucune étiquette ne doit précéder la directive ORG.

Si un programme est écrit sans la directive ORG, l'assembleur choisit par défaut l'adresse $0000.

Directive BSZ (Block Storage Zero)

Voir aussi la directive RMB ou ZMB

Cette directive est plus ou moins identique à RMB (ou ZMB) à la fois dans le principe de réservation des mémoires et le mode d'écriture des formes autorisées.

L'unique différence vient du fait que BSZ initialise à 0 au chargement toutes les mémoires réservées, alors que RMB n'écrit rien en mémoire, RMB ne détruit donc pas les contenus mémoires situés dans les blocs réservés.

BSZ impose à l'assembleur de réserver un bloc d'octets et d'assigner à chacun de ces octets la valeur 0

Le nombre d'octets est donné par l'opérande qui ne doit pas contenir de symbole, l'opérande ne doit pas être nul sinon il y aura erreur lors de l'assemblage.

La directive BSZ permet d'initialiser de la mémoire à 0

```
        ORG   $2000           ;
TOTO    BSZ   $10             ; TOTO vaudra $2000
TITI    EQU   *               ; TITI vaudra $2010
                              ; Et en mémoire entre $2000 et $2010, il y aura des 0
```

Le code ci-dessus est équivalent à :

```
        ORG   $2000
TOTO    FCB   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
TITI    EQU   *
```

Ou encore à ça :

```
        ORG   $2000
TOTO    FDB   0,0,0,0,0,0,0,0
TITI    EQU   *
```

Directive END (Fin)

Définit la fin logique d'un programme source.
De même qu'il est nécessaire de fournir à l'assembleur une indication de début de programme par ORG, l'assembleur doit connaître l'endroit ou se termine le programme.
Cette directive marque la fin logique d'un programme.

Les instructions derrière la directive END ne seront pas assemblées. L'oubli de la directive END à l'édition sera signalé par un avertissement.

END autorise à spécifier l'adresse d'exécution dans son champ opérande par un symbole.

```
DEBUT   LDA    $78      ;
        ADCA   #05      ;
        …               ;
        END    DEBUT    ; indique à ASM que la première instruction
                        ; à exécuter sera à l'adresse DEBUT.
                        ; Alors que ORG donne l'adresse de
                        ; chargement en mémoire du code objet.
```

Nota : dans le cas des kits K32 de la Sté DATA RD, l'assembleur met en plus un $3F en mémoire, le même OpCode que l'instruction **`SWI`**.

Directive EQU (Equate)

Voir aussi la directive SET

Permet de d'affecter une valeur 8 ou 16 bits à un nom de constante (nom symbolique).
L'opérande ne peut pas contenir un nom de constante qui est défini un peu plus loin dans le programme.

```
    COMPT   EQU   $05           ; donnée 8 bits
    ADRDEB  EQU   1000          ; donnée 16 bits
    FIN     EQU   DEBUT+$60     ;
```

On placera les directives EQU en début de programme.

A chaque fois que l'assembleur rencontre le nom symbolique, celui-ci est remplacé par la valeur hexadécimale indiquée après le mnémonique EQU.

EQU donne au nom symbolique une valeur qui n'est pas liée au compteur de programme PC.
Les noms définis par EQU ne peuvent pas être redéfinis dans la suite du programme.

La directive EQU accepte également un opérande du type expression avec tous les opérateurs arithmétiques et logiques.

```
    VALHEX EQU   $8000              ; VALHEX aura pour valeur $8000
    DEBBIN EQU   -$0200             ;
    FINATT EQU   NOMVAR+30          ;
    FINTIT EQU   NOMVAR-VARFIN      ;
    TOUCHA EQU   'A                 ; assigne $0041 au symbole TOUCHA
    FONCTA EQU   TOUCHA+%10000000   ; assigne $00C1 au symbole FONCTA
```

On peut affecter une adresse à un label, dans ce cas on utilises l'opérateur * qui donne l'adresse courante

```
        ORG   $2000
TOTO    EQU   *       ; TOTO aura pour valeur $2000, car on affecte la
                      ;            valeur courante de l'adresse à TOTO
TITI    EQU   (*-3)   ; TITI aura pour valeurs $1FFD ($2000-3)
```

Remarque : Pour une adresse courante on utilise toujours `EQU *` et pas `SET *`.

Directive FAIL

Une directive FAIL incluse dans une ligne quelconque du programme provoque l'affichage sur l'écran ou l'impression sur une imprimante, de l'ensemble de la ligne marqué par FAIL.
Elle est destinée à repérer le trajet choisi par l'assembleur.

Directive FILL

La directive FILL indique à l'assembleur d'initialiser une zone mémoire avec une valeur constante.
Le premier opérande donne la valeur de la constante et le deuxième opérande donne le nombre d'octets successifs à initialiser.
Le premier opérande doit se trouver dans la gamme de valeurs 0-255.
Les deux opérandes ne peuvent comporter ni de constante (EQU) défini plus loin dans le programme ni de constante non défini.

Directive OPT (OPTion système)

La directive OPT sert à fixer le format du fichier de sortie. Elle permet au programmeur de sélectionner ou de contrôler différentes opérations de sorties de l'assembleur.

Directive PAGE

Cette directive provoque un saut de page dans le fichier listing.
Le mot PAGE ne sera pas reproduit sur le listing.

Directive NAM

Donne un nom au programme en dehors du nom donné aux fichiers, le programmeur à la possibilité d'attribuer un autre nom au listing.

Ce nom sera composé d'au maximum 6 caractères ASCII visualisables.

Le nom dans le champ opérande de la directive NAM est répété à chaque début de page du listing, après le numéro de page et le nom du fichier.

Directive SET (Voir aussi la directive EQU)

Set est une assignation temporaire, les variables ou les noms de symboles (Constantes ou Etiquettes) de cette directive peuvent être définies à nouveau dans un même programme et dans des circonstances variées.

Les noms symboliques définit par SET peuvent être redéfinis dans la suite du programme. La directive SET est très utile pour définir temporairement des noms symboliques ou des étiquettes réutilisables dans la suite du programme.

A l'inverse de SET, une étiquette assignée par EQU conservera sa valeur tout le long d'un programme.

La séquence suivante, écrite avec la directive SET, ne sera pas tolérée si on utilise la syntaxe EQU.

```
ARG     SET   2          ; le symbole ARG prend la valeur 2
ARG     SET   ARG*2      ; le symbole ARG prend la valeur 4
ARG     SET   ARG*ARG    ; le symbole ARG prend la valeur 16 (4*4)
ARG     SET   ARG+2      ; le symbole ARG prend la valeur 18 (16+2)
```

Permettent d'affecter une valeur à un label

```
TOTO   SET   1          ; TOTO aura pour valeur 1
TATA   EQU   2          ; TATA aura pour valeur 2
TITI   SET   TOTO+TATA  ; TITI aura pour valeur 3
```

Un symbole assigné par SET peut apparaître simultanément dans les champs Etiquette et Opérande d'une même instruction.

Une modification quelconque d'un opérande d'une directive SET peut éventuellement avoir des répercussions sur plusieurs endroits d'un programme.

-- La directive **EQU** est liée plus "au Matériel". Une adresse d'interface entrée-sortie, une zone mémoire attribuée au programme utilisateur, l'adresse d'appel au moniteur résident, etc ...

-- La directive **SET** est liée à un programme spécifique.

Choisir une étiquette est en réalité une opération délicate, surtout quand on dispose que de 6 caractères. On peut mieux choisir une étiquette quand elle est liée au matériel.

Quand elle se rapporte à des notions abstraites comme un algorithme ou un résultat intermédiaire de calcul le programmeur est souvent embarrassé.
Il faut également tenir compte que multiplier les étiquettes alourdit le programme source et le rend touffu, voire peu compréhensible.

Dans ces circonstances la directive SET apporte une simplification considérable.

Directive SPC (Space)

Cette directive permet d'insérer une ligne blanche dans le listing après compilation, le plus souvent pour séparer les différents blocs de programme, le programme principal des sous-programmes ou les sous-programmes entre eux, pour améliorer la lisibilité.

3 formes connues de l'opérande pour SPC : la constante, le symbole ou l'expression.

```
SPC          ; équivaut à SPC  1
SPC   $A     ; laisser 10 lignes blanches
SPC   NLB    ; opérande du style symbole
SPC   2*CFLB ; opérande de type expression
```

Directive TTL (Title)

Donne la possibilité de titrer les différentes partie d'un programme, de sonner un entête différent pour chaque sous-programme ou chaque section.

```
TTL   ; ** Sous Programme d'attente ***
TTL   ---SECTION PARAMETRES---
TTL   Initialisation générale
```

Si NAM admet éventuellement un champ commentaire, la directive TTL exclut tout commentaire

L'opérande peut contenir jusqu'à 45 caractères ASCII visualisables pour un terminal comportant 80 colonnes.

Il est reproduit intégralement au début de chaque page et derrière le nom attribué par la directive NAM.

Ce nombre peut être néanmoins modifié par une option de la directive OPT.

La fin d'un sous-programme ou d'une section ne correspond pas toujours à la fin d'une page.

L'association des directives SPC et TTL permet de positionner la première instruction d'une section sur un début de page avec un nouvel entête, ce qui évite des coupures aléatoires.

```
NAM  COMFRS        COMmande FRaiSeuse
TTL  ---S/P Vérification Etat Capteurs---
...         ;
...         ; premier corps de programme
...         ;
TTL  ---S/P Lecture des Commandes---
SPC  100    ; le chiffre 100 est factice. N'importe quel
            ; chiffre > au nombre de ligne par page provoque
```

```
                        ; un positionnement du texte au début de la page
                        ; suivante avec un nouvel entête. Il ne faut pas
                        ; intervertir l'ordre des deux directives SPC et TTL
                        ;
...                     ; deuxième corps de programme
```

Ci-dessous, apparaissent les informations contenues dans la première ligne d'une page de listing, avec NAM et TTL combinées.

`PAGE 006`	`ASERIMGF.LO:0`	`COMFRS`	`Reconnaissance des Paramètres`
Numéro de page	Nom du fichier	Nom du programme	Entête d'une section particulière

Directive REG (REGistre)

D'une importance marginale, cette directive est conçue pour raccourcir l'écriture des instructions d'empilement PSHS, PSHU et de dépilement PULS, PULU.

Les noms des registres situés dans le champ opérande sont affectés au symbole du champ étiquette.

En l'absence de REG les instructions opérant sur les piles requièrent des opérandes formés par des successions de noms de registres. Exemples :

```
          PSHS  PC,Y,CC           ; deux registres 16 bits et un de 8 bit
          PSHS  PC,U,Y,X,DP,B,A   ;
```

A partir de 3 ou 4 registres, l'écriture peut paraître longue, surtout si ces instructions se répètent souvent. Le macro-assembleur autorise alors une définition globale des registres

```
RGT_S  REG   PC,U,Y,X,DP,D,CC   ; tous les registres sauf S
RGT_U  REG   PC,S,Y,X,DP,D,CC   ; tous les registres sauf U
RGABCC REG   A,B,CC             ; on peut aussi écrire  D,CC
RGUYX  REG   U,Y,X
RGUDCC REG   U,D,CC             ; on peut aussi écrire  U,A,B,CC
```

Dans la suite du programme, on peut écrire :

```
          PSHS  #RGT_U            ; Attention : le signe # obligatoire
          PULU  #RGT_S            ;             devant les symboles
          PSHS  #RGABCC!+RGUYX    ; cette écriture est équivalente à
                                  ; PSHS   U,Y,X,D,CC
```

Les noms des registres peuvent être spécifiés dans un ordre quelconque, comme pour les instructions portant sur les piles.

L'opérateur **!+** qui est en fait un OU inclusif permet d'associer les registres, il est le seul autorisé.

Plusieurs opérateurs en chaîne sont permis ex : `#RGA!+RGB!+RGCC`

A plus de trois définitions REG, au lieu d'utiliser l'opérateur !+ il est préférable d'utiliser la forme conventionnelle.

Quelques erreurs à ne pas commettre :

```
RGUS   REG   U,S      ; U et S ne doivent pas coexister
RGDB   REG   D,B      ; B est spécifié deux fois (avertissement)
```

Directive FCB (Form Constant Byte) Réservation d'un Octet)

Création d'une constante de 8 bits en mémoire, cet espace mémoire est initialisé à une valeur particulière placée dans le champ opérande.
FCB peut avoir des opérandes multiples séparés par une virgule.
La valeur de chaque opérande est tronquée à 8 bits puis est rangée dans un octet du programme objet.

Les constantes peuvent être une valeur numérique, une constante, un caractère, des autres noms de constantes ou des expressions.

L'étiquette placée devant les données prend la valeur de l'adresse courante.

L'opérande peut éventuellement être écrit sous forme d'une formule arithmétique comportant des noms de constantes, dans ce cas elle doit être définie par une directive EQU avant la ligne FCB.

```
                                        ;    prog GEN-01
                  0010  LGR   EQU   $10   ;
0000 4F                 VAL   FCB   $4F   ; place $4F dans la première case
                                          ; mémoire et lui assigne VAL
                                          ; La constante $4F est mise en mémoire
                                          ; à l'adresse pointée par le PC
                                          ; courant. Le PC est incrémenté de 1.
                                          ;
0100                            ORG   $0100            ;
0100 0C 10 41 00        DONN    FCB   $C,16,'A,,0,$0006 ; opérande multiple
0104 00 06                                              ; la , comme séparateur
0106 2D                         FCB   45               ;
0107 13                         FCB   %10011           ; opérande binaire
                                                       ; incomplètement rempli
0108 8A                         FCB   LGR*8+10         ; Opérande du type
                                                       ; expression
                                END                    ;
```

Les constantes sont dans le programme objet dans le champ Hexa opérande à raison de 4 octets par ligne.

Au chargement du programme en mémoire, les positions mémoire de $0100 à $0109 seront remplies par les valeurs ci-dessous :

```
{....} représente le contenu de la mémoire

{0100} = $0C  assignation simultanée de la valeur $0100 du compteur
              PC à l'étiquette DONN
{0100} = $0C  Valeur $C
{0101} = $10  conversion du nombre décimal 16 en hexa
{0102} = $41  code hexa du caractère "A"
{0103} = $00  il est permis de ne rien mettre entre deux virgule
              dans ce cas la valeur est nulle
{0104} = $00  c'est la valeur  0
{0105} = $06  c'est un faux opérande 16 bits. L'assembleur analyse
              entièrement l'opérande sur un mot de 16 bits,
              teste à posteriori l'octet le plus
              significatif. S'il est nul, comme c'est le cas,
              il n'y aura pas troncature, donc message
              d'erreur.
{0106} = $2D  équivalent hexa du nombre décimal 45
{0107} = $13  équivalent du nombre binaire %10011. Il est préférable
              d'écrire tous les 8 bits d'un nombre présenté
              sous la forme binaire %xxxxxxxx
{0108} = $8A  équivalent à (TAB * 8) + 10, la valeur de TAB étant 10
              ce qui fait en décimal (128 * 8) + 10 = 138 = $8A
```

Les formes suivantes sont incorrectes :

```
010A        FCB   LGR*8+$A0  ; après calcul, le nombre obtenu est $0120
                             ; l'octet le plus significatif MSB n'est pas nul
                             ; donc émission d'un message d'erreur
                             ; $0120 = 288 donc > à 127
                             ;
010B        FCB   $138,256   ; deux erreurs simultanées
```

FCB permet de réserver des cases mémoires afin de créer des tableaux de données. On donne une origine au tableau et on utilise la directive FCB, exemple :

```
                                ;----prog GEN-02
                    0002  Val     EQU    $02              ;
     3000                       ORG    $3000            ;
     3000 00 AA 19 FF       DEBTAB  FCB    0,$AA,25,@377,10 ;
     3004 0A                                             ;
```

Dans la mémoire à l'adresse $3000 on aura :

```
$3000 = $00
$3001 = $AA
$3002 = $19        ; $19 = 25
$3003 = $FF        ; $FF = @377  (octal)
$3004 = $0A        ; $0A = 10
```

Directive FCB et FDB exemples communs

On utilise ces pseudo code, pour, par exemple initialiser des tables. Par exemple

```
       ORG  $2000          ; En mémoire, à l'adresse $2000 on aura
       FCB  1,2,3          ;                    010203000400050006
       FDB  4,5,6          ;
```

On peut mélanger FCB, FDB, et EQU

```
       ORG  $2000          ; En mémoire, à l'adresse $2000 on aura
TOTO   EQU  *              ;                    010203000400050006
       FCB  1,2,3          ;
       FDB  4,5,6          ; et TOTO vaudra $2000
```

Le code juste au-dessus est EXACTEMENT pareil que le suivant :

```
       ORG  $2000
TOTO   FCB  1,2,3
       FDB  4,5,6
```

Le fait de mettre un label en début de ligne, permet d'affecter l'adresse courante au label

Directive FDB (Form Double constant Byte) (Réservation d'un Double Octet)

Crée une constante en 16 bits en mémoire. Presque identique à la directive FCB, mais cette fois on est en 16 bits. La directive FDB peut avoir un ou plusieurs opérandes qui sont alors séparés par des virgules.

La valeur codée sur 16 bits de chaque opérande est rangée dans deux octets consécutifs du programme objet. Le rangement commence à l'adresse courante et l'assembleur assigne au nom placé dans le champ étiquette la valeur de l'adresse courante.

Si la valeur des données se situe en dehors des 16 bits donc en dehors de la plage [$0000 , $FFFF], l'assembleur émet un message d'erreur.

L'étiquette placée devant les données prend la valeur de l'adresse courante.

La séquence ci-dessous illustre les formes autorisées de cette directive :

```
                        ;----prog GEN-03
0100                            ORG    $0100              ;
0100 23 45 00 24        ADR_1   FDB    $2345,36,'A        ; 3 data de 16 bits
0104 00 41                                                 ;
0106 01 06 00 00                FDB    *,,-3456           ; 3 data de 16 bits
010A F2 80                                                 ;
010C 00 18 00 F8                FDB    (*-ADR_1)*2,DEBTAB ; 2 data de 16 bits
                   00F8 DEBTAB  EQU    $F8                ;
0110 23 45 00 24        ADR_2   FDB    $2345,36,'A        ;
0114 00 41                                                 ;
0116 01 16 00 00                FDB    *,,-3456           ;
011A F2 80                                                 ;
```

Au chargement du programme en mémoire, les positions mémoire de $0100 à $010F seront remplies par les valeurs hexa ci-dessous :

```
{....} représente le contenu de la mémoire

{0100} = $23  assignation simultanée de la valeur $0100 du  compteur PC à
                    l'étiquette ADR_1
{0101} = $45  octet moins significatif de $2345
-------------------------------------------------------------
```

```
{0102} = $00  FDB place chaques données sur 16 bits,
{0103} = $24         la valeur 36 en décimale est égale à $0024
-------------------------------------------------------------
{0104} = $00  extension d'une donnée ASCII l'équivalent de
{0105} = $41         "A" en hexa est égal à $0041 en 16 bits
-------------------------------------------------------------
{0106} = $01  le signe * désigne en toute circonstance la valeur du
{0107} = $06         compteur programme au début de l'instruction
-------------------------------------------------------------
{0108} = $00  deux octets nuls remplacent le "rien" entre les
{0109} = $00         deux virgules
-------------------------------------------------------------
{010A} = $F2
{010B} = $80  nombre négatif qui est converti en hexa -3456 = $F280
-------------------------------------------------------------
{010C} = $00  le premier astérisque * représente la valeur du compteur
                     programme, soit $010C
{010D} = $18         (*-ADR_1)*2 = ($010C - $0100) * 2 = $0018 en 16 bits
-------------------------------------------------------------
{010E} = $00
{010F} = $F8  la valeur de la constante DEBTAB = $F8    $00F8 en 16 bits
-------------------------------------------------------------
{0110} = $23
{0111} = $45  valeur $2345
-------------------------------------------------------------
{0112} = $00
{0113} = $24  valeur 36 = $24    $0024 en 16 bits
-------------------------------------------------------------
{0114} = $00
{0115} = $41  valeur de A en hexa
-------------------------------------------------------------
{0116} = $01
{0117} = $16  valeur adresse courante donc $0116
-------------------------------------------------------------
{0118} = $00
{0119} = $00  valeur de 'rien' entre les deux virgules
-------------------------------------------------------------
{011A} = $F2
{011B} = $80  valeur de -3456 = $F280 (arithmétique signée sur 16 bits)
```

Lors d'une utilisation normale, la directive FCB inscrit les constantes qui seront sollicitées par la suite par des registres 8 bits du type A, B, DP ou CC.

Alors que la directive FDB est destinée à être employée pour le registre 16 bits du type PC, S, U, Y ou X.

Directive FCC (Form Constant Caractères) (Réservation d'un Bloc mémoire)

FCC permet de stocker en mémoire une chaîne de caractères ASCII.

Le premier octet est stocké à l'adresse courante. Autrement dit le nom placé dans le champ étiquette se voit assigner la valeur de l'adresse du premier octet de la chaîne.

Tout caractère ASCII imprimable codé de $20 à $7F peut se trouver dans la chaîne qui est enfermée entre deux délimiteurs identiques.

Le délimiteur est le premier caractère ASCII imprimable autre que l'espace situé après la directive FCC.
Il n'est donc plus nécessaire de les spécifier par le signe apostrophe '.

Dans l'assembleur **P30RS09** (que j'ai développé, disponible gratuitement), les délimiteurs autorisés sont :

- " Valeur Hexa $22
- | Valeur Hexa $7C

Dans le **P30RS09** la directive FCC peut contenir 10 données maximum par ligne, séparées par des virgules.

A la différence de FCB et FDB, la directive FCC inscrit dans les mémoires le code ASCII des caractères situés dans le champ opérande. $41 pour "A", $31 pour "1", $20 pour SPACE.

L'étiquette placée devant les données prend la valeur de l'adresse courante.

FCC est utile pour envoyer une chaîne de caractère sur un terminal, cette chaîne peut être :
-- Un message d'erreur
-- Un entête de programme
-- Une question ou une réponse
-- Une simple ligne de texte

```
                        ;----prog GEN-04
0000 45 72 72 65        MGS115  FCC    "Erreur AA115"     ; délimiteur est le "
0004 75 72 20 41                                          ;
0008 41 31 31 35                                          ;
000C 56 61 6C 65        Msg116  FCC    |Valeur "A"|       ; délimiteur est le |
0010 75 72 20 22                                          ;
0014 41 22                                                ;
0016 86   2D                    LDA    #45                ;
```

Dans le programme objet (code machine) après le champ Adresse met chaque caractère de la chaîne entre les délimiteurs à raison de 4 caractères par ligne.

Dans la partie hexa de l'opérande, on place la valeur du premier caractère, se sera ici le caractère E donc la valeur $45.

La valeur $0100 du compteur programme correspond à l'adresse du premier caractère de la chaîne affectée au symbole MSG115.

Répétition de caractère dans FCC

Dans le **P30RS09** il existe un format pour l'opérande de la directive FCC, un format plus commode pour mettre les titres ou les tabulations sur l'écran ou sur vers une imprimante :

```
0100 20 20 20 20     MSG216   FCC    (7/' '),"Titre 123"  ;
0104 20 20 20 54                                         ;
0108 69 74 72 65                                         ;
010C 20 31 32 33                                         ;
```

-- La valeur $0100 du compteur programme PCR en début de chaîne est affectée à l'étiquette MSG216
-- Le nombre décimal 7 (uniquement en décimal). Le chiffre doit être compris entre 1 et 99.
-- Le caractère / doit suivre immédiatement le chiffre.
-- Les 7 premiers octets seront remplis avec le code du caractère se trouvant entre les ' ' ici pour cet exemple se sera $20 (SPACE).

Pour répéter plusieurs fois le même caractère on utilise le format `(x/'c')` ou `(xx/'c')`

```
FCC  (20/'-'),"RESULTAT",(20/'-')
```

avec :

`x` ou `xx`	Un nombre en décimal sur 2 caractères maxi (1 à 99)
`/`	Un séparateur
`' '`	Les simples côtes, encapsulant le caractère à répéter
`c`	Le caractère ASCII a répéter dans les données de FCC, sauf le caractère ' ($27)

Insertion dans le texte de FCC d'une vigule ou d'un point-virgule

Si on souhaite mettre dans le texte :
-- Un point-virgule il faudra mettre le code **$3B**

```
0110 45 78 65 6D     FCC    "Exemple",$3B," pour le.. "  ;
0114 70 6C 65 3B                                         ;
0118 20 70 6F 75                                         ;
011C 72 20 6C 65                                         ;
0120 2E 2E                                               ;
```

-- Une virgule il faudra mettre le code **$2C**

```
0123 4D 6F 6E 74     FCC    "Monter le son",$2C," dans la.."  ;
0127 65 72 20 6C                                              ;
012B 65 20 73 6F                                              ;
012F 6E 2C 20 64                                              ;
0133 61 6E 73 20                                              ;
0137 6C 61 2E 2E                                              ;
```

Directive RMB (Reserve Memory Bytes) (Réservation d'octets en mémoire)

Voir aussi la directive BSZ

Sert à réserver des cases mémoires, soit un nombre d'octets égal à la valeur de l'opérande.
On peut utiliser une étiquette devant cette directive.
RMB provoque dans un programme un saut du compteur PC d'un nombre d'octets égal à la valeur spécifiée dans le champ opérande. Ces octets ne sont pas initialisés à une valeur donnée.

Parmi l'une des plus utilisées, cette directive assigne un ou plusieurs octets en mémoire pour un usage particulier. On utilise cette directive pour réserver une zone de "brouillon".

Les chiffres à gauche représentent les valeurs du compteur programme PC et non les numéros d'ordre. Ces nombres sont en hexadécimal et toujours codés sur 16 bits. Le signe $ n'est jamais utilisé pour les valeurs de PC. Cette convention a pour but d'alléger les écritures.

```
0080        RMB   16            ; réserve 16 octets à partir de l'adrs $0080
                                ;          L'instruction suivante sera donc
                                ;          mise 16 octets plus loin
0090        RMB   LGTAB         ; réserve un nombre d'octets égal à la
                                ;          valeur de LGTAB
1A00 DEBTAB RMB   LGTAB+$80     ; réserve LGTAB+$80 octets à partir de
                                ; l'adrs $1A00 et assigne simultanément
                                ; la valeur $1A00 au symbole DEBTAB
```

Bien qu'il existe apparemment une ressemblance de forme avec la directive EQU, la directive RMB fonctionne tout à fait différemment.

Pour RMB ce n'est pas la valeur de l'opérande qui est assignée à l'étiquette, mais une valeur 16 bits du compteur programme PC.

Tous les symboles dans l'opérande doivent être définis avant la rencontre d'une directive RMB. Ceci est dû au mode de fonctionnement de l'assembleur à deux passes.

Les exemples ci-dessous illustrent les possibilités d'emploi de cette directive comme moyen de réservation des mémoires pour le stockage des paramètres et des données temporaires.

```
0100        ORG   $100          ;
0101 IMGA   RMB   1              ; réservation d'un octet et assignation
                                 ; de l'adresse $0100 au symbole
                                 ; IMGA (registre image de A)
0101 IMGX   RMB   2              ; réservation de 2 octets et assignation
                                 ; de $0101 à IMGX
0103 TAB    RMB   $14            ; réservation de 20 octets pour un tableau
                                 ; et repérage de l'adresse $0103 pour le
                                 ; début du tableau par le symbole TAB
```

Plus loin dans le programme, les contenus mémoires aux adresses suivantes seront affectés si l'on rencontre des instructions du type :

```
            STA   IMGA           ; adrs $0100
            STX   IMGX           ; adrs $0101 et $0102
            STB   TAB+5          ; adrs $0108
            STX   TAB+10         ; adrs $010D et $010E
```

Les mémoires à lecture-écriture peuvent donc être réservées pour le stockage des données temporaires.
Ce type de réservation occupe en général la section finale d'un programme.

Autres exemples

```
            ORG   $2000          ;
     TOTO   RMB   $10            ; TOTO vaudra $2000
     TITI   EQU   *              ; TITI vaudra $2010
```

Et en mémoire entre $2000, et $2010, il y aura ????. On n'est pas capable de le dire.
-- Si c'est de la ROM, ça pourra être $00, ou $FF
-- Si c'est de la RAM, ça pourra être n'importe quoi (ce qu'il y avait avant en mémoire)

Pour pousser l'exemple encore plus loin : A quoi sert RMB alors ?

La directive RMB peut servir à définir une structure
Si on veut avoir les adresses d'un PIA, et l'adresser de manière indirecte, on peut définir

```
PiaSys            SET  $E7C8      ; PIA System & Printer

; Pia Systeme definition
;---------------------------------------
                  ORG 0
PiaSys.ddra       RMB 1
PiaSys.ora        SET PiaSys.ddra
PiaSys.ddrb       RMB 1
PiaSys.orb        SET PiaSys.ddrb
PiaSys.cra        RMB 1
PiaSys.crb        RMB 1
```

Dans le code 6809, on peut ainsi accéder au PIA de la manière suivante :

```
      LDX   #PiaSys
      LDA   PiaSys.ddra,X
```

Ou encore

```
      LDY  #PiaSys
      STB  PiaSys.cra,Y
```

Directive SETDP (SET Direct Page)

(désignation de la Page Direct à utiliser)

C'est une directive qui indique l'adresse d'une page mémoire (dans le 6809 pour 64Ko il y a 256 pages de 256 octets) où tout accès en mode étendu doit être converti en mode direct, ceci pour accélérer la vitesse d'exécution (voir "l'adressage Direct" avec page de base).

SETDP indique donc à l'assembleur quelle page de la mémoire utiliser en mode d'adressage en page directe**.**
Cette directive a pour but d'avertir l'assembleur de l'intention du programmeur.

Si la directive SETDP n'est pas utilisée se sera la page 0 qui sera utilisée par défaut pour l'adressage Direct.
La page 0 va de **$0000 à $00FF**
Le 6809 utilise 256 pages de 256 octets.

SETDP Configure l'assembleur, mais ne configure pas le 6809.

Pour ce faire il faut, en plus de la ligne SETDP, ajouter des autres lignes, LDA et TFR :

```
SETDP     $A0       ; configure l'assembleur
LDA       #$A0      ; ----
TFR       A,DP      ; ---- Configure le 6809
```

Exemple : La séquence ci-dessous définit une page de base dans l'espace mémoire, à l'exécution, s'étendant entre les adresses $1000 et $10FF.

Toute instruction écrite en mode direct prélève automatiquement la valeur $10 pour remplir l'octet le plus significatif (MSB) de l'adresse.

```
                  ORG  $8000       ; adresse de chargement du programme
8000 86 10        LDA  #$10        ; Cette séquence très caractéristique
                                   ; permet de charger la valeur $10 dans DP
8002 1F 8B        TFR  A,DP        ;
...                                ;
...                                ;
8100 B7 102A      STA  $102A       ; adrs à l'intérieur de la page de base
8103 F6 113A      LDB  $113A       ; adrs à l'extérieur de la page de base
8106 D7 3A        STB  $3A         ; adrs à l'extérieur de la page de base
```

La séquence ci-dessous "corrigée" sera comparée ligne par ligne à la séquence ci-dessus.

```
                  ORG  $8000       ;
8000 86 10        LDA  #$10        ; chargement de DP pour l'exécution
8002 1F 8B        TFR  A,DP        ;
                  SETDP  $10       ; pas de code objet généré.
...                                ; chargement de la page DP "fictive"
...                                ;
...                                ;
```

```
8100 97 2A      STA  $102A      ; 2 octets au lieu de 3
8103 F6 113A    LDB  $113A      ; code objet inchangé
8106 D7 003A    STB  $3A        ; 3 octets au lieu de 2
```

Voir aussi le registre DP et la rubrique Adressage Direct

Ci-dessous : les règles d'emploi de SETDP :

-- Elle n'a pas d'étiquette.

-- Le nombre de remises à jour de la page fictive est illimité

-- L'opérande peut avoir 3 formes connues : Constante, Etiquette ou Expression.

-- Dans le cas d'une expression, elle ne doit pas contenir des symboles définis plus bas dans le programme.

-- Lorsque la valeur d'une expression excède un mot de 8 bits, l'assembleur tronque le mot le plus significatif MSB et prend en compte l'octet le moins significatif LSB `SETDP $1028` chargera $28 dans le registre DP "fictif" avec un avertissement.

-- Il faut toujours spécifier dans l'opérande l'adresse comme s'il s'agissait d'un adressage étendu. L'assembleur choisit dans tous les cas le code optimal.

Dans l'exemple ci-dessus `STB $3A` est interprété comme `STB $003A` et non pas comme `STB $103A`.

Le programmeur n'a qu'à écrire `STB $103A` comme s'il s'agissait d'un adressage étendu; l'assembleur compare l'adresse avec sa page de base fictive et génère le code en mode direct, car $103A se trouve bien à l'intérieur de [$1000 , $10FF].

-- On a toujours la possibilité de forcer l'assembleur à prendre le mode d'adressage direct ou étendu en utilisant les signes < ou > devant l'opérande.

Toujours en se référant à l'exemple précédant, on peut écrire :

`STA`	`<$102A`	Forçage du mode direct. Inutile ! L'assembleur le fait automatiquement
`LDB`	`<$1113`	Forçage du mode direct pour une adresse située à l'extérieur de la page de base fictive. L'assembleur accepte en émettant un avertissement. On peut d'ailleurs écrire tout simplement `LDB <$13` le registre DP ayant la valeur `$11`
`STA`	`>$102A`	Forçage du mode étendu bien que l'adresse soit à l'intérieur de la page de base. Un code à 3 octets sera généré sans avertissement.
`LDB`	`<$1113`	Forçage du mode étendu pour une adresse à l'extérieur de la page de base. Inutile ! L'assembleur le fait automatiquement.

Autre exemple de l'emploi de la directive SETDP. Prenons comme exemple le programme suivant

```
00219               EFC2  VECTAB  EQU    *              ;VECTOR TABLE $EFC2
.
.
00342  F000 30   8C BF    BLTVTR  LEAX   <VECTAB,PCR    ;Origine table destination
00343  F003 1F   10              TFR    X,D            ;Obtient l'adresse de base
00344  F005 1F   8B              TFR    A,DP           ;
```

A la ligne 00342 la valeur de l'opérande est `$EFC2 - $F000 - $3 = $FFBF`

valeur de VECTAB	adresse courante	nombre d'octet pour cette ligne

Comme on force le résultat en 8 bits (présence du signe < devant l'opérande) on prendra **$BF** comme valeur de l'opérande

Comme **$BF** est un nombre négatif la valeur de l'offset pour cet instruction LEAX (adressage indexé direct) est de :
`$100 - $BF = $41` pour connaître le vrai déplacement
et `$F003 - $41 = $EFC2` cela nous donne `X = $EFC2`

A la ligne 00343 on transfère la valeur de X dans le registre D, ce qui donne `D = $EFC2`
Automatiquement la valeur du registre A passe à **$EF** et celui du registre B à **$C2**

A la ligne 00344 on transfère la valeur de A dans le registre DP, ce qui donne **DP = $EF**

Directive ZMB (Zero Memory Bytes)

La directive ZMB (ou BSZ) impose à l'assembleur de réserver un bloc d'octets et d'assigner à chacun de ces octets la valeur 0.
Le nombre d'octets est donné par l'opérande, qui ne doit pas contenir de nom de constante (par EQU) défini plus loin dans le programme ou de nom de constante non défini sous peine de provoquer une erreur à l'assemblage.

Programmes translatables

Un programme est translatable s'il fonctionne quelle que soit l'adresse à laquelle il est implanté. Un tel programme est implanté en mémoire et exécuté n'importe où.

Il est donc indépendant de sa position mémoire et ne contient pas de référence d'adresse en absolu, un tel programme n'utilise pas l'adressage étendu ou l'adressage direct mais plutôt l'adressage indexé

Contrairement au programme relogeable qui dispose d'adresses relatives transformées en adresses absolues après l'édition de liens, le programme translatable n'exige pas de passage par une édition de lien.

Dans un programme translatable il faut que les instructions faisant référence à la mémoire soient écrites en relatif, avec des possibilités de saut en avant et en arrière de 32 Ko.

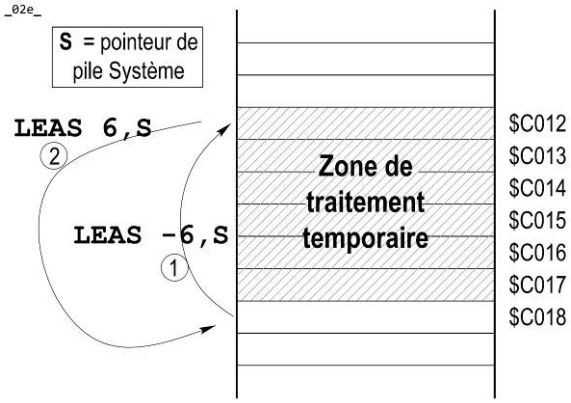
Le programme translatable peut être positionné par rapport au compteur ordinal PC grâce au mode d'adressage INDEXE (Direct ou indirect), dont la base est constituée par le PC lui-même (en absolue) et le déplacement codé sur 8 ou 16 bits.

On peut utiliser des zones de rangement temporaires sur la pile grâce à l'emploi de l'instruction chargement d'adresse effective.

Au début du programme l'instruction `LEAS -6,S` permet de déterminer une zone de travail temporaire sur la pile, cette zone se situe entre 0,S et 5,S.

Cette possibilité est intéressante pour les programmes translatables car on est certain de ne pas travailler dans une zone mémoire déjà utilisée et de ne pas détruire ainsi une partie du programme.

Il faut bien sûr, à la fin de l'utilisation de cette zone de traitement temporaire, réinitialiser le pointeur de pile avec l'instruction `LEAS 6,S` l'inverse de la celle ci-dessus.



Au début le pointeur de pile système S a la valeur $C018, puis S=$C012.
La zone de traitement temporaire se situe donc entre $C012 et $C017
L'instruction chargement d'adresse effective permet d'accéder à des tables placées dans un programme translatable.

Pour un programme translatables (programme indépendant de l'implantation en mémoire), il est à noter que l'on ne doit jamais utiliser l'adressage étendu. Tout l'adressage doit être relatif au PCR en utilisant les instructions LEAX, LEAY.

L'adressage indexé utilisant le compteur programme PCR comme base permet de charger dans l'index X l'adresse du début de la table.
L'adressage indexé permet ensuite d'accéder aux données de la table.

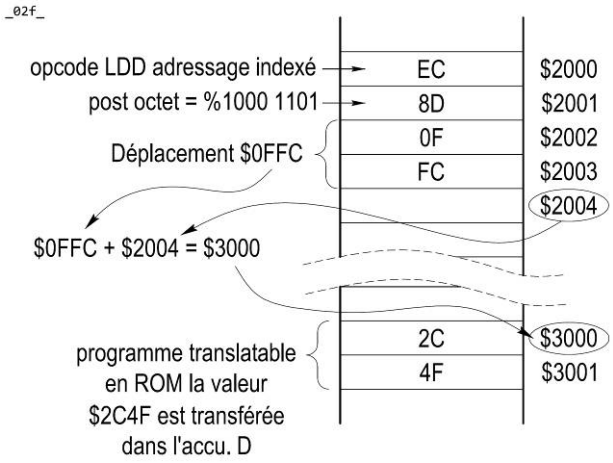
```
LEAX   TABLE,PCR       ; PC + Déplacement → X
```

On peut également avoir accès à des constantes placées dans un programme translatable. Cet accès se fait de manière indépendante de la position en utilisant l'indexation par rapport au compteur programme PC.

```
LDD    ETIQ,PCR        ; valeur à l'adrs ETIQ → D
```

Cette adresse n'est pas connue au moment de l'assemblage du programme translatable.
Il faut intégrer la ROM dans l'application ce qui définit automatiquement l'étiquette ETIQ et qui permet de calculer le déplacement correspondant à l'écart entre la valeur du compteur programme courant et l'adresse ETIQ.

```
LDD    GISSE,PCR    ; instruction LDD placée en $2000
                    ; programme GISSE est placé en $3000
```



Autre Exemple :

Soit un programme résident dans la plage **`$8000---$8FFF`** (soit 4 Ko) et possédant un tableau de donnée dans la plage **`$8000---$807F`**
Supposant qu'une instruction implantée à l'adresse **$8200** ait besoin d'une donnée 8 bits à l'adresse **$8000**

Dans un mode étendu on écrirait

```
8200  B6 8000        LDA   $8000
```

Si au lieu de la plage **$8000---$8FFF** on désire charger l'ensemble du programme dans la plage **$2000---$207F** le code à l'adresse $2200 serait toujours égal à **`B6 8000  LDA $8000`**

Le problème est que le programme ne se trouve plus en $8000, l'instruction **`LDA $8000`** fait appel à une adresse localisée à l'extérieur du tableau, ce dernier étant maintenant dans la plage **$2000---$207F**

Pour rendre opérationnel le nouveau programme il faudrait réécrire le programme source en

```
2200 B6 2000          LDA   $2000
```

Le mode indexé avec déplacement relatif au PC permet de résoudre ce problème de translabilité.
Le tableau de données étant toujours au même endroit **$8000---$807F** l'instruction **`LDA $8000`** est transformé en

```
8200 A6 8D  FDFC     LDA   $8000,PCR
8204
```

A6	OpCode du mode indexé
8D	Post byte indiquant qu'il y a mode indexé à déplacement sur 16 bits relatif au PC
FDFC	représente le code opérande (en 16 bits signé) qui mesure la distance relative entre l'adresse **$8000** et l'adresse de fin d'instruction **$8204**.

C'est un offset 16 bits qui est négatif dans cet exemple.

OFFSET =	Adrs inscrite	-	Adrs fin d'instruction
$FDFC =	**$8000**	**-**	**$8204**

A l'exécution, le processeur effectue le calcul inverse pour retrouver l'adresse de la donnée.

Adrs de la donnée =	Adrs fin d'instruction	+	OFFSET
$8000	**=** **$8204**	**+**	**$FDFC**

Lors d'un chargement de l'ensemble du programme à une autre adresse, par exemple entre **$2000---$2FFF** le même code objet se retrouve en **$2200**

```
2200 A6 8D  FDFC     LDA   $2000,PCR
8204
```

Cependant à la lecture des deux octets A6 et 8D, le processeur effectue le calcul suivant pour retrouver l'adresse de la donnée.

```
$2000 =    $2204       +  $FDFC
```

Cette fois-ci, la donnée se trouve bien en $2000 et non en $8000 comme dans la situation du mode étendu.

On dit que le programme en entier est rendu translatable, car aucune retouche du code objet n'est nécessaire lors d'un chargement à une adresse différente de l'adresse servant à la phase d'assemblage.

<u>Pour plus de renseignement concernant les programmes translatables</u>, voir l'ouvrage 03 "Programmation du 6809" par Rodnay ZAKS et Willam LABIAK (édition SYBEX). Page 206 et 207, concernant les PIC (Position Independant Code) Code indépendant de la position ou banalisation de l'implantation en mémoire.
<u>Voir également</u> : L'ouvrage n°06 Livre de BUI MINH DUC aux pages IV.23 et III.25.

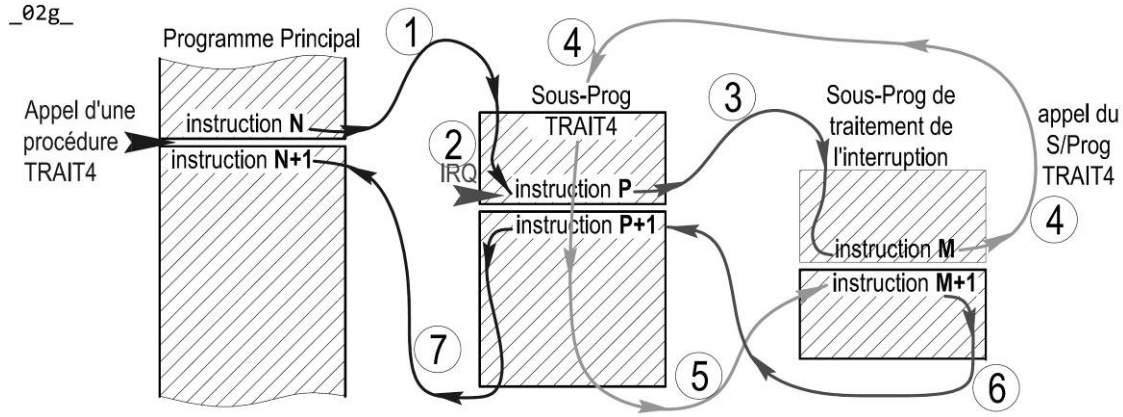
Programme réentrant

Un programme réentrant est un programme qui peut être utilisé à n'importe quel niveau de priorité.
La notion de réentrance n'est pas liée qu'aux interruptions, c'est aussi lié à la récursivité.

Si une interruption intervient pendant que le 6809 exécute une tâche dont le niveau de priorité est le plus bas, le traitement est interrompu.

Le programme d'interruption pourra utiliser la même séquence de programme que celle qui était traitée avant l'interruption, si cette séquence est réentrante, c'est-à-dire quelle permet de traiter la demande d'interruption de niveau plus élevé sans pour cela perturber les informations et les résultats de la tâche initiale.

La figure ci-dessous met en évidence le concept de la réentrance.



1. Appel d'un sous-programme appelé par exemple TRAIT4, à partir de l'instruction N.

2. Durant le déroulement du S/Prog TRAIT4 le 6809 reçoit une interruption IRQ| d'un niveau de priorité plus important que celui du traitement en cours (IRQ| n'est pas masquée). Le 6809 arrête le traitement du S/Prog TRAIT4 après l'instruction P.

3. Le 6809 va exécuter le programme de traitement d'interruption IRQ|.

4. Dans le programme de traitement de l'interruption IRQ|, à l'instruction M, il y a un appel au S/Prog TRAIT4.

5. Dès que le S/Prog TRAIT4 est achevé, on retourne au programme de traitement de l'interruption à l'instruction M+1.

6. Dès que le programme de traitement de l'interruption est terminé, on retourne au traitement initial du sous-programme TRAIT4, à l'instruction P+1.

7. Dès que le sous-programme TRAIT4 est terminé, on retourne au programme principal à l'instruction N+1.

Pour que toutes ces opérations se déroulent normalement, il faut :
-- D'une part que tous les registres internes du 6809 soient sauvegardés, ce qui est fait automatiquement sur la pile système.
-- D'autre part préserver toutes les données intermédiaires.

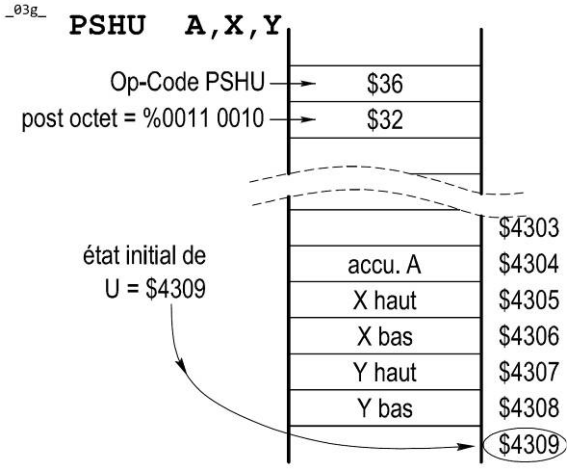
Ces données intermédiaires doivent être sauvegardées dans des endroits différents, ceci en fonction du niveau de priorité; seule l'utilisation d'une ou de plusieurs piles permet cela.

La réentrance est donc possible si le microprocesseur possède plusieurs pointeurs de piles qui permettront de gérer plusieurs zones de sauvegarde affectées aux différents niveaux de priorité.

Dans l'exemple précédent, les données intermédiaires utilisées lors du premier appel que la procédure sont sauvegardées dans la zone 1. Celle du second appelle dans la zone 2.

Le 6809 possède deux pointeurs de pile S et U dont la gestion est facilitée grâce aux instructions PSHU, PSHS et PULU, PULS qui permettent de ranger ou extraire de la pile n'importe quel registre.
Ces deux pointeurs à son gérés de façon hardware, l'ordre de rangement fixé.

L'instruction `PSHU A,X,Y` avec `U = $4309` occupe l'espace mémoire suivant :



Ces instructions PSHU, PSHS et PULU, PULS permettent donc une manipulation aisée des piles ce qui facilite l'écriture de programmes réentrants.

Il est de plus possible d'utiliser les registres d'index X et Y comme pointeurs de pile gérés par logiciel.

Le mode d'adressage indexé auto-incrémentation/décrémentation par un ou deux permet de gérer des piles logicielles dont les pointeurs sont X ou Y.

Exemple avec X = $2007

L'instruction `STA ,-X` sauvegarde le contenu de A à l'adresse `$2007 - $1`
L'instruction `LDA ,X+` va chercher le contenu de A dans la pile

De la même façon, on peut travailler sur tous les registres du 6809.

Les instructions suivantes permettent une sauvegarde des registre A, Y et U dans une pile logicielle (avec X = $2007)

```
STA  ,-X
STY  ,--X
STU  ,--X
```

La restitution des registres s'effectue en sens inverse :

```
LDU  ,X++
LDY  ,X++
LDA  ,X+
```

Sous Programmes

Lorsque l'on appelle un sous-programme, il existe trois méthodes pour transmettre des données en paramètres :

-- Par les registres

Plus avantageuse si les registres sont disponibles, pas de nécéssité de recourrir à des emplacements mémoire fixe. Le sous programme rest indépendant de la mémoire.

-- Par la mémoire

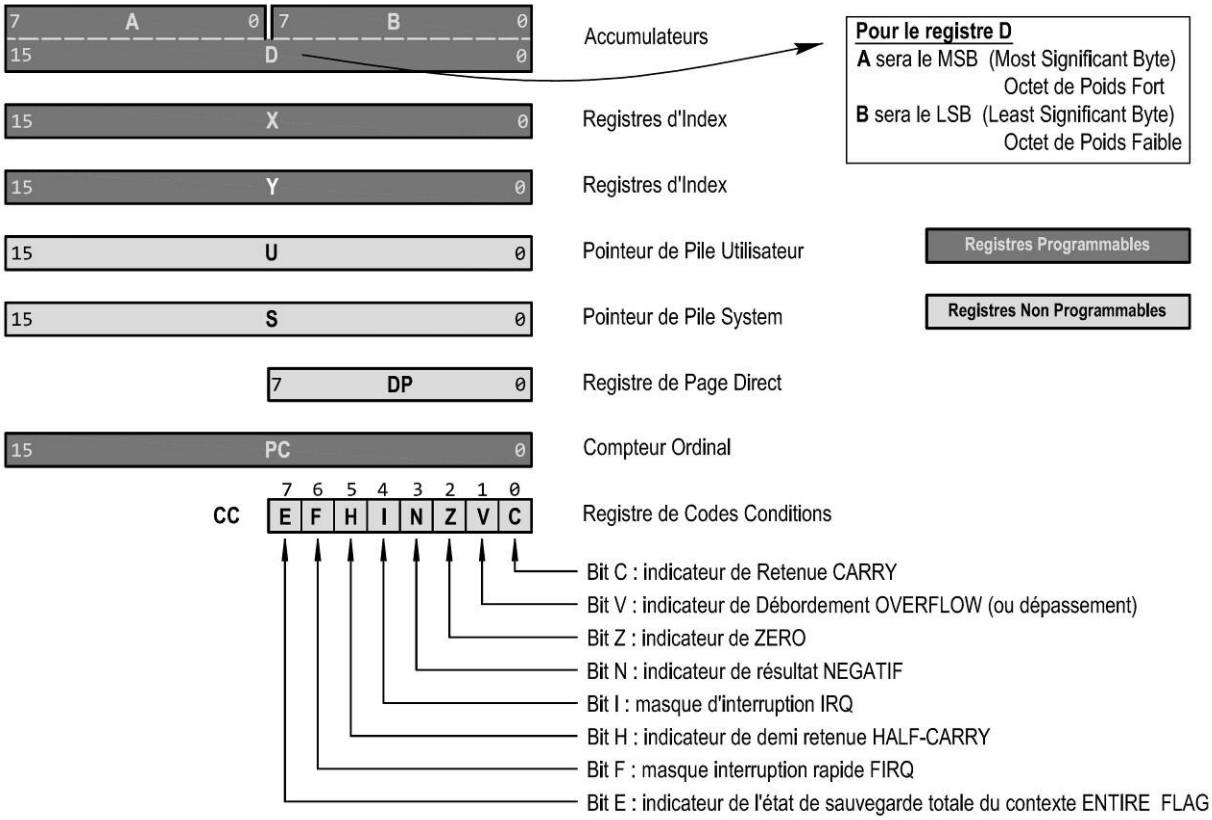
Offre la plus grande souplesse, mais moins performant. Cette méthode rest par définition dépendante de la mémoire.

-- Par la pile

Même avantage que la méthode A. Indépendant de la mémoire. Inconvénients ça rempli la pile de données et par conséquent réduit le nombre possible de niveau d'appels d'autres sous progammes.

Le plus souvent on recherche à rester le plus possible indépendant des adresses mémoires

_15a_



Les accumulateurs A, B et D

On effectuera dans ces deux registres toutes les opérations arithmétiques et logiques. Ils sont pour cela entièrement identiques et mis à part les instructions ABX et DAA, les registres A et B jouent exactement le même rôle sur 8 bits.

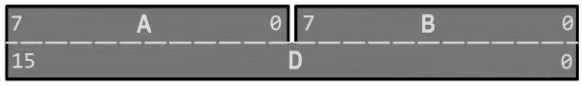
Deux accumulateurs de 8 bits, jouant exactement le même rôle.
Ils sont utilisés pour :

-- Les instructions arithmétique et logique.
-- Les instructions de comparaisons.
-- Les transferts de :
  -- mémoire → accumulateur
  -- accumulateur → mémoire
  -- accumulateur → registre.

A et B peuvent se réunir pour former un accumulateur **D** de 16 bits.

-- Le registre A représentant les poids Forts de l'accumulateur D MSB (Most Signifiant Bit)
-- Le registre B représentant les poids Faibles de l'accumulateur D LSB (Least Signifiant Bit)

_15c_



Les registres d'index X et Y

Ils sont dits registres d'index, car ils permettent d'adresser tout l'espace mémoire avec en plus la capacité d'être pré-décrémenté ou post-incrémenté pour faciliter le traitement de variables en tables.

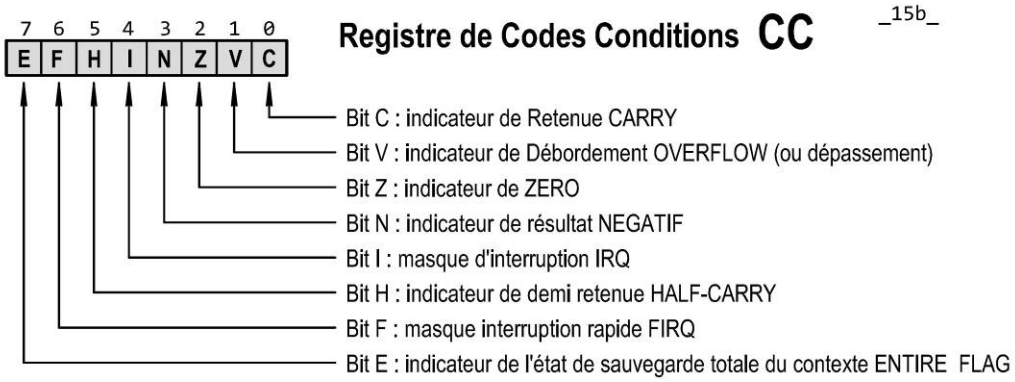
De préférence, il faut utiliser le registre X à la place du registre Y car les instructions comme LDY, STY, et CMPY sont plus longues à exécuter que LDX, STX et CMPX. X et Y sont en 16 bits.

Sont utilisés dans le cadre du mode d'adressage indexé direct et indexé indirect.
Ils peuvent être utilisés comme des registres d'usage général :

-- Pour stocker des résultats intermédiaires
-- Comme compteur de boucle.

Le registre de condition CC Condition Code (appelé aussi Registre d'état)

Ce registre définit à tout instant l'état du 6809 à la suite de l'exécution d'une instruction. Chaque bit joue un rôle important pour les instructions de saut ou de branchements conditionnels.



Chaque bit du registre CC est mis à 0 ou à 1 :
-- Soit lors de l'exécution d'une instruction
-- Soit par le programme en forçage direct.

Autrement dit, le registre CC définit l'état du 6809 après chaque opération arithmétique ou logique. Il aide le 6809 à prendre des décisions, sous la forme de branchements absolus ou relatifs.

Indicateurs arithmétiques :
_ _ H _ N Z V C C correspond au bit 0
Ils sont positionnés en fonction du résultat des instructions qui manipulent les données.

Indicateurs d'interruption :
E F _ I _ _ _ _ E correspond au bit 7
Ils sont liés au fonctionnement en interruption. Les états de chaque bit peuvent dépendre d'un calcul antérieur.

Les bits I et F sont utilisé comme masque d'interruption.
Lorsque l'un ou les deux bits sont définis les entrée IRQ et FIRQ ne sont pas reconnus.

Certaines instructions particulières peuvent modifier ou lire la valeur de certains bits de CC.
Certaines instructions de branchement ont un déroulement qui dépend de la valeur prise par ces bits.

Bit E (Entire flag) sauvegarde des registres dans la pile bit b7

Indique l'état de la sauvegarde des registres lors d'une interruption. Il indicateur la sauvegarde totale ou partielle d'un contexte. Il est très peu utilisé.

Mis à 0 Une partie seulement des registres est sauvegardée pendant le traitement de l'interruption.
Cas de l'interruption FIRQ ou seul les registre PC et CC sont sauvegardé.

Mis à 1 Tout le contexte est sauvegardé dans la pile système.
Cas de l'interruption IRQ ou tous les registres sont sauvegardé dans la pile S (System).

Ce bit E différencie les deux modes d'interruption FIRQ et IRQ et indique au 6809 le nombre de registre à dépiler lorsque le 6809 aura fini d'exécuter le programme d'interruption qui doit se terminer par l'instruction RTI (ReTurn from Interrupt).

Bit F (Fast interrupt mask) Masque d'interruption rapide bit b6

Il conditionne le traitement de la broche FIRQ|.

Il est positionné par le programmeur à l'aide des instructions ANDCC ou ORCC.

Mis à 0 : Lorsqu'il est à 0 ce bit F autorise le traitement des interruptions FIRQ.

Mis à 1 : pour interdire le traitement des instructions FIRQ. Le 6809 dans ce cas ne tient pas compte des demandes d'interruption arrivant sur cette broche FIRQ|.

Seules les interruptions NMI (Nom Masquable Interrupt) et SWI (SoftWare Interrupt) sont acceptées

Ce bit F est mis à 1 par les interruptions NMI, SWI et FIRQ. Il n'est donc pas affecté par les interruptions IRQ

Si les bits F et I ont été positionnés simultanément à 0, le 6809 accorde une priorité supérieure à la demande d'interruption rapide FIRQ.

Quand il y a demande d'interruption FIRQ, (à l'inverse de la demande IRQ), seul le registre PC et le registre d'état CC sont sauvegardés dans la pile système S et le bit E (Entire Flag), Etat de sauvegarde, est mis à 0.

Bit H (Half Carry) Demi retenue ou retenue intermédiaire bit b5

Ce bit indique si lors d'une opération, il y a une retenue à faire entre le bit 3 et le bit 4

Mis à 0 : S'il n'y a pas de retenue.

Mis à 1 : S'il y a retenue du bit 3 sur le bit 4.

Il intervient dans les opérations sur les chiffres codés en BCD (Binary Coded Decinal).

En code BCD chaque chiffre compris entre 0 et 9 est codé par un groupement de 4 bits.

Exemple : Le chiffre décimal 16 sera représenté sous la forme :

```
0001 0110 en BCD      au lieu de     0001 0000  en hexadécimal
  1      6                             1      0
```

Exemple : Addition de 9 + 9 = 1 2

```
0000 1001  +  0000 1001  =  0001 0010
```

Cet exemple conduit à un chiffre 12 en notation BCD Ce qui est incorrect

La demi retenue bit H indique alors au processeur qu'il faut ajouter encore 6 à ce résultat pour avoir une représentation correcte de 18 en BCD

```
0001 0010  +  0000 0110  =  0001 1000
                              1     8
```

Est un bit de demi retenue entre les 4 octets de poids Faibles
et les 4 octets de poids Forts.

Il n'y a pas d'instruction de branchement conditionnel testant ce bit, mais il faut savoir que l'on peut tester ce bit H en utilisant les instructions :

```
TFR    CC,A     pour ne pas modifier le registre CC et tester après le registre A
ANDCC  $20      ce qui transfère la valeur du bit H dans le bit Z.
```

Fonctionne comme le bit C mais au lieu de détecter le dépassement de capacité du bit 7, le bit H détecte un dépassement de capacité au niveau du bit 3 (utile pour les opérations codées BCD)

Ce bit H est mis à 1 si lors d'une opération une retenue passe du bit 3 au bit 4 dans l'octet concerné par l'opération. Dans ce cas le bit H est appelé retenu du bit 3.

L'instruction d'ajustement décimal DAA utilise ce bit H et le bit C pour corriger le résultat après une addition 8 bits du type ADDA ou ADCA.

Il n'existe pas d'instruction de branchement attribué à ce bit H, qui représente un intérêt mineur.

Exemples :

```
$02 + $17 = $19      -->  H sera mis à 0
$08 + $19 = $21      -->  H sera mis à 1

   0000   1000     = $08
   0001   1001     = $19
   -----------     -----
   .... 1 0001
```

Bit I (Interrupt mask) Masque d'interruption bit b4

Il est positionné par le programmeur à l'aide des instructions ANDCC ou ORCC.

Il permet d'inhiber (de masquer) les demandes d'interruption de la forme IRQ.

Mis à 0 : Lorsqu'il est à 0 ce bit **I** autorise le traitement des interruptions IRQ.

Mis à 1 : Par le programmeur pour interdire le traitement des instructions IRQ. Le 6809 dans ce cas ne tient pas compte des demandes d'interruption arrivant sur cette broche IRQ.

Ce bit est positionné à 1 par un RESET.

Quand il y a demande d'interruption IRQ, tous les registres sont sauvegardés dans la pile système S et le bit E (Entire Flag), Etat de sauvegarde, est mis à 1.

Bit N (Négatif) bit b3

Ce bit indique si le résultat d'une opération est négatif.

Il prend la valeur du bit de poids le plus fort d'un résultat, d'un octet ou d'une données 16 bits qui a été transférée.

Toute opération arithmétique ou logique, susceptible de modifier le bit le plus significatif des registres 8 ou 16 bits affecte le bit N.

Ce bit est positionné à la valeur du bit de poids fort du résultat d'une opération.

Mis à 0 : Si le bit 7 (bit de signe) résultat d'une opération est à 0.
Mis à 1 : Si le bit 7 (bit de signe) résultat d'une opération est à 1.

Dans le mode en complément à 2 si le bit de poids fort est à 1 cela indique que le nombre est négatif.

Il indique un résultat négatif, pour toutes les instructions, ce bit N prend la valeur du bit de poids fort de l'opérande ou de l'accumulateur en mouvement.

Il est très utile lors de travaux signés sur les entiers compris entre +127 et -128, il a donc une signification que pour les nombres signés.
On peut se servir du bit N pour déterminer la valeur du bit n°7 de l'octet résultat.

Dans une arithmétique signée
Les chiffres compris entre $0 et $7F (ou $7FFF) sont des chiffres Positifs
Les chiffres compris entre $80 et $FF (ou $8000 ou $FFFF) sont des chiffres Négatifs

Les opérations de chargement LD.... et de stockage ST.... agissement également sur ce bit N. comme pour les opérations arithmétique et logique.

Les instructions CLR, CLRA, CLRB, LSR, LSRA, LSRB mettent le bit N à 0.

Lors d'un dépassement de capacité dû à une opération utilisant le complément à deux, ce bit est incorrect.

Aussi, lors d'une telle opération, le signe est donné par l'opération logique N+V N ou V

Bit Z (Zero) Indicateur de zéro bit b2

Il indique un résultat nul de l'opération précédante, toutes les instructions positionnent ce bit.

Ce bit Z est utilisé pour indiquer que le résultat de l'opération précédante est Nul.

Le bit Z est souvent employé avec les instructions de comparaison pour indiquer qu'il y a correspondance.
Ce bit est positionné à 1 lorsque le résultat de l'opération précédente est nul.

Attention, ce bit est également positionné par les opérations de chargement LDA, LDB, ... et de stockage STA, STB ....

Mis à 0 Quand le résultat d'une opération quelconque produit un résultat non nul.
si (A ∧ Mém) ≠ 0 instruction BITA
si (B ∧ Mém) ≠ 0 instruction BITB

Mis à 1 Quand le résultat d'une opération quelconque produit un résultat égal à 0 (résultat null).
si (A ∧ Mém) = 0 instruction BITA
si (B ∧ Mém) = 0 instruction BITB

Bit V (oVer flow) Débordement bit b1

Il indique, lors d'une opération utilisant un complément à deux, s'il y a un dépassement de capacité
Le bit V est le résultat d'un OU EXCLUSIF entre les bits b6 et b7 de l'octet en question.

Autrement dit le bit V détecte un changement accidentel du signe résultant d'un débordement.

Le bit V est le bit de débordement en complément à 2.
Il indique un résultat supérieur à ce qu'un octet peut représenter en binaire signé.

Seules les opérations comme ADD, ADC, SUB, SBC, NEG et CMP positionnent le bit V à la valeur appropriée.

La multiplication non signée (instruction MUL) et les opérations de décalages à droite n'affectent pas le bit V.

Mis à 0 : (pas de dépassement de capacité)
Dans les opérations de chargements, de stockages et de transfert, dans les opérations logiques, dans les instructions TST, TSTA, TSTB.

Mis à 1 : (il y a dépassement de capacité)
Dans le cas de nombres non signés (de 0 à 255), s'il y a dépassement de capacité.
Les opérations arithmétiques sont seules à mettre le bit V à 1.

Quand le résultat d'une opération arithmétique ne peut être représenté correctement par le contenu des registres.

S'il y a dépassement de capacité (dans le cas ou le 6809 manipule des nombres signés de -128 à +127). Il est donc positionné si le résultat d'une opération arithmétique en complément à 2 déborde.

Bit V à 1 lorsque :
-- Soit si il y a une retenue du bit 6 vers le bit 7 ceci sans retenue du type Carry (bit C)
-- Soit il n'y a pas de retenue du bit 6 vers le bit 7, par contre il y a une retenue Carry (bit C)

Rappel :
Dans le cas de nombre signé (de -128 à +127) le bit de poids fort (bit 7) sert de signe :
-- 0 si le nombre est positif
-- 1 si le nombre est négatif

Exemple 1 pour le bit V : Addition de $4B et $71

```
   $4B      0100 1011    positif
 + $71   +  0111 0001    positif
------   ------------
 = $BC   =  1011 1100    résultat négatif
```

Le bit V = 1 car il y a retenue du bit 6 vers le bit 7 sans CARRY

Les 2 nombres (dans le cas arithmétique signé) $4B et $71 sont positif mais le résultat est négatif. Il est donc nécessaire d'indiquer que le résultat est erroné bit V = 1.

Exemple 2 pour le bit V : Addition de 2 nombres négatifs $-01 et $-05

```
   $-01        1111 1111   négatif
 + $-05   +    1111 1011   négatif
-------   ----------------
 = $-06   = (1) 1111 1010
```

Dans ce cas le bit V est mis à 0 (retenue du bit 6 vers le bit 7)
Et le bit C à 1

Lorsque V est positionné à 0 le résultat de l'addition est OK
Lorsque V est positionné à 1 le résultat de l'addition est faux

Bit C (Carry) Retenue bit b0

Ce bit prend la valeur 1 chaque fois que le résultat d'une instruction arithmétique ou logique dépasse 8 bits, c'est à dire nous avons une retenue.
Il est positionné lors d'une opération arithmétique uniquement (addition, soustraction, multiplication, comparaison, négation, complément à 2, décalage à gauche ou à droite).

Le bit C joue un double rôle :
-- Sert à indiquer si une opération d'addition ou de soustraction à produit une retenue.
-- Sert de neuvième bit dans les opérations de décalage et de rotation, le bit C indique dans ce cas une retenue ou un repport.

Donc les déplacements de données et les opérations logiques n'affectent pas ce bit.

Mis à 0 : (il n'y a pas de retenue)
Par les instructions CLR, CLRA, CLRB.

Mis à 1 : (Il y a une retenue à effectuer)
Quand le résultat d'une opération arithmétique ne peut être représenté correctement par le contenu des registres.
Par les instructions de complémentation à 1, exemple les instructions : COM, COMA, COMB.
Dans le cas d'une addition dont le résultat est supérieur à 255 ($FF) ou lorsque le résultat d'une soustraction (SUB, NEG, CMP, SBC) et positif.

Dans le cas de l'instruction MUL (A multiplié par B résultat sur 16 bits dans D), le bit de **C** est égal au bit b7 du registre D

Cas de l'addition : exemple effectuer la somme de $8A et de $D5

```
      $8A          1000 1010
   +  $D5       +  1101 0101
  -------     ---------------
   = $15F     = (1) 0101 1111
                    |
                    └──────────mise à 1 du bit C
```

On voit que le résultat est un nombre de 9 bits.
Le bit **C** est positionné à 1 chaque fois qu'il y a une retenue sur le bit de plus fort poids.

Rappel : en arithmétique Signé on ignore la retenue le bit C.

Le registre PC (Program Counter) ou PCR (Program Counter Register) le compteur ordinal

Un programme est exécuté par le microprocesseur de manière séquentielle.

Lorsque le microprocesseur exécute une instruction, il doit connaître l'adresse de la prochaine instruction. Le registre PC contient l'adresse de la prochaine instruction à exécuter

Chaque fois que le microprocesseur va chercher un octet en mémoire, le registre PC (16 bits) est incrémenté de 1 et pointe donc sur l'adresse de la prochaine instruction à exécuter.

Autrement dit, le registre PC détermine l'adresse mémoire à laquelle le 6809 doit exécuter une instruction. Il sert donc au 6809 pour stocker l'adresse de la prochaine instruction à exécuter. Il est donc modifié à chaque instruction exécutée.

Appelé compteur programme, il est utilisé par le 6809 pour pointer l'adresse de la mémoire devant être lue et décodé par l'unité centrale à l'étape suivante.

Le registre PC s'incrémente automatiquement à chaque lecture d'un octet a moins qu'une instruction de branchement ou de saut oblige le registre PC à prendre une autre valeur particulière.

Il peut donc être modifié par :
- -- Une instruction de saut
- -- Une interruption
- -- Une instruction de branchement
- -- Un transfert d'un registre 16 bits dans le registre PC.
- -- Par un retour de fonction RTS, d'un retour d'interruption RTI ou d'un dépilement **PULS PC** ou **PULU PC**

Certaines instructions du 6809 considèrent le registre PC comme un registre d'index au même titre que les registres X et Y, à la différence près que l'index varie automatiquement avec l'avancement du programme.

Cette propriété intéressante autorise, entre autre, l'écriture des programmes entièrement translatables dans tout l'espace mémoire.

Le registre PC peut également être utilisé comme registre d'index dans un des modes d'adressage.

Les registres S et U les pointeurs de PILE

S et U ont un fonctionnement identique. Ils opèrent en mode dernier entré - premier sorti (LIFO Last In First Out)

Comme pour les mémoires vives, les piles servent également à stocker les données ou les adresses à la différence près que la gestion des piles relève d'une procédure très ordonnée.

La rentrée d'une donnée dans la pile, ne détruit pas la donnée précédente et dans le cas du 6809, des instructions à deux octets peuvent charger l'ensemble des registres du 6809.

L'organisation des piles du 6809, mérite une étude approfondie car les erreurs commises dans l'usage des piles sont assez difficiles à détecter car les pointeurs de pile n'ont pas d'adresses fixes.

S et U peuvent à l'occasion servir de enregistre d'index avec la totalité des possibilités de X et de Y.

Le microprocesseur doit connaître à tout instant l'adresse de la prochaine instruction à exécuter.

Le programme principal se déroule jusqu'à l'appel au sous-programme.

Le compteur ordinal registre PC se charge avec l'adresse de début du sous-programme.

Puis il y a exécution du sous-programme jusqu'à la rencontre d'une instruction de retour.

A ce moment le microprocesseur ne sait plus à quelle adresse il doit reprendre le déroulement du programme principal, il faut donc pouvoir mémoriser ces adresses.

Le 6809 possède deux zones mémoire qui servent de PILE grâce aux registres **S** et **U** :

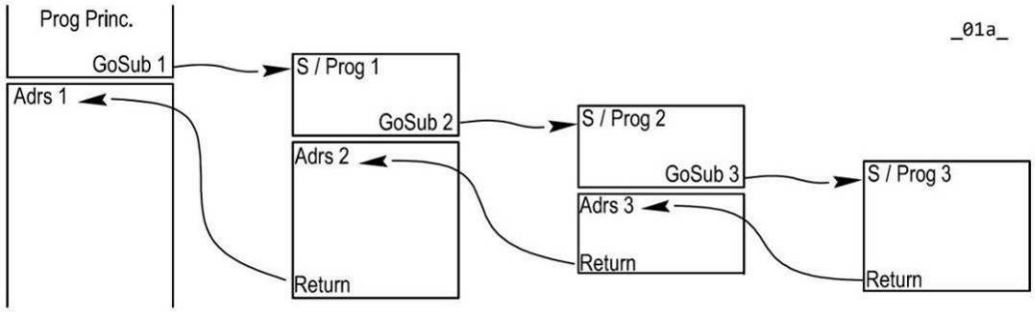
Registre S

(16 bits) pour la pile **S**YSTEME.
Le registre S est utilisé automatiquement et contrôlé exclusivement par le processeur.
Lors d'appels de sous-programmes et lors des interruptions le registre S peut être utilisé par le programmeur.
Il appartient d'office au 6809, il s'en sert pour mémoriser les états de la machine lors de l'exécution des sous-programmes (saut à un sous-programme) et en cas d'interruption. Il est donc utilisé automatiquement par le 6809
Sauvegarde des données nécessaires au fonctionnement du 6809 pour les adresses de retour de sous-programme et le contenu de certains registres internes dans le cas d'interruptions.

Registre U

(16 bits) pour la pile **U**TILISATEUR
Il est entièrement réservé à l'utilisateur.
Le programmeur peut l'utiliser pour transférer ses propres paramètres lors de l'appel des sous-programmes, de la même façon que X et Y.
Il peut être utilisé pour le stockage temporaire de certains résultats et d'aller les retrouver rapidement.

S et U peuvent être installés n'importe où dans l'espace adressable. Leur contenu doit être programmé chaque fois que le 6809 est mis sous tension, par l'intermédiaire d'instruction spécifique.



_01a_

La pile système contiendra les octets suivants après l'appel du sous-programme n°3 :
Si le contenu de S était à l'initial **$1F07** par exemple, la répartition des adresse serai la suivante :

_01b_

	7 ... 0
$1F01	Adrs S/Pgm 3 H
$1F02	Adrs S/Pgm 3 L
$1F03	Adrs S/Pgm 2 H
$1F04	Adrs S/Pgm 2 L
$1F05	Adrs S/Pgm 1 H
$1F06	Adrs S/Pgm 1 L
$1F07	

Lors du stockage d'une adresse de retour de sous-programme les opérations suivantes sont effectuées :
-- Le pointeur de pile est décrémenté
-- L'octet de poids faible de l'adresse de retour est chargé
-- Le pointeur de pile système est décrémenté
-- L'octet de poids fort de l'adresse de retour est chargé

Lors d'une instruction de retour de sous-programme l'opération inverse se produit :
-- Le compteur ordinal est chargé par l'octet de poids fort puis par celui de poids faible de l'adresse de retour.
-- Le pointeur est incrémenté de 2, l'incrémentation se fait après le chargement (contrairement au cas d'un appel de sous-programme.

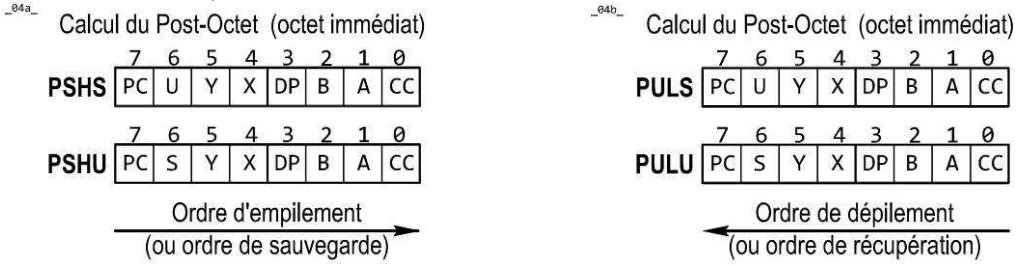
Le registre U (utilisateur) fonctionne de la même façon que le registre S.

Les pointeurs de pile S et U pointent sur la dernière donnée stockée dans la pile.

La Pile

Le 6809 ne travaille pas toujours avec des emplacements mémoires connus, il se sert très souvent d'une pile.

Les registres à empiler ou à dépiler sont indiqués dans l'octet qui suit immédiatement le code hexa du mnémonique. Voir le descriptif de cet octet dans l'explication des instructions PSHS, PSHU, PULS et PULU.



Pour ce Post-Octet (Post-byte) chaque bit est spécifique d'un registre à sauvegarder.

Lorsque un de ces bits est à 1, cela indique que le registre est concerné dans l'opération d'empilement ou de dépilement.

Elle peut être implantée n'importe où dans l'espace mémoire à l'aide des instructions
`LDS` **`LDU`** **`TFR X,S`** **`TFR Y,U`** **`EXG D,S`** **`EXG Y,U`**

Ou tout autre mode autorisé de ces 6 instructions,

Exemple :
`LDS #$8100` qui définit une pile système à l'adresse $8100
L'expansion de la pile s'effectue par la suite vers les adresses basses en commençant par l'adresse $80FF qui reçoit le nom de "haut de pile".

La **pile S** est gérée ultérieurement par un ensemble d'instructions du type :

`PSHS`	qui "pousse" les données dans la pile S (système)
`PULS`	qui "retire" les données de la pile S
`JSR` ou **`BSR`**	qui "pousse" la valeur du PC (compteur programme) dans la pile S pour sauvegarder l'adresse de retour d'un sous-programme.
`RTS`	qui "restitue" l'adresse de retour du programme appelant à la fin d'exécution d'un sous-programme.
`LEAS`	qui "déplace" volontairement le pointeur à n'importe quel autre endroit de la pile, ce déplacement relatif est compté à partir de la position courante du pointeur.
`SWI, SW2, SW3, RTI`	action sur une ligne d'interruption.

La **pile U** est gérée par les instructions du type :

`PSHU`	qui "pousse" les données dans la pile U (utilisateur)
`PULU`	qui "retire" les données de la pile
`LEAU`	qui "déplace" le pointeur relativement à n'importe quel emplacement mémoire.

Il y a aussi **`LDA ,U+`** **`STA ,U+`** **`LDX ,U++`** etc.…

La pile est une portion de mémoire vive destinée à la sauvegarde temporaire des informations.
Ce sont des emplacements mémoires pour lesquels le CPU possède un générateur d'adresses spécifiques.

Ce générateur d'adresses utilise le concept d'empilement et de dépilement sous la règle du dernier entrée, premier sorti.
On parle de pile LIFO (**L**ast **I**n, **F**irst **O**ut) (Dernier entré, Premier sorti) La donnée poussée en dernier dans la pile doit être retirée en premier.

Cependant, on peut accéder à n'importe quelle donnée en déplaçant volontairement les pointeurs avec les instructions du type LEA.…

La pile **S** (Système) et la pile **U** (Utilisateur) sont des registres internes au 6809. La présence d'une pile système va faciliter :
-- Les retours après interruptions
-- Les retours de sous programmes
-- Le stockage de données temporaires

La pile commence toujours à l'adresse la plus haute et elle régresse, le pointeur étant décrémenté à chaque empilement.
Le sommet dune pile est en permanence repérée, son pointeur S ou U qui :
-- Décrémente avant qu'un octet soit écrit dans la pile.

-- Incrémente lorsqu'un octet vient d'être lu dans la pile.

Pointeur de pile SP (Stack Pointer)

-- SP contient l'adresse du sommet de la pile située en mémoire.

-- Une pile est indispensable pour les interruptions et les sous-programmes.

-- Structure LIFO (**L**ast **I**n, **F**irst **O**ut) (Dernier entré, Premier sorti)

-- On ne peut accéder à la pile que par des instructions du type :
- -- Push (empiler) avec les instructions PSHS et PSHU
- -- Pull (dépiler) " " " PULS et PULU

-- La présence d'une pile simplifie l'exécution :
- -- des sous-programmes.
- -- des interruptions.
- -- du stockage temporaire des données.

Chaque fois qu'un registre 8 bits est mis dans une pile, le pointeur de pile est décrémenté de 1.
Chaque fois qu'un registre 16 bits est mis dans une pile, le pointeur de pile est décrémenté de 2.

L'ordre d'empilement de 16 bits met d'abord l'octet de poids faible dans la pile, puis celui de poids fort.

Le registre de page direct DP (Direct Page Register)

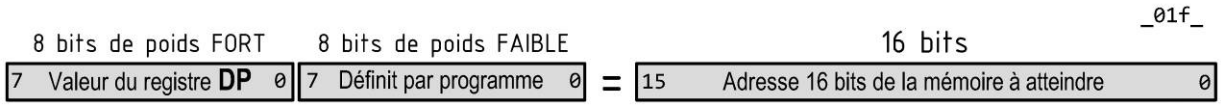
Format 8 bits, utilisé uniquement dans la mode d'adressage Direct.

Il forme la partie haute de l'adresse à pointer dans le cas d'un adressage direct.
Il est automatiquement remis à 00 par un RESET.

Dans le mode d'adressage direct, une mémoire de 16 bits requiert deux octets en mémoires.

Par contre dans ce même mode d'adressage direct avec la page DP, requiert seulement un octet en mémoire et un cycle machine en moins.

L'unité centrale UC prélève automatiquement le contenu du registre DP pour constituer les 8 bits de poids les plus fort de l'adresse. Seuls les 8 bits de poids faibles doivent être spécifiés par le programmeur.



Octets de poids FAIBLE…….. LSB (Least Signifiant Bit) mis dans l'opérande
Octets de poids FORT…….. MSB (Most Signifiant Bit)

Le mode d'adressage a pour but d'exécuter le code binaire et d'accroître la vitesse d'exécution dans les échanges de données avec les périphériques rapides.

MODES D'ADRESSAGE DU 6809

Les modes d'adressage complètent le rôle rempli par les instructions et permettent au 6809 d'accéder à n'importe quelle case mémoire interne ou externe.

Divers types d'adressages.

1 Inhérent
2 Immédiat #
3 Direct <
4 Etendu >
5 Etendu indirect [ ]
6 Indexé
- Avec déplacement constant Nul
- Avec déplacement constant 5 bits
- Avec déplacement constant 8 bits
- Avec déplacement constant 16 bits
- Avec déplacement du contenu d'un accumulateur A ou B
- En auto incrémentation de 1
- En auto incrémentation de 2
- En auto décrémentation de 1
- En auto décrémentation de 2
- Relatif au PCR

7 Indexé indirect [ ]
- Avec déplacement constant Nul
- ~~Avec déplacement constant 5 bits~~
- Avec déplacement constant 8 bits
- Avec déplacement constant 16 bits
- Avec déplacement du contenu d'un accumulateur A ou B
- ~~En auto incrémentation de 1~~
- En auto incrémentation de 2
- ~~En auto décrémentation de 1~~
- En auto décrémentation de 2
- Relatif au PCR

8 Relatif Court
9 Relatif Long

Ces **9 modes d'adressage** (9 façons de coder les adresses) et les **59 instructions** font du 6809 le meilleur microprocesseur 8 bits de l'époque, lui procurant 1464 solutions possibles.

Définitions – Données

Les informations traitées par le 6809 se présentent sous la forme de données codées sur 8 ou 16 bits.
A part quelques instructions comme NOP, SYNC qui sont dépourvues de données, les autres instructions impliquent toujours une transformation ou un mouvement des données.

Dans une instruction de branchement l'adresse de branchement est considérée comme une donnée.
Une donnée est ce que contient un registre ou une mémoire. Elle est encore appelée contenu du registre ou contenu de la mémoire.

Définitions – Adresses

A une donnée est affectée une adresse qui indique l'endroit où la donnée se trouve.
Ces cases de rangement (adresses) portent un code sur 16 bits.
L'espace mémoire interne est très restreint, il correspond à l'espace que prend les registres internes du 6809 c'est-à-dire les registres PC, S, U, Y, X, DP, A, B, D et CC.
L'espace mémoire externe est très étendu, chaque case mémoire porte un numéro codé en hexa.

Voici les points repères à retenir en hexadécimal.

$10 = 16	$1000 = 4096 (4 Ko en jargon informatique)
$80 = 128	$2000 = 8192 (8 Ko en jargon informatique)
$FF = 255 valeur maxi d'un mot de 8 bits	$4000 = 16384
$800 = 2048	$8000 = 32768
$FFFF = 65535 valeur maxi d'un mot de 16 bits	

Définitions – Conventions d'écriture

-- Une adresse mémoire sera toujours écrite en base Hexadécimale.

-- Pour ranger une donnée de 16 bits le 6809 mettra :
-- l'octet de poids **Fort** (MSB) dans l'adresse **Base**
-- l'octet de poids **Faible** (LSB) dans l'adresse **Haute**

La syntaxe assembleur indique que l'adresse Base, quelle que soit la longueur de la donnée.

Une donnée 16 bits = **MSB** & **LSB**
MSB ..... (Most Significant Byte) le plus significatif à l'adresse n.
..... **LSB** (Least Significant Byte) le moins significatif à l'adresse n+1.

Exemple

si `{X} = $4A3F` {.....} = contenu de

Après exécution de `STX $F000` On aura
`{$F000} = $4A`
`{$F001} = $3F` sous une forme plus condensée `{$F000}{$F001} = $4A3F`

Définitions – Utilité des différents modes d'adressage

La principale tâche du 6809 consiste à traiter les données contenues dans les mémoires.

Les modes d'adressage peuvent, en complément aux instructions :
-- Spécifier que la donnée se trouve dans un registre interne du 6809.
-- Spécifier le numéro de la case mémoire à traiter dans l'espace mémoire externe.
-- Indiquer la distance relative en octets de l'adresse à traiter par rapport à une adresse de référence.
-- Indiquer une adresse intermédiaire ou le 6809 peut trouver une adresse effective.

Le 6809 comporte ainsi plusieurs possibilités différentes pour spécifier une adresse de donnée.
L'ensemble de ces possibilités constitue les modes d'adressage du 6809.

Cette méthode évite la multiplication des instructions, au lieu de créer des mnémoniques différents, on a préféré conserver intact le champ opération de l'assembleur et faire varier les modes d'adressages, donc les écritures dans le champ opérande.

Le mnémonique est inchangée, par contre c'est l'op-code qui changera en fonction du mode d'adressage.

Définitions – Notion d'adresse effective EA

Quelque soit le mode d'adressage, à l'exécution, le 6809 détermine toujours l'adresse finale dont le contenu est à modifier.
Cette détermination fait intervenir un certain nombre d'adresses intermédiaires.
L'ultime adresse dont le contenu sera traité est désignée comme ADRESSE EFFECTIVE.

Exemple

`ADDA B,X` Ajouter à A le contenu d'une case mémoire dont l'adresse s'obtient en ajoutant les contenus de B et de X

```
{A} = $45
{B} = $16
{X} = $1024
```

{.....} = contenu de

On aura
`{B} + {X} = $16 + $1024 = $103A`

Comme le contenu de l'adresse `{$103A}` est égal à `$26`

alors après exécution de `ADDA B,X` on aura `{A} = $6B` `$45 + $26 = $6B`

Définitions – l'indirection

L'espace mémoire du 6809 est constitué de cases de 8 bits.

Lors du stockage d'une adresse, deux octets adjacents sont nécessaires, on est en présence d'un degré d'indirection. Ce degré n'est jamais poussé au-delà de 1.

Exemple

Adressage DIRECT

```
LDA     $F080

{A} = $...      {.....} = contenu de
{F080} = $6C

Après exécution
{A} = $6C
```

adressage INDIRECT

```
LDA     [$F080]

{A} = $...
{F080} = $30
{F081} = $B5
Adresse effective =  $30B5
Et à l'adresse effective il y a  $4F

Après exécution
{A} = $4F
```

L'adressage INHÉRENT

Appelé aussi adressage Implicite.

Le code opératoire contient toute l'information nécessaire à l'exécution de l'instruction. Il n'y a pas d'opérandes.

Cet adressage opère exclusivement sur les registres.

N'est pas en réalité un véritable mode d'adressage, car toutes les informations nécessaires à l'exécution de l'instruction se trouvent groupées dans de mnémonique (ou code opération appelé aussi op-code).

L'adressage Inhérent est utilisé par des instructions qui agissent sur les registres internes et non pas sur la mémoire. Ces instructions n'ont pas besoin qu'on leur ajoute des opérandes.

```
ABX,  ASLA, ASLB, ASRA, ASRB, CLRA, CLRB, COMA, COMB, DAA,  DECA, DECB
INCA, INCB, LSLA, LSLB, LSRA, LSRB, MUL,  NEGA, NEGB, NOP,  ROLA, ROLB
RORA, RORB, RTI, RTS,  SEX,  SWI, SWI2, SWI3,  SYNC, TSTA, TSTB
```

Exemple :

l'instruction **ABX** `{X} + {B} → {X}` {…..} = contenu de

Avant exécution `{X} = $8034  et  {B} = $15`

Après exécution `{X} = $8049  et  {B} = $15`

L'adressage IMMÉDIAT

Le code opératoire est directement suivi par un opérande de 1 ou 2 octets.

```
CMPA   #$2B06        ; Comparaison de A avec la valeur hexa 2B06;
ADDB   #%1001100     ; Addition de la valeur binaire 1001100 à B.
```

On trouvera toujours le signe **#** pour notifier le mode immédiat.

Dans cet adressage le mnémonique est suivi d'un opérande de 1 ou 2 octets.

Cet opérande permet d'initialiser les registres du 6809, ou de fournir des valeurs à une variable.

Exemples :

```
ADDA #$05       ; soit additionner  $05  au contenu de A
CMPS #$12F3     ; soit comparer le contenu de  S  avec la valeur $12F3
CMPX #$4BF6     ; comparaison de X avec la valeur hexa $4BF6
ADDD #%010110   ; addition de la valeur en binaire %010110
```

Ces instructions peuvent atteindre 4 octets comme LDS ou CMPU

En mémoire on aura {.....} = contenu de

```
Exemple :  8F00   LDA  #$27     {8F00}=86      {8F01}=27
Exemple :  8C00   LDU  #$2506   {8C00}=CE      {8C01}=25   {8C02}=06
```

L'adressage DIRECT <

Le symbole **<** précise l'adressage direct. Voir également la directive d'assemblage SETDP.

Ce mode d'adressage n'est pas à utiliser pour les programmes translatables.

L'adresse effective 16 bits à atteindre est "scindée" en deux :

-- L'octet de poids Fort **MSB** (Most Significant Byte), est fourni par le contenu du registre de page direct **DP**.
Cette valeur peut être chargée dans le registre **DP** par l'intermédiaire de l'instruction `TRF A,DP`

-- L'octet de poids Faible **LSB** (Least Significant Byte), est dans l'opérande.

Donc en collaboration avec le registre **DP** (Direct Page register), ce mode permet de charger un accumulateur avec le contenu de n'importe quelle adresse mémoire et uniquement avec 2 octets de code source..

Le registre **DP** permet de spécifier l'une des 256 pages de 256 octets à utiliser
(pour rappel 256 x 256 = 64 536 octets).

La page 0 étant les 256 premiers octets des 64 Ko de l'espace total du 6809. La page 0 va de `$0000 à $00FF`

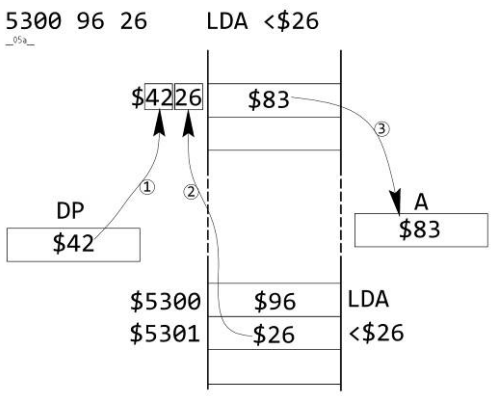
Après une mise sous tension le registre DP est toujours à zéro.

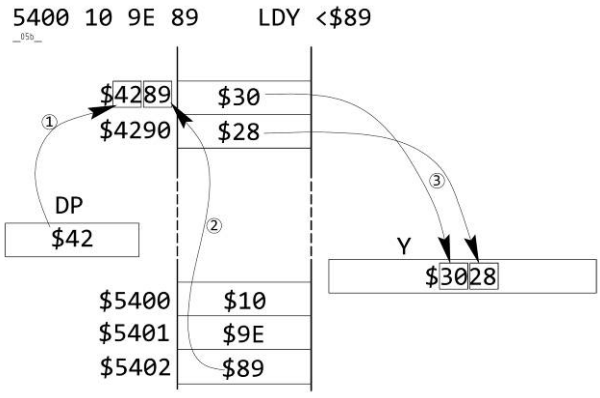
Exemples avec DP = $20

```
LDA    <$55      charge A avec {$2055}                {.....} = contenu de
CMPX   <$35      compare X avec {$2035} et {$2036}
```

Le contenu du registre DP doit être préalablement chargé, pour fixer le registre DP il faut utiliser l'instruction TFR pour transférer la valeur de A ou de B dans DP.

```
                    ;----prog MA-07
                0010  Val    EQU    $10    ;
                0042         SETDP  $42    ; configure l'assembleur
0000 86   42                 LDA    #$42   ; }
0002 1F   8B                 TFR    A,DP   ; }-- configure le 6809,
                                           ; initialisation du registre DP
5300                         ORG    $5300  ;
5300 96   26                 LDA    <$26   ; charge A avec {$4226}
                                           ;
5400                         ORG    $5400  ;
5400 109E 89                 LDY    <$89   ; charge Y avec {$4289} et {$4290}
```





L'exécution en mode Direct est plus rapide que le mode étendu, du fait de l'économie d'un octet en mémoire et d'un cycle machine.

A l'assemblage, le programmeur doit écrire l'instruction comme en mode étendu, la traduction de l'instruction en code "mode direct" n'est effectuée que si la page de base virtuelle est correctement positionnée par la directive SETDP.

L'adressage ÉTENDU >

C'est un mode qui va concerner le contenu de n'importe quel octet de la mémoire et donc permet d'atteindre toute la plage mémoire du 6809 mais avec un opérande à 2 octets.

Ce mode d'adressage n'est pas à utiliser pour des programmes translatables (Programmes indépendants de l'implantation en mémoire).

Pour ce mode le symbole d'assemblable est **>**.

Exemples :

`LDA >$50` L'accumulateur **A** ne sera pas chargé par le nombre `$50` comme dans l'adressage immédiat. Mais il sera chargé par le contenu de la case mémoire à l'adresse `$0050`.

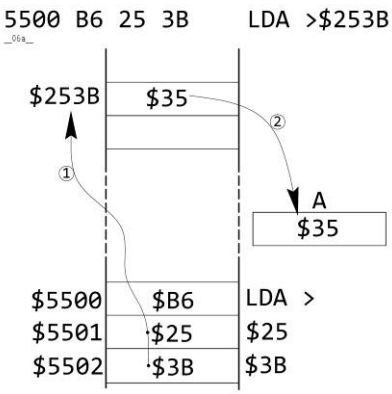
`LDA >$5100` Charge le registre A avec le contenu de l'adresse `$5100`.

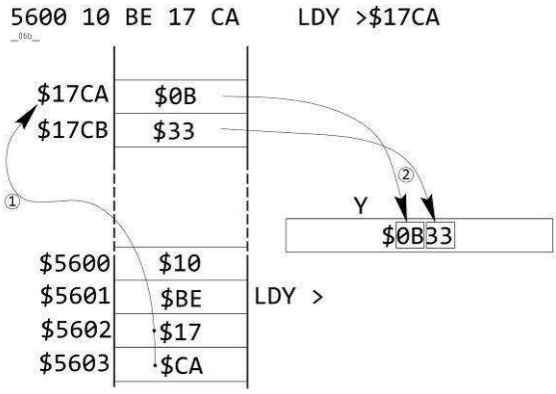
L'adressage ÉTENDU est la façon la plus simple de donner une adresse, de donner "en clair" cette adresse à l'aide d'un nombre de 4 chiffres hexa. L'opérande est ici une adresse de 16 bits.

L'adressage Etendu spécifie toujours une adresse effective (adresse particulière) qui ne change pas pendant l'exécution du programme. Donc ce code ne peut être utilisé dans un programme indépendant de la position mémoire.

Cette adresse effective peut être écrite dans n'importe quelle base pourvu que sa valeur soit dans la plage **[$0000 , $FFFF]**

```
                    ;----prog MA-08
                0010  Val    EQU    $10     ;
5500                         ORG    $5500   ;
5500 B6   253B               LDA    >$253B  ; charge A avec {$253B}
                                            ;
5600                         ORG    $5600   ;
5600 10BE 17CA               LDY    >$17CA  ; charge Y avec {$17CA} et {$17CB}
```





L'adressage ÉTENDU indirect [ ] groupe 14 B

C'est le contenu de l'adresse citée comme opérande qui va indiquer l'adresse mémoire à manipuler.

Est identique au mode Etendu mais il possède en plus une indirection. On accède à une Adresse Effective en passant par une adresse intermédiaire.

A l'assemblage l'op-code est celui du mode indexé et un post-octet fixe de valeur **$9F** est inséré entre l'op-code et l'adresse spécifiée dans le champ opérande. La longueur globale de l'instruction atteint alors 4 ou 5 octets.

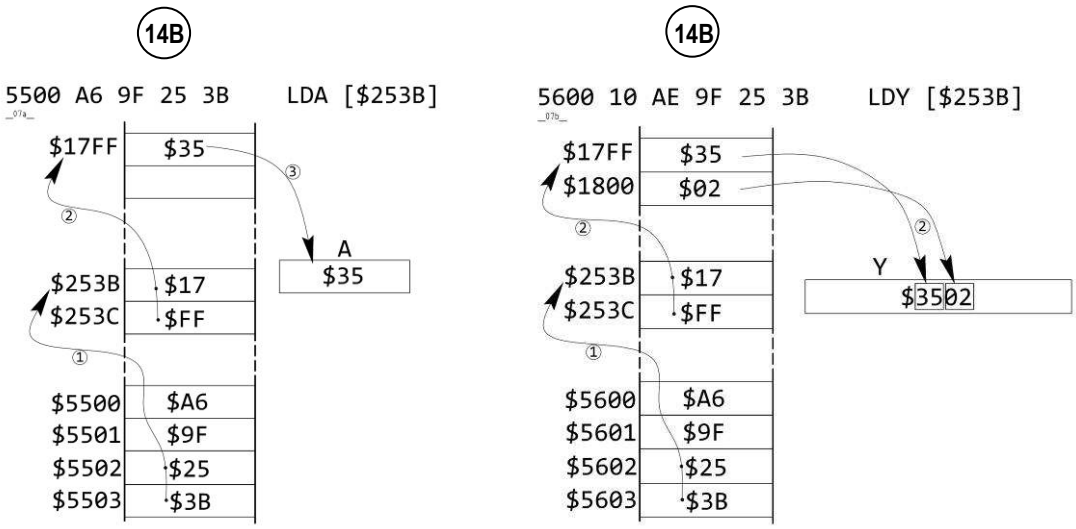
Supposons le contenu mémoire s `{$253B} = $17`
`{$253C} = $FF` {…..} = contenu de

L'instruction `LDA [$253B]` on charge le registre A avec le contenu de l'adresse `$17FF`

Cet adressage est très efficace lorsqu'une routine unique doit traiter différentes valeurs. La routine n'a pas besoin d'être doublée ce sont des points de repère qui bougent.

On y trouve aussi une grande utilité dans l'emploi de "tableaux d'adresse" et non des tableaux de données. Ces tableaux d'adresses sont souvent localisés au début ou à la fin d'un programme.

```
                    ;----prog MA-09
                0010  Val     EQU     $10         ;
5500                          ORG     $5500       ;
5500 A6   9F 253B             LDA     [$253B]     ;
                                                  ;
5600                          ORG     $5600       ;
5600 10AE 9F 253B             LDY     [$253B]     ;
                              END                 ;
```



Les adressages INDEXES

Ce mode d'adressage en à utiliser pour les programmes translatables.

Dans ce monde d'adressage, un registre d'index 16 bits (X, Y, U, S ou PCR) spécifie une base à laquelle on ajoute un déplacement signé de 0, 5, 8 ou 16 bits.

EA (Effective Adresse) L'adresse effective de l'opérande est obtenue par la somme :

-- du registre d'index choisi X, Y, U, S ou PCR

-- et un déplacement appelé Offset qui suit immédiatement le code opération.

L'offset, indique la distance relative en octets entre l'adresse effective et l'adresse de base.

```
  EA  =  Offset  +  Base
Effective     |          |
Adresse       |          |_____Base peut être  X, Y, S, U  ou  PCR
              |
              |_______Déplacement peut être :
                              -- Constant (nul, 5, 8 ou 16 bits)
                              -- Variable   utilisation de A, B ou D
                              -- Auto Incrémenté sur X, Y U ou S
                              -- Auto Décrémenté sur X, Y U ou S
                              -- Depuis PC en 8 bits
                              -- Depuis PC en 16 bits
```

Ce déplacement peut-être défini par une constante faisant partie de l'instruction ou par le contenu d'un des autres registres du 6809.

Ce mode permet aussi la pré-décrémentation ou la post-incrémentation simple ou double.

Ça permet de travailler sur des emplacements mémoires adjacents, possibilités intéressante pour les traitements des tables de données, par exemple pour accéder aux éléments successifs d'un bloc de données.

Toutes ces possibilités sont précisées par le post octet qui suit le code opératoire.

le mode indexé nécessite un post-octet destiné à renseigner le 6809 sur le type exact d'adressage à utiliser.

A noter que l'offset Nul (01A) et (01B) ainsi que l'offset sur 5 bits (1bit de signe + 4 bits) (02A) ont une taille en mémoire minimum et donc une grande vitesse d'exécution.

On peut forcer le mode 8 bits ou le mode 16 bits, il suffit de mettre le signe < ou > devant l'opérande :

- `<` adressage direct
- `>` adressage étendu

```
TST    $83,Y          ; génerera 3 octets déplacement sera sur 8 bits

TST   >$83,Y          ; génerera 4 octets déplacement sera sur 16 bits

NEG    <DEPLAC,X      ; forcera le mode 8 bits, par contre si le déplacement n'est pas dans
                        l'intervalle [ -128 , +127 ] une erreur sera produite lors de l'assemblage.
```

Tableau Regroupant Tous les Types d'Adressage Indexé

	Offset	Base	Groupe	INDEXÉ direct Syntaxe	Binaire	Post Byte	Nbre Cycles	Nbre d'Octets	Groupe	INDEXÉ indirect Syntaxe	Binaire	Post Byte	Nbre Cycles	Nbre d'Octets
A déplacement constant	Nul	X	01A	0,X ou ,X	1000 0100	84	0	0	01B	[0,X] ou [,X]	1001 0100	94	3	0
		Y		0,Y ou ,Y	1010 ''''	A4	0	0		[0,Y] ou [,Y]	1011 ''''	B4	3	0
		U		0,U ou ,U	1100 ''''	C4	0	0		[0,U] ou [,U]	1101 ''''	D4	3	0
		S		0,S ou ,S	1110 ''''	E4	0	0		[0,S] ou [,S]	1111 ''''	F4	3	0
	5 bits s = bit de signe dddd valeur de n de -16 à +15	X	02A	n,X	000s dddd (bit 64, bit 32)	10 à 1F / 00 à 0F	1	0	Décalage = 0	Si n est positif (s = 0) PostByte = n + Décalage Si n est Négatif (s = 1) PostByte = n + 32 + Décalage				
		Y		n,Y	001s dddd	30 à 3F / 20 à 2F	1	0	Décalage = 32					
		U		n,U	010s dddd (n négatif)	50 à 5F / 40 à 4F (n positif)	1	0	Décalage = 64					
		S		n,S	011s dddd	70 à 7F / 60 à 6F	1	0	Décalage = 96					
	8 bits de -128 à +127	X	03A	n,X	1000 1000	88	1	1	03B	[n,X]	1001 1000	98	4	1
		Y		n,Y	1010 ''''	A8	1	1		[n,Y]	1011 ''''	B8	4	1
		U		n,U	1100 ''''	C8	1	1		[n,U]	1101 ''''	D8	4	1
		S		n,S	1110 ''''	E8	1	1		[n,S]	1111 ''''	F8	4	1
	16 bits de -32 768 à +32 767	X	04A	n,X	1000 1001	89	4	2	04B	[n,X]	1001 1001	99	7	2
		Y		n,Y	1010 ''''	A9	4	2		[n,Y]	1011 ''''	B9	7	2
		U		n,U	1100 ''''	C9	4	2		[n,U]	1101 ''''	D9	7	2
		S		n,S	1110 ''''	E9	4	2		[n,S]	1111 ''''	F9	7	2
Déplacement = accu A, B ou D	Accu A	X	05A	A,X	1000 0110	86	1	0	05B	[A,X]	1001 0110	96	4	0
		Y		A,Y	1010 ''''	A6	1	0		[A,Y]	1011 ''''	B6	4	0
		U		A,U	1100 ''''	C6	1	0		[A,U]	1101 ''''	D6	4	0
		S		A,S	1110 ''''	E6	1	0		[A,S]	1111 ''''	F6	4	0
	Accu B	X	06A	B,X	1000 0101	85	1	0	06B	[B,X]	1001 0101	95	4	0
		Y		B,Y	1010 ''''	A5	1	0		[B,Y]	1011 ''''	B5	4	0
		U		B,U	1100 ''''	C5	1	0		[B,U]	1101 ''''	D5	4	0
		S		B,S	1110 ''''	E5	1	0		[B,S]	1111 ''''	F5	4	0
	Accu D	X	07A	D,X	1000 1011	8B	4	0	07B	[D,X]	1001 1011	9B	7	0
		Y		D,Y	1010 ''''	AB	4	0		[D,Y]	1011 ''''	BB	7	0
		U		D,U	1100 ''''	CB	4	0		[D,U]	1101 ''''	DB	7	0
		S		D,S	1110 ''''	EB	4	0		[D,S]	1111 ''''	FB	7	0
Auto-incrémentation Auto-décrémentation	Auto Incrémenté + 1	X	08A	,X+	1000 0000	80	2	0	le + 1 n'est pas possible car en indirect on recherche toujours sur une adresse intermédiaire qui nécessiste 2 octets					
		Y		,Y+	1010 ''''	A0	2	0						
		U		,U+	1100 ''''	C0	2	0						
		S		,S+	1110 ''''	E0	2	0						
	Auto Incrémenté + 2	X	09A	,X++	1000 0001	81	3	0	09B	[,X++]	1001 0001	91	6	0
		Y		,Y++	1010 ''''	A1	3	0		[,Y++]	1011 ''''	B1	6	0
		U		,U++	1100 ''''	C1	3	0		[,U++]	1101 ''''	D1	6	0
		S		,S++	1110 ''''	E1	3	0		[,S++]	1111 ''''	F1	6	0
	Auto décrémenté - 1	X	10A	,-X	1000 0010	82	2	0	le - 1 n'est pas possible car en indirect on recherche toujours sur une adresse intermédiaire qui nécessiste 2 octets					
		Y		,-Y	1010 ''''	A2	2	0						
		U		,-U	1100 ''''	C2	2	0						
		S		,-S	1110 ''''	E2	2	0						
	Auto décrémenté - 2	X	11A	,--X	1000 0011	83	3	0	11B	[,--X]	1001 0011	93	6	0
		Y		,--Y	1010 ''''	A3	3	0		[,--Y]	1011 ''''	B3	6	0
		U		,--U	1100 ''''	C3	3	0		[,--U]	1101 ''''	D3	6	0
		S		,--S	1110 ''''	E3	3	0		[,--S]	1111 ''''	F3	6	0
Relatif au PC	Depuis PCR 8 bits	X	12A	n,PCR	1000 1100	8C	1	1	12B	[n,PCR]	1001 1100	9C	4	1
		Y			1010 ''''	AC	1	1			1011 ''''	BC	4	1
		U			1100 ''''	CC	1	1			1101 ''''	DC	4	1
		S			1110 ''''	EC	1	1			1111 ''''	FC	4	1
	Depuis PCR 16 bits	X	13A	n,PCR	1000 1101	8D	5	2	13B	[n,PCR]	1001 1101	9D	8	2
		Y			1010 ''''	AD	5	2			1011 ''''	BD	8	2
		U			1100 ''''	CD	5	2			1101 ''''	DD	8	2
		S			1110 ''''	ED	5	2			1111 ''''	FD	8	2
Etendu Indirect									14B	[n]	1001 1111	9F	5	2

Adressage INDEXE et INDEXE Indirect à déplacement NUL

groupe 01A et 01B

L'adresse de la donnée correspond au contenu du registre spécifié dans l'opérande.
Le registre d'index contient donc l'adresse effective de l'octet à manipuler.

C'est le mode d'indexation le plus rapide.

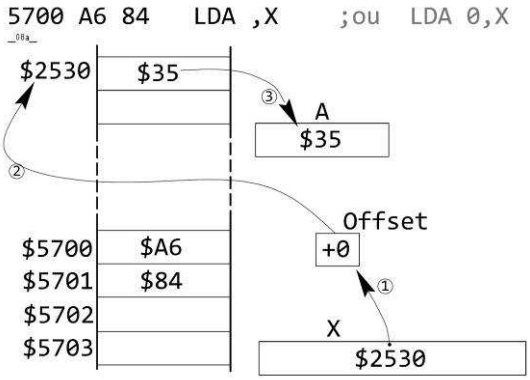
Exemple :

`LDA 0,X` est identique à `LDA ,X` On charge A avec le contenu de l'adresse pointée par la valeur de X.

```
                    ;----prog MA-10
                0010  Val     EQU     $10         ;
5700                          ORG     $5700       ;
5700 A6   84                  LDA     ,X          ; groupe 01A
                                                  ;
5800                          ORG     $5800       ;
5800 A6   94                  LDA     [,X]        ; groupe 01B
                              END                 ;
```

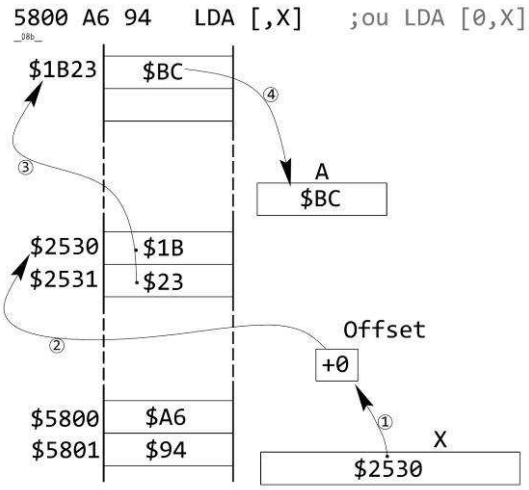
INDEXE direct

01A



INDEXE indirect

01B



Adressage INDEXE à déplacement 5 bits groupe 02A

Le déplacement (appelé Offset) est codé sur 5 bits en complément à deux (c'est-à-dire en arithmétique signé).

Le déplacement est compris entre **[-16 et +15]**

Les 5 bits se décomposent **xxxS VVVV** :

VVVV de 4 bits de déplacement (cest la valeur de **n** dans l'opérande)
De -16 à + 15

S d'un bit pour le signe. C'est le 5ième bit
La valeur du déplacement est codée en complément à 2
-- bit de signe = 0 déplacement positif
-- bit de signe = 1 déplacement négatif

Si **s=0** n est positif
Valeur du PostByte = n + Décalage

Si **s=1** n est négatif
Valeur du PostByte = n + **32** + Décalage

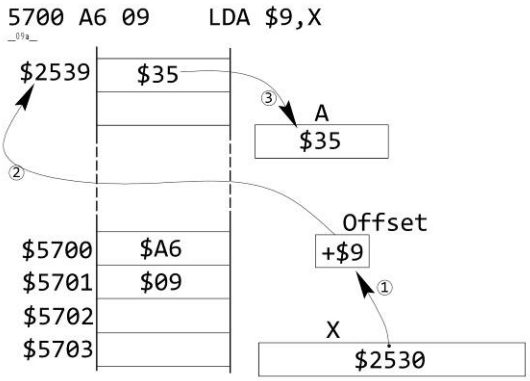
	déplacement Négatif	déplacement Positif	Décalage
Pour le registre **X**	de $10 à $1F	de $00 à $0F	0
Pour le registre **Y**	de $30 à $3F	de $20 à $2F	32
Pour le registre **U**	de $50 à $5F	de $40 à $4F	64
Pour le registre **S**	de $70 à $7F	de $60 à $6F	128

L'adresse de la donnée correspond au contenu du registre spécifié dans l'opérande additionné au déplacement compris dans l'opérande.

```
                    ;----prog MA-11
                0010  Val    EQU    $10        ;
5700                         ORG    $5700      ;
5700 A6   09                 LDA    $9,X       ; groupe 02A
                             END                ;
```

INDEXE direct

02A



INDEXE indirect

Le mode indexé Indirect n'est pas utilisé dans le déplacement à 5 bits

Adressage INDEXE et INDEXE indirect à déplacement 8 bits

groupe 03A et 03B

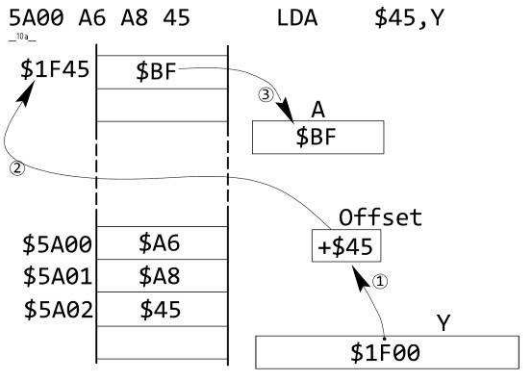
L'adresse de la donnée est obtenue en faisant la somme du contenu du registre d'index et du déplacement codé sur 8 bits.

Le déplacement est compris entre **[-128 et +127]**

```
                        ;----prog MA-12
                  0010  Val     EQU     $10         ;
5900                            ORG     $5900       ;
5900 A6   88 66                 LDA     102,X       ;
                                                    ;
5A00                            ORG     $5A00       ;
5A00 A6   A8 45                 LDA     $45,Y       ; groupe 03A
                                                    ;
5B00                            ORG     $5B00       ;
5B00 A6   B8 45                 LDA     [$45,Y]     ; groupe 03B
                                END                 ;
```

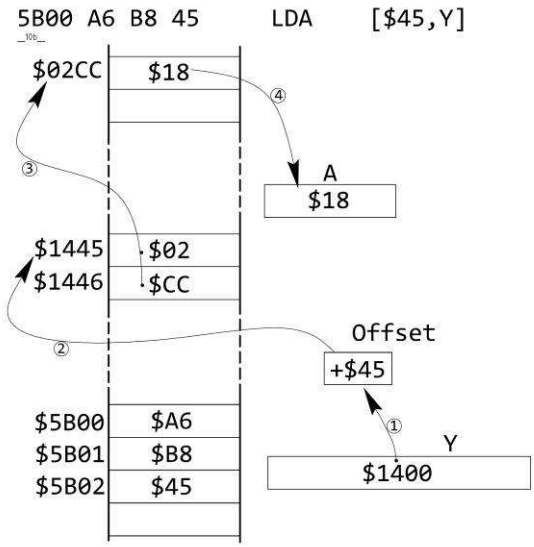
INDEXE direct

(03A)



INDEXE indirect

(03B)



Adressage INDEXE et INDEXE indirect à déplacement 16 bits

groupe 04A et 04B

Similaire au précédent sauf qu'ici on est sur 16 bits.

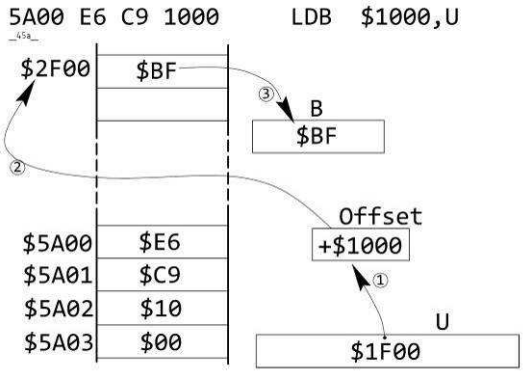
Le déplacement est compris entre **[-32768 et +32767]**

```
                        ;----prog MA-13
                  0010  Val     EQU     $10         ;
5A00                            ORG     $5A00       ;
5A00 E6   C9 1000               LDB     $1000,U     ; groupe 04A
                                                    ;
5B00                            ORG     $5B00       ;
5B00 E6   D9 1000               LDB     [$1000,U]   ; groupe 04B
                                END                 ;
```

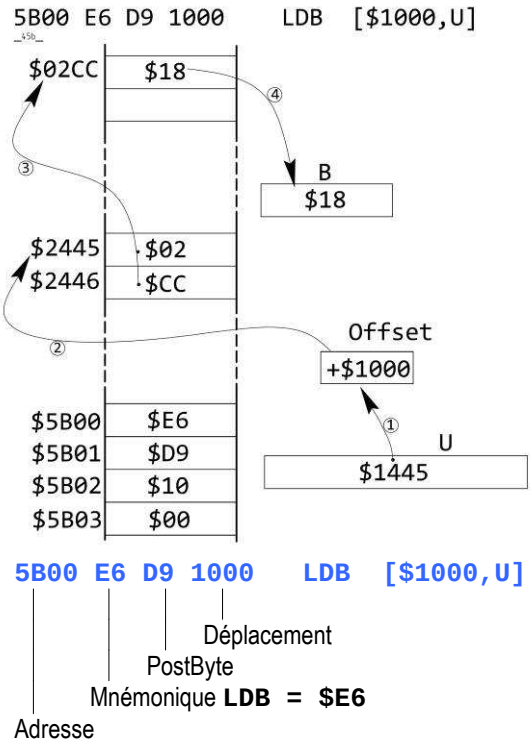
INDEXE direct LDB $2500,U

INDEXE indirect LDB [$2500,U]

04A



04B



5B00 E6 D9 1000 LDB [$1000,U]

Déplacement

PostByte

Mnémonique **LDB = $E6**

Adresse

Adressage INDEXE et INDEXE indirect offset par accumulateur

Le contenu des accumulateurs A, B ou D sert de déplacement.

L'adresse de la donnée considérée est obtenue en ajoutant le contenu de l'un des accumulateurs avec le contenu du registre d'index spécifié.

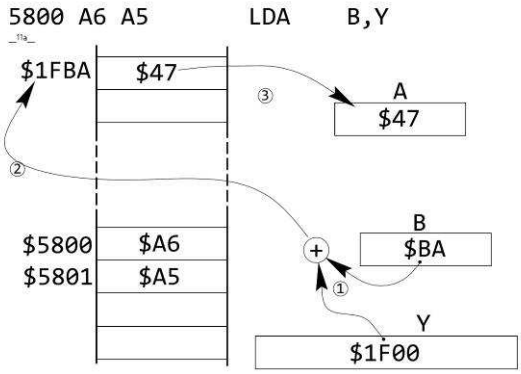
Si l'accumulateur est A ou B, alors la variation n'est que de 256 octets.

Si l'accumulateur est D, alors ou pourra accéder à la totalité de l'espace adressable du 6809. (déplacement sur 16 bits Signés)

```
                    ;----prog MA-14
                0010  Val    EQU    $10        ;
5800                         ORG    $5800      ;
5800 A6   A5                 LDA    B,Y        ; groupe 06A
                                                ;
5900                         ORG    $5900      ;
5900 A6   B5                 LDA    [B,Y]      ; groupe 06B
                             END                ;
```

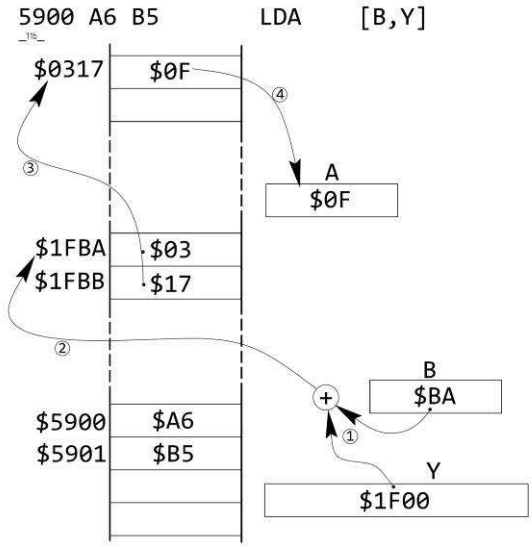
INDEXE direct

05A 06A 07A



INDEXE indirect

05B 06B 07B



Adressage INDEXE et INDEXE indirect auto-incrémenté

groupe 08A, 09A et 09B

Extrêmement utiles pour l'écriture des boucles.
Il est **ENSUITE incrémenté automatique de 1 ou 2 unités**, ce qui lui permet de pointer à l'adresse de la donnée suivante. Le contenu registre d'index est donc incrémenté après avoir pointé l'adresse effective.

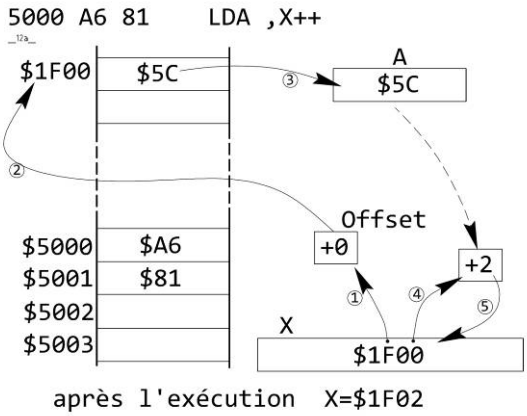
```
    STX 0,U+     ; auto-incrémenté de 1 unité
    STX 0,U++    ; auto-incrémenté de 2 unités
```

En adressage INDEXE INDIRECT on ne peut pas avoir la décrémentation d'une seule unité, puisqu'il faut 2 octets mémoire pour définir une adresse, donc `[,R+]` **est interdit en Indexé Indirect auto-incrémenté.**

```
                    ;----prog MA-15
                0010  Val    EQU    $10        ;
5000                         ORG    $5000      ;
5000 A6   81                 LDA    ,X++       ; groupe 09A
                                                ;
5100                         ORG    $5100      ;
5100 A6   91                 LDA    [,X++]     ; groupe 09B
                             END                ;
```

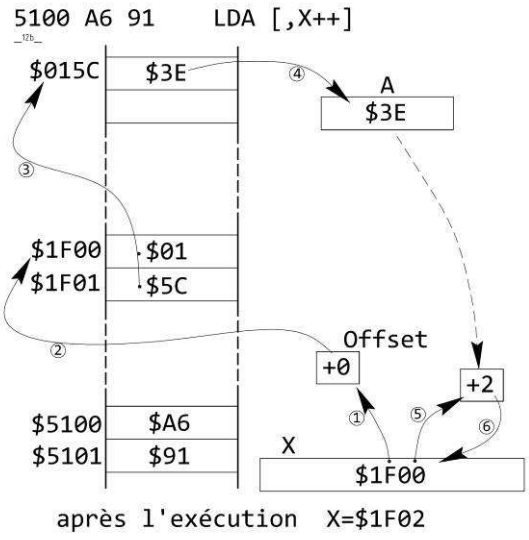
INDEXE direct

08A 09A



INDEXE indirect

~~08B~~ 09B



Adressage INDEXE et INDEXE indirect auto-décrémenté

groupe 10A, 11A et 11B

Extrêmement utiles pour l'écriture des boucles.
Le contenu du registre **est TOUT D'ABORD décrémenté de 1 ou 2 unités**, avant de pointer l'adresse effective, le registre contient ensuite l'adresse de la donnée désirée.

```
    LDD ,-Y      ; auto-décrémenté de 1 unité
    LDA ,--Y     ; auto-décrémenté de 2 unités
```

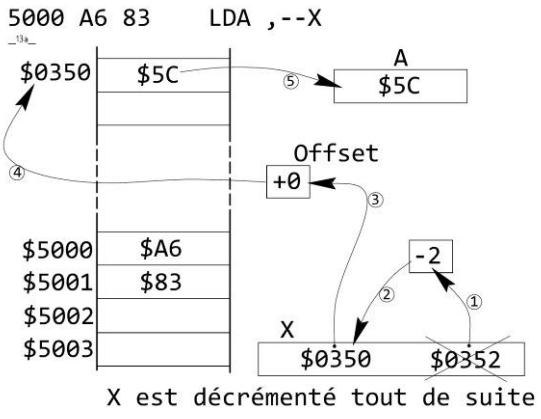
En adressage INDEXE INDIRECT on ne peut pas avoir l'incrémentation d'une seule unité, puisqu'il faut 2 octets mémoire pour définir une adresse donc `[,-R]` **est interdit en Indexé Indirect auto-décrémenté.**

Pour ces deux adressages. Le post-octet contient le code de 2 bits caractéristique du registre d'index utilisé.

```
                    ;----prog MA-16
                0010  Val    EQU    $10        ;
5000                         ORG    $5000      ;
5000 A6   83                 LDA    ,--X       ; groupe 11A
                                                ;
5100                         ORG    $5100      ;
5100 A6   93                 LDA    [,--X]     ; groupe 11B
                             END                ;
```

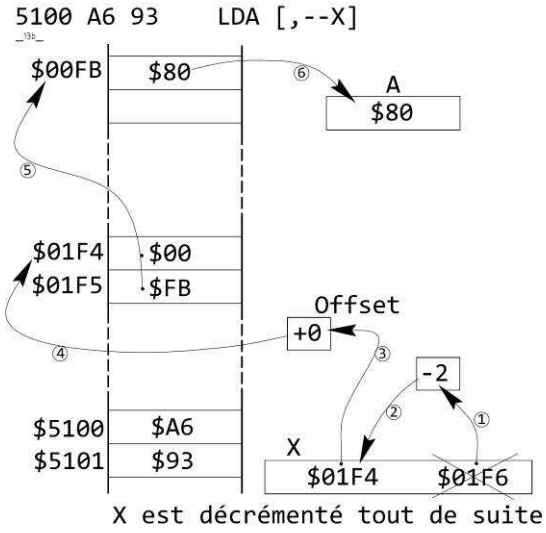
INDEXE direct

10A 11A



INDEX E indirect

~~10B~~ 11B



Adressage INDEXE Direct relatif au compteur ordinal PCR groupe 12A, 13A

Fonctionnement identique aux modes à déplacement constant su 8 ou 16 bits, sauf que le registre est le compteur ordinal PC, il est utilisé comme registre pointeur pour l'adressage.

Le programmeur peut obliger l'Assembleur à mettre un opérande sur 8 bits en mettant le signe **<** devant l'opérande.

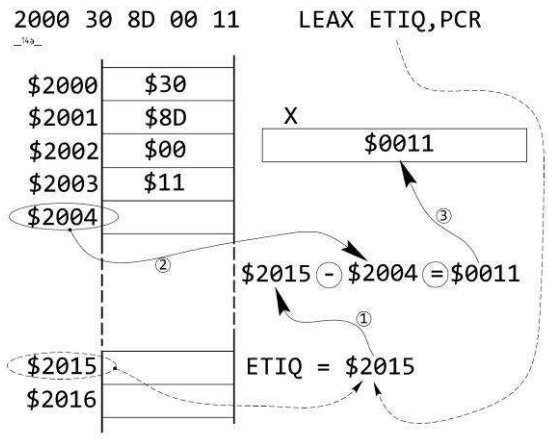
Un déplacement signé (Complément à 2) est ajouté au contenu du registre PC pour créer l'Adresse Effective. Cette adresse effective sert à la recherche de la donnée.

A l'inverse de l'adressage relatif (Branchements) le registre PC n'est pas changé par l'addition.

```
                        ;----prog MA-17 -- 1ière partie
                 0010   Val     EQU     $10          ;
2000                            ORG     $2000        ;
2000 30   8D 0011               LEAX    ETIQ,PCR     ;
                                                     ;
2015                            ORG     $2015        ;
                 2015   ETIQ    EQU     *            ;
                                                     ;
```

INDEXE Direct relatif au PCR

12A 13A



Avantage : écriture de programme fonctionnant à n'importe quelle adresse mémoire et ceci sans modifications.

Le code objet est indépendant de la position mémoire, le programme peut être logé à n'importe quelle adresse voir la rubrique **translatable (Programme)**

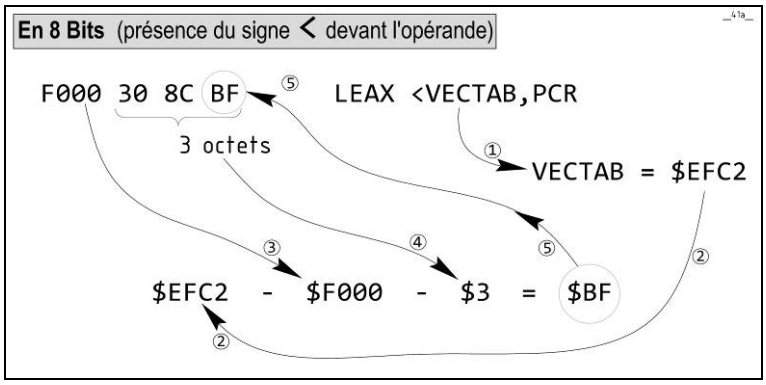
Programme relogeable		Programme NON relogeable
LEAX VECTAB,PCR	est équivalent à	**LDX VECTAB**

L'adresse de l'opérande est obtenue par la somme du déplacement (8 ou 16 bits) et le registre PC.

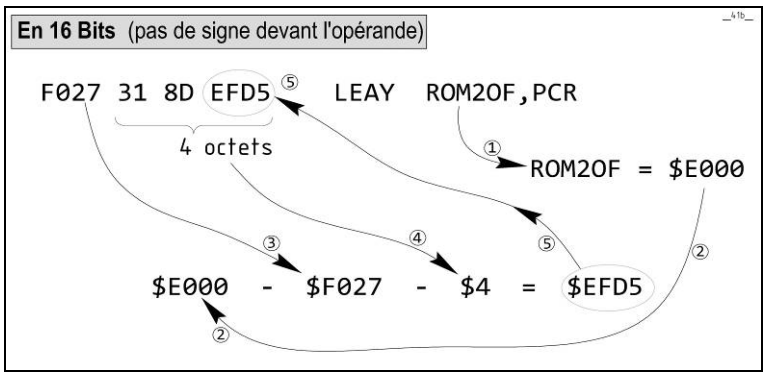
```
        LDA   $26,PCR     ; déplacement sur 8 bits
        LDA   $23F4,PCR   ; déplacement sur 16 bits
ETIQ    EQU   $01FF       ; attribut $01FF à la constante ETIQ
        LDA   ETIQ,PCR    ; déplacement sur 16 bits en fonction d'une Etiquette
        LEAX  ETIQ,PCR    ; charge X avec l'Adresse Effective qui est donnée
                          ; par la position de l'étiquette par rapport au PC
```

Exemple 01 : `LDA $12,PCR` charge A avec le contenu de la mémoire située 18 adresses plus loin (18 = $12).

Exemple 02 : En 8 bits car il y a la présence du signe **<** devant l'opérande



Exemple 03 : En 16 bits car il n'y a pas de présence du signe **<** devant l'opérande



Un programme implanté en [$7000, $7FFF] ne fonctionnera pas :

-- Si le code objet est transposé en [$1000, $1FFF].

-- Si ce programme contient des instructions en mode étendu (symbole >) ou en mode direct (symbole <) faisant référence aux adresses situées à l'intérieur de la zone d'implantation.

Sauf si on utilise le mode indexé (direct ou indirect) relatif au compteur-programme PC, on parle alors de programme entièrement translatables.

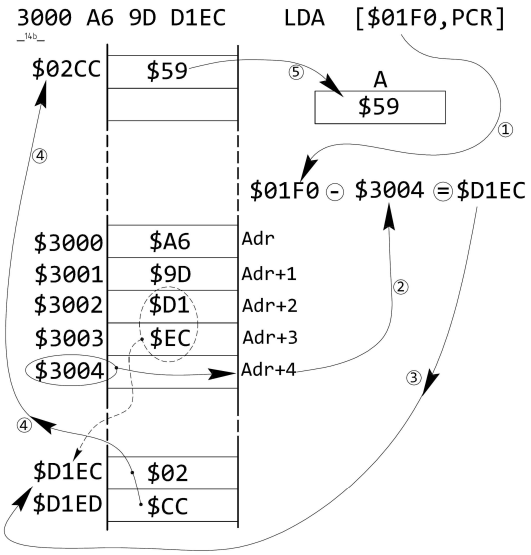
Adressage INDEXE Indirect relatif au compteur ordinal PCR groupe 12B, 13B

Le contenu du compteur ordinal ajouté à un déplacement constant donne l'adresse de la donnée nécessaire à l'instruction.

```
                        ;----prog MA-17 -- 2ième partie
3000                            ORG    $3000        ;
3000 A6   9D D1EC               LDA    [$01F0,PCR]  ;
                                END
```

INDEXE Indirect relatif au PCR

12B 13B



Utilisation des modes d'adressage

Cette partie contient des exemples de programmes courts illustrant l'utilisation de plusieurs modes d'adressage.

Utilisation de l'indexation pour des accès séquentiels à un bloc de données

Accès à un tableau de 100 éléments afin de rechercher le caractère @
L'adresse de départ de ce tableau s'appelle BASE.

```
1000                            ORG    $1000  ;  Prog MA-01
                  2500  BASE    EQU    $2500  ;
                  2000  COMPT   EQU    $2000  ;
1000 8E   2500          CHERCH  LDX    #BASE  ;
1003 86   40                    LDA    #'@    ;
1005 C6   00                    LDB    #COMPT ;
1007 A1   80            TEST    CMPA   ,X+    ; comparaison B avec A et +1 sur X
1009 27   03      100E          BEQ    TROUVE ; le car recherché est trouvé
100B 5A                         DECB           ; décrémenté B
100C 26   F9      1007          BNE    TEST   ; est ce le dernier élément ?
100E                    TROUVE                 ;                        TROUVE
```

Transfert d'un bloc de données comportant moins de 256 éléments

COMPT est le nombre d'éléments du bloc à déplacer. On suppose ce nombre < 256
La valeur DE est l'adresse de début du bloc, la valeur VERS est l'adresse de début de la zone mémoire où le bloc de donnée doit être transféré.

On déplace un octet à la fois, on garde la trace de l'octet déplacé en stockant sa position dans B.

```
1000                            ORG    $1000  ;  Prog MA-02
                  0500  DE      EQU    $0500  ;
                  0100  VERS    EQU    $0100  ;
                  2000  COMPT   EQU    $2000  ;
1000 8E   0500          BLKMOV  LDX    #DE    ; init de l'adresse pointeur Source
1003 108E 0100                  LDY    #VERS  ; init de l'adresse pointeur Destination
1007 C6   00                    LDB    #COMPT ; B nombre d'élément à transférer
1009 A6   80            SUITE   LDA    ,X+    ; {adrs X}=>A, puis +1 sur X
100B A7   A0                    STA    ,Y+    ; A=>{adrs Y}, puis +1 sur Y
100D 5A                         DECB           ; décrémentation du compteur B
100E 26   F9      1009          BNE    SUITE  ; tant que B n'est pas à 0 --> SUITE
                                               ; B=0 tous les éléments sont transférés
```

Même programme mais en utilisant le mode d'indexation à déplacement accumulateur. Dans ce cas B set en même temps de valeur de déplacement et de compteur.

Le programme ci-dessous se déroule plus vite car le mode à déplacement par accumulateur nécessite un cycle machine de moins que le mode par incrémentation.

```
1000                            ORG    $1000  ;  Prog MA-03
                  0500  DE      EQU    $0500  ;
                  0100  VERS    EQU    $0100  ;
                  2000  COMPT   EQU    $2000  ;
1000 8E   0500          BLKMOV  LDX    #DE    ;
1003 108E 0100                  LDY    #VERS  ;
1007 C6   00                    LDB    #COMPT ;
1009 A6   85            SUITE   LDA    B,X    ; ces deux lignes on été
100B A7   A5                    STA    B,Y    ; modifiées
100D 5A                         DECB           ;
100E 26   F9      1009          BNE    SUITE  ;
```

Transfert de bloc de données de plus de 256 éléments

Ce programme transfère 2 octets à la fois, toujours un nombre pairs d'octets.
C'est pourquoi LONG est divisé par 2.

Si LONG, avant la division était impair, le dernier octet le serait.

Ceci est pris en compte par un test de la retenu après la division.

S'il y a une retenue, on fait +1 sur D.

```
1000                            ORG    $1000  ;  Prog MA-04
                  0500  DE      EQU    $0500  ;
                  0100  VERS    EQU    $0100  ;
                  0050  LONG    EQU    $0050  ;
1000 CC   0050                  LDD    #LONG  ;
1003 44                         LSRA          ; diviser LONG de 16 bits par 2
1004 56                         RORB          ;
1005 24   03      100A          BCC    PAIR   ;
1007 C3   0001                  ADDD   #1     ; incrémenter D si LONG est impair
100A 108E 0500          PAIR    LDY    #DE    ;
100E CE   0100                  LDU    #VERS  ;
1011 AE   A1            SUITE   LDX    ,Y++   ; ++ pour 2 octets à transférer à la fois
1013 AF   C1                    STX    ,U++   ; ++
1015 83   0001                  SUBD   #1     ; pas d'instruction de décrémentation pour D
                                              ; alors on soustrait 1 à D
1018 26   F7      1011          BNE    SUITE  ;
```

X est utilisé comme registre de transfert Y et U servent de registre d'index

Le code ci-dessus a été optimisé par **(Jacques BRIGAUD** du forum.system-cfg.com) mars 2016.

Il vaut mieux ajouter le 1 avant la division. ***Si c'est pair, le +1 sera effacé.***
Si c'est impair, il sera pris en compte

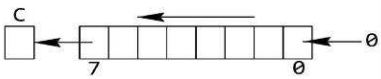
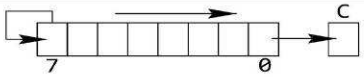
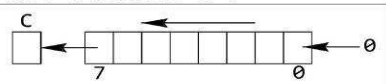
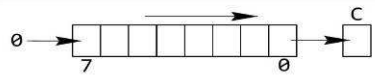
```
1000                            ORG    $1000  ;  Prog MA-05
                  0500  DE      EQU    $0500  ;
                  0100  VERS    EQU    $0100  ;
                  0050  LONG    EQU    $0050  ;
1000 CC   0050                  LDD    #LONG  ;
1003 C3   0001                  ADDD   #1     ;
1006 44                         LSRA          ; diviser LONG de 16 bits par 2
1007 56                         RORB          ;
1008 108E 0500                  LDY    #DE    ;
100C CE   0100                  LDU    #VERS  ;
100F AE   A1            SUITE   LDX    ,Y++   ; ++ pour 2 octets à transférer à la fois
1011 AF   C1                    STX    ,U++   ; ++
1013 83   0001                  SUBD   #1     ; pas d'instruction décrémentation pour D
                                              ; alors on soustrait 1 à D
1016 26   F7      100F          BNE    SUITE  ;
```

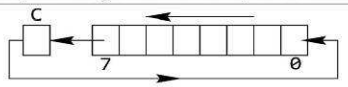
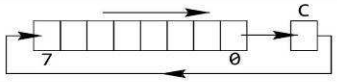
Addition de 2 blocs de données

Addition élément par élément, 2 blocs qui débutent respectivement aux adresses BLK1 et BLK2.
Ces 2 blocs ont le même nombre d'élément COMPT.

```
0000                            ORG    $0000  ;  Prog MA-06
                  1000  BLK1    EQU    $1000  ;
                  2000  BLK2    EQU    $2000  ;
                  0050  COMPT   EQU    $0050  ;
0000 8E   1000          BLKADD  LDX    #BLK1  ;
0003 108E 2000                  LDY    #BLK2  ;
0007 C6   50                    LDB    #COMPT ; Nb d'élément à additionner mis dans B
0009 4F                         CLRA          ; RAZ du bit de retenue en prévision de la
                                              ; première addition
000A A6   84            BOUCLE  LDA    ,X     ; premier élément mis dans A
000C A9   A0                    ADCA   ,Y+    ; ajout du 2ième élément, puis +1 sur Y
000E A7   80                    STA    ,X+    ; résult sauv dans case mém BLK1, +1 sur X
0010 5A                         DECB          ; B décrémenté
0011 26   F7      000A          BNE    BOUCLE ; aussi longtemps que B n'est pas = à 0
```

Tableau Regroupant Toutes les Instructions

	# Immediat			< Direct			Indexé ①			> Etendu			Inhérent			← Modes d'adressage	Bit de CC				
																	5	3	2	1	0
	Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#	{.....} = le contenu de	H	N	Z	V	C
ABX													3A	3	1	B + X → X					
ADCA	89	2	2	99	4	2	A9	4+	2+	B9	5	3				A + Mem + C → A	H	N	Z	V	C
ADCB	C9	2	2	D9	4	2	E9	4+	2+	F9	5	3				B + Mem + C → B	H	N	Z	V	C
ADDA	8B	2	2	9B	4	2	AB	4+	2+	BB	5	3				A + Mem → A	H	N	Z	V	C
ADDB	CB	2	2	DB	4	2	EB	4+	2+	FB	5	3				B + Mem → B	H	N	Z	V	C
ADDD	C3	4	3	D3	6	2	E3	6+	2+	F3	7	3				D + Mem:Mem+1 → D		N	Z	V	C
ANDA	84	2	2	94	4	2	A4	4+	2+	B4	5	3				A ∧ Mem → A		N	Z	0	
ANDB	C4	2	2	D4	4	2	E4	4+	2+	F4	5	3				B ∧ Mem → B		N	Z	0	
ANDCC	1C	3	2													CC ∧ Mem → CC	?	?	?	?	?
ASLA	Idem que												48	2	1		⑧	N	Z	V	C
ASLB	LSL...												58	2	1		⑧	N	Z	V	C
ASL	voir + bas ⇓			08	6	2	68	6+	2+	78	7	3					⑧	N	Z	V	C
ASRA													47	2	1		⑧	N	Z		C
ASRB													57	2	1		⑧	N	Z		C
ASR				07	6	2	67	6+	2+	77	7	3					⑧	N	Z		C
BITA	85	2	2	95	4	2	A5	4+	2+	B5	5	3				{Mem ∧ A} seul CC est modifié		N	Z	0	
BITB	C5	2	2	D5	4	2	E5	4+	2+	F5	5	3				{Mem ∧ B} seul CC est modifié		N	Z	0	
CLRA													4F	2	1	0 → A		0	1	0	0
CLRB													5F	2	1	0 → B		0	1	0	0
CLR				0F	6	2	6F	6+	2+	7F	7	3				0 → Mem		0	1	0	0
CMPA	81	2	2	91	4	2	A1	4+	2+	B1	5	3				{Mem - A} CC modifié	⑧	N	Z	V	C
CMPB	C1	2	2	D1	4	2	E1	4+	2+	F1	5	3				{Mem - B} CC modifié	⑧	N	Z	V	C
CMPD	1083	5	4	1093	7	3	10A3	7+	3+	10B3	8	4				{Mem:Mem+1 - D} CC modifié		N	Z	V	C
CMPS	118C	5	4	119C	7	3	11AC	7+	3+	11BC	8	4				{Mem:Mem+1 - S} CC modifié		N	Z	V	C
CMPU	1183	5	4	1193	7	3	11A3	7+	3+	11B3	8	4				{Mem:Mem+1 - U} CC modifié		N	Z	V	C
CMPX	8C	4	3	9C	6	2	AC	6+	2+	BC	7	3				{Mem:Mem+1 - X} CC modifié		N	Z	V	C
CMPY	108C	5	4	109C	7	3	10AC	7+	3+	10BC	8	4				{Mem:Mem+1 - Y} CC modifié		N	Z	V	C
COMA													43	2	1	complément{A} → A		N	Z	0	1
COMB													53	2	1	complément{B} → B		N	Z	0	1
COM				03	6	2	63	6+	2+	73	7	3				complément{Mem} → Mem		N	Z	0	1
CWAI	3C	≥20	2													CC ∧ Mem → CC attend interrup	⑦	⑦	⑦	⑦	⑦
DAA													19	2	1	Ajustement Décimal + A → A		N	Z	0	C
DECA													4A	2	1	A - 1 → A		N	Z	V	
DECB													5A	2	1	B - 1 → B		N	Z	V	
DEC				0A	6	2	6A	6+	2+	7A	7	3				Mem - 1 → Mem		N	Z	V	
EORA	88	2	2	98	4	2	A8	4+	2+	B8	5	3				A ⊻ Mem → A		N	Z	0	
EORB	C8	2	2	D8	4	2	E8	4+	2+	F8	5	3				B ⊻ Mem → B		N	Z	0	
EXG ②	1E	8	2													R1 ⇄ R2					
INCA													4C	2	1	A + 1 → A		N	Z	V	
INCB													5C	2	1	B + 1 → B		N	Z	V	
INC				0C	6	2	6C	6+	2+	7C	7	3				Mem + 1 → Mem		N	Z	V	
JMP				0E	3	2	6E	3+	2+	7E	4	3				Adrs Effective③ → PC					
JSR				9D	7	2	AD	7+	2+	BD	8	3				Saut vers Sous-programme					
LDA	86	2	2	96	4	2	A6	4+	2+	B6	5	3				Mem → A		N	Z	0	
LDB	C6	2	2	D6	4	2	E6	4+	2+	F6	5	3				Mem → B		N	Z	0	
LDD	CC	3	3	DC	5	2	EC	5+	2+	FC	6	3				Mem:Mem+1 → D		N	Z	0	
LDS	10CE	4	4	10DE	6	3	10EE	6+	3+	10FE	7	4				Mem:Mem+1 → S		N	Z	0	
LDU	CE	3	3	DE	5	2	EE	5+	2+	FE	6	3				Mem:Mem+1 → U		N	Z	0	
LDX	8E	3	3	9E	5	2	AE	5+	2+	BE	6	3				Mem:Mem+1 → X		N	Z	0	
LDY	108E	4	4	109E	6	3	10AE	6+	3+	10BE	7	4				Mem:Mem+1 → Y		N	Z	0	
LEAS							32	4+	2+							Adrs Effective③ → S					
LEAU							33	4+	2+							Adrs Effective③ → U					
LEAX							30	4+	2+							Adrs Effective③ → X			Z		
LEAY							31	4+	2+							Adrs Effective③ → Y			Z		
LSLA	Idem que												48	2	1			N	Z	V	C
LSLB	ASL...												58	2	1			N	Z	V	C
LSL	voir + haut ⇑			08	6	2	68	6+	2+	78	7	3						N	Z	V	C
LSRA													44	2	1			0	Z		C
LSRB													54	2	1			0	Z		C
LSR				04	6	2	64	6+	2+	74	7	3						0	Z		C

	# Immediat			< Direct			Indexé ①			> Etendu			Inhérent			← Modes d'adressage	Bit de CC				
																	5	3	2	1	0
	Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#	{.....} = le contenu de	H	N	Z	V	C
MUL													3D	11	1	A x B → D			Z		⑨
NEGA													40	2	1	{Complément à 2 de A} → A	⑧	N	Z	V	C
NEGB													50	2	1	{Complément à 2 de B} → B	⑧	N	Z	V	C
NEG				00	6	2	60	6+	2+	70	7	3				{Complément à 2 de Mem} → Mem	⑧	N	Z	V	C
NOP													12	2	1	Pas d'opération					
ORA	8A	2	2	9A	4	2	AA	4+	2+	BA	5	3				A ∨ Mem → A		N	Z	0	
ORB	CA	2	2	DA	4	2	EA	4+	2+	FA	5	3				B ∨ Mem → B		N	Z	0	
ORCC	1A	3	2													CC ∨ Mem → CC	⑦	⑦	⑦	⑦	⑦
PSHS	34	5+④	2													Empile registres dans pile S					
PSHU	36	5+④	2													Empile registres dans pile U					
PULS	35	5+④	2													Dépile registres dans pile S					
PULU	37	5+④	2													Dépile registres dans pile U					
ROLA													49	2	1			N	Z	V	C
ROLB													59	2	1			N	Z	V	C
ROL				09	6	2	69	6+	2+	79	7	3						N	Z	V	C
RORA													46	2	1			N	Z		C
RORB													56	2	1			N	Z		C
ROR				06	6	2	66	6+	2+	76	7	3						N	Z		C
RTI													3B	6/15	1	Retour d'interruption	⑦	⑦	⑦	⑦	⑦
RTS													39	5	1	Retour de sous-programme					
SBCA	82	2	2	92	4	2	A2	4+	2+	B2	5	3				A - Mem - bit C → A	⑧	N	Z	V	C
SBCB	C2	2	2	D2	4	2	E2	4+	2+	F2	5	3				B - Mem - bit C → B	⑧	N	Z	V	C
SEX													1D	2	1	Extention de signe		N	Z	0	
STA				97	4	2	A7	4+	2+	B7	5	3				A → Mem		N	Z	0	
STB				D7	4	2	E7	4+	2+	F7	5	3				B → Mem		N	Z	0	
STD				DD	5	2	ED	5+	2+	FD	6	3				D → Mem:Mem+1		N	Z	0	
STS				10DF	6	3	10EF	6+	3+	10FF	7	4				S → Mem:Mem+1		N	Z	0	
STU				DF	5	2	EF	5+	2+	FF	6	3				U → Mem:Mem+1		N	Z	0	
STX				9F	5	2	AF	5+	2+	BF	6	3				X → Mem:Mem+1		N	Z	0	
STY				109F	6	3	10AF	6+	3+	10BF	7	4				Y → Mem:Mem+1		N	Z	0	
SUBA	80	2	2	90	4	2	A0	4+	2+	B0	5	3				A - Mem → A	⑧	N	Z	V	C
SUBB	C0	2	2	D0	4	2	E0	4+	2+	F0	5	3				B - Mem → B	⑧	N	Z	V	C
SUBD	83	4	3	93	6	2	A3	6+	2+	B3	7	3				D - Mem:Mem+1 → D		N	Z	V	C
SWI ⑥													3F	19	1	Interruption logicielle 1					
SWI2⑥													103F	20	2	Interruption logicielle 2					
SWI3⑥													113F	20	2	Interruption logicielle 3					
SYNC													13	≥4	1	Synchro événement extérieur					
TFR ②	1F	6	2													TFR R1,R2 R1 → R2					
TSTA													4D	2	1	Test du contenu de A		N	Z	0	
TSTB													5D	2	1	Test du contenu de B		N	Z	0	
TST				0D	6	2	6D	6+	2+	7D	7	3				Test du contenu de Mem		N	Z	0	

①	Cette colonne donne le nombre de cycle de base et le décompte d'octets. Afin d'obtenir le décompte total, il faut ajouter les valeurs obtenues dans la table Mode d'Adressage Indexé.
②	R1 et R2 paire de registres 8 bits ou paire de registres 16 bits
③	EA est l'Adresse Effective
④	Les instructions PSH et PUL nécessitent cinq cycles plus un cycle pour chaque octet empilé ou dépilé.
⑤	Instructions branchement : 5(6) signifie, 5 cycles si branche non fait, 6 cycles si branchement réalisé
⑥	SWI fixe les bits F et I, SWI2 et SWI3 n'affecte pas les bits F et I
⑦	L'état du registre CC résulte directement de cette instruction.
⑧	Valeur du Flag de demi-retenue non défini.
⑨	Cas particulier : bit C = 1 si le b7 de B est égal à 1

Op	**Op-Code** ou Code instruction en hexadécimal
~	**Nombre de cycle**. Lorsqu'il est suivit d'un + le nombre total de cycles s'obtient en ajoutant la valeur supplémentaire contenue dans le tableau de l'adressage indexés.
#	**Nombre d'octets** traduisant l'encombrement mémoire. Lorsqu'il est suivit d'un + le nombre total de cycles s'obtient en ajoutant la valeur supplémentaire contenue dans le tableau de l'adressage indexés.

⊻	OU Logique exclusif
∧	ET Logique
∨	OU Logique
:	Concaténation

Explications sur le nombre d'octets dans les différents tableaux

ADRESSAGE NON INDEXE

	Nombre d'Otects
Adressage Immédiat	2, 3 ou 4
Adressage Direct	2 ou 3
Adressage Etendu	3 ou 4
Adressage Inhérent	1 ou 2

	# Immediat			< Direct			Indexé ①			> Etendu			Inhérent		
	Op	~	#	Op	~	#	Op	~	#	Op	~	#	Op	~	#
ABX													3A	3	1
ADCA	89	2	2	99	4	2	A9	4+	2+	B9	4	2			
ADCB	C9	2	2	D9	4	2	E9	4+	2+	F9	5	3			
ADDA	8B	2	2	9B	4	2	AB	4+	2+	BB	5	3			
ADDB	CB	2	2	DB	4	2	EB	4+	2+	FB	5	3			
ADDD	C3	4	3	D3	6	2	E3	6+	2+	F3	7	3			

Le Post Byte est déjà compté dans le chiffre 2+ ou 3+

ADRESSAGE INDEXE direct et indirect

	Nombre d'Otects
Adressage Indexé	2+ ou 3+

exemples pour le cas du 2+

Le Post Byte est déjà compté dans le chiffre 2+ ou 3+

Opérande

```
EF 6A          STU   10,S
30 88 AE       LEAX  <CMEMN,X
30 8D 017C     LEAX  ERRMSG,PCR
```

exemples pour le cas du 3+

Post Byte

Mnémonique ou Op Code

```
11A3 A1        CMPU  ,Y++
10EE 88 EC     LDS   -20,X
10AE C9 42F3   LDY   >ERRMSG,U
```

Offset	Base	Groupe	INDEXÉ direct Syntaxe	Binaire	Post Byte	Nbre Cycles	Nbre d'Octets	Groupe	INDEXÉ indirect Syntaxe	Binaire	Post Byte
Nul	X	01A	0,X ou ,X	1000 0100	84	0	0	01B	[0,X] ou [,X]	1001 0100	94
	Y		0,Y ou ,Y	1010 ''''	A4	0	0		[0,Y] ou [,Y]	1011 ''''	B4
	U		0,U ou ,U	1100 ''''	C4	0	0		[0,U] ou [,U]	1101 ''''	D4
	S		0,S ou ,S	1110 ''''	E4	0	0		[0,S] ou [,S]	1111 ''''	F4
5 bits s = bit de signe dddd valeur de n de -16 à +15	X	02A	n,X	000s dddd (bit 64, bit 32)	10 à 1F / 00 à 0F	1	0	Décalage = 0	Si n est positif (s = 0) PostByte = n + Décalage		
	Y		n,Y	001s dddd	30 à 3F / 20 à 2F	1	0	Décalage = 32			
	U		n,U	010s dddd	50 à 5F / 40 à 4F	1	0	Décalage = 64	Si n est Négatif (s = 1) PostByte = n + 32 + Décalage		
	S		n,S	011s dddd	70 à 7F / 60 à 6F	1	0	Décalage = 96			
	X	03A	n,X	1000 1000	88						
	Y		n,Y	1010 ''''							

Tableau Regroupant Toutes les Instructions Triées par OpCode (par code opération)

OpCode		Mnémo.	Adress.	#
00		NEG	Direct	2
01		---	---	
02		---	---	
03		COM	Direct	2
04		LSR	Direct	2
05		---	---	
06		ROR	Direct	2
07		ASR	Direct	2
08		**ASL, LSL**	Direct	2
09		ROL	Direct	2
0A		DEC	Direct	2
0B		---	---	
0C		INC	Direct	2
0D		TST	Direct	2
0E		JMP	Direct	2
0F		CLR	Direct	2

OpCode		Mnémo.	Adress.	#
20		BRA	Relatif	2
21		BRN	Relatif	2
22		BHI	Relatif	2
23		BLS	Relatif	2
24		**BCC,BHS**	Relatif	2
25		**BCS,BLO**	Relatif	2
26		BNE	Relatif	2
27		BEQ	Relatif	2
28		BVC	Relatif	2
29		BVS	Relatif	2
2A		BPL	Relatif	2
2B		BMI	Relatif	2
2C		BGE	Relatif	2
2D		BLT	Relatif	2
2E		BGT	Relatif	2
2F		BLE	Relatif	2

OpCode		Mnémo.	Adress.	#
70		NEG	Etendu	3
71		---	---	
72		---	---	
73		COM	Etendu	3
74		LSR	Etendu	3
75		---	---	
76		ROR	Etendu	3
77		ASR	Etendu	3
78		**ASL,LSL**	Etendu	3
79		ROL	Etendu	3
7A		DEC	Etendu	3
7B		---	---	
7C		INC	Etendu	3
7D		TST	Etendu	3
7E		JMP	Etendu	3
7F		CLR	Etendu	3

OpCode		Mnémo.	Adress.	#
C0		SUBB	Immediat	2
C1		CMPB	Immediat	2
C2		SBCB	Immediat	2
C3		ADDD	Immediat	3
C4		ANDB	Immediat	2
C5		BITB	Immediat	2
C6		LDB	Immediat	2
C7		---	---	
C8		EORB	Immediat	2
C9		ADCB	Immediat	2
CA		ORB	Immediat	2
CB		ADDB	Immediat	2
CC		LDD	Immediat	3
CD		---	---	
CE		LDU	Immediat	3
CF		---	---	

10				
10	21	LBRN	Relatif	4
10	22	LBHI	Relatif	4
10	23	LBLS	Relatif	4
10	**24**	**LBCC,LBHS**	Relatif	4
10	**25**	**LBCS,LBLO**	Relatif	4
10	26	LBNE	Relatif	4
10	27	LBEQ	Relatif	4
10	28	LBVC	Relatif	4
10	29	LBVS	Relatif	4
10	2A	LBPL	Relatif	4
10	2B	LBMI	Relatif	4
10	2C	LBGE	Relatif	4
10	2D	LBLT	Relatif	4
10	2E	LBGT	Relatif	4
10	2F	LBLE	Relatif	4
10	3F	SWI2	Inhérent	2
10	83	CMPD	Immediat	4
10	8C	CMPY	Immediat	4
10	8E	LDY	Immediat	4
10	93	CMPD	Direct	3
10	9C	CMPY	Direct	3
10	9E	LDY	Direct	3
10	9F	STY	Direct	3
10	A3	CMPD	Indexé	3+
10	AC	CMPY	Indexé	3+
10	AE	LDY	Indexé	3+
10	AF	STY	Indexé	3+
10	B3	CMPD	Etendu	4
10	BC	CMPY	Etendu	4
10	BE	LDY	Etendu	4
10	BF	STY	Etendu	4
10	CE	LDS	Immediat	4
10	DE	LDS	Direct	3
10	DF	STS	Direct	3
10	EE	LDS	Indexé	3+
10	EF	STS	Indexé	3+
10	FE	LDS	Etendu	4
10	FF	STS	Etendu	4

11				
11	3F	SWI3	Inhérent	2
11	8C	CMPS	Immediat	4
11	83	CMPU	Immediat	4
11	93	CMPU	Direct	3
11	9C	CMPS	Direct	3
11	A3	CMPU	Indexé	3+
11	AC	CMPS	Indexé	3+
11	B3	CMPU	Etendu	4
11	BC	CMPS	Etendu	4

12		NOP	Inhérent	1
13		SYNC	Inhérent	1
14		---	---	
15		---	---	
16		LBRA	Relatif	3
17		LBSR	Relatif	3
18		---	---	
19		DAA	Inhérent	1
1A		ORCC	Immediat	2
1B		---	---	
1C		ANDCC	Immediat	2
1D		SEX	Inhérent	1
1E		EXG	Immediat	2
1F		TFR	Immediat	2

30		LEAX	Indexé	2+
31		LEAY	Indexé	2+
32		LEAS	Indexé	2+
33		LEAU	Indexé	2+
34		PSHS	Immediat	2
35		PULS	Immediat	2
36		PSHU	Immediat	2
37		PULU	Immediat	2
38		---	---	
39		RTS	Inhérent	1
3A		ABX	Inhérent	1
3B		RTI	Inhérent	1
3C		CWAI	Immediat	2
3D		MUL	Inhérent	1
3E		---	---	
3F		SWI	Inhérent	1

40		NEGA	Inhérent	1
41		---	---	
42		---	---	
43		COMA	Inhérent	1
44		LSRA	Inhérent	1
45		---	---	
46		RORA	Inhérent	1
47		ASRA	Inhérent	1
48		**ASLA,LSLA**	Inhérent	1
49		ROLA	Inhérent	1
4A		DECA	Inhérent	1
4B		---	---	
4C		INCA	Inhérent	1
4D		TSTA	Inhérent	1
4E		---	---	
4F		CLRA	Inhérent	1

50		NEGB	Inhérent	1
51		---	---	
52		---	---	
53		COMB	Inhérent	1
54		LSRB	Inhérent	1
55		---	---	
56		RORB	Inhérent	1
57		ASRB	Inhérent	1
58		**ASLB,LSLB**	Inhérent	1
59		ROLB	Inhérent	1
5A		DECB	Inhérent	1
5B		---	---	
5C		INCB	Inhérent	1
5D		TSTB	Inhérent	1
5E		---	---	
5F		CLRB	Inhérent	1

60		NEG	Indexé	2+
61		---	---	
62		---	---	
63		COM	Indexé	2+
64		LSR	Indexé	2+
65		---	---	
66		ROR	Indexé	2+
67		ASR	Indexé	2+
68		**ASL,LSL**	Indexé	2+
69		ROL	Indexé	2+
6A		DEC	Indexé	2+
6B		---	---	
6C		INC	Indexé	2+
6D		TST	Indexé	2+
6E		JMP	Indexé	2+
6F		CLR	Indexé	2+

80		SUBA	Immediat	2
81		CMPA	Immediat	2
82		SBCA	Immediat	2
83		SUBD	Immediat	3
84		ANDA	Immediat	2
85		BITA	Immediat	2
86		LDA	Immediat	2
87		---	---	
88		EORA	Immediat	2
89		ADCA	Immediat	2
8A		ORA	Immediat	2
8B		ADDA	Immediat	2
8C		CMPX	Immediat	3
8D		BSR	Relatif	2
8E		LDX	Immediat	3
8F		---	---	

90		SUBA	Direct	2
91		CMPA	Direct	2
92		SBCA	Direct	2
93		SUBD	Direct	2
94		ANDA	Direct	2
95		BITA	Direct	2
96		LDA	Direct	2
97		STA	Direct	2
98		EORA	Direct	2
99		ADCA	Direct	2
9A		ORA	Direct	2
9B		ADDA	Direct	2
9C		CMPX	Direct	2
9D		JSR	Direct	2
9E		LDX	Direct	2
9F		STX	Direct	2

A0		SUBA	Indexé	2+
A1		CMPA	Indexé	2+
A2		SBCA	Indexé	2+
A3		SUBD	Indexé	2+
A4		ANDA	Indexé	2+
A5		BITA	Indexé	2+
A6		LDA	Indexé	2+
A7		STA	Indexé	2+
A8		EORA	Indexé	2+
A9		ADCA	Indexé	2+
AA		ORA	Indexé	2+
AB		ADDA	Indexé	2+
AC		CMPX	Indexé	2+
AD		JSR	Indexé	2+
AE		LDX	Indexé	2+
AF		STX	Indexé	2+

B0		SUBA	Etendu	3
B1		CMPA	Etendu	3
B2		SBCA	Etendu	3
B3		SUBD	Etendu	3
B4		ANDA	Etendu	3
B5		BITA	Etendu	3
B6		LDA	Etendu	3
B7		STA	Etendu	3
B8		EORA	Etendu	3
B9		ADCA	Etendu	3
BA		ORA	Etendu	3
BB		ADDA	Etendu	3
BC		CMPX	Etendu	3
BD		JSR	Etendu	3
BE		LDX	Etendu	3
BF		STX	Etendu	3

D0		SUBB	Direct	2
D1		CMPB	Direct	2
D2		SBCB	Direct	2
D3		ADDD	Direct	2
D4		ANDB	Direct	2
D5		BITB	Direct	2
D6		LDB	Direct	2
D7		STB	Direct	2
D8		EORB	Direct	2
D9		ADCB	Direct	2
DA		ORB	Direct	2
DB		ADDB	Direct	2
DC		LDD	Direct	2
DD		STD	Direct	2
DE		LDU	Direct	2
DF		STU	Direct	2

E0		SUBB	Indexé	2+
E1		CMPB	Indexé	2+
E2		SBCB	Indexé	2+
E3		ADDD	Indexé	2+
E4		ANDB	Indexé	2+
E5		BITB	Indexé	2+
E6		LDB	Indexé	2+
E7		STB	Indexé	2+
E8		EORB	Indexé	2+
E9		ADCB	Indexé	2+
EA		ORB	Indexé	2+
EB		ADDB	Indexé	2+
EC		LDD	Indexé	2+
ED		STD	Indexé	2+
EE		LDU	Indexé	2+
EF		STU	Indexé	2+

F0		SUBB	Etendu	3
F1		CMPB	Etendu	3
F2		SBCB	Etendu	3
F3		ADDD	Etendu	3
F4		ANDB	Etendu	3
F5		BITB	Etendu	3
F6		LDB	Etendu	3
F7		STB	Etendu	3
F8		EORB	Etendu	3
F9		ADCB	Etendu	3
FA		ORB	Etendu	3
FB		ADDB	Etendu	3
FC		LDD	Etendu	3
FD		STD	Etendu	3
FE		LDU	Etendu	3
FF		STU	Etendu	3

(caractères en gras) un même op-code peut avoir deux instructions différentes ?

Dans les colonnes **#** si il y a un **+** Alors il y a des octets suplémentaires voir la table d'adressage indexé (PostByte)

Les Branchements

Adressage Indexé

ABX

Permet d'ajouter les 8 bits Non Signé de B dans le contenu du registre X
`{B} + {X} ---> {X}` {.....} = contenu de

Exemple : addition BCD sur 16 bits de 2 nombres stockés initialement aux adresses Adr1 et Adr2 le résultat est mis dans Adr3. (Adr1 et Adr2 constitué d'une partie Haute H=High et d'une partie Base L= Low).

```
LDA    Adr1L    ;--charge l'octet poids faible
ADDA   Adr2L    ;  additionne l'octet de poids faible
DAA             ;  ajustement décimal
STA    Adr3L    ;  sauvegarde l'octet de poids faible
LDA    Adr1H    ;--charge l'octet poids fort
ADDA   Adr2H    ;  additionne l'octet de poids fort
DAA             ;  ajustement décimal
STA    Adr3H    ;  sauvegarde l'octet de poids fort
SWI             ;
```

efhinzvc
Après l'instruction `CC = _____` _ = inchangé

ADCA ADCB (ADdition with Carry)

Permet d'ajouter à A ou B le contenu d'une case mémoire ou d'une valeur binaire dans le cas d'adressage immédiat, en plus on ajoute le bit C

`{A} + bit C + {Mém} ---> {A}` {.....} = contenu de
`{B} + bit C + {Mém} ---> {B}`

Exemple : `ADCA #$71` Avec initialement : bit C=1 et {A} = $4B

```
   $4B            0100 1011
+  $71         +  0111 0001
+    1 (bit C) +         1
=  $BD         =  1011 1101
```

Après on aura `{A}=$BD`

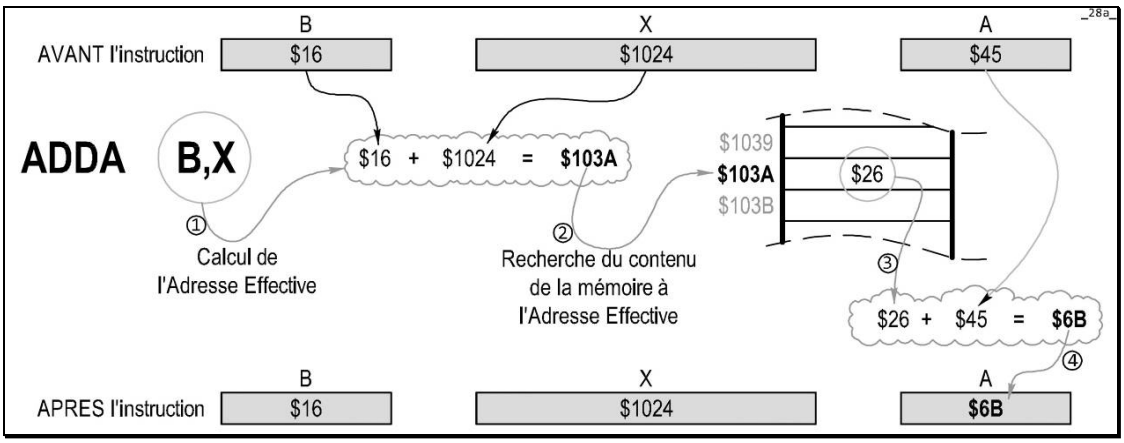
efhinzvc
Après l'instruction `CC = __0_1010` _ = inchangé

ADDA ADDB

Fonctionnement identique à **ADCA** et **ADCB** sauf que le bit C n'est pas pris en considération.

`{A} + {Mém} ---> {A}`
`{B} + {Mém} ---> {B}`

Exemple : en adressage indexé de l'instruction `ADDA B,X`



efhinzvc
Après l'instruction `CC = __H_NZVC` _ = inchangé

ADDD

Permet d'ajouter à D le contenu de la case mémoire ou une valeur sur 16 bits dans le cas d'adressage immédiat.

```
{D} + {Mém, Mém+1} ---> {D}
```

{.....} = contenu de

Exemple :

```
ADDD $12,X      ; adressage indexé à déplacement sur 5 bits
```

Avant l'instruction **{X}=$1000 {D}=$2000 {$1012}=$12 {$1013}=$F4**
Après l'instruction **{X}=$1000 {D}=$32F4 {cc}=xxxx 0000**

```
                 efhinzvc
Après l'instruction  CC = ____NZVC
```

_ = inchangé

ANDA ANDB (symbole & , parfois * ou Λ)

Dans les opérations Binaire le ET s'effectue bit par bit.

	AND	&
0	0	0
0	1	0
1	0	0
1	1	1

Ces instructions ANDA et ANDB effectuent un ET logique entre :
-- D'une part le contenu d'une case mémoire ou d'une valeur binaire dans le cas d'adressage immédiat
-- Et d'autre part l'un des accumulateurs A ou B.

ANDA

```
{A} & {Mém} ---> {A}
{A} & {%xxxxxxxx} ---> {A}
```

ANDB

```
{B} & {Mém} ---> {B}
{B} & {%xxxxxxxx} ---> {B}
```

{.....} = contenu de

```
                 efhinzvc
Après l'instruction  CC = ____NZ0_
```

_ = inchangé avec **V=0**

Utile pour mettre à 0 ou masquer un ou plusieurs bits dans un octet.

Exemple : on souhaite mettre à 0 les quatre bits de droite d'un octet

```
LDA     Valeur      ; Valeur = %1010 0110
ANDA    %11110000   ; l'opérande est le masque
```

Après le ANDA **{Valeur} = %1010 0000**

ANDCC

Effectue un ET logique **&** entre l'opérande et le registre de condition **CC**

	AND	&
0	0	0
0	1	0
1	0	0
1	1	1

Peut être utile pour positionner à 0 certains Bits du registre CC.

```
{CC} & {%xxxxxxxx} ---> {CC}
```

Exemple : On souhaite positionner à 0 les bits : 3, 2, 1 et 0 du registre CC

```
ANDCC   #$F0              ; CC=$12 Avant l'instruction
                          ; CC=$10 Après l'instruction

   {CC} = $12    0001 0010
Opérande = $F0   1111 0000
    $12 & $F0  = 0001 0000 = $10  les 4 derniers bits de CC ont été remis à 0
```

{.....} = contenu de

```
                 efhinzvc
Après l'instruction  CC = __H_NZVC
```

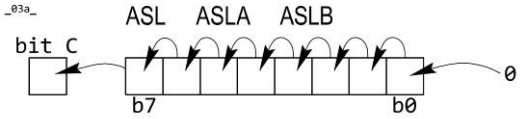
_ = inchangé avec **V=0**

ASL ASLA ASLB (**A**rithmétique **S**hift **L**eft)

Décalage arithmétique d'un rang vers la gauche, tous les bits de l'accumulateur A ou B, ou d'un octet en mémoire.
Le bit 0 est remplacé par un 0
Le bit 7 est mis dans le bit C du registre de condition CC, ce bit C est appelé bit de retenue (Carry)

Après ce décalage, le résultat est une multiplication par 2 de l'opérande

Identique aux instructions LSL (même fonction et même OpCode).



efhinzvc
Après l'instruction **CC = ____NZVC** _ = inchangé

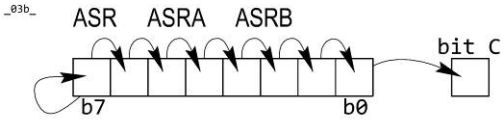
V = N ⊕ C après décalage ⊕ C'est un XOR

ASR ASRA ASRB (**A**rithmétique **S**hift **R**ight)

Décalage arithmétique d'un rang vers la droite.
Le bit 0 est mis dans le bit C
Le bit 7 est recopié à la place qu'il occupait précédemment

Après ce décalage, le résultat est une division par 2 de l'opérande

Lorsque l'on divise par 2 un nombre négatif (en complément à 2) le bit de signe Bit 7 doit être conservé.



efhinzvc
Après l'instruction **CC = ____NZ_C** _ = inchangé

BITA BITB

Effectue un Et logique **&** bit par bit entre le contenu d'un accumulateur A ou B et le contenu de l'opérande.:
Seul le registre de condition CC est modifié.

	AND	&
0	0	0
0	1	0
1	0	0
1	1	1

Le bit N Le bit N est positionné à 1 ou 0 selon que le résultat de l'instruction est négatif ou positif.

Le bit Z **Z = 0** si le résulat est non nul
Z = 1 si le résulat est = 0

Exemple : `BITA ,X` type indexé à déplacement constant (nul en l'occurrence).

Avant l'instruction **{X}=$0150** **{A}=$2A** **{$0150}=$51**
{.....} = contenu de

```
  $51    0101 0001
  $2A    0010 1010
$51 ʌ $2A  0000 0000 = $00
```

Le résultat du ET logique est égal à 0 alors le **bit Z = 1**

Après l'instruction le **bit Z= 1** **bit N = 0** car le résultat est positif

efhinzvc
Après l'instruction **CC = ____NZ0_** _ = inchangé avec **V=0**

INSTRUCTIONS DE BRANCHEMENTS (Adressage Relatif)

Les instructions de branchements :

-- peuvent être :
Conditionnels (avec une condition)
Inconditionnel (sans aucune condition)

-- sont précédées la pluspart du temps par des instructions de test qui reposent essentiellement sur l'emploi du registre d'état CC.

-- provoque un branchement à une adresse donnée (Adresse effective)

Branchements Relatifs

Utilisé uniquement pour les instructions de branchement conditionnel (branchement uniquement si la condition est réalisé). L'équivalent du IF….THEN en basic.

Dans ce mode, l'opérande sera ajoutée ou retranchée au compteur ordinal PC (suivant que l'opérande est positif ou négatif) au compteur ordinal PC. (arithmétique Signé ou dite aussi en complément à deux)

Rappel : le PC pointe sur la prochaine instruction à exécuter.

Choix d'écriture :

Mode 1 : Spécification de l'adresse absolue
```
BNE     $840F
```

Mode 2 : On laisse l'assembleur gérer les adresses. Ce mode sera utilisé lorsque les distances ne sont pas trop importante, ce qui permet déviter le nombre d'étiquette.

```
BNE     *+$F
```
***** remplace la valeur du PC au début de l'instruction
+ spécifie un branchement vers l'avant
- spécifie un branchement vers l'arrière

```
2F40  26 F8          BNE   *-6     ;
```
on aura comme code objet 26 F8 et non pas 26 FA
car il faut soustraire 2 octets supplémentaire à la valeur -6
pour obtenir la valeur de l'offset `-6 = $FA`
`-8 = $F8`

Certaine valeur ne seront pas à utiliser !

`*+1`	Conduit à une adresse au milieu de l'instruction
`*+0`	Conduit à une boucle sans fin
`*+2`	Donne le contrôle à l'instruction qui suit l'instruction de branchement. Le saut est non nécessaire et est équivalent à deux instructions NOP

Mode 3 : Définition d'étiquettes de branchement
Plus facile, cela évite tout calcul d'adresse
Pas besoin de connaître le nombre d'octet généré par l'instruction
Ce mode est à éviter pour les déplacement très court, on utilisera le mode n°2

Branchements relatifs courts

Le déplacement est codé sur un seul octet.

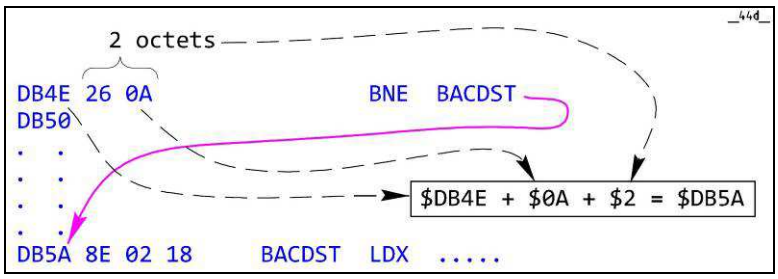
L'instruction de branchement est toujours précédée par une opération qui peut s'assimiler à un test.
Le résultat de ce test affecte un ou plusieurs bits du registre CC, le branchement se fait ou pas suivant l'état de ces bits.

Il permet d'atteindre toute la mémoire par un déplacement de **-128 à +127** octets.

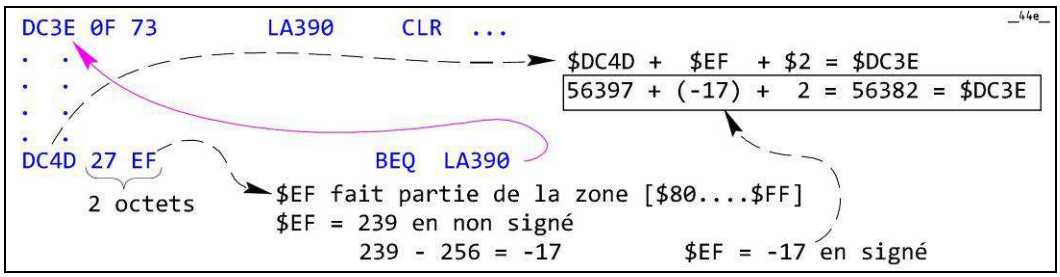
Cet adressage permet donc de "brancher" le programme à une instruction se trouvant :

`- 128 à -1`	`($80 à $FF)`	branchement **en arrière** par rapport au PC
`0 à + 127`	`($00 à $7F)`	branchement **en avant** par rapport au PC

Branchement Court en Avant :



Branchement Court en Arrière :



Branchements Relatifs longs

Le "L" devant l'instruction signale le mode relatif long

Fonctionne comme d'adressage relatif court mais le déplacement (offset) est codé sur deux octets 16 bits signés. Les branchements longs permettent un branchement à n'importe quelle adresse de l'espace mémoire de 64 Ko.

Il permet d'atteindre donc toute la mémoire par un déplacement de **-32768 à +32767**

- 32768 à -129 ($8000 à $FF7F) branchement **en arrière** par rapport au PC
+128 à + 32767 ($0080 à $7FFF) branchement **en avant** par rapport au PC

Pour la zone -128 à +127 on utilisera les branchements Courts

Il occupe 4 octets mais reste très peu pratique pour la mise en place des routines devant pouvoir s'implanter n'importe où dans la mémoire.

Au niveau du code machine, ce qui différencie l'adressage long de l'adressage court, c'est la présence d'un pré-octet, pour l'op-code du mnémonique, dont la valeur est constante, ce pré-octet est de valeur $10.
Sauf pour **LBRA** et **LBSR** qui restent sur un op-code de 1 octet.
Ces deux branchements **LBRA** et **LBSR** sont souvent utilisé en cas d'optimisation de la taille d'un programme.

Le code opératoire proprement dit est le même dans les deux cas.

Exemple :

```
        LBEQ $2000      ; branche le déroulement à l'adresse $2000
                        ; si le bit Z du registre CC est positionné.
```

Si on utilise une étiquette

```
        BEQ   FIN       ; branchement COURT
        LBEQ  FIN       ; branchement LONG
```

Si le déplacement est connu

```
        BEQ   *+10      ; branchement COURT
        LBEQ  *+$12F6   ; branchement LONG
```

Tableau Regroupant Tous les Branchements

BRANCHEMENTS INCONDITIONNELS

		Mnémonique	Op	~	#	
BRA	**Branchement Toujours** (équivalement à un saut JMP)	**BRA $xx**	20	3	2	BRanch Always
		LBRA $xxxx	16	5	3	Long BRanch Always
BRN	**Branchement Jamais** (instruction de non opération)	**BRN $xx**	21	3	2	BRanch Never
		LBRN $xxxx	1021	5	4	Long BRanch Never
BSR	**Branchement à un sous-programme**	**BSR $xx**	8D	7	2	Branch to SubRoutine
		LBSR $xxxx	17	9	3	Long Branch to SubRoutine

BRANCHEMENTS CONDITIONNELS

Signé			Branchement Si...			Mnémonique	Op	~	#	
	non	**BCC**	**Branch si pas de retenue** Donc si retenue à 0	si C = 0	bit **C**	**BCC $xx**	24	3	2	Branch if Carry Clear
						LBCC $xxxx	1024	5(6)⑤	4	Long Branch if Carry Clear
	non	**BHS**	**Branch si supérieur ou égal** Reg ⩾ Mém	il n'y a **pas** de retenue		**BHS $xx**	24	3	2	Branch if Higher or Same
						LBHS $xxxx	1024	5(6)⑤	4	Long Branch if Higher or Same
	non	**BCS**	**Branch si retenue**	si C = 1		**BCS $xx**	25	3	2	Branch if Carry Set
						LBCS $xxxx	1025	5(6)⑤	4	Long Branch if Carry Set
	non	**BLO**	**Branch si inférieur** Reg < Mém	il y a une retenue		**BLO $xx**	25	3	2	Branch if Lower
						LBLO $xxxx	1025	5(6)⑤	4	Long Branch if Lower
		BPL	**Branch si positif** Le résultat d'une opération est à 0	si N = 0	bit **N**	**BPL $xx**	2A	3	2	Branch if Plus
						LBPL $xxxx	102A	5(6)⑤	4	Long Branch if Plus
		BMI	**Branch si négatif** Le résultat d'une opération est à 1	si N = 1		**BMI $xx**	2B	3	2	Branch if Minus
						LBMI $xxxx	102B	5(6)⑤	4	Long Branch if Minus
oui		**BVC**	**Branch si pas de débordement** Il n'y a pas de dépassement de capacité	si V = 0	bit **V**	**BVC $xx**	28	3	2	Branch if oVerflow Clear
						LBVC $xxxx	1028	5(6)⑤	4	Long Branch if oVerflow Clear
		BVS	**Branch si débordement** Il a dépassement de capacité	si V = 1		**BVS $xx**	29	3	2	Branch if oVerflow Set
						LBVS $xxxx	1029	5(6)⑤	4	Long Branch if oVerflow Set
oui	**non**	**BNE**	**Branch si différent** Reg ≠ Mém Produit un résultat différent de 0	si Z = 0	bit **Z**	**BNE $xx**	26	3	2	Branch if Not Equal
						LBNE $xxxx	1026	5(6)⑤	4	Long Branch if Not Equal
		BEQ	**Branch si égal** Reg = Mém Produit un résultat égal à 0	si Z = 1		**BEQ $xx**	27	3	2	Branch if Equal
						LBEQ $xxxx	1027	5(6)⑤	4	Long Branch if Equal
oui		**BLT**	**Branch si inférieur** Reg < Mém	N ⊕ V = 1	Ne dépendant pas des bits de CC	**BLT $xx**	2D	3	2	Branch if Less Than
						LBLT $xxxx	102D	5(6)⑤	4	Long Branch if Less Than
oui		**BGT**	**Branch si supérieur** Reg > Mém	(N ⊕ V) + Z = 0		**BGT $xx**	2E	3	2	Branch if Greater Than
						LBGT $xxxx	102E	5(6)⑤	4	Long Branch if Greater Than
	non	**BHI**		C + Z = 0		**BHI $xx**	22	3	2	Branch if HIgher
						LBHI $xxxx	1022	5(6)⑤	4	Long Branch if HIgher
oui		**BGE**	**Branch si supérieur ou égal** Reg ⩾ Mém	N ⊕ V = 0		**BGE $xx**	2C	3	2	Branch if Greater than Equal to
						LBGE $xxxx	102C	5(6)⑤	4	Long Branch if Greater than Equal to
oui		**BLE**	**Branch si inférieur ou égal** Reg ⩽ Mém	(N ⊕ V) + Z = 1		**BLE $xx**	2F	3	2	Branch if Less than or Equal to
						LBLE $xxxx	102F	5(6)⑤	4	Long Branch if Less than or Equal to
	non	**BLS**		C + Z = 1		**BLS $xx**	23	3	2	Branch if Lower or Same
						LBLS $xxxx	1023	5(6)⑤	4	Long Branch if Lower or Same

	ET Logique	AND
+	OU Logique	OR
⊕	OU Exclusif Logique	XOR
:	Concaténation	
Reg	Registe	(Reg) = contenue du registre
Mém	Mémoire	(Mém) = contenue de la case mémoire
⑤	Instructions branchement : 5(6) signifie, 5 cycles si branche non fait, 6 cycles si branchement réalisé	
$xx	Adresse en 8 bits	
$xxxx	Adresse en 16 bits	

Op	**Op-Code** ou Code instruction en hexadécimal
~	**Nombre de cycle**. Lorsqu'il est suivit d'un + le nombre total de cycles s'obtient en ajoutant la valeur supplémentaire contenue dans le tableau de l'adressage indexés.
#	**Nombre d'octets** traduisant l'encombrement mémoire. Lorsqu'il est suivit d'un + le nombre total de cycles s'obtient en ajoutant la valeur supplémentaire contenue dans le tableau de l'adressage indexés.

Branchements Inconditionnels

Instructions : **BRA** et **LBRA** **BRN** et **LBRN** **BSR** et **LBSR** **JMP** et **JSR**

Un branchement est une rupture de séquence. La nouvelle valeur du registre PC s'obtient en ajoutant ou en retranchant une valeur (appelée "Déplacement" ou "Offset") à l'ancien contenu du registre PC.

Le déplacement (ou Offset) est un nombre en Arithmétique Signé sur :

8 bits **Branchement COURT**
Instruction du type **Bxx** (hormis les instructions BITA et BITB)
La plage est de `-128 octets` (déplacement en arrière)
à `+127 octets` (déplacement en avant).

16 bits **Branchement LONG**
Instruction du type **LBxx**
La plage est de `-32768 octets` (déplacement en arrière)
à `+32767 octets` (déplacement en avant).

L'utilisation d'un programme "Assembleur" évite le calcul du déplacement : il suffit de mettre un nom dans la colonne étiquette. Les branchements (ou adressages relatifs) respectent la translatabilité des programmes.

Branchements Inconditionnels **BRA LBRA** (**BR**anch **A**lways) "Branchement toujours"

Brancher toujours, correspond à JMP avec un adressage Relatif.

Pour avoir un programme structuré, cette instruction est à éviter ! Car c'est l'équivalent du GOTO en Basic !

L'adresse du branchement Relatif est calculée en ajoutant au registre PC un déplacement signé (valeur en complément à deux). `{PC + Déplacement} ---> PC` {.....} = contenu de

`BRA $xx` (Branch Always) pour accéder à un espace adressable de 256 octets
`LBRA $xxxx` (Long Branch Always) pour accéder à la totalité de l'espace adressable.

Branchements Inconditionnels **BRN LBRN** (**BR**anch **N**ever) "Branchement jamais"

Ne jamais brancher, équivalent à NOP. Cette instruction n'effectue aucune opération.
Utilisé uniquement pour disposer de programmes plus facilement lisibles.
Existe pour maintenir une symétrie dans le jeu d'instructions.

L'adresse du branchement Relatif est calculée en ajoutant au registre PC un déplacement signé (valeur en complément à deux). `{PC + Déplacement} ---> PC` {.....} = contenu de

`BRN $xx` (Branch Never) pour accéder à un espace adressable de 256 octets
`LBRN $xxxx` (Long Branch Never) pour accéder à la totalité de l'espace adressable

Branchements Inconditionnels **BSR LBSR** (**B**ranch to **S**ub**R**outine) "....à un sous programme"

Branchement inconditionnel à un sous-programme, équivalent à JSR avec un adressage Relatif.
L'adresse du branchement Relatif est calculée en ajoutant au registre PC un déplacement signé (valeur en complément à deux). `{PC + Déplacement} ---> PC` {.....} = contenu de

`BSR $xx` (Branch to Subroutine) pour accéder à un espace adressable de 256 octets
`LBSR $xxxx` (Long Branch to Subroutine pour accéder à la totalité de l'espace adressable

Utilisé dans les programmes relogeables (pour info : l'instruction RTS effectue un retour vers le programme d'origine)
L'avantage de BSR et de LBSR par rapport à JSR est que si la totalité du programme est déplacée alors les instructions de branchement du sous-programme permettront encore de se brancher à la bonne adresse. C'est parce que l'adresse de départ est calculée par rapport au PC.
Par contre LBSR à un temps d'exécution plus lent que JSR.

Branchement Conditionnel

Ces branchements sont des ruptures de séquence.

Ces instructions de branchements conditionnels testent les valeurs du registre de conditions CC, ainsi que les combinaisons de ces valeurs.

Le calcul du branchement ou plutôt du déplacement est dit aussi Offset.

-- Si la condition est réalisée, le branchement a lieu.
-- Si la condition n'est pas réalisée le branchement est ignoré, c'est-à-dire que le registre PC (ou PCR) est inchangé.

Les conditions sont relatives à certains bits du registre de condition ou registre d'état, il s'agit du registre CC.

Branchement si	N=1	(bit 3)	**N**égatif	si le résultat ou chargement est négatif
	Z=1	(bit 2)	**Z**éro	si le résultat ou chargement est nul.
	V=1	(bit 1)	o**V**erflow	si il y a dépassement de capacité.
	C=1	(bit 0)	**C**arry	si il y a une retenue.

Exemple 01

```
        LDA     #$FF    ; charge A avec la valeur $FF
TOTO    DECA            ; décrémente A
        BNE     TOTO    ; branch à l'étiquette TOTO si A <> 0
```

Exemple 02

```
        LDA     $B200   ; charge A avec le contenu de la mémoire $B200
        CMPA    #$41    ; compare A avec la valeur $41 en faisant le calcul  A-$41
                        ; (si A=$41 alors A-$41=0) puis positionne le bit Z à 1 si A=$41
        BEQ     TITI    ; test du bit Z et branch à TITI car Z=1
```

Branchements traduisant une condition simple (dépendant de la valeur d'un bit de CC)
Voir tableau ci-dessus.

Branchements Conditionnels	**BCC BHS**	(voir tableau ci-dessus)
Branchements Conditionnels	**BCS BLO**	(voir tableau ci-dessus)
Branchements Conditionnels	**BPL BMI**	(voir tableau ci-dessus)
Branchements Conditionnels	**BVC BVS**	(voir tableau ci-dessus)
Branchements Conditionnels	**BNE BEQ**	(voir tableau ci-dessus)
Branchements Conditionnels	**BLT**	(voir tableau ci-dessus)
Branchements Conditionnels	**BGT BHI**	(voir tableau ci-dessus)
Branchements Conditionnels	**BGE**	(voir tableau ci-dessus)
Branchements Conditionnels	**BLE BLS**	(voir tableau ci-dessus)

CLR CLRA CLRB (CL…= CLear)

Remettre à 0 le contenu d'un accumulateur ou d'une case mémoire {.....} = contenu de

```
CLR      0 --->   {Mém}
CLRA     0 --->   {A}
CLRB     0 --->   {B}
                       efhinzvc
Après l'instruction    CC = ____0100        _ = inchangé
```

CMPA CMPB CMPD CMPS CMPU CMPX CMPY

Afin d'effectuer une comparaison, le contenu de l'opérande est soustrait au contenu du registre spécifié.
Les indicateurs `----NZVC` du registre CC sont positionnés suivant le résultat de cette soustraction.

Les contenus de la case mémoire et des registres ne sont pas affectés.

```
{A} - {Mém}              {B} - {Mém}              {D} - {Mém, Mém + 1}
{X} - {Mém, Mém + 1}     {Y} - {Mém, Mém + 1}
{S} - {Mém, Mém + 1}     {U} - {Mém, Mém + 1}
```

{.....} = contenu de

```
                  efhinzvc
Après l'instruction   CC = ____NZVC        _ = inchangé
```

```
N ⊕ V = 1   -->  {Reg} < {Mém}     en arithmétique signé          ⊕ C'est un XOR
    C = 1   -->  {Reg} < {Mém}     en arithmétique non signé
    Z = 1   -->  {Reg} = {Mém}     soustraction de 2 valeurs identiques, résultat = 0 donc Z =1
```

Exemple :

```
                    ;----Prog INS-03
0000                        ORG    $0000    ;
0000      0100      VarTest EQU    $100     ;
0000 B6   0100              LDA    VarTest  ; Lire l'octet à tester
0003 81   00                CMPA   #$00     ; Comparer à 0
0005 27   05                BEQ    E_Zéro   ; Est-ce un 0
0007 81   01                CMPA   #$01     ; Comparer à 1
0009 27   03                BEQ    E_Un     ; Est-ce un 1
000B 39                     RTS              ;
                                             ;
000C C6   00        E_Zéro  LDB    #0       ;
000E C6   01        E_Un    LDB    #1       ;
```

Exemple : Programme qui compare 2 chaînes de caractères. La longueur de ces 2 chaînes est à l'adresse $50.
1ière chaîne stockée à partir de $00 2ième chaîne stockée à partir de $20

```
0000                        ORG    $0000     ;   Prog INS-01
                  0083  Val83 EQU    $83       ;
0000 D6   FF                LDB    $FF       ; indicateur de chaînes différentes
0002 9E   00                LDX    $0000     ; pointe sur la 1ière chaîne
0004 109E 20                LDY    $0020     ; pointe sur la 2ième chaîne
0007 A6   80          COMPT LDA    ,X+       ; 1er caract de 1ière chaîne
0009 A1   A0                CMPA   ,Y+       ; 1ier caract de 2ième chaîne
000B 26   06      0013      BNE    FIN       ; non égalité alors terminé
000D 9C   50                CMPX   $50       ; dernier caractère ?
000F 26   F6      0007      BNE    COMPT     ; non, recommence
0011 C6   00                LDB    #$00      ; indicateur de chaîne égales
0013 D7   51          FIN   STB    $51       ; (B)=$00 si les deux chaînes sont égales
0015 3F                     SWI              ; (B)=$FF si les deux chaînes sont différentes
```

COMA COMB COM

Donne le complément à 1, bit à bit d'un accumulateur ou d'un octet mémoire.

$\overline{\{Mém\}}$ ---> {Mém} $\overline{\{A\}}$ ---> {A} $\overline{\{B\}}$ ---> {B}

x	$\overline{x}$
0	1
1	0

{.....} = contenu de

```
                  efhinzvc
Après l'instruction   CC = ____NZ01        _ = inchangé avec V=0 et C=1
```

CWAI (Clear WAit Interrupt) Attente d'interruption

Le 6809 doit pouvoir se mettre en attente ou se synchroniser sur un évènement extérieur dont la présence est signalée par une ou des broches d'entrée d'interruptions (Voir instructions SYNC et CWAI).

L'instruction CWAI synchronise le 6809 sur un évènement externe par le biais d'une interruption.

`{CC  %xxxxxxxx} ---> CC` avec bit **E=1** {....} = le contenu de

CWAI fonctionne dans sa première phase comme ANDCC (elle effectue un ET logique entre un octet immédiat et le registre de conditions CC) ce qui mettra à 0 certains bits de CC, donc à effacer les masques d'interruptions et sauvegarde l'ensemble des registres dans la pile le système S puis se met en attente d'interruption.

registre CC

7	6	5	4	3	2	1	0
E	F	H	I	N	Z	V	C

Ceci a pour but de positionner à 0 certains bits de CC, on peut effacer ici les masques d'interruptions.
Ensuite le bit E est mis à 1 ce qui a pour but de provoquer une sauvegarde totale du contexte (états de tous les registres internes) dans la pile matérielle.

Cet état est donc sauvegardé, le 6809 se met en position d'attente d'interruption (il n'exécute donc plus d'instructions).

Lorsqu'une interruption intervient, le 6809 se branche à la routine de gestion d'interruption et l'exécute.

Lorsque le 6809 rencontre une instruction RTI (retour d'interruption) il restaure complètement l'état du 6809.
Le bit E de CC est positionné à 1.

Avant l'interruption CWAI :
-- Si le bit F=1 avant CWAI, seule IRQ est activée.
-- Si le bit F=0 avant CWAI, FIRQ et IRQ peut interrompre le 6809.

NMI peut également interrompre le 6809 bien que ce mode de fonctionnement ne soit pas très habituel.
Exemple d'attente d'interruption NMI

```
CWAI   #$FF        ; car IRQ et FIRQ sont masqué.
```

CWAI permet au 6809 de synchroniser le traitement de la séquence d'exception sur une interruption IRQ ou FIRQ.

La réponse est rapide puisque le 6809 a déjà sauvegardé l'ensemble des registres.

Il ne lui reste plus qu'à charger le vecteur d'interruption pour le compteur programme, soit à partir des mémoires :

`$FFF8 et $FFF9` pour IRQ
`$FFF6 et $FFF7` pour FIRQ

Notons toutefois que le bit E est égal à 1, par conséquent, même si interruption est un FIRQ, la sauvegarde est totale et non partielle.

Au dépilement par l'instruction RTI, l'ensemble des registres sera restitué, d'où un mouvement du pointeur S de + ou - 12 unités.

On trouve CWAI dans des applications lorsque l'on a besoin d'une interruption simultanée de toutes les unités de traitement sur une seule impulsion de départ.

CWAI force toutefois le 6809 à se brancher vers une séquence d'exception, ce qui n'est pas le cas avec SYNC.

Dans d'autres circonstances, CWAI permet aux programmeurs de sélectionner un endroit précis dans son programme pour lancer un traitement d'exception, même lorsqu'il s'agit d'une interruption matérielle.

C'est pour la raison pour laquelle nous l'appelons "interruption matériel programmée" alliant les deux concepts matériel et logiciel.
CWAI écarte en quelque sorte de l'aspect aléatoire d'une interruption matérielle.

Contrairement à l'instruction SYNC l'instruction CWAI ne met pas les bus de données et d'adresses en haute impédance.

DAA

Utilisé dans les opérations en BCD.
Permet un ajustement décimal sur A. Consiste à faire +06 à A.

Exemple : 2 nombres BCD

```
   8       0000 1000
 + 7     + 0000 0111
------   ------------
=  15    =  0000 1111  = $0F au lieu de 15 en BCD
```

Il faut ajouter 6

```
  $0F      0000 1111
+ $6     + 0000 0110
-------  ------------
         =  0001 0101  = $15 codé DCB
```

```
                    efhinzvc
Après l'instruction CC = ____NZ_C        _ = inchangé
```

DEC DECA DECB

(**DEC**rement....)

Soustrait 1 à l'accumulateur ou à l'octet mémoire, sans modifier le bit C.

```
{Mém} - 1 ---> {Mém}
{A}   -  1 ---> {A}
{B}   -  1 ---> {B}
```

{.....} = contenu de

Pour Décrémenter les registres X, Y, U ou S voir LEAX, LEAY, LEAU, LEAS

Si il faut détecter le changement de signe d'une valeur alors il faudra tester les bits N et V.

```
                    efhinzvc
Après l'instruction CC = ____NZV_        _ = inchangé
```

EORA EORB

(symbole ⊕ ou parfois ⩛)

	XOR	⊕
0	0	0
0	1	1
1	0	1
1	1	0

Dans les opérations Binaire le OU Exclusif s'effectue bit par bit, entre l'opérande et l'accumulateur A ou B.

On peut utiliser le OU exclusif pour des comparaisons, si un bit est différents alors le OU exclusif de ces deux octets sera non nul.

On peut aussi utiliser le OU exclusif pour complémenter un octet en faisant un OU exclusif avec un octet $FF %1111 1111.

```
LDA     Valeur        ; Valeur = %1010 1010
EORA    %11111111     ; =$FF
                      ; après exécution  {A}=%0101 0101
```

EORA

```
{A} ⊕ {Mém} ---> {A}
{A} ⊕ {%xxxxxxxx} ---> {A}
```

EROB

```
{B} ⊕ {Mém} ---> {B}
{B} ⊕ {%xxxxxxxx} ---> {B}
```

⊕ C'est un XOR

{.....} = contenu de

```
                    efhinzvc
Après l'instruction CC = ____NZ0_        _ = inchangé  avec V=0
```

EXG et TFR (ExchanGe TransFeR)

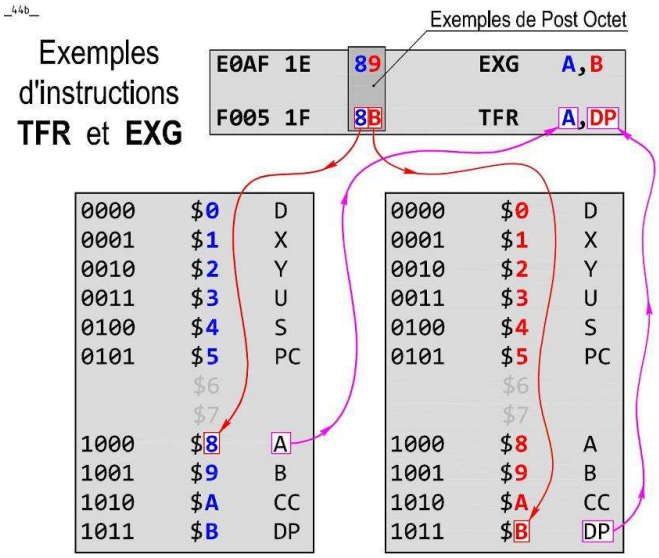
Pour EXG et TFR le contenu de l'octet suivant le code opération est appelé
Le transfert ou l'échange ne peut se faire que sur des registres de même taille.

POST-OCTET (PostByte), il précise la paire de registre sur laquelle s'applique les instructions.

TFR Transfert de registre à registre **R1 ----> R2** `TFR R1,R2`

EXG Echange de registre **R1 <----> R2** `EXG R1,R2`

Ces deux instructions, ne sont que dans le mode d'adressage Immédiat.



Instruction EXG

Permet l'échange de deux registres 8 bits ou deux registres 16 bits.
Le registre de condition CC n'est pas modifié. **R1 <----> R2**

```
EXG R1,R2     ; échange du contenu de registre de même taille
EXG A,B       ; avant {A}=$10 {B}=$40
              ; après {A}=$40 {B}=$10
```

{.....} = contenu de

```
              efhinzvc
Après l'instruction   CC = _________
```

_ = inchangé

Instruction TFR

Permet le transfert d'un registre R1 dans un autre registre R2 de même taille.
Le registre de condition CC n'est pas modifié. . **R1 ----> R2**

```
TFR R1,R2     ; transfert le contenu de R1 dans R2
              efhinzvc
Après l'instruction   CC = _________
```

_ = inchangé

Exemples d'Instructions de Transfert de Données

Transfert 8 bits

```
LDA    Adrss1
STB    Adrss2
TFR    A,DP    ; copie d'un registre dans un autre
EXG    A,B     ; échange de contenus
```

Transfert 16 bits

```
LDY    Adrss3
STX    Adrss4
TFR    X,Y
EXG    PC,X
```

Opérations avec la pile

Instruction d'empilement Push et de dépilement Pull sur 2 pointeurs (U utilisateur et S système)

INC INCA INCB (**INC**rement....)

Ajoute 1 à l'accumulateur ou à l'octet mémoire, sans modifier le bit C.

```
{Mém} + 1 ---> {Mém}
{A} + 1 ---> {A}
{B} + 1 ---> {B}
```

{.....} = contenu de

Pour Incrémenter les registres X, Y, U ou S voir LEAX, LEAY, LEAU, LEAS

Si il faut tester le changement de signe alors on devra tester les bits N et V.

```
                  efhinzvc
Après l'instruction   CC = ____NZV_
```

_ = inchangé

JMP (**JuMP** = saut)

C'est un saut, une rupture de séquence. Il est obtenu en chargeant le compteur ordinal (PC Program Counter) avec l'adresse spécifiée par l'opérande, adresse à laquelle il faut sauter. **ATTENTION** : pour du code relogeable, le JMP est à bannir.

Adresse Effective ----> PC

Exemple 01

```
JMP   $F005
```

Transfert inconditionnel à l'adresse $F005
Après éxécution de l'instruction PC = $F005

Exemple 02

```
JMP   [A,X]
```

Saut inconditionnel indexé indirect

```
{X} = $8000   {A} = $06
{$8006} = $F6
{$8007} = $79
```

{.....} = contenu de

après l'instruction `JMP [A,X]` `{PC} = $F679`

Exemple 03

```
JMP   $D855,PCR
```

Saut inconditionnel relatif au PC
`$D855` n'est pas un offset mais une adresse
Avant l'instruction `{PC} = $7083`
Après l'instruction `{PC} = $D855`

Equivalente au GOTO en Basic

```
JMP DEBUT        ;
JMP $F03E        ; charge le PC et effectue le saut vers l'adresse $F03E
```

Pour avoir un programme structuré, cette instruction est à éviter !
Car c'est l'équivalent du GOTO en Basic !

JSR (**J**ump to **S**ub**R**outine)

Equivalent au Gosub en Basic. Voir aussi les instructions RTS et BSR (Branch To Subroutine)

Dès que le 6809 rencontre **JSR**, il charge le contenu du compteur ordinal PC (ou PCR) dans la pile puis se branche à l'adresse spécifié. Le pointeur de pile système est donc décrémenté de 2.

```
JSR   $F005
```

Saut inconditionnel à un sous programmes à l'adresse $F005

La différence avec l'instruction JMP est qu'avec JSR la valeur du registre PC à la fin de l'instruction est sauvegardée dans la pile système S et cette valeur sera restituée à la fin du sous programme lors de la rencontre de l'instruction RTS.

Le registre S est également décrémenté de 2 octets, en même temps le registre PC prend la valeur $F005

ATTENTION : pour du code relogeable, le JSR est à bannir

Voir aussi l'instruction BSR (Branch To Subroutine)

LDA LDB (LD…= LoaD)

Charger l'accumulateur A ou B à l'aide d'une contenu d'une adresse mémoire.

```
LDA      {Mém} ---> A                                          {.....} = contenu de
LDB      {Mém} ---> B

LDA #$94      ; après l'instruction {A}=$94        # indique l'adressage immédiat
LDA $10FF     ; après l'instruction {A}={$10FF}
```

L'instruction LDA met la valeur `$94` dans l'accumulateur **A**
$94 étant négatif (car le bit 7=1) et non nul alors le bit N=1 et bit Z=0

Bit Z Dans le cas des instructions LD… :
-- Z mis à 1 si A ou B est chargé à $00
-- Z mis à 0 si A ou B est chargé avec une valeur <> de 0

Bit N Un octet peut se présenter :
-- Soit un nombre positif entre 0 et 255
-- Soit un nombre signé entre -128 et +127 (le nombre négatif étant représenté par son complément à 2)

Le bit **N** est tout simplement la recopie du bit 7 de l'octet.

```
                    efhinzvc
Après l'instruction CC = ____100_                              _ = inchangé
```

LDD LDX LDY LDS LDU (LD…= LoaD)

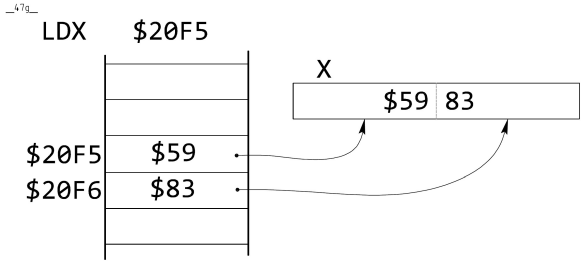
Chargement les registres 16 bits D, X, Y, S et U avec le contenu de 2 adresses mémoire contiguës ou avec une valeur binaire (cas d'adressage immédiat)

```
{Mém, Mém+1} --->  {Registre}                                  {.....} = contenu de
```

L'octet de poids fort est chargé en premier (contenu de l'adresse Mém)
L'octet de poids faible est chargé en second (contenu de l'adresse Mém+1)

```
                    efhinzvc
Après l'instruction CC = ____100_                              _ = inchangé
```

```
LDX  $20F5
```



LEAS LEAU LEAX LEAY (LEA = Load Effective Adress)

Charge le registre désigné **S**, **U**, **X** ou **Y** avec l'adresse effective, concerne les pointeurs de donnée (pointeur d'index et piles). **AE ----> Registre** (AE = Adresse Effective)

Le calcul de l'AE (Adresse Effective) se fait en fonction du seul mode **d'adressage INDEXE** (Direct ou Indirect) et charge cette valeur dans le pointeur désiré (Registre X, Y, S ou U).

Ces instructions ne fonctionnent donc qu'avec **l'adressage INDEXE** (Direct ou Indirect).
Puisque ce sont les seuls modes qui nécessitent un calcul d'adresse effective.

Les instructions LEA…… servent à mouvoir les pointeurs ou les index de façon très souple à travers tout l'espace mémoire pour rechercher les adresses des données.

L'instruction LEA... charge l'adresse et non pas la donnée pointée par le registre d'adresse.

On peut ainsi définir facilement des blocs de données relatifs à d'autres adresses pendant l'exécution d'un programme.

```
LEAS $80,X      ; adressage INDEXE à déplacement 8 bits
                ; si X=$2000 l'adresse effective = $2080
                ; CC ne sera pas affecté
LEAX -1,X       ; décrémentation de X d'une unité
LEAX 4,X        ; incrémentation de X de +4
```

Pour une optimisation du source, la séquence

```
STA   ,X        ;
LEAX  +1,X      ;
DECB            ;
```

Peut être remplacer par

```
STA   ,X+       ; stockage avec autoincrémentation
DECB            ;
```

`LEAX 0,X` équivaut à 2 instructions NOP avec mise à 1 de Z si l'index X passe à zéro.

Ces instructions LEAS, LEAU, LEAX, LEAY permettent de reloger des programmes.

LEAX et LEAY : N'agissent que sur le bit Z, ce bit est mis à 1 quand X ou Y passe par la valeur 0

Donc ces deux instructions peuvent précéder les branchements courts du type BEQ ou BNE ou long du type LBEQ ou LBNE.

```
                    efhinzvc
Après l'instruction CC = _____Z__   ( _ = inchangé  Z=1 si X ou Y passe à 0)
```

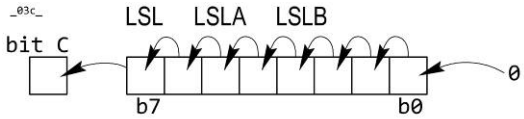
LEAU et LEAS : N'agissent sur aucun bit du registre CC, ces deux instructions ne peuvent être employées conjointement à des branchements conditionnels.

```
                    efhinzvc
Après l'instruction CC = _________  ( _ = inchangé  )
```

L'usage des registres S et U comme compteurs est fortement déconseillé

LSL LSLA LSLB (Logical Shift Left) Décalage Logique vers la gauche

Après ce décalage, le résultat correspond à une multiplication par 2 de l'opérande



Ces instructions sont identiques aux instructions ASL (Même fonction et même OpCode)

```
                 efhinzvc
Après l'instruction  CC = ____NZVC                    _ = inchangé
```

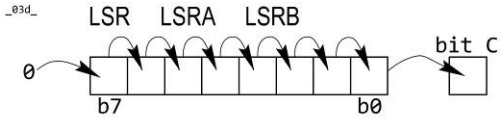
V = N ⊕ C après décalage ⊕ C'est un XOR

Exemple : **LSL [$0310]**

Avant	Après	
{$0310} = $4B	{$0310} = $4B	{.....} = contenu de
{$0311} = $27	{$0311} = $27	
{$4B27} = $B5 = % 1011 0101	{$4B27} = $6A = % 0110 1010	

LSR LSRA LSRB (Logical Shift Right) Décalage Logique vers la droite

Après ce décalage, le résultat correpond à une division par 2 de l'opérande




```
                 efhinzvc
Après l'instruction  CC = ____0Z_C                    _ = inchangé avec N=0
```

MUL

Multiplication des contenus de A et de B le résultat stocké dans D. Les registres A et B étant considérés comme non signé. Cette instruction est une particularité du 6809 rarement trouvé sur les microprocesseurs 8 bits du marché de l'époque.

{A x B} ---> {D} {.....} = contenu de

Exemple : au maxi on peut avoir 255 ($FF) x 255 ($FF) = 65 025 ($FE01)

```
                 efhinzvc
Après l'instruction  CC = _____Z_C                    _ = inchangé
```

NEG NEGA NEGB

Donne le complément à 2 du contenu de la case mémoire, de A ou de B.
Autrement dit ces instructions remplacent l'opérande par son opposé.

{Mém} + 1 ---> {Mém} {A} + 1 ---> {A} {B} + 1 ---> {B}
{.....} = contenu de

```
                 efhinzvc
Après l'instruction  CC = ____NZVC                    _ = inchangé
```

NOP (**No OP**eration)

Elle ne fait rien, elle se contente juste d'incrémenter le Compteur Ordinal, durant 2 cycles (soit 2µS pour un quartz de 4 MHz)

Elle peut servir en autre à :

-- Remplacer une instruction non utile, ce qui permet de ne pas avoir à réécrire tout le programme lors d'un assemblage à la main
-- Elle sert pendant la mise au point, en remplaçant une instruction par NOP, pour éviter de recalculer tous les branchements.
-- La mise au point d'un programme partie par partie en remplaçant par exemple certains sous-programmes par de NOP
-- Provoquer un délai de durée fixe dans l'exécution d'un programme.

Il est évident que le programme définitif devra être débarrassé de ces instructions inutiles afin d'en diminuer le temps d'exécution.

ORA ORB

(symbole + ou parfois V , ≥1)

Dans les opérations Binaire le OU inclusif s'effectue bit par bit, entre l'opérande et l'accumulateur A ou B.

	OR	+
0	0	0
0	1	1
1	0	1
1	1	1

Le OU logique met à 1 n'importe quel bit d'un octet.
Exemple : mettre à 1 les quatre bits de droite de la variable **Valeur**

```
LDA   Valeur       ; {Valeur} = %1010 1010
ORA   %00001111    ; le résultat {Valeur}=%10101111
```

Ces instructions ORA et ORB effectuent un OU logique entre :

-- D'une part le contenu d'une case mémoire ou d'une valeur binaire dans le cas d'adressage immédiat

-- Et d'autre part l'un des accumulateurs A ou B.

ORA	**ORB**	{.....} = contenu de
{A} + {Mém} ---> {A}	{B} + {Mém} ---> {B}	
{A} + {%xxxxxxxx} ---> {A}	{B} + {%xxxxxxxx} ---> {B}	

```
                    efhinzvc
Après l'instruction CC = ____NZ0_
```

_ = inchangé avec **V=0**

ORCC

Effectue un OU logique + entre l'opérande et le registre de condition **CC**

	OR	+
0	0	0
0	1	1
1	0	1
1	1	1

Exemple : On souhaite positionner à 1 les bits : 3, 2, 2 et 0 de CC

```
    ORCC #$0F      ; CC=$12 avant l'instruction
                   ; CC=$1F Après l'instruction
```

```
                 efhi nzvc
    {CC} = $12   0001 0010
Opérande = $0F   0000 1111
      $12 + $0F = 0001 1111 = $1F
```

{.....} = contenu de

les 4 derniers bits de CC ont été remis à 1

Exemple : On souhaite positionner à 1 les bits : I et F

```
    ORCC I,F                ; met les bits b6 et b4 à 1

                 efhi nzvc
      $12 + $0F = 0101 0000
```

Sauvegarde dans la pile. Permet de stocker le contenu d'un ou plusieurs registres internes au sommet de la pile :

Système	pile **S**	**PSHS**
Utilisateur	pile **U**	**PSHU**

Exemple : Sauvegarde des registres X et Y dans la pile Système **PSHS Y,X**
Sauvegarde des registres A, B, et CC dans la pile utilisateur **PSHU A,B,CC**

PSHU A,B,CC
le post Octet sera = % 0000 0111 = $07
l'OpCode instruction = $36

PSHU Y,X
le post Octet sera = % 0011 0000 = $30
l'OpCode instruction = $36

PSHS A,Y,X
le post Octet sera = % 0011 0010 = $32
l'OpCode instruction = $34

Calcul du Post-Octet (octet immédiat)

Exemple d'empilement sur le pointeur de pile S système, **PSHS** (il en serait de même pour le pointeur U)
On commence par le bit de poids fort b7 pour l'ordre d'empilement.

{.....} = contenu de

Si b7=1	alors	S – 1 → S	et	PC Low → {S}	Octet de poids faible du registre PC
		S – 1 → S	et	PC Hight → {S}	Octet de poids fort du registre PC
Si b6=1	alors	S – 1 → S	et	U Low → {S}	Octet de poids faible du registre U
		S – 1 → S	et	U Hight → {S}	Octet de poids fort du registre U
Si b5=1	alors	S – 1 → S	et	Y Low → {S}	Octet de poids faible du registre Y
		S – 1 → S	et	Y Hight → {S}	Octet de poids fort du registre Y
Si b4=1	alors	S – 1 → S	et	X Low → {S}	Octet de poids faible du registre X
		S – 1 → S	et	X Hight → {S}	Octet de poids fort du registre X
Si b3=1	alors	S – 1 → S	et	DP → {S}	
Si b2=1	alors	S – 1 → S	et	B → {S}	
Si b1=1	alors	S – 1 → S	et	A → {S}	
Si b0=1	alors	S – 1 → S	et	CC → {S}	

Exemple :
{S} = $8100 {X} = $10B6 {Y} = $4F03 {.....} = contenu de

Après l'instruction **PSHS X,Y**

S - 4	{$80FC} = $10	correspondant à XH
S - 3	{$80FD} = $B6	correspondant à XL
S - 2	{$80FE} = $4F	correspondant à YH
S - 1	{$80FF} = $03	correspondant à YL

Lorsqu'un mélange de plusieurs registres 8 ou 16 bits est mis dans une instruction **PSHS** ou **PULS**, l'ordre d'exécution est celui imposé par le 6809 et non par celui spécifié par le programmeur

Ordre d'empilement pour **PSHS**
PCL,PCH UL,UH YL,YH XL,XH DP B A CC

Ordre de dépilement pour **PULS**
CC A B DP XH,XL YH,YL UH,UL PCH,PCL

Ainsi pour pousser les registres A et B dans la pile S on peut écrire :
PSHS A,B ou **PSHS B,A** ou **PSHS D** le registre B sera poussé en premier suivi de A

Pour dépiler les données vers les registres X et CC, ont peut écrire indifféremment
PULS X,CC ou **PULS CC,X** l'ordre imposé sera CC puis XH puis XL

Récupération depuis la pile. Permet de charger un ou plusieurs registres internes à l'aide d'octets stocké dans la pile

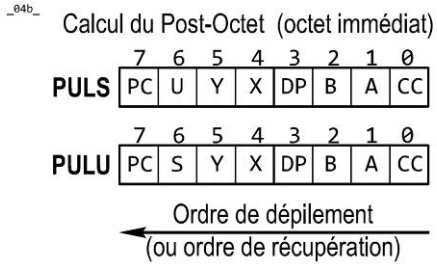
- Système pile **S** **PULS**
- Utilisateur pile **U** **PULU**

Restaurer les registres X et Y depuis la pile Système **PULS Y,X**
Restaurer les registres A, B, et CC depuis la pile utilisateur **PULU A,B,CC**

PULU A,B,CC
le post Octet sera = % 0000 0111 = $07
l'OpCode instruction = $37

PULS Y,X
le post Octet sera = % 0011 0000 = $30
l'OpCode instruction = $35

PULU A,Y,X
le post Octet sera = % 0011 0010 = $32
l'OpCode instruction = $37



Exemple de dépilement sur le pointeur de pile S système, **PULS** (il en serait de même pour le pointeur U)

{…..} = contenu de

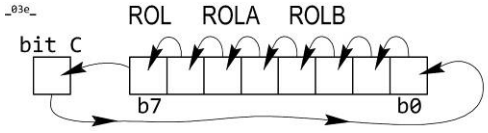
On commence par le bit de poids faible b0 pour l'ordre de dépilement.

Si b0=1	alors	S + 1 → S	et	CC → {S}	
Si b1=1	alors	S + 1 → S	et	A → {S}	
Si b2=1	alors	S + 1 → S	et	B → {S}	
Si b3=1	alors	S + 1 → S	et	DP → {S}	
Si b4=1	alors	S + 1 → S	et	X Hight → {S}	Octet de poids fort du registre X
		S + 1 → S	et	X Low → {S}	Octet de poids faible du registre X
Si b5=1	alors	S + 1 → S	et	Y Hight → {S}	Octet de poids fort du registre Y
		S + 1 → S	et	Y Low → {S}	Octet de poids faible du registre Y
Si b6=1	alors	S + 1 → S	et	U Hight → {S}	Octet de poids fort du registre U
		S + 1 → S	et	U Low → {S}	Octet de poids faible du registre U
Si b7=1	alors	S + 1 → S	et	PC Hight → {S}	Octet de poids fort du registre PC
		S + 1 → S	et	PC Low → {S}	Octet de poids faible du registre PC

Pour gagner du temps et de la mémoire
Le code **PULS A,B** peut être
RTS remplacer par **PULS A,B,PCR**

ROL ROLA ROLB (**RO**tate **L**eft) Rotation vers la gauche

Chaque bit est décalé vers la gauche.

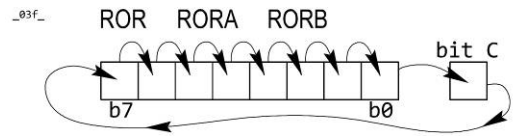



```
                    efhinzvc
Après l'instruction CC = ____NZVC          _ = inchangé

                V = N ⊕ C   après décalage  ⊕ C'est un  XOR
```

ROR RORA RORB (**RO**tate **R**ight) Rotation vers la droite

Chaque bit est décalé vers la droite.




```
                    efhinzvc
Après l'instruction CC = ____NZ_C          _ = inchangé
```

RTI (**R**e**T**urn from **I**nterrupt)

Toutes les séquences de traitement d'interruption doivent se terminer par l'instruction RTI pour éviter toute mauvaise manipulation de la pile.

L'utilisation de cette instruction comme les instructions CWAI, SYNC, … sera très rare.

L'appel à un sous programme peut se faire de 2 façons :
-- Par logiciel par l'utilisation de JSR et le retour par RTS
-- Par le matériel par une interruption, l'instruction de retour dans ce cas là est RTI (voir le détail dans le chapitre des interruptions)

Dès que le 6809 rencontre cette instruction, il teste tout d'abord la valeur du bit E de CC, registre CC est restauré en premier :

registre CC

7	6	5	4	3	2	1	0
E	F	H	I	N	Z	V	C

Bit E = 1 le restant des registres est restauré dans l'ordre A, B, DP, X haut, X bas, Y haut, Y bas, U haut, U bas, PC haut puis PC bas)
Autrement dit : Si E=1 alors l'empilement était total.

Le dépilement correspond à l'ordre connu pour l'opération PSHS, soit CC, A, B, DP, XH, XL, YH, YL, UH, UL, PCH, PCL avec une variation du pointeur S de + ou - 12 unités à la fin de l'instruction RTI.

Bit E = 0 Seul le registre PC est restauré dans l'ordre PC haut puis PC bas (le registre CC ayant été restauré en premier lieu).
Autrement dit : Si E=0 alors il s'agissait d'un empilement partiel au début de l'exécution de l'interruption.
Le 6809 dépile dans l'ordre CC, PCH, PCL et décroît aussi le pointeur de 3 unités

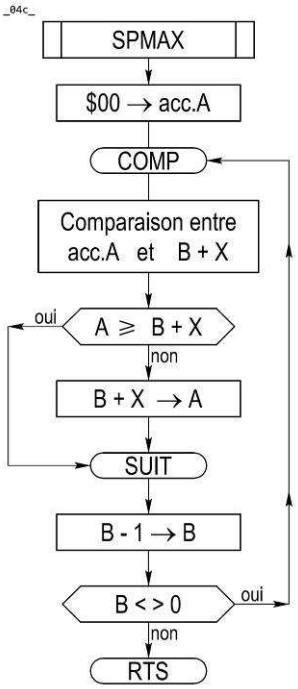
RTI est la dernière instruction d'un sous programme d'interruption, à sa rencontre le 6809 dépile d'abord le registre d'état CC, il examine la valeur du bit E pour connaître l'étendue du dépilement.

Equivalent au Return en Basic.

Voir aussi l'instruction JMP
Dès que le 6809 rencontre **RTS**, il va chercher l'adresse qui se trouve en haut de la pile système S, l'incrémente et la charge ensuite dans le compteur ordinal.

Le pointeur de pile est donc incrémenté de 2.

Exemple :



Programme faisant appel un sous-programme permettant de trouver le plus grand élément d'une table de valeurs.
Longueur du bloc à scruter ---> B
Adresse de début de bloc ---> X

Le résultat sera sauvegardé à l'adresse $0200

```
                        ;----Prog INS-02
0000                            ORG     $0000       ;
                  0100  LONG    EQU     $100        ;
                  5000  ADRES   EQU     $5000       ;
                                                    ;
0000 F6   0100                  LDB     LONG        ; charge la longueur de la table
0003 BE   5000                  LDX     ADRES       ; charge l'adresse de début
0006 9D   0D                    JSR     SPMAX       ; appel au S-Programme
0009 B7   0200                  STA     $0200       ; sauve le résultat
000C 3F                         SWI                 ;
                                                    ;
                        ;--------------------
000D 86   00            SPMAX   LDA     #$00        ; init de la valeur maxi
000F A1   85            COMP    CMPA    B,X         ; nouvelle valeur maxi ???
0011 2C   02      0015          BGE     SUIT        ; non, on recommence
0013 A6   85                    LDA     B,X         ; oui, charge ce nouveau maxi
0015 5A                 SUIT    DECB                ; bloc terminé ?
                                                    ; Décrémentation de B
0016 26   F7      000F          BNE     COMP        ; non, on continue
0018 39                         RTS                 ;
```

SBCA SBCB (SuBtraCt with Borrow ....)

Soustraction **avec** retenue.
Analogue aux instructions ADCA et ADCB.

```
{A} - bit C - {Mém} ---> {A}
{B} - bit C - {Mém} ---> {B}
```

{.....} = contenu de

```
                efhinzvc
Après l'instruction  CC = _____NZVC
```

_ = inchangé

SEX (Sign EXtended) "Extension de signe"

Transforme un nombre de 8 bits signé en complément à 2 en un nombre signé en complément à 2 de 16 bits.

En fait le nombre codé en complément à deux de 8 bits, stocké dans B est tout simplement transformé en nombre en complément à deux, sur 16 bits, pour être stocké dans D. L'ancienne valeur de A est perdue.

Porte uniquement sur A Si bit7 de B égal à **1** (cas d'un nombre négatif) Alors `{A}=$FF`
Si bit7 de B égal à **0** (cas d'un nombre positif) Alors `{A}=$00`

```
                efhinzvc
Après l'instruction  CC = _____NZ__
```

_ = inchangé

STA STB (STore) ranger

```
A ---> {Mém}      B ---> {Mém}
```

{.....} = contenu de

Pour ces instructions, pas d'adressage immédiat puisque STA et STB range le contenu de l'accumulateur A ou B dans une case mémoire d'adresse définie.

```
STB A,X     ; mode d'adressage Indexé Direct à déplacement accumulateur.
```

X est le registre d'index
A est l'accumulateur utilisé pour le déplacement

Avant l'instruction `STB A,X` si on a : `{B}=$32` `{A}=$80` `{X}=$2000` {.....} = contenu de
Après l'instruction la valeur $32 sera stockée à l'adresse $2080

```
                efhinzvc
Après l'instruction  CC = _____000_
```

_ = inchangé

STD STX STY STS STU (STore) ranger

```
Registre --->  {Mém, Mém+1}
```

{.....} = contenu de

L'octet de poids fort est chargé en premier Mém
L'octet de poids faible est chargé en second Mém+1

```
STD ,U++    ; adresse double incrémentation
```

Avant l'instruction `STD ,U++` si on a : `{U}=$53F2` `{D}=$1000`

Après l'instruction, étant donné que {D} = $10 00
$10 {octet de poids fort de D} est mis dans une case mémoire à l'adresse `$53F2`
$00 {octet de poids faible de D} est mis dans une case mémoire à l'adresse `$53F3`
{U} est incrémenté deux fois et passe à `$53F4`

```
                efhinzvc
Après l'instruction  CC = _____000_
```

_ = inchangé

SUBA SUBB (**SUB**tract) Soustraction

Soustraction **sans** retenue.
Analogue aux instructions ADDA et ADDB.

```
{A} - {Mém} ---> {A}
{B} - {Mém} ---> {B}
```

{.....} = contenu de

```
                    efhinzvc
Après l'instruction CC = ____NZVC
```

_ = inchangé

SUBD (**SUB**tract) Soustraction

```
{D} - {Mém, Mém+1} ---> {D}
```

{.....} = contenu de

```
                    efhinzvc
Après l'instruction CC = ____NZVC
```

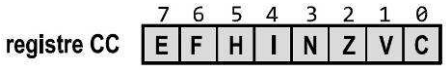
_ = inchangé

Instruction d'interruption logicielle

Avant la sauvegarde des registres internes le bit E de CC est positionné à 1, pour indiquer que l'état total du 6809 est sauvegardé dans la pile S.

L'interruption logiciel SWI est une instruction assembleur possédant un code opératoire **$3F**

Après la sauvegarde des registres internes, les bits I et F de CC sont positionné à 1 (ce qui signifie que les interruptions matérielles sont interdites).



Cette interruption logicielle est plus prioritaire que FIRQ| et IRQ| car son traitement entraîne le masquage de FIRQ| et IRQ|.

Généralement, SWI est utilisée dans un moniteur de mise au point de programme, pour faire des arrêts sur adresses.

Le PC est chargé avec le contenu des adresses :

{$FFFA} + {$FFFB} {.....} = contenu de

LSB (Least Signifiant Bit) poids Faible
MSB (Most Signifiant Bit) poids Fort

A la rencontre de cette instruction SWI le 6809 déclenche la série d'opération suivante :

1) Mise à 1 du bit E avertissant que l'ensemble des registres sont sauvegardés.

2) sauvegarde de l'ensemble des registres dans l'ordre suivant : PCL, PCH, UL, UH, YL, YH, XL, XH, DP, B, A, CC. Le pointeur S décroît de 12 cases mémoire à la fin de cette opération.

3) Interdiction d'IRQ (bit I=1) et FIRQ (bit F=1). La séquence d'exception SWI ne sera pas interrompue par les sources connectées à IRQ et FIRQ.

4) Chargement du contenu des adresses $FFFA et $FFFB dans le compteur programme. Cette séquence d'exception doit se terminer également par l'instruction RTI qui restitue le contexte initial.

SWI est une interruption logicielle conçue avec une priorité importante puisqu'elle interdit IRQ et FIRQ.

Pour cette raison, SWI trouve son utilité dans les utilitaires systèmes destinés à la mise au point des programmes utilisateurs.

Cela permet également de faire appel à un OS, à des routines systèmes.
L'intérêt est que l'utilisateur n'a pas à connaître l'adresse, c'est l'OS qui s'occupe de tout.

Par exemple, pour poser un point d'arrêt à une adresse donnée, le moniteur remplace le premier octet de l'instruction pour le code $3F.

A la rencontre d'un code, le sous-programme d'interruption SWI visualise les registres du 6809 et arrête momentanément exécution en entrant dans une phase d'attente.

Par la suite, si l'opérateur désire continuer, le moniteur établit l'op-code correct sauvegardé dans la zone système, positionne le prochain point d'arrêt, puis exécute le programme utilisateur à partir de l'ancien point d'arrêt.

Cette méthode implique que le programme sous test doit se trouver dans un espace mémoire lecture écriture.

SWI2 (**S**oft**W**are **I**nterrupt) Interruption logicielle

Instruction d'interruption logicielle
Réservée à l'usage du programme utilisateur.

Fonctionnement similaire à SWI, sauf que les masques d'interruptions bits I et F du registre CC ne sont pas affectés. Il conservent leur valeur au commencement du traitement d'exception, donc les sous-programmes correspondants demeurent interruptibles.

Ne masquent pas les interruptions IRQ et FIRQ

Autrement dit, SWI2 et SWI3 ont un fonctionnement identique à SWI, mais elles peuvent être interrompues par toutes autres interruptions du 6809 et sont par conséquent les moins prioritaires

Le PC est chargé avec le contenu des adresses :

{$FFF4} + {$FFF5} {…..} = contenu de
LSB (Least Signifiant Bit) poids Faible
MSB (Most Signifiant Bit) poids Fort

Les interruptions matérielles sont donc autorisées durant l'exécution du sous-programme d'interruption logicielle SWI2, elle a une priorité inférieure à SWI.

On dit aussi que SWI2 et SWI3 ont une priorité inférieure à celle de SWI.

SWI3 (**S**oft**W**are **I**nterrupt) Interruption logicielle

Instruction d'interruption logicielle
Réservée à l'usage du programme utilisateur.

Fonctionnement similaire à SWI, sauf que les masques d'interruptions bits I et F du registre CC ne sont pas affectés. Il conservent leur valeur au commencement du traitement d'exception, donc les sous-programmes correspondants demeurent interruptibles.

Ne masquent pas les interruptions IRQ et FIRQ

Autrement dit, SWI2 et SWI3 ont un fonctionnement identique à SWI, mais elles peuvent être interrompues par toutes autres interruptions du 6809 et sont par conséquent les moins prioritaires

Le PC est chargé avec le contenu des adresses :

{$FFF2} + {$FFF3} {…..} = contenu de
LSB (Least Signifiant Bit) poids Faible
MSB (Most Signifiant Bit) poids Fort

SWI, SWI2 et SWI3

Ces trois instructions SWI, SWI2 et SWI3 sont analogues à un appel de sous-programme.

Ces instructions stockent tous les registres dans la pile, sauf le pointeur de pile système et l'adresse de retour.

SYNC (**SYNC**hronize to external event) Attente d'une synchronisation externe

Le 6809 doit pouvoir se mettre en attente ou se synchroniser sur un évènement extérieur dont la présence est signalée par une ou des broches d'entrée d'interruptions (Voir instructions SYNC et CWAI).

L'instruction SYNC permet de synchroniser le 6809 sur un évènement extérieur.
Le 6809 se met en attente d'une interruption grâce aux broches d'interruption NMI|, IRQ| et FIRQ|.

Utile par exemple dans une application biprocesseur où les taches sont partagées.

Elle permet également de réaliser des synchronisations rapides avec les périphériques, cette méthode permet éventuellement d'éviter l'utilisation d'un circuit d'accès direct mémoire.

Dès la rencontre de cette instruction le 6809 s'arrête et attend qu'une interruption se produise.

Cette interruption peut être masquée par le biais des bits I ou F du registre CC.

Dès qu'une interruption apparaît le 6809 reprend son programme et exécute les instructions suivant l'instruction SYNC.

C'est une interruption matérielle programmée.

Si une interruption quelconque est inhibée par un masquage (sauf pour NMI), ou si le niveau Bas appliqué dure moins de trois cycles machine, le 6809 continue l'exécution normale de l'instruction suivant SYNC.

Si le niveau Bas appliqué se prolonge au-delà de trois cycles machine avec l'indicateur correspondant non masqué (égal à 0), alors le 6809 entame le traitement habituel de l'interruption sollicitée.

On remarque que :

-- Tous niveaux Bas appliqués sur l'une des entrées IRQ, FIRQ ou NMI, quelle que soit sa durée, quel que soit l'état des bits I et F, provoque un redémarrage du 6809 et le sort de l'état HALT.

-- Si une impulsion dure moins de trois cycles machine, son rôle essentiel est de redémarrer le 6809 sur un programme commençant immédiatement après l'instruction SYNC, et ceci très rapidement car il n'y a ni opération de sauvegarde, ni vectorisation.
A ce titre, on peut se servir de SYNC et d'une impulsion externe courte pour synchroniser le traitement du 6809 sur un événement externe, d'où nom de cette instruction.

-- Par comparaison avec l'instruction CWAI, l'instruction SYNC peut sortir le 6809 de son état latent sans orienter ce dernier vers une séquence interruption, alors que CWAI oblige le 6809 à se diriger vers un traitement d'exception.
SYNC ne contient pas d'opérande permettant d'agir sur les indicateurs du registre d'état comme CWAI.

-- L'instruction SYNC arrête le fonctionnement du 6809 et met les bus de données et d'adresses en haute impédance. Le 6809 reprend son fonctionnement lorsqu'un signal d'interruption est reçu.

TFR

Voir l'instruction EXG

TST TSTA TSTB

Permet de tester le contenu d'une case mémoire ou d'un accumulateur A ou B.
Après l'instruction la valeur de l'accumulateur **A** n'est pas modifiée.

Bit **N = 1** si la valeur est négative

Bit **Z = 1** si la valeur est nulle.

Bit **V** est mis à 0

Exemple : `TSTA ; avec {A} = $B8` {.....} = contenu de

Le bit Z=0 car $B8 différent de 0

Le bit N=1 car $B8 inférieur à 0 (positif de 0 à 127, négatif de 128 à 255)

```
          efhinzvc
Après l'instruction CC = ____NZ0_
```

= inchangé avec **V=0**

Qu'est ce qu'une interruption ?

C'est un moyen MATERIEL, un signal sur une broche du 6809 qui change d'état.
Ou une procédure LOGICIEL, une instruction placée dans un programme en cours d'exécution.

L'interruption LOGICIEL ou MATERIEL permet :
-- d'interrompre un programme en cours
-- de traiter prioritairement un programme par rapport à un autre.

Le microprocesseur scrute à chaque cycle, c'est-à-dire toutes les microsecondes, une éventuelle interruption, ce qui permet une réponse instantanée.
De ce fait le 6809 n'a pas besoin de scruter les périphériques, il se contente d'attendre qu'on l'avertisse.

Lorsque le 6809 détecte une interruption et si celle-ci est autorisée, le 6809 termine avant tout l'instruction qu'il était en train de d'exécuté. Dans tous les cas, le programme principal est interrompu.

Il interdit ensuite les autres interruptions moins prioritaires et empile le registre PC et CC et éventuellement d'autres registres. Il se branche alors à l'adresse de la routine d'interruption. Voir les vecteurs d'interruptions.

L'interruption étant terminée, le 6809 dépile les registres empilés et poursuit le programme qui à été interrompu.

Le 6809 doit être capable de répondre rapidement aux sollicitations de ces périphériques.
Ces demandes externes au 6809 sont vues comme des demandes d'interruption, le 6809 possède pour cela :
-- 3 Broches d'entrées d'interruption Matérielle NMI, IRQ et FIRQ et une séquence d'initialisation RESET
-- 3 Instructions d'interruption Logicielle SWI, SWI2, SWI3
-- 2 Instructions d'Attente d'Interruptions SYNC CWAI

Système d'interruption du 6809

Dans les systèmes à base de microprocesseur, on désire souvent intervenir sporadiquement dans le système, même lorsqu'il est en train d'exécuter une tâche quelconque (exemples : arrêt d'urgence pour prévenir d'un danger, arrêt d'une imprimante par manque de papier, arrêt d'un traitement pour traiter un autre programme plus prioritaire).

Un système permettant d'interrompre le déroulement d'un programme n'est réellement utile que si le contexte d'exécution peut être sauvegardé.

Le 6809 comporte un système de d'interruptions très complet permettant de solutionner la majorité des applications dites "à temps réel".

Par rapport au 6800, le 6809 possède en plus : une interruption matérielle rapide FIRQ, 2 interruptions logicielles SW2 et SW3, 2 instructions originales CWAI et SYNC.

Différentes catégories d'interruptions

Une interruption est une instruction (logiciel) ou une impulsion (matériel) provoquant un arrêt ordonné du programme en cours, avec sauvegarde partielle ou totale du contexte d'exécution dans la pile système et un branchement du 6809 vers une séquence spéciale appelée sous-programme d'interruption.

Le sous-programme d'interruption restitue le contexte d'exécution programme principal (à partir de la pile système) dès la rencontre de l'instruction RTI. L'instruction RTI ressemble un peu à RTS, à la différence près qu'il examine de l'Etat du E (bit7 du registre CC) pour connaître l'étendue du dépilage.

Pour faciliter la description du système d'interruption du 6809 nous distinguons :
-- Les arrêts manuels RESET, HALT
-- Les interruptions matérielles IRQ, FIRQ, NMI
-- Les interruptions logicielles SWI, SWI2, SWI3
-- Les instructions pour attendre une interruption CWAI, SYNC

Une interruption matérielle est provoquée par l'apparition d'une impulsion ou d'un front actif sur l'une des broches d'interruption du 6809, broches IRQ, FIRQ, NMI.

Mise à part NMI qui ne peut être "masqué", les deux autres IRQ et FIRQ sont masquables par des instructions logiques appropriées.

Au commencement d'un traitement le programmeur spécifie explicitement s'il autorise la prise en compte par le 6809 des interruptions IRQ ou FIRQ. Si elles sont autorisées elles pourront par la suite apparaître à n'importe quel endroit du programme utilisateur. Le 6809 ne tient compte de leur présence qu'après l'exécution complète de l'instruction en cours.

Avant de passer le contrôle à la séquence d'exécution, le 6809 effectue une sauvegarde :
-- Totale pour IRQ et NMI
-- Partielle pour FIRQ

Pour les interruptions logicielles SWI, SWI2, SWI3, elles doivent être insérées dans le programme utilisateur sous forme d'instructions régulières.

Le fonctionnement d'une interruption logicielle s'identifie à celui d'une interruption matérielle.
La différence réside dans le fait qu'une interruption logicielle doit être programmée à l'avance, qu'elle est placée dans des endroits bien déterminés, d'où disparition de l'aspect aléatoire rencontré dans une interruption matérielle.

Le fonctionnement des instructions CWAI et SYNC est assez particulier. Elles font intervenir simultanément les deux aspects matériels et logiciels.

Le 6809 comporte aussi deux interruptions matérielles d'un genre particulier :

RESET qui effectue une initialisation générale du système.

HALT qui suspend toute activité du 6809 sans perte de contexte et sans traitement de séquence d'exception.

Remarques sur la programmation des interruptions

Remarque 01

Les interruptions ne sont pas très faciles à tester car elles impliquent un minimum de matériel pour produire des niveaux Bas, des impulsions à largeur réglable (SYNC), etc...

D'autre part étant donné le caractère aléatoire des interruptions matérielles, dans certains cas, se posent des problèmes de localisation.

Remarque 02

Pour les interfaces, les interruptions matérielles ne conduisent pas forcément à une vitesse de transfert maximum étant donné le nombre de registres empilés et dépilés à chaque activation de IRQ.

Certes, l'introduction de FIRQ apporte une amélioration considérable. Le nombre minimal de registres à empiler est laissé à l'appréciation du programmeur. Il peut mettre PSHS au début du traitement d'exception et PULS avant l'exécution de RTI.

Voici quelques indications sur le nombre de cycles machines nécessaires à la prise en compte de l'interruption.

21 cycles	Réponse à NMI à IRQ	6 cycles	Exécution de RTI avec bit E=0
12 cycles	Réponse à FIRQ	15 cycles	Exécution de RTI avec bit E=1
19 cycles	Réponse à SWI	20 cycles	Réponse à SWI2, SWI3
20 cycles	Réponse à CWAI		
9 cycles	Réponse à n'importe quelle interruption après CWAI		
1 cycles	Exécution de SYNC si l'interruption est masquée		
2 cycles	Exécution de SYNC		

Remarque 03

Lorsque plusieurs sources d'interruptions sont câblées à une même entrée IRQ ou FIRQ, l'identification du logicielle (polling) de la source émettrice de la demande peut présenter quelques inconvénients sur la vitesse de réponse.

D'autre part les adresses des registres des interfaces sont rarement contiguës dans l'espace mémoire du processeur si bien qu'on éprouve parfois quelques difficultés pour employer les modes d'indexés.

La vectorisation des interruptions à l'aide des circuits spécialisés peut apporter une solution élégante.

Remarque 04

Dans les séquences d'exception, le 6809 permet de manipuler des données de la pile système facilement à l'aide du mode indexé **nn,S**. les offsets sont les suivantes :

```
PCL 11,S        PCH 10,S        UL 9,S          UH 8,S
YL 7,S          YH 6,S          XL 5,S          XH 4,S
DP 3,S          B 2,S           A 1,S           CC 0,S
```

Position du pointeur S à l'entrée du sous-programme d'interruption.

Si les instructions PSHS sont nécessaires dans le sous-programme d'interruption, il faut apporter les corrections nécessaires à ces valeurs d'offset.

Par exemple, si l'on veut interdire les interruptions dans le programme principal après le traitement d'une séquence d'exception, on peut écrire :

```
    LDA     0,S             ;
    ORA     #%01010000      ; interdire IRQ et FIRQ
    STA     0,S             ;
```

Après dépilement, les masques (bit I et bit F du registre de contrôle) se retrouvent avec la valeur 1, ce qui correspond résultat souhaité. Il est également possible de modifier l'adresse de retour.

Remarque 05

Pour le 6809, les interruptions sont inhibées quand les bits **F** et **I** du registre d'état sont mis à 1.

Pour les interfaces PIA 6821 et ACIA 6850, c'est exactement le contraire, les bits correspondants des registres de contrôle doivent être mis à 0.

Au RESET général, les modes par interruption du PIA 6821 sont inhibées. Pour l'ACIA 6850, l'initialisation générale qui consiste à charger %00000011 (Master reset) dans le registre le contrôle de l'ACIA 6850 n'affecte pas le bit de contrôle d'interruption CR7.

Remplissage d'un buffer de commande par une interruption d'une ligne série

Hypothèse d'un terminal écran-clavier connecté au système à travers une ligne série contenant un **ACIA 6850**.

La sortie IRQ du **6850** est connectée à l'entrée IRQ du processeur et le **6850** est configuré en mode réception par interruption (bit CR7 du registre de contrôle égal à 1).

Chaque fois qu'un caractère est reçu, il est renvoyé à l'écran par la même interface, selon le mode habituel de test de bit SR1 du registre d'état. Rappelons que se bit passe à 1 si le registre de transmission de le **6850** est disponible.

Le programme débute à l'adresse $8000. Les caractères reçus à partir du clavier seront rangés dans un buffer placé à l'adresse $8100. La fin du programme se termine par la détection du code $0D correspondant à un retour chariot.

Le premier octet du buffer contient l'état de remplissage :
-- 0 si le buffer est vide
-- 1 si le buffer est plein

Le deuxième octet contient le nombre total de caractères reçus sans compter le caractère "retour chariot".
A partir du troisième octet, se rangent les données proprement dites.

Exemple : si on rentre au clavier MERGE puis "retour chariot" :

```
;     prog FEI-01
; ($8100) = $01   indicateur "buffer non vide"
; ($8101) = $05   Cinq caractères reçus
```

```
                        ; ($8102) = $4D   'M
                        ; ($8103) = $45   'E
                        ; ($8104) = $52   'R
                        ; ($8105) = $47   'G
                        ; ($8106) = $45   'E
                        ; ($8107) = $0D   'Retour chariot
                                          ;
                        ;*****************************************************
                        ;   Remplissage d'un buffer de commande par
                        ;   interruption sur ligne série
                        ;*****************************************************
8000                          ORG    $8000       ;
                        ;------adresses des registres gérant le terminal clavier
                  EC80  RCRTER EQU    $EC80       ;  reg. contrôle ACIA
                  EC80  RSRTER EQU    $EC80       ;  reg. Etat ACIA
                  EC81  RRXTER EQU    $EC81       ;  reg. Réception Clavier
                  EC81  RTXTER EQU    $EC81       ;  reg. Transmission écran
                                          ;
                        ;------partie exécutée à l'initialisation générale
8000 10CE 8200                LDS    #$8200      ;  adrs pile système
8004 86   03                  LDA    #%00000011  ;  reset ACIA
8006 B7   EC80                STA    RCRTER      ;
8009 86   91                  LDA    #%10010001  ;  interruption réception + 8 bits
                                                 ;  + 2 stop + clock 1/16
800B B7   EC80                STA    RCRTER      ;
                                                 ;==================================;
800E 8E   8080                LDX    #$8080      ;  adrs S/P interruption         ;
8011 BF   FFF8                STX    $FFF8       ;  adrs vecteur IRQ  FFF8 et FFF9  ;
8014 8E   8010                LDX    #$8010      ;  adrs buffer                   ;
8017 8D   01      801A        BSR    BUFFER      ;  remplissage buffer            ;
8019 3F                       SWI                ;                                ;
                                                 ;==================================;
                                          ;
                        ;*****************************************************
                        ; S/P remplissage Buffer
                        ; Caractère final "RC"    nom d'appel : BUFFER
                        ; Entrée :     X: adrs point départ buffer
                        ; Sortie :     Aucun registre affecté
                        ; (0,X)=0  buffer vide
                        ; (0,X)=1  buffer plein
                        ; (1,X):   nombre de caractères du buffer sans "RC"
                        ; à partir de (2,X): Contenu du buffer
                        ; Encombrement pile système :  2+5+12+1 = 20 octets
                        ;*****************************************************
801A 34   17            BUFFER PSHS   X,B,A,CC    ;
801C 6F   84                   CLR    0,X         ;  buffer vide initialement
801E 6F   01                   CLR    1,X         ;  Nb caractères
8020 C6   02                   LDB    #2          ;  offset premier caractère
8022 1C   EF                   ANDCC  #%11101111  ;  interruption CPU autorisée
8024 6D   84            CARSVT TST    0,X         ;  fin procédure remplissage ???
8026 27   FC      8024         BEQ    CARSVT      ;  sinon caractère suivant
8028 35   97                  PULS   PC,X,B,A,CC ;  fin S/P BUFFER
                                          ;
                        ;*****************************************************
                        ; Début S/P d'interruption
                        ; l'adrs $8080 connue doit être chargée au préalable
                        ; dans ($FFF8),($FFF9) au début de l'appel du S/P BUFFER
                        ; Registres échangés: B (voir explication à la fin du prog
                        ;*****************************************************
8080                          ORG    $8080       ;
8080 B6   EC81                LDA    RRXTER      ;  lire reg. réception
                                                 ;  mise à 0 bit interruption SR7
8083 A7   85                  STA    B,X         ;  rangement donnée
8085 81   0D                  CMPA   #$0D        ;  caractère "RC" return ???
8087 26   07      8090        BNE    VISU        ;  sinon visualiser
8089 6C   84                  INC    0,X         ;  mettre indicateur Buffer
                                                 ;  plein. Fin remplissage
808B C0   02                  SUBB   #2          ;  obtenir Nb caractère
808D E7   01                  STB    1,X         ;  stockage
808F 3B                       RTI                ;  fin S/P interruption
                                          ;
                        ;*****************************************************
8090 34   02            VISU   PSHS   A           ;
8092 B6   EC80                 LDA    RSRTER      ;  reg. Etat terminal
```

```
8095 85   02                BITA   #%00000010  ;  reg. transmission vide ???
8097 27   F9      8092       BEQ    VISU+2      ;
8099 35   02                PULS   A           ;
809B B7   EC81               STA    RTXTER      ;  transmettre sur écran
809E 6C   62                 INC    2,S         ;  avancer valeur de B dans pile
                                                ;  système pour prochaine entrée
                                                ;  dans le S/P interrup. (INCB
                                                ;  serait une erreur)
80A0 3B                      RTI                ;  fin S/P interruption
```

Quelques explications sur le programme ci-dessus

L'**ACIA 6850** est configuré avec CR7=1, donc la réception fonctionne en interruption.

Chaque fois que le registre de réception est plein, le bit SR0 est mis à 1, de même que le bit SR7.
La ligne IRQ subit une transition négative et le 6809 reconnaît une interruption.

Le traitement d'exception débute à l'adresse $8080 par une lecture simple du registre de réception.

Cette lecture provoque une mise à 0 des bits SR0 et SR7 et rétablit un niveau Haut sur la ligne IRQ.

On remarquera qu'ici, le programme ne teste plus l'état du bit de réception SR0.

Le sous-programme BUFFER fonctionne par interruption de type IRQ.

A l'appel de ce sous programme, il est nécessaire de sauvegarder le contenu de $FFF8 et $FFF9 dans un endroit quelconque, puis charger un autre vecteur IRQ avant l'appel de BUFFER.

En fin d'exécution du sous-programme, on restituera l'ancienne valeur du vecteur IRQ avant de continuer. Ici, nous avons enlevé cette opération de sauvegarde.

Le programmeur pourra la rétablir en écrivant par exemple :
A partir de l'adresse $800E

```
                                                ;==============Modif================;
800E BE   FFF8               LDX    $FFF8       ; sauvegarde vecteur IRQ courant    ;
8011 34   10                 PSHS   X           ;              dans la pile système ;
8013 8E   8080               LDX    #$8080      ; nouveau vecteur IRQ pour BUFFER   ;
8016 BF   FFF8               STX    $FFF8       ; mise en place du nouveau vecteur  ;
8019 8E   8100               LDX    #$8100      ; index du buffer de caractères     ;
801C 8D   06      8024       BSR    BUFFER      ; appel du sous-programme BUFFER    ;
801E 35   10                 PULS   X           ; rétablir ancien vecteur IRQ       ;
8020 BF   FFF8               STX    $FFF8       ;                                   ;
8023 3F                      SWI                ;                                   ;
                                                ;===================================;
                                                ;
8024 34   17           BUFFER PSHS   X,B,A,CC    ;
```

La mémoire (0,X) dans le sous-programme BUFFER sert de paramètres de communication entre BUFFER et son sous-programme d'interruption.

La boucle CARSVT se referme jusqu'à ce que (0,X) soit mis à 1 par la séquence d'exception.
(0,X) est mis à 1 quand le caractère reçu est un retour chariot $0D.

Il faut remarquer que l'interruption peut se produire différemment après TST 0,X ou après BEQ CARSVT. Le déclenchement reste aléatoire.

Une autre difficulté est rencontrée pour l'incrémentation de l'offset contenu dans B.
A la 1ière entrée dans la boucle CARSVT, {B} = 2. {.....} = contenu de

Après interruption, le contenu de B est poussé dans la pile.

Si l'on incrémente B simplement par INCB, après dépilement par RTI, l'ancienne valeur de B, en l'occurrence 2 sera rétabli dans B et l'offset B restera en réalité inchangé.

Pour faire évoluer ce paramètre, il est nécessaire d'actualiser sa valeur par INC 2,S dans la pile S avant l'opération de dépilement RTI.

Ce type d'erreur est difficile à prévoir et à tester. C'est le désavantage des interruptions.

Donc, ne pas employer les registres pour transférer les paramètres lorsqu'il s'agit de sous programme d'interruption, car les registres sont empilés et dépilés automatiquement. Ce qui n'est pas le cas pour les appels BSR, LBSR, JSR, RTS qui n'invoquent que le compteur programme PC.

Si l'on touche aux registres dans la pile système S dans les séquences exception, il y a effectivement transfert de paramètres entre les deux programmes. Or, l'interruption matérielle peut se produire de façon aléatoire, ce qui rend la conception des sous-programmes d'interruption difficile.

Pour s'affranchir de ce problème, il faut s'abstenir d'employer les mêmes registres dans les deux programmes.

Ici, B est employé par la séquence d'exception; ce registre n'est pas employé par la boucle d'attente dans le programme principal. Cette caractéristique est notifiée explicitement dans la partie commentaire.

Dans des cas spécifiques où tous les registres doivent être mobilisés par les deux programmes, le dernier recours consiste à autoriser l'interruption dans des endroits bien précis du programme principal en manipulant adroitement les instructions de masquage ANDCC et ORCC.

Prog. D'une touche d'arrêt temporaire avec visu des registres 6809 par interruption IRQ.

Etude simultanée d'une interruption par SWI

La mise au point des programmes nécessite des outils logiciels permettant de visualiser l'exécution d'un programme étape par étape.

Cette application ci-dessous, montre comment confectionner un programme traitant une interruption logiciel SWI et comment concevoir un arrêt momentané d'un programme avec la possibilité d'exécution ultérieure sans perte de contexte.

Dans les deux cas, l'ensemble des registres du 6809 est visualisé sur terminal écran-clavier connecté au système étudié à travers une liaison série gérée par un ACIA 6850.

Nous supposons que le système utilisateur possède un moniteur résident dont l'adresse MONIT est connue par le programmeur.

A la rencontre d'une instruction SWI, le 6809 entame une séquence d'interruption en sauvegardant l'ensemble des registres dans la pile système S.

Le sous programme de visualisation des registres appelé VISURG doit chercher les valeurs instantanées du programme utilisateur dans la pile système par le mode de indexé nn,S.

Les valeurs binaires sont ensuite converties en caractères ASCII visualisables.

Ces caractères sont rangés ensuite dans une zone mémoire appelée mémoire bloc-notes dont l'adresse est repérée par l'étiquette assembleur BLNOTE.

Au cours du remplissage, le buffer ASCII subit également une opération de formatage qui consiste à glisser les "espaces" entre les nombres, les sauts et retour de ligne pour améliorer la lisibilité.

Ce buffer se termine par un caractère de contrôle ETX de code ASCII $04.

Un autre sous-programme appelé VISU transmet l'ensemble du buffer vers l'écran de visualisation à la fin du module VISURG. Cette organisation en modes indépendants permet un emploi ultérieur de VISURG dans le mode ARRET TEMPORAIRE.

Après la phase de visualisation des registres, le contrôle du 6809 est transféré moniteur par un branchement inconditionnel `JMP MONIT`.

L'ARRET TEMPORAIRE de l'exécution du programme utilisateur est réalisé par une pression sur la touche "H" du clavier.

Les registres instantanés du 6809 seront visualisés sur l'écran.

Si l'opération désire reprendre l'exécution à l'endroit où s'est produite l'interruption, il suffit d'appuyer sur la touche "C" du clavier.

Au besoin, l'utilisateur peut affecter d'autres codes de contrôle pour réaliser ces deux fonctions.

Si l'opérateur appuie sur des touches autre que "H" et "C", le programme ignore ces touches, car les sous-programmes d'exception savent distinguer les touches et imposent l'ordre "H", "C", "H", "C", etc...

Les touches "H" et "C" interrompent le 6809 par la ligne IRQ.

Les sous-programmes d'exception correspondants sont répartis volontairement ici en deux niveaux, montrant comment on peut interrompre une séquence d'exception IRQ par une autre interruption IRQ et organiser un retour donné sans perte de contexte.

Dans cette application, le programme principal est parfaitement factice. Il sert simplement à faire varier continuellement les registres pour contrôler la bonne marche des sous-programmes d'interruption, c'est une boucle sans fin.

Pour sortir de cette boucle, il faut effectuer un RESET général ou employer une interruption NMI à condition que cette dernière existe sur le système étudié sous forme de bouton "ABORT" ou tout autre.

Pour tester l'interruption logicielle SWI, il faut enlever la boucle sans fin en effectuant une retouche mineure du code objet. Cette retouche est indiquée dans les commentaires.

Ce programme apparemment complexe est modulaire. Il est conseillé d'isoler en premier lieu des différents sous-programmes :

-- VISURG Visualisation des registres qui appelle BINHEX et VISU.
-- BINHEX Conversion d'un octet binaire en deux caractères ASCII
-- VISU Transmission du buffer ASCII sur écran
-- SPINT1 Sous-programme d'interruption par IRQ du 1er niveau
-- SPINT2 Sous-programme d'interruption par IRQ du 2ième niveau
-- SPSWI Sous-programme d'interruption par SWI

L'usage des couleurs peut améliorer la lisibilité.
Isolé également la boucle sans fin simulant le programme principal qu'on veut interrompre.

```
                        ;    prog FEI-03
                        ;************************************************************
                        ;   Arrêt Temporaire avec Visualisation des registres
                        ;   du 6809 à l'écran. Interruption logicielle par SWI
                        ;   Interruption Matérielle par IRQ      ouvrage 06 pVII.18
                        ;************************************************************
8000                          ORG    $8000          ;
                        ;------Adrs Registre gérant le terminal écran-clavier
                  EC80  RSCR1  EQU    $EC80          ; Reg. contrôle ACIA
                  EC80  RSET1  EQU    $EC80          ; Reg. Etat ACIA
                  EC81  RSRD1  EQU    $EC81          ; Reg. Réception Clavier
                  EC81  RSTD1  EQU    $EC81          ; Reg. Transmission Ecran
                  F000  MONIT  EQU    $F000          ; Adrs Moniteur
                  8040  SPSWI  EQU    $8040          ; Adrs S/P SWI
                  8080  SPINT1 EQU    $8080          ; Adrs 1er niveau d'interrup par IRQ
                  8050  SPINT2 EQU    $8050          ; Adrs 2èm niveau d'interrup par IRQ
                  8300  BLNOTE EQU    $8300          ; Adrs mémoire bloc-notes
                                                      ;
                        ;------Partie exécutée à l'initialisation générale
8000 10CE 8400                LDS    #$8400         ; Adrs de la pile système
8004 86   03                  LDA    #%00000011     ; Master Rest de l'ACIA
8006 B7   EC80                STA    RSCR1          ;
8009 86   91                  LDA    #%10010001     ; interruption réception + 8 bits
                                                    ; + 2 stops + clock 1/16
800B B7   EC80                STA    RSCR1          ;
800E 8E   8080                LDX    #SPINT1        ; adrs S/P IRQ
8011 BF   FFF8                STX    $FFF8          ;
8014 8E   8040                LDX    #SPSWI         ; adrs S/P SWI
8017 BF   FFFA                STX    $FFFA          ;
                                                    ;
                        ;************************************************************
                        ;   la séquence suivante s'exécute indéfiniment.
                        ;   Appuyer de temps à autre sur la touche H pour
                        ;   arrêter le déroulement du programme et de visualiser
                        ;   les registres du 6809 à l'écran
                        ;   Pour continuer, appuyer sur la touche C
                        ;************************************************************
801A 4F                       CLRA                  ;
801B 5F                       CLRB                  ;
801C 8E   0000                LDX    #0             ;
```

```
801F 1F   12                TFR    X,Y             ;
8021 1F   13                TFR    X,U             ;
8023 1C   EF                ANDCC  #%11101111      ; autoriser IRQ
8025 4C                TSTINT INCA                 ;
8026 5C                       INCB                 ;
8027 30   01                LEAX   1,X             ;
8029 31   21                LEAY   1,Y             ;
802B 33   41                LEAU   1,U             ;
802D 20   F6      8025      BRA    TSTINT          ; enlever pour tester SWI
802F 3F                     SWI                    ;
                                                   ;
                       ;************************************************************
                       ;   S/P interruption par SWI. Pour tester cette séquence,
                       ;   remplacer l'op-code de BRA  TSTINT par NOP NOP $12 $12
                       ;************************************************************
8040                          ORG    SPSWI           ; implantation S/P SWI
8040 17   00BD    8100        LBSR   VISURG          ; VISUaliser les ReGistres
8043 1C   AF                  ANDCC  #%10101111      ; Autoriser IRQ et FIRQ
8045 32   6C                  LEAS   12,S            ; rétablir position pointeur
8047 7E   F000                JMP    MONIT           ; retour au moniteur
                                                   ;
                       ;------S/P interruption IRQ 1er niveau
8080                          ORG    SPINT1          ; adrs d'implantation
8080 B6   EC81                LDA    RSRD1           ; lire caractère reçu
8083 81   48                  CMPA   #'H             ; mode "arrêt temporaire"
8085 27   01      8088        BEQ    ARRET           ; si oui vers ARRET
8087 3B                       RTI                    ; sinon retour au programme départ
8088 8D   76      8100  ARRET BSR    VISURG          ; visualiser registres
                                                   ;
                       ;------définition d'un 2ième niveau d'interruption
808A 8E   8050                LDX    #SPINT2         ; adrs 2ième niveau IRQ
808D BF   FFF8                STX    $FFF8           ; Charger vecteur IRQ
8090 3C   EF                  CWAI   #%11101111      ; enlever le masquage sur IRQ et
                                                   ; attendre arrivée autre caractère
                       ;------A cet endroit, 6809 attend une interruption type IRQ
                       ;      Le S/P d'interruption du 2ième niveau vérifie s'il
                       ;      s'agissait bien d'un caractère C provenant de
                       ;      l'ACIA1, puis retourne le contrôle au 1er niveau
                       ;------(voir S/P interruption 2ième niveau)
8092 8E   8080                LDX    #SPINT1         ; rétablir le vecteur IRQ niveau 1
8095 BF   FFF8                STX    $FFF8           ;
8098 3B                       RTI                    ; retour au prog principal
                                                   ;
                       ;------S/P interruption IRQ 2ième niveau
8050                          ORG    SPINT2          ; adrs d'implantation
8050 7D   EC80                TST    RSET1           ; examiner bit SR7 de ACIA 1
8053 2A   0E      8063        BPL    ATTEN           ; si non ACIA1 attendre
8055 B6   EC81                LDA    RSRD1           ; lecture registre réception
8058 81   43                  CMPA   #'C             ; mode "Continuer" ???
805A 26   07      8063        BNE    ATTEN           ; sinon attendre autre caractère
805C 8E   838A                LDX    #BLNOTE+128+10  ; index sur chaîne EXECUTION
805F 17   01AA    820C        LBSR   VISU            ; visualiser
8062 3B                       RTI                    ; caractère C identifié
                                                   ; continuer S/P 1er niveau
                       ;------dans l'un des 2 cas suivans : IRQ non issue de
                       ;      l'ACIA 1 ou caractère différent de C, revenir
                       ;------au niveau 1 mais sur l'instruction CWAI
8063 AE   6A            ATTEN LDX    10,S            ; adrs retour dans pile S
8065 30   1E                  LEAX   -2,X            ; 2 octets en avant
8067 AF   6A                  STX    10,S            ; mettre dans la pile avant
                                                   ; dépilement par RTI
8069 3B                       RTI                    ;
                                                   ;
                       ;************************************************************
                       ;    S/P Visualisation des registres du 6809
                       ;    nom d'appel : VISURG
                       ;    Entrée: sans paramètre
                       ;    Sortie: aucun registre affecté
                       ;    sous programme appelés :  BINHEX,  VISU
                       ;    encombrement pile système : 9+7 = 16 octets
                       ;************************************************************
8100                          ORG    $8100           ;
8100 34   37            VISURG PSHS   Y,X,B,A,CC     ;
                       ;------A cause de cette opération de sauvegarde le pointeur S
```

```
                        ;      se déplace de 9 octets vers le bas. Ajouter +9 pour
                        ;      retrouver les offsets des registres sauvegardés à
                        ;------l'interruption
8102 8E   8300          	LDX    #BLNOTE        ; Adrs mémoire bloc-notes
8105 CC   5043          	LDD    #$5043         ; car. PC $50="P" et $43="C"
8108 ED   81            	STD    ,X++           ;
810A 86   3A            	LDA    #$3A           ; car. $3A=":"
810C A7   80            	STA    ,X+            ;
810E A6   E8 13         	LDA    19,S           ; mot poids fort PC
8111 17   00DF    81F3  	LBSR   BINHEX         ; conversion
8114 ED   81            	STD    ,X++           ;
8116 A6   E8 14         	LDA    20,S           ; mot poids faible PC
8119 17   00D7    81F3  	LBSR   BINHEX         ;
811C ED   81            	STD    ,X++           ;
811E CC   2020          	LDD    #$2020         ; deux espaces
8121 ED   81            	STD    ,X++           ;
8123 CC   533A          	LDD    #$533A         ; S:
8126 ED   81            	STD    ,X++           ;
8128 1F   40            	TFR    S,D            ; position courante pointeur S
812A C3   0015          	ADDD   #21            ; 9 pour compenser VISURG
                                                  ; + 12 pour compenser interruption
812D 1F   02            	TFR    D,Y            ; sauvegarde provisoire
812F 17   00C1    81F3  	LBSR   BINHEX         ; conversion poids fort S
8132 ED   81            	STD    ,X++           ;
8134 1F   20            	TFR    Y,D            ; obtenir poids faible S
8136 1E   89            	EXG    A,B            ;
8138 17   00B8    81F3  	LBSR   BINHEX         ;
813B ED   81            	STD    ,X++           ;
813D CC   2020          	LDD    #$2020         ; deux espaces
8140 ED   81            	STD    ,X++           ;
8142 CC   553A          	LDD    #$553A         ; U:
8145 ED   81            	STD    ,X++           ;
8147 A6   E8 11         	LDA    17,S           ; mot poids fort U
814A 17   00A6    81F3  	LBSR   BINHEX         ;
814D ED   81            	STD    ,X++           ;
814F A6   E8 12         	LDA    18,S           ; mot poids faible U
8152 17   009E    81F3  	LBSR   BINHEX         ;
8155 ED   81            	STD    ,X++           ;
8157 CC   2020          	LDD    #$2020         ; deux espaces
815A ED   81            	STD    ,X++           ;
815C CC   593A          	LDD    #$593A         ; Y:
815F ED   81            	STD    ,X++           ;
8161 A6   6F            	LDA    15,S           ; mot poids fort Y
8163 17   008D    81F3  	LBSR   BINHEX         ;
8166 ED   81            	STD    ,X++           ;
8168 A6   E8 10         	LDA    16,S           ; mot poids faible Y
816B 17   0085    81F3  	LBSR   BINHEX         ;
816E ED   81            	STD    ,X++           ;
8170 CC   2020          	LDD    #$2020         ; deux espaces
8173 ED   81            	STD    ,X++           ;
8175 CC   583A          	LDD    #$583A         ; X:
8178 ED   81            	STD    ,X++           ;
817A A6   0D            	LDA    13,X           ; mot poids fort X
817C 8D   75      81F3  	BSR    BINHEX         ;
817E ED   81            	STD    ,X++           ;
8180 A6   6E            	LDA    14,S           ; mot poids faible X
8182 8D   6F      81F3  	BSR    BINHEX         ;
8184 ED   81            	STD    ,X++           ;
8186 86   0D            	LDA    #$D            ; retour chariot
8188 A7   80            	STA    ,X+            ;
818A 86   0A            	LDA    #$A            ; saut de ligne
818C A7   80            	STA    ,X+            ;
818E CC   0450          	LDD    #$450          ; lettre  DP
8191 ED   81            	STD    ,X++           ;
8193 86   3A            	LDA    #$3A           ; signe ":"
8195 A7   80            	STA    ,X+            ;
8197 A6   6C            	LDA    12,S           ; registre DP
8199 8D   58      81F3  	BSR    BINHEX         ;
819B ED   81            	STD    ,X++           ;
819D CC   2020          	LDD    #$2020         ; deux espaces
81A0 ED   81            	STD    ,X++           ;
81A2 CC   423A          	LDD    #$423A         ; B:
81A5 ED   81            	STD    ,X++           ;
81A7 A6   6B            	LDA    11,S           ; registre B
```

```
81A9 8D   48      81F3          BSR    BINHEX          ;
81AB ED   81                    STD    ,X++            ;
81AD CC   2020                  LDD    #$2020          ; deux espaces
81B0 ED   81                    STD    ,X++            ;
81B2 CC   413A                  LDD    #$413A          ; A:
81B5 ED   81                    STD    ,X++            ;
81B7 A6   6A                    LDA    10,S            ; registre A
81B9 8D   38      81F3          BSR    BINHEX          ;
81BB ED   81                    STD    ,X++            ;
81BD CC   2020                  LDD    #$2020          ; deux espaces
81C0 ED   81                    STD    ,X++            ;
81C2 CC   4343                  LDD    #$4343          ; lettre  CC
81C5 ED   81                    STD    ,X++            ;
81C7 86   3A                    LDA    #$3A            ; signe ":"
81C9 A7   80                    STA    ,X+             ;
81CB A6   69                    LDA    9,S             ; registre CC
81CD 108E 0008                  LDY    #8              ; Nb d'opération
81D1 C6   30                    LDB    #$30            ; 0 ASCII
81D3 E7   80            BINBIN  STB    ,X+             ; initialiser à $30
81D5 48                         LSLA                   ; bit nul ???
81D6 24   02      81DA          BCC    *+4             ; si oui, bit suivant
81D8 6C   1F                    INC    -1,X            ; sinon, mettre $31
81DA 31   3F                    LEAY   -1,Y            ; Cpt. Opération - 1
81DC 26   F5      81D3          BNE    BINBIN          ;
81DE CC   0D0A                  LDD    #$0D0A          ; retour et saut de ligne
81E1 ED   81                    STD    ,X++            ;
81E3 86   04                    LDA    #$04            ; caractère ETX
81E5 A7   80                    STA    ,X+             ;
                  81E7  ADRCR   EQU    *               ; mémoriser adrs courante  <--[RS]
                                                       ;                   avant c'était  SET  *
                                                       ;
                        ;------Stockage des chaînes de caractères à l'assemblage
8380                            ORG    BLNOTE+128      ; 128 octets au-delà du bolc-notes
8380 0D 0A                      FDB    $0D0A           ; retour chariot saut ligne CRLF
8382 50 41 55 53                FCC    /PAUSE/         ; Chaîne à écrire
8386 45                                                ;
8387 20 20                      FDB    $2020           ; deux espaces
8389 04                         FCB    $04             ; caractère ETX
838A 0D 0A                      FDB    $0D0A           ; retour chariot saut ligne CRLF
838C 45 58 45 43                FCC    /EXECUTION/     ;
8390 55 54 49 4F                                       ;
8394 4E                                                ;
8395 0D 0A                      FDB    $0D0A           ; retour chariot saut ligne CRLF
8397 04                         FCB    $04             ; caractère ETX
                                                       ;
81E7                            ORG    ADRCR           ; Suite de VISURG
                                                       ;
                        ;------visualisation de la chaîne PAUSE
81E7 8E   8380                  LDX    #BLNOTE+128     ; adresse chaîne à transmettre
81EA 8D   20      820C          BSR    VISU            ;
                                                       ;
                        ;------Visualiser les caractères contenus dans le
                        ;------bloc-note jusqu'à la rencontre de ETX
81EC 8E   8300                  LDX    #BLNOTE         ; début du bloc-notes
81EF 8D   1B      820C          BSR    VISU            ; visualiser
81F1 35   B7                    PULS   PC,Y,X,B,A,CC   ; fin S/P VISURG
                                                       ;
                        ;*************************************************************
                        ;        S/P Convertion d'un octet binaire en 2
                        ;        caractères ASCII hexadécimaux
                        ;   Nom: BINHEX
                        ;   Entrée: A: donnée à convertir
                        ;   Sortie: A: code ASCII 4 bits poids fort
                        ;           B: code ASCII 4 bits poids faible
                        ;   Exemple: $3A sera converti en $33="3" $41="4"
                        ;   Encombrement pile système: 2 octets
                        ;*************************************************************
81F3 1F   89            BINHEX  TFR    A,B             ;
81F5 44                         LSRA                   ; obtenir 4 bits poids fort
81F6 44                         LSRA                   ;
81F7 44                         LSRA                   ;
81F8 44                         LSRA                   ;
81F9 81   0A                    CMPA   #$A             ; < 10 ???
81FB 25   02      81FF          BLO    *+4             ;
```

```
81FD 8B   07                ADDA   #7             ;
81FF 8B   30                ADDA   #$30           ;
8201 C4   0F                ANDB   #%00001111     ; 4 bits poids faible
8203 C1   0A                CMPB   #$A            ; < 10 ???
8205 25   02      8209      BLO    *+4            ;
8207 CB   07                ADDB   #7             ;
8209 CB   30                ADDB   #$30           ;
820B 39                     RTS                   ; Fin S/P BINHEX
                                                  ;
                      ;*******************************************************
                      ;   S/P visualisation d'une zone mémoire bloc-note
                      ;   jusqu'à la rencontre d'un ETX
                      ;   les registres de l'ACIA1 sont employés ici
                      ;   Nom Appel: VISU
                      ;   Entrée: X: Adresse début bloc
                      ;   Sortie: aucun registre affecté
                      ;   Encombrement pile système: 7 octets
                      ;*******************************************************
820C 34   17          VISU   PSHS   X,B,A,CC      ;
820E A6   80                 LDA    ,X+           ; caractère à transmettre
8210 81   04                CMPA   #4             ; ETX ???
8212 26   02      8216       BNE    *+4            ;
8214 35   97                PULS   PC,X,B,A,CC    ; fin S/P VISU
8216 F6   EC80               LDB    RSET1          ;
8219 54                      LSRB                  ; registre réception libre ???
821A 54                      LSRB                  ;
821B 24   F9      8216       BCC    *-5            ; sinon, attendre
821D B7   EC81               STA    RSTD1          ; transmettre
8220 81   0A                 CMPA   #$A            ; saut de ligne ???
8222 27   06      822A       BEQ    TDNUL          ; si oui, transmettre des nuls
8224 81   0D                 CMPA   #$D            ; retour chariot ???
8226 27   02      822A       BEQ    TDNUL          ;
8228 20   E4      820E       BRA    VISU+2         ; caractère suivant
822A C6   1E           TDNUL  LDB    #30            ; 30 nuls à transmettre
822C B6   EC80               LDA    RSET1          ;
822F 44                      LSRA                  ;
8230 44                      LSRA                  ;
8231 24   F9      822C       BCC    TDNUL+2        ;
8233 4F                      CLRA                  ;
8234 B7   EC81               STA    RSTD1          ;
8237 5A                      DECB                  ;
8238 26   F2      822C       BNE    TDNUL+2        ;
823A 20   D2      820E       BRA    VISU+2         ; caractère suivant
                                                  ;....                      ....
                                                  ;....                      ....
823C 39                      RTS                   ; fin S/P Visu
```

Quelques explications sur le programme ci-dessus

-- Toutes les étiquettes assembleur sont définies au début du programme. Au besoin adapter les adresses au système sous test avant de procéder à une compilation. La pile système est mise ici à $8400.

-- Après avoir configuré l'interface de communication sur le mode interruption en réception et non en transmission, le programme de charge les vecteurs d'interruption SWI et IRQ dans les adresses réservées du 6809. Avant d'entrer dans la boucle sans fin, le 6809 autorise l'interruption par IRQ.

-- Examinons tout d'abord l'interruption logicielle SWI. Pour la rendre opérationnelle, il est nécessaire de remplacer BRA TSTINT par 2 instructions NOP puis lancer l'exécution à partir de $8000.

-- A l'entrée dans la séquence d'exception SWI, les interruptions par IRQ et FIRQ sont inhibées. L'ensemble des registres est sauvegardé dans la pile système et le pointeur S décroît de 12 unités.

Après la visualisation des registres par l'appel de VISURG, la séquence d'exception autorise les interruptions par IRQ et FIRQ.

Ceci n'est pas tout à fait nécessaire, puisque le contrôle est donné au moniteur

L'instruction LEAS 12,S rétablit la position du pointeur pour simuler les conditions du programme principal juste avant la rencontre de SWI.

Rappelons que cette séquence d'exception ne charge aucun registre du 6809 à part CC et PC. Donc, à

l'entrée du moniteur par JMP MONIT, on retrouve les conditions du programme principal si le moniteur du système sous test sait sauvegarder le contexte.
Au besoin, supprimer les instructions ANDCC #%10101111 et LEAS 12,S.

-- Pour tester les interruptions par IRQ, rétablir l'instruction BRA TSTINT. L'interruption peut se produire de façon aléatoire à l'intérieur de la boucle TSTINT. Chaque fois qu'une donnée est rentrée au clavier, l'impulsion sur IRQ provoque une séquence d'interruption.

Si la donnée reçue n'est pas la touche "H", la séquence d'exception du premier niveau retourne le contrôle programme principal en restituant l'ensemble des registres.

-- Lorsque la touche "H" est d'identifiée, le 6809 visualise les registres selon le format suivant :

```
PAUSE  PC:....  S:...  U:....  Y:....  X:....  DP:....  B:....
A:....  CC:........
```

Après la transmission de l'ensemble du buffer vers l'écran, la séquence d'exception du premier niveau modifie le vecteur IRQ à l'adresse $FFF8, $FFF9 avant d'exécuter CWAI et autoriser simultanément IRQ.

L'opérateur peut alors rentrer une autre commande. Chaque impulsion sur IRQ provoque un traitement de 2ième niveau.

Si l'identification révèle une touche autre que "C", le contrôle est rendu au 1er niveau mais sur l'instruction CWAI.

Ce détail implique une modification de la valeur du compteur programme dans la pile système avant de dépilement.

Si le caractère reçu est "C", le sous-programme du 2ième niveau affiche le message "EXECUTION" puis retourne au 1er niveau, à l'instruction juste après CWAI.

Avant de rendre le contrôle au programme principal, le 1er niveau rétablit encore une fois le vecteur IRQ correspondant.

Donc, il est important de souligner qu'on peut imbriquer un grand nombre d'interruptions en rétablissant chaque fois correctement le vecteur IRQ du niveau correspondant.

Cette manipulation est normalement interdite puisque le masque bit I est mis à 1 chaque fois qu'on actionne la ligne IRQ.

Néanmoins, le programmeur peut concevoir son système particulier, le 6809 possède toutes les ressources logicielles pour satisfaire le programmeur exigeant.

Trois broches d'entrées d'interruption Matérielle NMI IRQ et FIRQ

Interruptions matérielles NMI, IRQ et FIRQ

Séquence d'initialisation RESET, traitée comme l'interruption la plus prioritaire du système.

Descriptions : voir chapitre Brochage du 6809

Instructions d'interruption Logicielle SWI, SWI2 et SWI3

Instructions permettant l'arrêt du programme en cours (pour voir le chapitre des instructions) :

SWI (SoftWare Interrupt)

SW2 (SoftWare Interrupt 2)

SWI3 (SoftWare Interrupt 3)

Traitement des Interruptions Logicielles SWI SWI2 SWI3

Dès que le 6809 rencontre l'une des trois instructions ci-dessous SWI, SWI2, SWI3, celui-ci sauvegarde l'état complet des registres internes dans la pile système. Dans l'ordre suivant :

S – 1 --> S	**PC bas --> {S}**	
S – 1 --> S	**PC haut --> {S}**	
S – 1 --> S	**U bas --> {S}**	{….} = le contenu de
S – 1 --> S	**U haut --> {S}**	
S – 1 --> S	**Y bas --> {S}**	
S – 1 --> S	**Y haut --> {S}**	
S – 1 --> S	**X bas --> {S}**	
S – 1 --> S	**X haut --> {S}**	
S – 1 --> S	**DP --> {S}**	{….} = le contenu de
S – 1 --> S	**B --> {S}**	
S – 1 --> S	**A --> {S}S – 1 --> S**	**CC --> {S}**

Puis le 6809 continu son déroulement.

Instructions d'Attente d'Interruptions SYNC CWAI

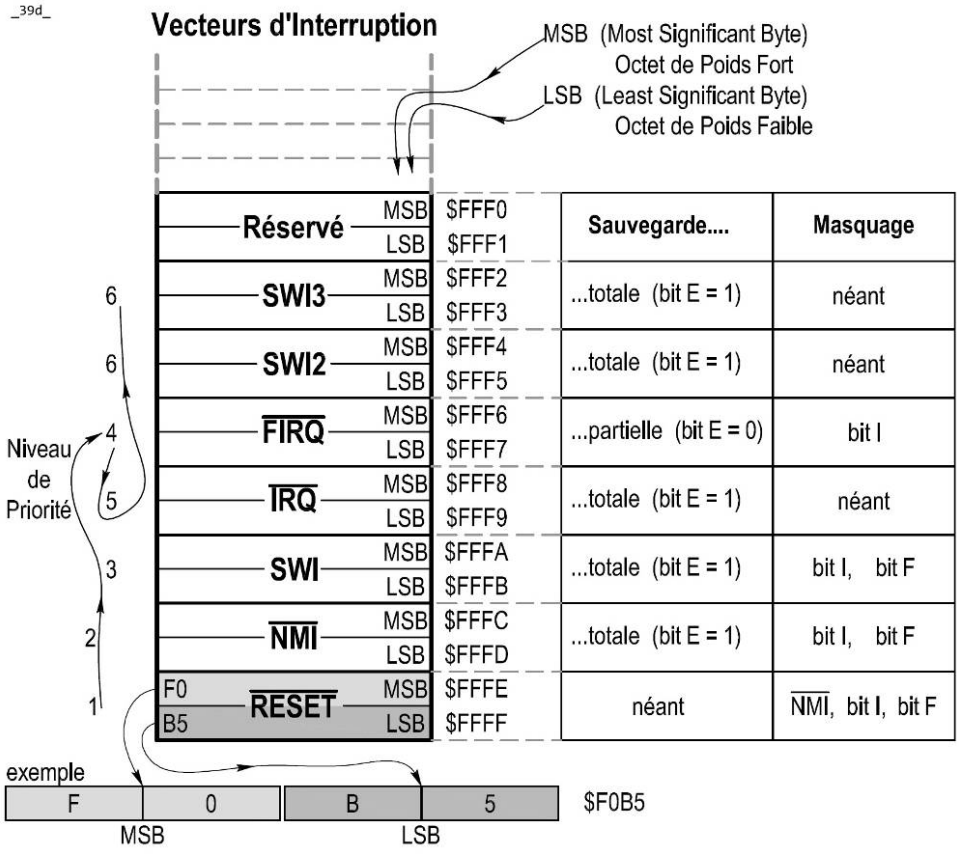
Voir l'instruction Instruction CWAI (Clear WAit Interrupt) Attente d'interruption
Voir l'instruction Instruction SYNC (SYNChronize to external event) Attente d'une synchronisation externe

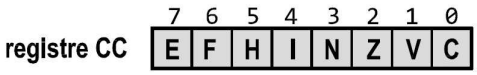
Tableau des Vecteurs d'Interruption

Les interruptions du 6809 font apparaître des niveaux de priorité, généralement liés à la structure Matérielle.

Lors d'une interruption le 6809 se positionne automatiquement à une adresse contenue dans deux octets (MSB et LSB).

Cette adresse représente l'adresse du programme de traitement de l'interruption demandée, en voici le tableau.





Le but de cette partie est de mettre en évidence les méthodes de gestion des requêtes d'Entrée-Sortie des périphériques du 6809.

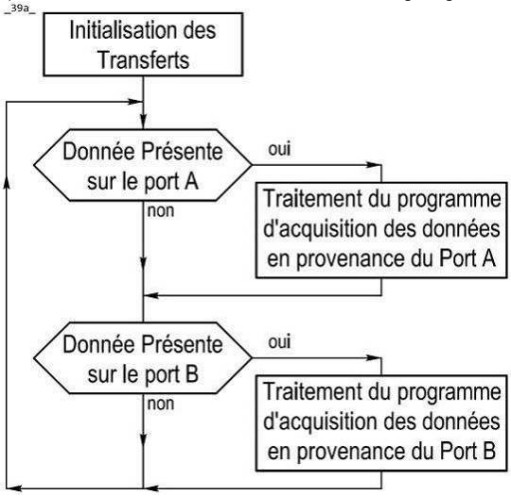
Le 6809 utilise les circuits intégrés d'Entrée-Sorties comme des projections en mémoires.
Ces circuits sont connectés aux bus de données et d'adresses comme des boitiers mémoires.
Ils sont vu par le programmeur comme des emplacements mémoires.

On peut adresser les circuit d'entrée-sortie en :
-- Les considérant comme des emplacement mémoire
-- en utilisant des instructions spécifiques d'Entré-Sortie

Scrutation Systématique

Dans ce cas le 6809 initialise les transferts, puis exécute une boucle d'attente d'événement extérieur en testant systématiquement les indicateurs d'état de chacun des périphériques.

Pour une application comportant un 6809 et un PIA 6821, voici l'organigramme de la boucle de scrutation.



L'immobilisation du 6809 pénalise fortement cette méthode, le temps d'activité du 6809 est très faible devant celui des périphériques connectés (une imprimante par exemple).

Pour remédier à cela, le fonctionnement en interruptions apporte une solution appréciable.

Principe de Fonctionnement en interruption

On interrompt le programme en cours et on exécute un sous-programme de traitement d'interruption.
Le sous-programme de traitement contient les routines de transfert, il tient compte du niveau de priorité des interruptions.

Ce mode de transfert permet au 6809 d'exécuter des traitements indépendants.
Le 6809 est sollicité seulement lorsque l'interface est prête à dialoguer en vue de transférer des informations à la mémoire centrale.

Gestions des Interruptions

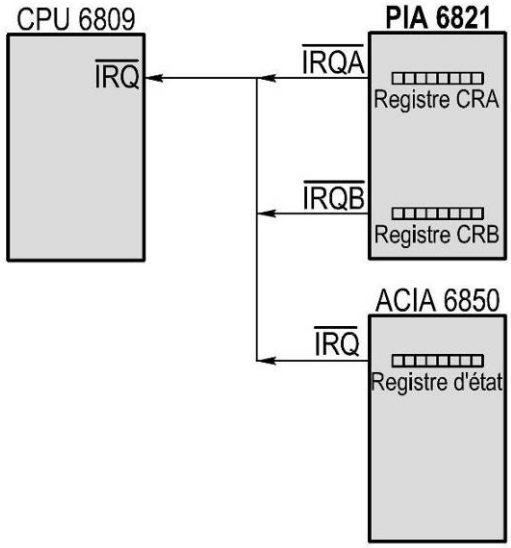
En général, l'entrée IRQ| du 6809 doit supporter plusieurs interfaces. Les interfaces possèdent des niveaux de priorité égaux.

Pour définir un niveau de priorité d'une interface par rapport à une autre il existe plusieurs méthodes :
-- La méthode logicielle
-- La méthode matérielle

Méthode logicielle

Toutes les lignes des interfaces sont câblées en "OU" et reliées à l'entrée demande d'interruption du 6809.

_39b_

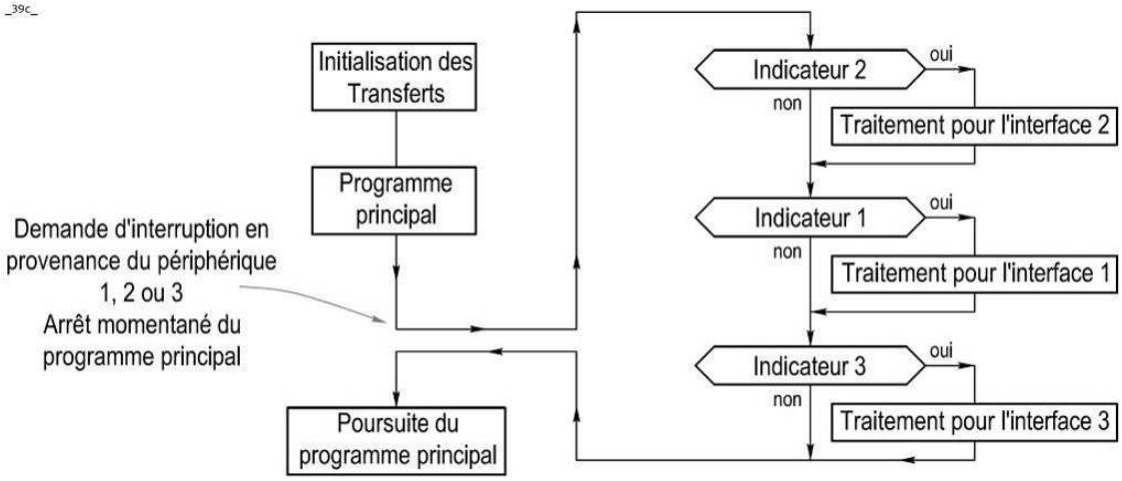


Lorsque l'entrée IRQ| du 6809 est activée, le 6809 fournit l'adresse du vecteur d'interruption (pour le 6809 le vecteur pour IRQ| est une adresse de 16 bits stockée en $FFF8 et $FFF9.

C'est l'informaticien qui détermine la priorité pour tester chacun des indicateurs d'état, afin de savoir qui a créé l'interruption. La première interface testée dans le sous-programme sera la plus prioritaire.

La figure suivante montre que l'interface n°2 est la plus prioritaire. La moins prioritaire étant la n°3 dans cet exemple. Une scrutation (ou "POLLING") est réalisée seulement après une demande d'interruption.

_39c_



Méthode matérielle

On peut établir la hiérarchie entre les interfaces en utilisant une logique électronique externe qui permet d'identifier directement l'origine de la demande.

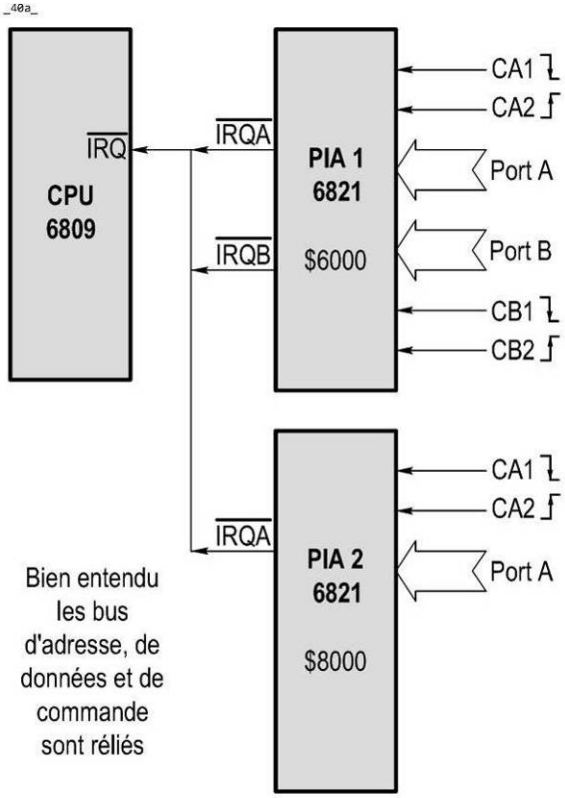
Cette méthode permet au 6809 d'accéder directement au vecteur d'interruption correspondant à la demande, il n'y a donc plus besoin de scrutation logicielle des interfaces.

Il existe des circuits intégrés spécialisés dont le rôle est de contrôler les priorités d'interruption.

Exemple 01 à base de 2 PIA 6821

Dans cet exemple, une application est à base d'un 6809 et de deux PIA assurant les liaisons avec la périphérie.

Ex01 Structure de l'Application



Les ports A et B du PIA1 et le port A du PIA2 sont en entrée.

Toutes les lignes de dialogue sont programmées en entrées.

Les lignes CA1 et CB1 sont actives sur des fronts descendants.
Les lignes CA2 et CB2 sont actives sur des fronts montants.

A chaque ligne de dialogues (CA1, CB1, CB1, CB2) correspond un périphérique différent, et par conséquent un sous-programme de traitement approprié.

Le S/Prog CA1 traite l'interruption générée par la ligne CA1 du PIA1

Le S/Prog CA2 traite l'interruption générée par la ligne CA2 du PIA1

L'entrée d'interruption IRQ| du 6809 est reliée aux lignes IRQA et IRQB des PIA (voir schéma ci-dessus).

Ex01 Fonctionnement

On ne s'occupe dans cet exemple que des informations en provenance de l'extérieur.
Comme toutes les lignes d'interruption sont reliées entre elles, le contrôle de priorité est réalisé par logiciel.

Le programme de gestion des PIA se décompose en 3 parties :

- Initialisation du transfert
- Scrutation des interfaces
- Exécution du transfert.

Un front actif sur CA1 du PIA1 entraîne le positionnement le bit indicateur d'état CRA7, une interruption validée par le bit CRA0 = 1 est envoyé vers le 6809.

Le 6809 termine l'instruction en cours puis exécute le sous-programme de traitement des interruptions.

L'indicateur est réinitialisé, les données sont transférées, puis le processeur reprend le programme principal.

Ex01 Organisation de la mémoire

Chacun des PIA occupe 4 octets mémoire.

Les 6 sous-programmes de traitement des transferts d'information sont implantés entre les adresses $3000 et $4000.

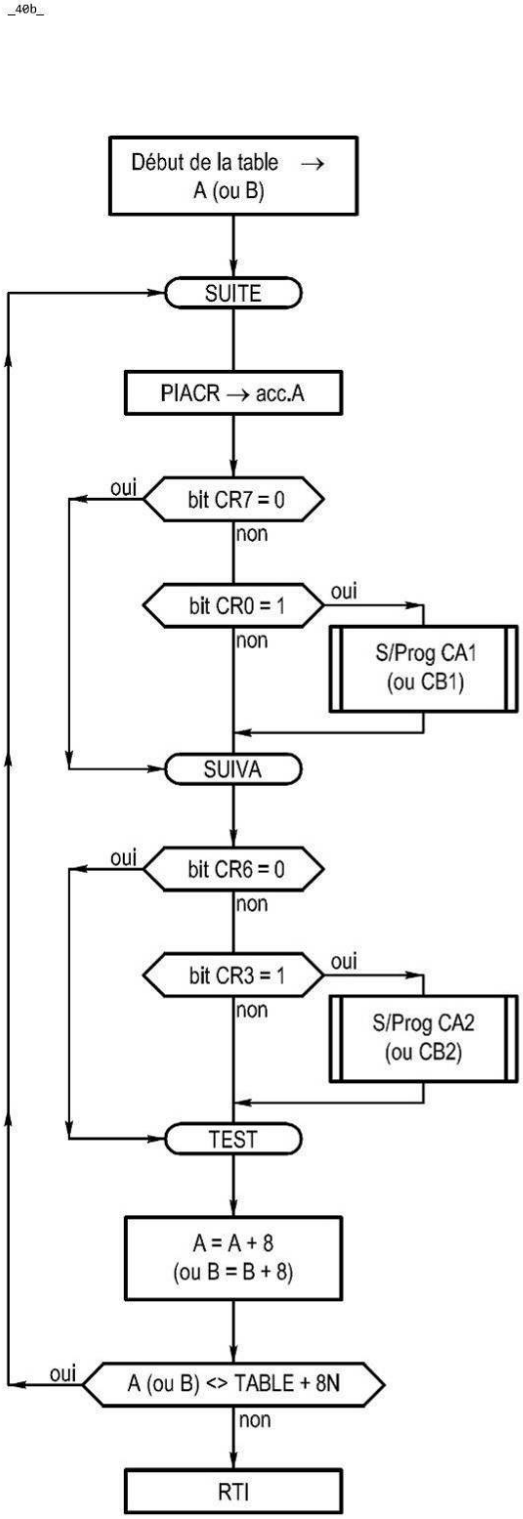
L'ensemble des adresses de base des sous-programmes de traitement et des octets PIA est regroupé dans une table à partir de l'adresse $2000.

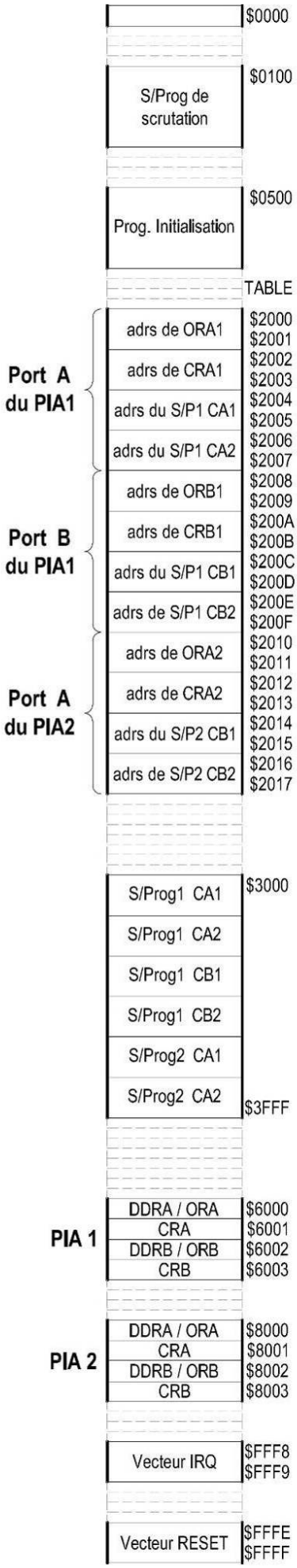
Le sous-programme de scrutation est implanté à l'adresse $0100.

Le sous-programme d'initialisation à l'adresse $0500.

Voir l'organisation de la mémoire apposé contre l'organigramme suivant.

Ex01 Organigramme de la scrutation





Ex01 Programmation

Cette partie permet d'initialiser le PIA

Il faut programmer les ports PIA1 A, PIA1 B et PIA2 A en entrées en initialisant les registres DDRA et DDAB.
Le fonctionnement est défini ensuite par le contenu des registres de contrôle

```
                        ;    prog FEI-04
                        ;
                   A000 MEM    EQU    $A000      ;
                   F3FF PILE   EQU    $F3FF      ;
                        ;---- Définition des registres de programmation
                   6000 ADPIA1 EQU    $6000      ;
                   8000 ADPIA2 EQU    $8000      ;
                   6001 CRA1   EQU    ADPIA1+1   ;
                   6000 ORA1   EQU    ADPIA1     ;
                   6000 DDRA1  EQU    ADPIA1     ;
                   6003 CRB1   EQU    ADPIA1+3   ;
                   6002 ORB1   EQU    ADPIA1+2   ;
                   6002 DDRB1  EQU    ADPIA1+2   ;
                   6001 CRA2   EQU    ADPIA1+1   ;
                   8000 ORA2   EQU    ADPIA2     ;
                   8000 DDRA2  EQU    ADPIA2     ;
                   8003 CRB2   EQU    ADPIA2+3   ;
                   8002 ORB2   EQU    ADPIA2+2   ;
                   8002 DDRB2  EQU    ADPIA2+2   ;
0500                        ORG    $0500      ; ---------------------------------------
0500 7F   6001              CLR    CRA1       ;}
0503 7F   6003              CLR    CRB1       ;} tous les registres de contrôle
0506 7F   6001              CLR    CRA2       ;} sont à zéro
0509 7F   8003              CLR    CRB2       ;}
050C 7F   6000              CLR    DDRA1      ; port A du PIA 1 en entrée
050F 7F   6002              CLR    DDRB1      ; port B du PIA 1 en entrée
0512 7F   8000              CLR    DDRA2      ; port A du PIA 2 en entrée
0515 86   FF                LDA    #$FF       ; le port B PIA2 en sortie
0517 B7   8002              STA    DDRB2      ;
051A 86   1D                LDA    #%00011101 ;}
051C B7   6001              STA    CRA1       ;}
051F B7   6003              STA    CRB1       ;}—initialisation de CRA et CRB
0522 B7   6001              STA    CRA2       ;}
0525 86   2D                LDA    #%00101101 ;}
0527 B7   8003              STA    CRB2       ;}
052A 8E   0100              LDX    #$0100     ; init du vecteur
052D BF   FFF8              STX    $FFF8      ; d'interrup IRQ ($FFF8 et $FFF9)
```

Scrutation des interfaces

La scrutation consiste dans ce cas à venir tester tous les indicateurs d'états de chaque interface.

```
                        ;    prog FEI-05
                        ;
                   7001 ORA1   EQU    $7001      ;
                   7002 ORA2   EQU    $7002      ;
                   7004 CRA1   EQU    $7004      ;
                                                 ;
                   7101 ORB1   EQU    $7101      ;
                   7102 ORB2   EQU    $7102      ;
                   7104 CRB1   EQU    $7104      ;
                   7106 CRB2   EQU    $7106      ;
                                                 ;
                   2000 TABLE  EQU    $2000      ;
                                                 ;
0000                        ORG    $0000      ;
0000 8E   7001              LDX    #ORA1      ; port A du PIA1
0003 BF   2000              STX    TABLE      ;
0006 8E   7004              LDX    #CRA1      ;
0009 BF   2002              STX    TABLE+2    ;
000C 8E   3000              LDX    #$3000     ; adrs S/Prog 1 CA1
000F BF   2004              STX    TABLE+4    ;
0012 8E   3200              LDX    #$3200     ; adrs S/Prog 1 CA2
0015 BF   2006              STX    TABLE+6    ;
```

```
0018 8E   7101              LDX    #ORB1        ; port B du PIA1
001B BF   2008              STX    TABLE+8      ;
001E 8E   7104              LDX    #CRB1        ;
0021 BF   200A              STX    TABLE+10     ;
0024 8E   3400              LDX    #$3400       ; adrs S/Prog 1 CB1
0027 BF   200C              STX    TABLE+12     ;
002A 8E   3600              LDX    #$3600       ; adrs S/Prog 1 CB2
002D BF   200E              STX    TABLE+14     ;
0030 8E   7002              LDX    #ORA2        ; port A du PIA2
0033 BF   2010              STX    TABLE+16     ;
0036 8E   7106              LDX    #CRB2        ;
0039 BF   2012              STX    TABLE+18     ;
003C 8E   3800              LDX    #$3800       ; adrs S/Prog 2 CA1
003F BF   2014              STX    TABLE+20     ;
0042 8E   4000              LDX    #$4000       ; adrs S/Prog 2 CA2
0045 BF   2016              STX    TABLE+22     ;
                                                ;
                    ;-------------------------------------------
                    ;On commence à travailler sur le port A du PIA 1
                    ;------programme d'interruption scrutation des interfaces.
0100                        ORG    $0100        ;
0100 BE   2000              LDX    TABLE        ; initialisation de la recherche
0103 A6   94          SUITE LDA    [,X]         ; lecture de CRA1
0105 2A   07      010E      BPL    SUIVA        ; test sur CA1,
                                                ; on va à SUIVA si CA1 inactive
0107 85   01                BITA   #$1          ; l'interruption est-elle valide ?
0109 27   03      010E      BEQ    SUIVA        ; branch. à SUIVA si interrup est masquée
010B AD   98 04             JSR    [4,X]        ; chercher l'adresse S/Prog de traitement
                                                ; d'interruption due à Cx1 des PIA
010E 49               SUIVA ROLA                ; rotation à gauche ? test sur CA2
010F 2A   07      0118      BPL    TEST         ; test sur CA2 on va à TEST si CA2 inactive
0111 84   10                ANDA   #$10         ; l'interruption est elle valide
0113 27   03      0118      BEQ    TEST         ; branch. à TEST si interrup est masquée
0115 AD   98 06             JSR    [6,X]        ; chercher l'adresse du sous-programme de
                                                ; traitement d'interruption due à Cx2
                                                ;
                    ;-----------------------------------------------------
                    ;On travaille ensuite sur les autres ports :
                    ;N = nombre de ports (N=3 dans cet exemple)
                0003 N      EQU    3            ;
0118 30   08          TEST  LEAX   8,X          ; on change de port
011A 8C   2018              CMPX   #TABLE+(8*N) ; test pour voir si la scrutation est
                                                ; terminée. Si elle est en cours, on
                                                ; se branche à SUITE
011D 26   E4      0103      BNE    SUITE        ;
011F 3B                     RTI                 ;
```

EX01 Exécution du transfert

Le programme d'exécution du transfert consiste à lire le contenu du registre de sortie du port concerné puis à le transférer dans la mémoire.

Pour le S/Prog SP1 CA1, dont l'adresse de départ est contenue dans la table à l'adresse TABLE + 4.

```
                    ;    prog FEI-06
                    ;
                7001 ORA1   EQU    $7001     ;
                1000 MEM    EQU    $1000     ;
                                              ;
                    ;------programme de transfert exemple sur S/Prog 1 CA1
3000                        ORG    $3000     ;
3000 B6   7001              LDA    ORA1      ; lecture port A du PIA1 (CRA7=0 pour PIA1)
3003 B7   1000              STA    MEM       ; le contenu est transféré à l'adresse MEM
3006 39                     RTS              ; on retourne au sous-programme de scrutation
                            END              ;
```

Exemple 02 Interfaçage d'un clavier Hexadécimal

Ex02 Sujet

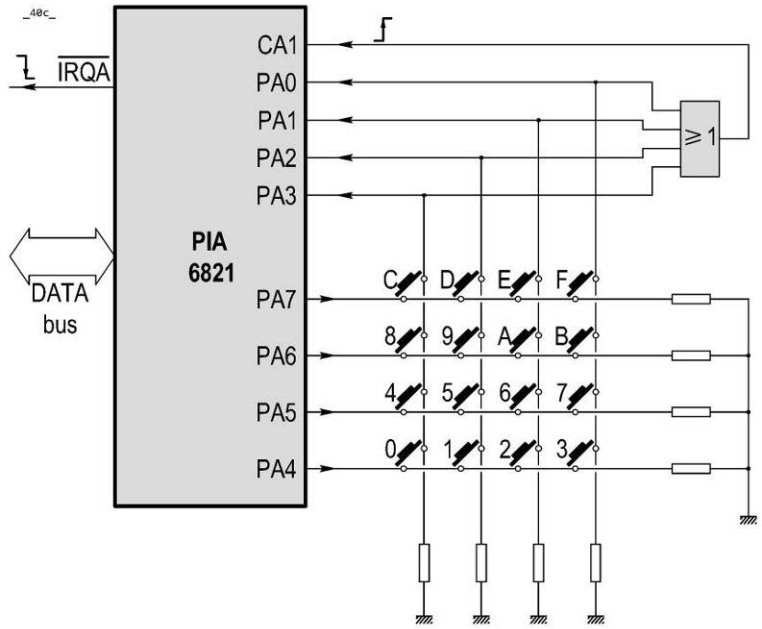
C'est l'exemple de l'interfaçage d'un clavier hexadécimal à travers un PIA en utilisant la ligne IRQ|.
Le clavier est composé de 16 boutons poussoirs organisés en matrice 4x4.

Il s'agit de générer dans l'accumulateur **A** le code hexadécimal correspondant à la touche enfoncée.
Exemple si c'est la touche B il faudra générer %0000 1011.

De plus la technologie des boutons poussoirs étant sommaire ils fourniront un effet de rebondissements qu'il faudra éliminer par logiciel.

Enfin, si l'on enfonce plusieurs touches simultanément, on souhaite appeler un sous-programme ERROR (non développé ici), il pourra par exemple commander l'écriture d'un message ERROR sur l'affichage associé.

Ex02 Schéma



Ex02 Principe

Au repos, aucune touche n'étant enfoncée, l'entrée CA1 du PIA est au niveau Bas.
Si on presse une touche quelconque, un front montant sera envoyé sur la broche CA1, déclenchant le processus d'interruption.

A l'état d'initialisation du PIA les broches PA4 à PA7 sont en sortie et sont toutes au niveau Haut. Les broches PA0 à PA3 sont en entrée.

Lorsque l'on presse une touche du clavier, le programme de traitement d'interruption lira d'abord sur PA0 à PA3 le numéro de la colonne de la touche enfoncée.

Puis l'on inverse les fils d'entrée-sortie sur le port A, on lit alors sur PA4 à PA7 le numéro de rangée de la touche enfoncée tandis que les lignesPA0 à PA3 sont en sortie et au niveau Haut.

On remarque qu'à chaque touche, correspond un numéro de colonne et un numéro de rangée particuliers. C'est à partir de ces numéros que l'on peut trouver différents procédés pour le codage en hexadécimal de chacune des touches.

On a créée une table comportant 16 codes à l'aide des instructions FCB correspondant aux 16 touches du clavier.

Chaque code est composé :
-- D'un premier chiffre pour le n° de colonne
-- D'un deuxième chiffre pour le n° de rangée

Le code généré par la touche enfoncée est comparé à la table en partant de la première position de cette table et jusqu'à trouver l'identité des codes.

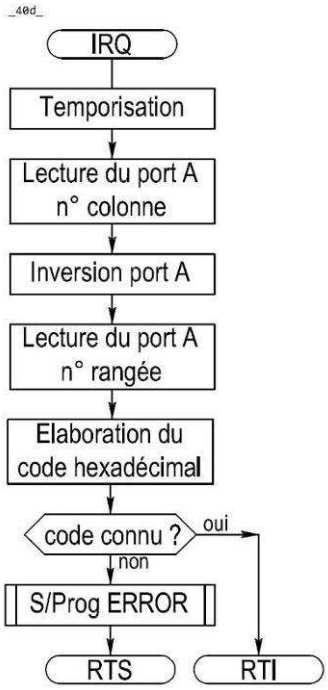
Un compteur initialisé à zéro, est incrémenté après chaque comparaison non satisfaisante. Lorsqu'il y aura égalité ce compteur contiendra le code hexadécimal recherché.

Si l'on n'a pas trouvé le code après avoir balayé toute la table, il est vraisemblable que plusieurs touches ont été pressées simultanément, on appelle alors le sous-programme ERROR.

Ce programme étant un programme d'interruption, l'adresse de départ $1000 aura au préalable été chargé dans les vecteurs de l'interruption IRQ à savoir $FFF9 et $FFF8 par la séquence suivante :

```
LDX     #$1000
STX     $FFF8
```

Ex02 Organigramme



Ex02 Programmation

```
00001                                                   ;    prog FEI-07
00002                                   ;
00003  0100                                 ORG    $0100           ;
00004                          1040  ERROR  EQU    $1040           ;
00005                          8005  PIACRA EQU    $8005           ;
00006                          8004  PIADRA EQU    $8004           ;
00007                          8004  PIAORA EQU    PIADRA          ;
00008                          0000  RAM0   EQU    $0000           ; 1ère adresse de la mémoire RAM.
00009                          0001  RAM1   EQU    $0001           ; 2ème adresse de la mémoire RAM.
00010                                                               ;
00011                                   ; Initialisation du port A du PIA
00012                                   ; Interruption sur le front montant de CA1
00013                                   ; 4 broches LSB en entrée
00014                                   ; 4 broches MSB en sortie état Haut
00015                                                               ;
00016  0100 4F                          INIT   CLRA                 ;
00017  0101 B7   8005                         STA    PIACRA          ;
00018  0104 86   F0                           LDA    #$F0            ; %11110000
00019  0106 B7   8004                         STA    PIADRA          ;
00020  0109 C6   07                           LDB    #$07            ; %00000111
00021  010B F7   8005                         STB    PIACRA          ;
00022  010E B7   8004                         STA    PIAORA          ;
00023  0111 39                               RTS                    ;
00024                                                               ;
00025                                   ; programme d'interruption pour décodage
00026                                   ; de clavier hexadécimal
00027                                                               ;
00028  1000                                 ORG    $1000           ;
```

```
00029                          ;------tableau des codes du clavier
00030  1000 18 14 12 11        TAB     FCB     $18,$14,$12,$11 ;
00031  1004 28 24 22 21                FCB     $28,$24,$22,$21 ;
00032  1008 48 44 42 41                FCB     $48,$44,$42,$41 ;
00033  100C 88 84 82 81                FCB     $88,$84,$82,$81 ;
00034                                                          ;
00035                          ;------début de la tempo anti-rebond
00036                          ;                              du clavier
00037  1010 86   04            TEMPO   LDA     #$04            ; = 2µs
00038  1012 97   00                    STA     RAM0            ; = 4µs
00039  1014 86   02            BCL4    LDA     #$02            ; = 2µs
00040  1016 97   01                    STA     RAM1            ; = 4µs
00041  1018 0A   01            BCL3    DEC     RAM1            ;                          } 6µs
00042  101A 26   FC      1018          BNE     BCL3            ; si [RAM1]`"0 alors BCL3 } 3µs
00043                                                          ;           9µs x 02 = 18µs
00044  101C 0A   00                    DEC     RAM0            ;           6µs
00045  101E 26   F4      1014          BNE     BCL4            ; si [RAM0]`"0 alors BCL4
00046                                                          ;           3µs 18µs x 04 =72µs
00047                                                          ;
00048                          ;------fin de la tempo
00049  1020 F6   8004                  LDB     PIAORA          ;  lecture colonne
00050  1023 86   03                    LDA     #$03            ;
00051  1025 B7   8005                  STA     PIACRA          ;  inversion port A
00052  1028 86   0F                    LDA     #$0F            ;  %0000FFFF 4 MSB en entrée
00053  102A B7   8004                  STA     PIADRA          ;            4 LSB en sortie
00054  102D B7   8005                  STA     PIACRA          ;  état haut
00055  1030 B7   8004                  STA     PIAORA          ;
00056  1033 F4   8004                  ANDB    PIAORA          ;  lecture rangée
00057  1036 8E   1000                  LDX     #TAB            ;  code touche dans acc.B
00058  1039 4F                        CLRA                    ;  registre comptage
00059  103A E1   84            ENCORE  CMPB    0,X             ;
00060  103C 27   09      1047          BEQ     FIN             ;
00061  103E A6   80                    LDA     ,X+             ; pour incrémenter X de 1
00062                                                          ; (remplace l'instruction INX du vieux
6800)
00063  1040 4C                        INCA                    ;
00064  1041 81   10                    CMPA    #$10            ;  code non trouvé
00065  1043 27   FB      1040          BEQ     ERROR           ;  aller à ERROR
00066  1045 20   F3      103A          BRA     ENCORE          ;
00067                                                          ;
00068                          ;------résultat dans acc.A,  Retour interruption
00069  1047 BD   0100          FIN     JSR     INIT            ;  réinitialisation PIA
00070  104A 3B                        RTI                     ;  pour nouveau codage
00071                                  END                     ;
```

MISE AU POINT D'UN PROGRAMME EN ASSEMBLEUR

Généralités

Plusieurs étapes dans la création d'un programme :
- -- Poser le problème
- -- L'organigramme
- -- Ecriture du programme
- -- Programmation Structurée
- -- Mise au point (MauP)
- -- Economie de place mémoire, quelques conseils

Poser le problème

Savoir exactement quelles fonctions le programme devra réaliser, quelles Entrées-sorties on utilisera. Cette phase est peut être un peu astreignante mais elle oblige à avoir les idées claires et permet d'aborder plus sereinement la suite.

L'organigramme (appelé aussi ordinogramme)

Quand on réalise un programme, surtout s'il est complexe, il est toujours bon de savoir comment va se dérouler son exécution avant de commencer à programmer. D'ailleurs, dans certains programmes, c'est essentiel, pour plusieurs raisons :

- -- La détection et le traitement des erreurs.
- -- Les structures de contrôle entraînant de longues conséquences sur votre programme.
- -- L'organisation des tâches pendant la création d'un programme en équipe.

A -- C'est la représantation symbolique d'un programme.
B -- L'organigramme est à faire avant le codage (avant la programmation).
C -- Seul 10 % des programmeurs sont capable d'écrire un programme correct sans passer par un organigramme.
D -- Malheuresement 90 % des programmeurs croient faire partie des 10 % cité au point C.
E -- Il en résulte que 80 % des programmes écrit par les programmeurs cité au point D ne fonctionne pas au premier passage et de plus ces programmeurs passent un temp considérable à tester, à corriger, à mettre au point (MauP) leurs programmes.

Voici quelques règles utiles

- -- Diviser le programme en plusieurs parties, chacune d'elles réalisant une ou plusieurs fonctions élémentaires.
- -- Eviter les "astuces géniales" qui peuvent ne pas être comprises par d'autres ou par soi même dans plusieurs mois.
- -- Eviter de mettre des instructions de programmation.
- -- Concevoir son programme de façon structurée, séquence après séquence avec une entrée et une sorties par séquence.
- -- Mettre un maximum de commentaires.

Un organigramme est normalisé, c'est à dire que tout le monde s'est mis d'accord pour dessiner les mêmes symboles. Dans notre cas c'est la norme ISO 5807.

L'écriture du programme

A ses débuts, le programmeur inexpérimenté dans le langage Assembleur a tendance à fixer son attention sur la fonctionnalité à produire, quelque soit la quantité de ligne de code, les procédures et les fonctions utilisées pour produire le résultat final. Et ceci sans comprendre parfois ce qu'il fait vraiment ou les spécificités de ce langage.

Le but est de faire un programme **très lisible**. Ne pas oubliez que dans l'industrie on travaille en équipe. Quelqu'un d'autre doit être capable de reprendre le programme.

Ne pas être avare de commentaires détaillés dans le corps même du programme. Le fait d'écrire un maximum de commentaires, très utiles lors de la mise au point ou lors de modifications ultérieures.

Enfin la rédaction de la documentation est primordiale, elle doit commencer à la genèse du programme, être mise à jour en fonction des évolutions majeur de la programmation, pour être finalisée et mise à jour dès que le programme a été complètement testé et débugué.

En règle générale, il faut faire des programmes "le plus simple que possible".
Après avoir écrit un programme compliqué, il suffit de quelques mois pour que le programmeur n'arrive pas le modifier. Alors qu'en est-il des autres programmeurs qui vont devoir prendre la suite !

Voici quelques règles d'or, pour éviter de faire trop d'erreurs

Etudier au préalable le problème posé (et surtout rester simple)

Réfléchir et imaginer un algorithme avant d'écrire la première ligne du programme. L'analyse du problème à résoudre doit rester simple. Cette analyse écrite doit être claire et facile à comprendre par d'autres programmeurs. L'organigramme doit être lisible par des non-spécialistes

Début de la création d'une documentation

En fonction d'une analyse écrite, déterminer les points principaux à traiter.

Ecriture d'un ordinogramme lisible par des non-spécialistes

(Simple et surtout très structuré avec des modules une entrée une sortie), évitez les instructions du style GOTO, par exemple comme les mnémoniques JMP, BRA.... Les instructions JMP (jump) sont à proscrire, elles sont analogues au GOTO en BASIC. Concevoir son programme de façon structurée, séquence après séquence avec une entrée et une sortie par séquence.

Mettre un maximum de commentaires.

Commenter chaque morceau du programme et de l'ordinogramme de manière explicative et non descriptive.

Ne pas hésiter à faire des sous-programmes.

Diviser le programme en plusieurs parties, chacune d'elles réalisant une ou plusieurs fonctions élémentaires. Elles pourront être des sous-programmes indépendants, afin de pouvoir les tester séparément lors d'une mise au point.

Ne pas écrire de longues procédures.

Une procédure ne devrait pas avoir plus de 20 lignes de code. Chaque procédure doit avoir un objectif clair. Un bon programme doit avoir des procédures claires, sans cumul.
Se méfier des "copier collé". Il faut relire à chaque fois.

Evitez les étiquettes du style "DFGHTYG", soyer clair.

-- Lors du codage, choisir des noms parlants pour représenter les objets manipulés dans le programme, éviter l'abstrait et opter toujours pour le concret.
-- Eviter le vocabulaire peu suggestif du type (TOTO, TATA, ESSAI, CHOSE, TRUC, XXX, ...)
-- Faire attention à des ressemblances formelles du type : O et 0, I et 1, Z et 2, B et 8, A et 4, S et 5
-- Lors d'un chiffrage, une bonne habitude consiste à utiliser deux chiffres (00, 01, 02, ...) ou trois chiffres (000, 001, 002, ...) si l'on sait que la plage pourrait aller au-delà de 100.

Eviter les "astuces géniales"

Qui peuvent ne pas être comprises par d'autres ou par soi même dans plusieurs mois. Ne pas utiliser des astuces de programmation qui rendrait les programmes illisibles. Les programmeurs ne doivent pas utiliser les fonctions fantaisistes du langage. L'utilisation des fonctions simples oblige le programmeur à réfléchir à ce qu'il écrit. Ne jamais utiliser les fonctionnalités du langage dont vous n'êtes pas sûr(e) du résultat ou du rôle.

Tester chaque module, procédure, fonction séparément.

En test, prévoyez tous les cas de figure possibles, faire des simulations de tous les cas.

Finalisation de la documentation

Elle se voudra claire et concise. Faire la mise au propre et en final éditer une version papier (ne pas oublier d'indexer les pages et de mettre le numéro de version). Faire la relecture de cette documentation en corrélation avec les commentaires laissés tout au long du programme et de l'ordinogramme.
Ne pas oublier de mettre à jour la Documentation après des éventuelles maintenances ou des modifications.

Dernière règle

On applique ces règles ci-dessus chaque jour pendant au moins six mois.

La pratique de la programmation en suivant ces règles d'or peut s'avérer très gênante. Mais c'est un excellent moyen d'apprendre l'assembleur du 6809 et surtout la pratique d'une programmation structurée.

Programmation Structurée

En programmation structurée il existe 3 formes extrêmement employées, voici leur équivalent en assembleur :

IF……THEN……ELSE……END IF

```
             LDA    VAR     ; charge VAR
             CMPA   $00     ;
             BLT    TEST1   ; si négatif
             JSR    PROG2   ;
             BRA    TEST2   ; si positif
      TEST1  JSR    PROG1   ;
      TEST2  SWI             ;
```

DO…WHILE

```
             LDA    VAR     ;
      TEST1  CMPA   #$00    ;
             BLE    TEST2   ;
             .              ;
             . séquence d'instructions dans la boucle
             .              ;
             BRA    TEST1   ;
      TEST2  SWI             ;
```

REPEAT…UNTIL

```
             LDA    VAR     ;
      TEST1  .              ;
             .              ;
             . séquence d'instructions dans la boucle
             .              ;
             CMPA   $00     ;
             BGR    TEST1   ;
             .              ;
             .              ;
```

Mise au Point (aussi appelé **MauP**)

Mise au point ou déverminage (Debugging).

Utilisation de point d'arrêt, utilisation des instructions SWI. Elles permettent d'arrêter un programme là ou on le désire. On peut ensuite visualiser le contenu des registres internes et de la mémoire.

Erreurs classiques

-- Erreurs de branchement des sauts conditionnels.
-- Ordres des opérandes.
-- Modes d'adressages.
-- Ne pas oublier d'initialiser les compteurs de boucle ou les pointeurs de pile.
-- Attention aux modifications que peuvent apporter des sous-programmes sur les bits du registre CC

Economie de place mémoire, quelques conseils

-- Utilisation au maximum de sous-programme pour effectuer des tâches répétitives.
-- Utilisation d'instruction utilisant peu d'octets et notamment l'adressage direct pour les données fréquemment utilisées.
-- Utilisation de la pile Utilisateur pour le passage de paramètres entre les diverses parties du programme.
-- Utilisation d'instruction opérant directement sur les registres ou les cases mémoires.

Prog 01 : Création d'une table de données

Prog 02 : Dénombrement de données spécifiques dans une table

Prog 03 : Multiplication

Prog 04 : Détermination du maximum ou du minimum d'une table

Prog 05 : Transfert d'une table de données d'une zone mémoire vers une autre

Prog 06 : Détermination logicielle de la parité croisée d'une table de données

Prog 07 : Tri des données d'une table

Prog 08 : Détection et correction d'erreurs

Prog 09 : Table de correspondance hexadécimal décimal

Prog 10 : Conversion DCB−binaire

Prog 11 : Multiplication

Prog 12 : Division

Prog 13 : Kit MC09-B Sté DATA RD : Interface Parallèle

Prog 14 : Kit MC09-B Sté DATA RD : Etude des Ports Entrée / Sortie

Prog 15 : Kit MC09-B Sté DATA RD : Etude des Interruptions

Prog 16 : Kit MC09-B Sté DATA RD : Etude des Lignes de Dialogues

Prog 17 : Ouvrage 06 : Mouvements de données 8 et 16 bits par LOAD et STORE

Prog 18 : Décrémenter le nombre $0E cinq fois et de stocker le résultat dans la case mémoire $0800

Prog 19 : Programme capable de calculer la somme des 10 premiers entiers, le résultat doit être stocké à l'adresse $4000

Prog 20 : Programme capable de stocker les 100 premiers nombres entiers dans le bloc mémoire dont la première adresse est $1200

Prog 21 : Addition sur 8 bits

Prog 22 : Addition sur 16 bits

Prog 23 : Addition sur 16 bits (variante avec l'accumulateur D)

Prog 24 : Addition sur 32 bits (avec l'accumulateur D)

Prog 25 : Recherche de la valeur $40 ('@') dans un tableau de 100 éléments

Prog 26 : Addition en 16 bits

Prog 27 : Comptage des données positives, négatives et nulles d'une table de nombres signés de 8 bits.

Prog 28 : Soustraction en 16 bits

Prog 29 : Multiplication de nombres de 16 bits

Prog 01 : Création d'une table de données

Prog 01 : Sujet

Une table de données consiste en une liste de données quelconques logées en mémoire à des adresses successives. L'adresse de la première donnée est qualifiée d'adresse de base de la table.

Prog 01 : Question A

Proposer un programme permettant de ranger en mémoire dans l'ordre croissant l'ensemble des données 8 bits non signées à partir de l'adresse de base $0100.
Commentaires

La plage des nombres non signés s'étend de $00 à $FF. Il faudra donc charger la mémoire avec ces 256 valeurs.

Prog 01 : Programme A

Création d'une table de données en bits non signés

```
0000                          ORG   $0000      ; Début du programme
0000 8E   0100                LDX   #$0100     ; Début de table
0003 86   00                  LDA   #$00       ; 1ere données $00
0005 A7   80            Boucle STA   ,X+       ; Chargement et incrémentation du pointeur
0007 81   FF                  CMPA  #$FF       ; Dernière donnée = $FF alors fin de programme
0009 27   03      000E        BEQ   Fin        ;
000B 4C                       INCA             ; Incrémentation de la donnée
000C 20   F7      0005        BRA   Boucle     ;
000E 3F                 Fin   SWI              ;
```

Etat de la mémoire après exécution du programme

```
0100 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0110 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
0120 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
0130 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
0140 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
0150 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
0160 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
0170 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
0180 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
0190 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
01A0 A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
01B0 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
01C0 C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
01D0 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
01E0 E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
01F0 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
```

Prog 01 : Question B

Faire la même chose pour l'ensemble des données 8 bits signées à partir de l'adresse de base $0200.

Prog 01 : Commentaires B

Il faudra en premier lieu charger la mémoire avec les nombres négatifs en décrémentant de $FF à$80, puis charger les nombres positifs en incrémentant de $00 à$7F.

Prog 01 : Programme B

Création d'une table de données en bits signés

```
0000                          ORG    $0000     ; Début du programme
0000 8E   0200                LDX    #$0200    ; Début 1ere donnée négative
0003 108E 0280                LDY    #$0280    ; Début 1ere donnée positive
0007 86   FF                  LDA    #$FF      ; 1ere donnée négative $FF
0009 A7   80            BOUCLE STA    ,X+       ; Chargement et incrémentation du pointeur X
000B 81   80                  CMPA   #$80      ; Si donnée = $80 fin des données négatives
000D 27   03      0012        BEQ    POSITIF   ;
000F 4A                       DECA             ; Décrémentation de la donnée
0010 20   F7      0009        BRA    BOUCLE    ;
0012 86   00            POSITIF LDA    #$00      ; 1ere donnée positive
0014 A7   A0            BOUCLE1 STA    ,Y+       ; Chargement et incrémentation du pointeur
0016 81   7F                  CMPA   #$7F      ; Si donnée = $7F fin des données positives
0018 27   03      001D        BEQ    FIN       ;
001A 4C                       INCA             ; Incrémentation de la donnée
001B 20   F7      0014        BRA    BOUCLE1   ;
001D 3F                 FIN   SWI              ;
```

Etat de la mémoire après exécution du programme

```
0200 FF FE FD FC FB FA F9 F8 F7 F6 F5 F4 F3 F2 F1 F0
0210 EF EE ED EC EB EA E9 E8 E7 E6 E5 E4 E3 E2 E1 E0
0220 DF DE DD DC DB DA D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
0230 CF CE CD CC CB CA C9 C8 C7 C6 C5 C4 C3 C2 C1 C0
0240 BF BE BD BC BB BA B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
0250 AF AE AD AC AB AA A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
0260 9F 9E 9D 9C 9B 9A 99 98 97 96 95 94 93 92 91 90
0270 8F 8E 8D 8C 8B 8A 89 88 87 86 85 84 83 82 81 80
0280 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0290 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
02A0 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
02B0 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
02C0 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
02D0 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
02E0 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
02F0 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
```

Prog 02 : Dénombrement de données spécifiques dans une table

Prog 02 : Sujet

On souhaite, dans ce problème, évaluer le nombre de données d'une table qui répondent à une même caractéristique.

Prog 02 : Question A

Proposer un programme permettant d'effectuer le comptage des données positives, négatives et nulles d'une table de nombres signés de 8 bits. Le programme devra permettre de stocker ces résultats aux adresses $0050, $0051,$0052 par exemple.

Prog 02 : Commentaires A

Après avoir chargé la valeur dans le registre A, qui automatiquement positionne les bits N et Z, on peut utiliser les instructions de branchements qui en découlent.

Prog 02 : Programme A

Tri de données positives, négatives ou nulle

```
0000                          ORG    $0000      ;
                1000  TABLE   EQU    $1000      ; Déclaration du début de table
                1009  FIN_TAB EQU    $1009      ; Déclaration du pointeur de fin de table
0000                          ORG    $0000      ; Début du programme
0000 8E   1000                LDX    #TABLE     ; Chargement du pointeur
0003 8C   100A         Boucle CMPX   #FIN_TAB+1 ; Si le pointeur dépasse la fin de la table
0006 27   21      0029        BEQ    FIN        ; alors FIN
0008 A6   80                  LDA    ,X+        ; Chargement et incrémentation du pointeur
000A 2B   0B      0017        BMI    Negatif    ; Si l'opération est négative -> Négatif
000C 27   12      0020        BEQ    Nul        ; Si A = 0 -> Nul
000E F6   0050                LDB    >$0050     ; Sinon la données est positive
0011 5C                       INCB              ; Incrémente le compteur situé en $0050
0012 F7   0050                STB    >$0050     ; On mémorise la valeur
0015 20   EC      0003        BRA    Boucle     ;
0017 F6   0051         Negatif LDB    >$0051     ; La données est négative
001A 5C                       INCB              ; Incrémente le compteur situé en $0051
001B F7   0051                STB    >$0051     ; On mémorise la valeur
001E 20   E3      0003        BRA    Boucle     ;
0020 F6   0052         Nul     LDB    >$0052     ; La données est nulle
0023 5C                       INCB              ; Incrémente le compteur situé en $0052
0024 F7   0052                STB    >$0052     ; On mémorise la valeur
0027 20   DA      0003        BRA    Boucle     ;
0029 3F                FIN    SWI               ;
                                                ;
1000                          ORG    $1000      ; Début de la TABLE
1000 FF FF 00 05              FCB    -1,-1,0,5  ;
1004 08 F9 00 F7              FCB    8,-7,0,-9  ;
1008 02 06                    FCB    2,6        ;
```

Etat de la mémoire après exécution du programme

Résultats du dénombrement

```
0050 04 04 02 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Table des données

```
1000 FF FF 00 05 08 F9 00 F7 02 06 00 00 00 00 00 00
```

Prog 02 : Question B

Proposer un programme permettant d'effectuer le comptage du nombre de données paires et impaires d'une table.

Prog 02 : Commentaires B

Pour connaître la parité d'un mot de 8 bit, il suffit de faire un ET logique entre le mot et $11. Si le résultat est zéro alors le nombre est pair, sinon il est impair.

Prog 02 : Programme B

Tri de données paires ou impaires

```
                1000  TABLE   EQU    $1000      ; Déclaration du début de table
                1009  FIN_TAB EQU    $1009      ; Déclaration du pointeur de fin de table
0000                          ORG    $0000      ; Début du programme
0000 8E   1000                LDX    #TABLE     ; Chargement du pointeur
0003 8C   100A          Boucle  CMPX   #FIN_TAB+1 ; Si le pointeur dépasse la fin de la table
0006 27   1A      0022        BEQ    FIN        ; alors FIN
0008 A6   80                  LDA    ,X+        ; Chargement et incrémentation du pointeur
000A 84   11                  ANDA   #$11       ; ET logique pour connaître la parité
000C 81   00                  CMPA   #$00       ; Si A = 0 la donnée est paire -> Pair
000E 27   09      0019        BEQ    Pair       ;
0010 F6   0050                LDB    >$0050     ; Sinon la donnée est impaire
0013 5C                       INCB              ; Incrémentation du compteur
0014 F7   0050                STB    >$0050     ; Mémorisation du compteur
0017 20   EA      0003        BRA    Boucle     ;
0019 F6   0051          Pair  LDB    >$0051     ; La donnée est paire
001C 5C                       INCB              ; Incrémentation du compteur
001D F7   0051                STB    >$0051     ; Mémorisation du compteur
0020 20   E1      0003        BRA    Boucle     ;
0022 3F                 FIN   SWI              ;
                              ;
1000                          ORG    $1000      ; Début de la TABLE
1000 01 02 03 04              FCB    1,2,3,4,5  ;
1004 05                       ;
1005 06 07 08 09              FCB    6,7,8,9,0  ;
1009 00                       ;
```

Etat de la mémoire après exécution du programme

Résultat du dénombrement

```
0050 05 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Table des données

```
1000 01 02 03 04 05 06 07 08 09 00 00 00 00 00 00 00
```

Prog 02 : Question C

Proposer un programme permettant de compter le nombre de données d'une table dont le bit b3 est égal à 1.

Prog 02 : Commentaires C

Pour connaître l'état du bit 3 d'un nombre de 8 bit, il suffit de faire un ET logique entre ce mot et $08, si le résultat est égal à 0, le bit 3 est à 0, sinon le bit 3 est à 1.

Prog 02 : Programme C

Tri de données suivant la valeur du bit 3 de la donnée

```
                1000  TABLE   EQU    $1000      ; Déclaration du début de table
                1009  FIN_TAB EQU    $1009      ; Déclaration du pointeur de fin de table
0000                          ORG    $0000      ; Début du programme
0000 8E   1000                LDX    #TABLE     ; Chargement du pointeur Boucle
0003 8C   100A          Boucle  CMPX   #FIN_TAB+1 ; Si le pointeur dépasse la fin de la table
0006 27   11      0019        BEQ    FIN        ; alors FIN
0008 A6   80                  LDA    ,X+        ; Chargement et incrémentation du pointeur
000A 84   08                  ANDA   #$08       ; ET logique avec $08 pour savoir si bit3=1
000C 81   00                  CMPA   #$00       ; Si A = 0 bit3=0 ?> Boucle
000E 27   F3      0003        BEQ    Boucle     ;
0010 F6   0050                LDB    >$0050     ; Sinon bit3=1
0013 5C                       INCB              ; Incrémentation du compteur
```

```
0014 F7   0050                STB    >$0050     ; Mémorisation du compteur
0017 20   EA      0003        BRA    Boucle     ;
0019 3F                 FIN   SWI               ;
                                                ;
1000                          ORG    $1000      ; Début de la TABLE
1000 01 02 03 04              FCB    1,2,3,4,5  ;
1004 05                                         ;
1005 06 07 08 09              FCB    6,7,8,9,0  ;
1009 00                                         ;
```

Etat de la mémoire après exécution du programme

Résultat du dénombrement

```
0050 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Table des données

```
1000 01 02 03 04 05 06 07 08 09 00 00 00 00 00 00 00
```

Prog 03 : Multiplication

Prog 03 : Question

Soit le nombre hexadécimal X1=$23.
Mettre au point un programme permettant de trouver le nombre X2 tel que le produit X1*X2 soit strictement inférieur à $0299.

Prog 03 : Commentaire

Pour connaître X2, on incrémente un mot de 8 bits que l'on multiplie à $23, puis on teste le résultat à pour savoir s'il est supérieur ou égal à la valeur que l'on recherche. Si c'est le cas, la valeur de X2 est donc le mot de 8 bits –1, puisque l'on désire obtenir un résultat strictement inférieur.

Prog 03 : Programme

Recherche du résultat − 1 d'une division

```
0000                          ORG    $0000      ; Début du programme
                0000  Val01   EQU    #0         ;
0000 86   23                  LDA    #$23       ; Chargement de la valeur à multiplier X1
0002 C6   01                  LDB    #$01       ; Chargement de la 1ere valeur
0004 F7   1000          BOUCLE STB    $1000      ; Mise en mémoire de l'accumulateur B
0007 3D                       MUL                ; Multiplication de A par B
0008 1083 0299                CMPD   #$0299     ; Si A.B est inférieur ou égal à $0299
000C 24   08      0016        BHS    RESULT     ; alors RESULT
000E F6   1000                LDB    >$1000     ; Recharge de l'accumulateur B
0011 5C                       INCB               ; Incrémentation de l'accumulateur B
0012 86   23                  LDA    #$23       ; Recharge de l'accumulateur A
0014 20   EE      0004        BRA    BOUCLE     ;
0016 F6   1000          RESULT LDB    >$1000     ; Recharge de l'accumulateur B
0019 5A                       DECB               ; Décrémentation de l'accumulateur B
001A 3F                       SWI                ;
```

Etat de la mémoire après exécution du programme

Résultat en $1000

```
1000 13 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Etat du registre B après exécution du programme

B = 12 C'est le résultat que l'on cherchait !

Prog 04 : Détermination du maximum ou du minimum d'une table

Prog 04 : Question A

On dispose d'une table de 10 données de 8 bits choisis arbitrairement. Proposer un programme de recherche de la donnée maximale et de la donnée minimale de la liste, les nombres considérés étant non signés.

Prog 04 : Commentaire A

Pour connaître le MIN et le MAX d'une table, on enregistre d'abord la 1ère donnée dans MIN et dans MAX, puis on vient les comparer avec la valeur suivante. Si la nouvelle donnée est plus grande que la valeur contenue dans MAX, on met la nouvelle valeur dans MAX, on procède de manière identique pour la valeur MIN.
Dans le cas ou la valeur n'est ni un MIN, ni un MAX, on pointe sur la valeur suivante de la table.

Prog 04 : Programme A

Tri de données MAX et MIN en non signé

```
                  0050  MIN     EQU    $0050      ; Déclaration de l'adresse du MAX
                  0060  MAX     EQU    $0060      ; Déclaration de l'adresse du MIN
                  1200  TABLE   EQU    $1200      ; Déclaration du pointeur de fin de table
0000                            ORG    $0000      ; Début du programme
0000 8E   1200                  LDX    #TABLE     ; Chargement du pointeur
0003 A6   80                    LDA    ,X+        ; Chargement et incrémentation du pointeur
0005 B7   0060                  STA    >MAX       ; Mémorise la 1ere valeur dans MAX
0008 B7   0050                  STA    >MIN       ; Mémorise la 1ere valeur dans MIN
000B 8C   120A          Boucle  CMPX   #TABLE+10  ; Si le pointeur dépasse la fin de la table
000E 27   1E      002E          BEQ    FIN        ; alors FIN
0010 A6   84                    LDA    ,X         ; Chargement et incrémentation du pointeur
0012 B1   0060                  CMPA   >MAX       ; Si A > MAX ?> HightBHI Hight
0015 A6   84                    LDA    ,X         ; Chargement et incrémentation du pointeur
0017 B1   0050                  CMPA   >MIN       ; Si A < MIN ?> Low
001A 25   0B      0027          BLO    Low        ;
001C A6   80                    LDA    ,X+        ; Chargement et incrémentation du pointeur
001E 20   EB      000B          BRA    Boucle     ;
0020 A6   80            Hight   LDA    ,X+        ; Chargement et incrémentation du pointeur
0022 B7   0060                  STA    >MAX       ; Mémorise la valeur dans MAX
0025 20   E4      000B          BRA    Boucle     ;
0027 A6   80            Low     LDA    ,X+        ; Chargement et incrémentation du pointeur
0029 B7   0050                  STA    >MIN       ; Mémorise la valeur dans MIN
002C 20   DD      000B          BRA    Boucle     ;
002E 3F                 FIN     SWI                ;
                                                   ;
1200                            ORG    $1200      ; Début de la TABLE
1200 02 02 03 04                FCB    2,2,3,4,5  ;
1204 05                                            ;
1205 00 07 07 07                FCB    0,7,7,7,7  ;
1209 07                                            ;
```

Etat de la mémoire après exécution du programme

```
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Valeur MIN = 0

```
0060 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Valeur MAX = 7 Table de données

```
1200 02 02 03 04 05 00 07 07 07 07 00 00 00 00 00 00
```

Prog 04 : Question B

Compléter ce programme de sorte qu'il soit capable de déterminer également le maximum et le minimum lorsque les données sont signées.

Prog 04 : Commentaire B

La méthode générale est la même que l'exercice précédent, seules les instructions de branchement BGT et BLT sont modifiées pour travailler sur des données signées.

Prog 04 : Programme B

Tri de données MAX et MIN en signé

```
                  0050  MIN     EQU    $0050      ; Déclaration de l'adresse du MAX
                  0060  MAX     EQU    $0060      ; Déclaration de l'adresse du MIN
                  1200  TABLE   EQU    $1200      ; Déclaration du pointeur de fin de table
                                                   ;
0000                            ORG    $0000      ; Début du programme
0000 8E   1200                  LDX    #TABLE     ; Chargement du pointeur
0003 A6   80                    LDA    ,X+        ; Chargement et incrémentation du pointeur
0005 B7   0060                  STA    >MAX       ; Mémorise la 1ere valeur dans MAX
0008 B7   0050                  STA    >MIN       ; Mémorise la 1ere valeur dans MIN
000B 8C   120A          Boucle  CMPX   #TABLE+10  ; Si le pointeur dépasse la fin de la table
000E 27   20      0030          BEQ    FIN        ; alors FIN
0010 A6   84                    LDA    ,X         ; Chargement et incrémentation du pointeur
0012 B1   0060                  CMPA   >MAX       ; Si A > MAX -> Hight
0015 2E   0B      0022          BGT    Hight      ;
0017 A6   84                    LDA    ,X         ; Chargement et incrémentation du pointeur
0019 B1   0050                  CMPA   >MIN       ; Si A < MIN -> Low
001C 2D   0B      0029          BLT    Low        ;
001E A6   80                    LDA    ,X+        ; Chargement et incrémentation du pointeur
0020 20   E9      000B          BRA    Boucle     ;
0022 A6   80            Hight   LDA    ,X+        ; Chargement et incrémentation du pointeur
```

```
0024 B7   0060                STA    >MAX       ; Mémorise la valeur dans MAX
0027 20   E2      000B        BRA    Boucle     ;
0029 A6   80            Low   LDA    ,X+        ; Chargement et incrémentation du pointeur
002B B7   0050                STA    >MIN       ; Mémorise la valeur dans MIN
002E 20   DB      000B        BRA    Boucle     ;
0030 3F                 FIN   SWI                ;
                                                 ;
1200                          ORG    $1200      ; Début de la TABLE
1200 FE 02 03 FC              FCB    -2,2,3,-4  ;
1204 05 00 07 07              FCB    5,0,7,7    ;
1208 07 07                    FCB    7,7        ;
```

Etat de la mémoire après exécution du programme

```
0050 FC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Valeur MIN = FC soit 4

```
0060 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Valeur MAX = 07 soit 7 Table de données

```
1200 FE 02 03 FC 05 00 07 07 07 07 00 00 00 00 00 00
```

Prog 05 : Transfert d'une table de données d'une zone mémoire vers une autre

Prog 05 : Question A

On dispose d'une table de 10 données de 8 bits, choisies arbitrairement, dont l'adresse de base est ADR1. Proposer un programme permettant de transférer cette table à l'adresse de base ADR2.

Prog 05 : Commentaire A

La méthode utilisée ici consiste à charger une valeur dans le registre A en se servant du pointeur X (identifiant de la table source), et de stoker cette valeur à l'adresse désignée par le pointeur Y (identifiant la table de destination).

Prog 05 : Programme A

Transfert d'une table de 10 données de `ADR1= $0050 ---> ADR2=$0060`

```
                0050  ADR1    EQU    $0050      ; Déclaration de l'adresse ADR1
                0060  ADR2    EQU    $0060      ; Déclaration de l'adresse ADR2
                                                 ;
0000                          ORG    $0000      ;
0000 8E   0050                LDX    #ADR1      ; Chargement du pointeur X
0003 108E 0060                LDY    #ADR2      ; Chargement du pointeur Y
0007 A6   80            Boucle LDA    ,X+        ; Chargement et incrémentation du pointeur X
0009 A7   A0                  STA    ,Y+        ; Chargement et incrémentation du pointeur Y
000B 8C   005A                CMPX   #ADR1+10   ; Si le pointeur dépasse la fin de la table
000E 26   F7      0007        BNE    Boucle     ; alors FIN
0010 3F                       SWI                ;
                                                 ;
0050                          ORG    $0050      ;
0050 00 01 09 03              FCB    0,1,9,3,4  ;
0054 04                                          ;
0055 05 02 07 05              FCB    5,2,7,5,9  ;
0059 09                                          ;
```

Etat de la mémoire après exécution du programme

Table à l'adresse ADR1

```
0050 00 01 09 03 04 05 02 07 05 09 00 00 00 00 00 00
```

Table à l'adresse ADR2

```
0060 00 01 09 03 04 05 02 07 05 09 00 00 00 00 00 00
```

Prog 05 : Question B

Proposer un programme permettant de ranger les nombres hexadécimaux $00 à $09 aux adresses $0100 à $0109 et leur complément à 1 aux adresses $0200 à $0209.
Sur le même principe, proposer un programme qui n'utilise qu'un seul pointeur.

Prog 05 : Commentaire B

Ce programme comporte une petite difficulté car l'utilisation d'un seul pointeur implique que pour pointer sur la table2 d'adresse de base $0200, il faut rajouter au pointeur un déplacement égal à $FF ($100−1).
Car l'adresse 2 est décalée de $0100 dans la mémoire par rapport à l'adresse de base de la table 1 (la soustraction du 1 vient du fait de l'auto incrémentation de 1) est induite par le dernier chargement du registre A.

Prog 05 : Programme B

Transfert d'une table de 10 données de ADR1 -> ADR2, 1 pointeur

```
                0100  TABLE   EQU    $0100      ; Déclaration de l'adresse de TABLE
                                                ;
0000                          ORG    $0000      ;
0000 8E   0100                LDX    #TABLE     ; Chargement du pointeur X
0003 86   00                  LDA    #$00       ; Initialisation de l'accumulateur A
0005 A7   80            Boucle STA    ,X+        ; Mémorisation de A à l'adresse pointée par X
0007 43                       COMA               ; Complément à 1 de A
0008 A7   89 00FF             STA    $FF,X      ; Mémorisation de A à l'adresse X+$FF
000C 43                       COMA               ; Complément à 1 de A ?> valeur initiale
000D 4C                       INCA               ; Incrémentation de la valeur de A
000E 81   0A                  CMPA   #$0A       ; Si A = $0A
0010 27   02      0014        BEQ    FIN        ; alors FIN
0012 20   F1      0005        BRA    Boucle     ;
0014 3F                 FIN   SWI                ;
```

Etat de la mémoire après exécution du programme

Table de données en ADR1

```
0100 00 01 02 03 04 05 06 07 08 09 00 00 00 00 00 00
```

Complément à 1 en ADR2

```
0200 FF FE FD FC FB FA F9 F8 F7 F6 00 00 00 00 00 00
```

Prog 05 : Question C

On dispose maintenant de deux tables de 10 données de 16 bits choisis arbitrairement. ADR1 et ADR2 sont les adresses de base de ces tables. On souhaite construire une troisième table, d'adresse de base ADR3, dont chaque élément résulte de l'addition des éléments de même rang des deux premières tables. Proposer le programme correspondant.

Prog 05 : Commentaire C

Le fait d'avoir trois tables de données, impose d'utiliser le pointeur X, pour pointer à la fois la table 1 et la table 2. Le déplacement rajouté à X, sera calculé de la manière suivante (ADR2 ADR1 2), la soustraction de 2 est ici dûe au fait que nous travaillons sur des données de 16 bits, de même les auto-incrémentations sont aussi sur 16 bits. Le pointeur Y quand a lui sert à pointer sur l'adresse de la table 3.

Prog 05 : Programme C

Addition de ADR1 + ADR2 -> ADR3, données de 16 bits

```
                0050  ADR1    EQU    $0050           ; Déclaration de l'adresse ADR1
                0090  ADR3    EQU    $0090           ; Déclaration de l'adresse ADR3
                                                     ;
0000                          ORG    $0000           ;
0000 8E   0050                LDX    #ADR1           ; Chargement du pointeur X
0003 108E 0090                LDY    #ADR3           ; Chargement du pointeur Y
0007 EC   81            Boucle LDD    ,X++            ; Chargement de D et incrément de 2
0009 E3   88 1E               ADDD   $1E,X           ; Addition de D avec le contenue de X+$1E
000C ED   A1                  STD    ,Y++            ; Mémorisation de D et incrément de 2
000E 8C   0064                CMPX   #ADR1+20        ; Si X = ADR1+20 alors fin de la table
0011 26   F4      0007        BNE    Boucle          ; et du programme
0013 3F                       SWI                    ;
                                                     ;
0050                          ORG    $0050           ; Début de la ADR1
0050 00 00 00 01              FCB    $00,$00,$00,$01 ;
0054 10 13 52 30              FCB    $10,$13,$52,$30 ;
0058 56 89 21 54              FCB    $56,$89,$21,$54 ;
005C 14 25 01 25              FCB    $14,$25,$01,$25 ;
0060 87 28 45 78              FCB    $87,$28,$45,$78 ;
                                                     ;
0070                          ORG    $0070           ; Début de la ADR2
0070 01 01 01 01              FCB    $01,$01,$01,$01 ;
0074 01 01 01 01              FCB    $01,$01,$01,$01 ;
0078 01 01 01 01              FCB    $01,$01,$01,$01 ;
007C 01 01 01 01              FCB    $01,$01,$01,$01 ;
0080 01 01 01 01              FCB    $01,$01,$01,$01 ;
```

Etat de la mémoire après exécution du programme

Table 1 à l'adresse ADR1

```
0050 00 00 00 01 10 13 52 30 56 89 21 54 14 25 01 25
0060 87 28 45 78 00 00 00 00 00 00 00 00 00 00 00 00
```

Table 2 à l'adresse ADR2

```
0070 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
0080 01 01 01 01 00 00 00 00 00 00 00 00 00 00 00 00
```

ADR1+ADR2 à l'adresse ADR3

```
0090 01 01 01 02 11 14 53 31 57 8A 22 55 15 26 02 26
00A0 88 29 46 79 00 00 00 00 00 00 00 00 00 00 00 00
```

Prog 06 : Détermination logicielle de la parité croisée d'une table de données

Prog 06 : Question A

On dispose d'une table de 10 données correspondant à des caractères ASCII (codés sur 7 bits).Proposer un programme permettant de déterminer la clé de parité paire de chaque élément de la table et, le cas échéant, de rajouter cette clé devant la donnée.

Prog 06 : Exemple A

Supposons que le premier élément de la table soit le caractère ASCII " a ", qui se traduit par la combinaison 110 0001(soit $61 en hexadécimal). La clé de parité paire de ce caractère étant égale à 1, le programme devra permettre de modifier la donnée $61 et de la remplacer par $E1.

Prog 06 : Commentaire A

Pour connaître la clé de parité paire d'un nombre sur 7 bits, il faut compter le nombre de bit à 1 qui le compose, pour cela nous avons utilisé le décalage logique à gauche qui a la particularité de faire rentrer le bit de poids fort dans le bit C (CARRY), et lors du décalage à gauche d'insérer un zéro dans le bit de poids faible.

Il suffira ensuite d'incrémenter un compteur de " 1 ", l'opération s'arrêtera quand le nombre traité dans le registre A sera égal à $00.

Une fois ceci fait il faudra déterminer si le nombre de 1 est pair ou impair cf. ED1−P2−Q2, enfin dans le cas ou celui−ci serait impair pour mettre à 1 le bit 8 du nombre il suffira de faire un OU logique entre le nombre et $80. Et pour finir de stocker le nouveau nombre en remplacement de l'ancien.

Prog 06 : Programme A

Détermination logicielle de la parité d'un mot de 8 bits

```
                  0080  CPTEUR  EQU    $0080          ; Déclaration de la variable COMPTEUR
                  00A0  TABLE   EQU    $00A0          ; Déclaration de la table TABLE
                                                      ;
0000                            ORG    $0000          ; Début du programme
0000 8E   00A0                  LDX    #TABLE         ; Chargement du pointeur X
0003 5F                         CLRB                  ; RAZ de B
0004 0F   80                    CLR    CPTEUR         ; RAZ de COMPTEUR
0006 A6   80            DATA    LDA    ,X+            ; Chargement et incrémentation de X
0008 8C   00AB                  CMPX   #TABLE+11      ; Si on a atteint la fin de la table
000B 27   21      002E          BEQ    FIN            ; alors FIN
000D 48                 BOUCLE  LSLA                  ; Décalage logique à gauche de A
000E 25   06      0016          BCS    INCREM         ; Branchement si Carry = 1
0010 81   00            RETOUR  CMPA   #$00           ; Comparaison de A avec $00
0012 27   06      001A          BEQ    PARITE         ; Si A = $00 -> PARITE
0014 20   F7      000D          BRA    BOUCLE         ; sinon -> BOUCLE
0016 0C   80            INCREM  INC    CPTEUR         ; Incrément de CPTEUR
0018 20   F6      0010          BRA    RETOUR         ; Aller à RETOUR
001A 96   80            PARITE  LDA    CPTEUR         ; Chargement de A avec CPTEUR
001C 84   11                    ANDA   #$11           ; ET logique entre A et $11 pour
                                                      ;          déterminer la parité
001E 81   00                    CMPA   #$00           ; Comparaison de A avec $00
0020 26   02      0024          BNE    IMPAIR         ; Si A est différent de $00 -> IMPAIR
0022 20   E2      0006          BRA    DATA           ; Sinon -> DATA
0024 A6   82            IMPAIR  LDA    ,-X            ; Chargement et incrémentation de X
0026 8A   80                    ORA    #$80           ; OU logique entre A et $08
0028 A7   80                    STA    ,X+            ; Mémo. de A et incrément pointeur
002A 0F   80                    CLR    CPTEUR         ; RAZ COMPTEUR
002C 20   D8      0006          BRA    DATA           ; Retour à DATA
002E 3F                 FIN     SWI                   ; Fin du programme
                                                      ;
00A0                            ORG    $00A0          ; Début de la table TABLE
00A0 61 62 63                   FCB    'a','b','c'    ;
00A3 61 62 63                   FCB    'a','b','c'    ;
00A6 61 62 63 64                FCB    'a','b','c','d' ;
```

Etat de la mémoire avant exécution du programme

Table de données

```
00A0 61 62 63 61 62 63 61 62 63 64 00 00 00 00 00 00
```

Etat de la mémoire après exécution du programme

Données + clé de parité

```
00A0 E1 E2 63 E1 E2 63 E1 E2 63 E4 00 00 00 00 00 00
```

Prog 06 : Question B

Le programme précédent permettant d'adjoindre aux données un contrôle transversal de la parité, le modifier de sorte qu'il soit également susceptible de déterminer puis d'ajouter à la fin de la table un octet de vérification de la parité longitudinale.

Prog 06 : Commentaire B

Le programme qui suit est quasiment similaire au précédant, il a juste fallu rajouter un compteur transversal, c'est à dire qui s'incrémente à chaque fois que l'on remplace une valeur dans la table d'origine en ajoutant une clé de parité. Il suffit ensuite de tester la parité de ce compteur et de rajouter en fin de table un 1 si il est impair, un zéro si il est pair.

Prog 06 : Programme B

Détermination logicielle de la parité croisée d'une table de données

```
                  0080  CPTEUR  EQU    $0080          ; Déclaration de la variable CPTEUR
                  0081  CPTEURT EQU    $0081          ; Déclaration de la variable CPTEURT
                  00A0  TABLE   EQU    $00A0          ; Déclaration de la table TABLE
                                                      ;
0000                              ORG    $0000          ; Début du programme
0000 8E   00A0                    LDX    #TABLE         ; Chargement du pointeur X
0003 5F                           CLRB                  ; RAZ de B
0004 0F   80                      CLR    CPTEUR         ; RAZ de CPTEUR
0006 A6   80            DATA     LDA    ,X+            ; Chargement et incrémentation de X
0008 8C   00AB                    CMPX   #TABLE+11      ; Si on a atteint la fin de la table
000B 27   23      0030            BEQ    TRANS          ; alors -> TRANS
000D 48                 BOUCLE    LSLA                  ; Décalage logique à gauche de A
000E 25   06      0016            BCS    INCREM         ; Branchement si Carry = 1
0010 81   00            RETOUR    CMPA   #$00           ; Comparaison de A avec $00
0012 27   06      001A            BEQ    PARITE         ; Si A = $00 -> PARITE
0014 20   F7      000D            BRA    BOUCLE         ; Sinon retour à BOUCLE
0016 0C   80            INCREM    INC    CPTEUR         ; Incrémentation de CPTEUR
0018 20   F6      0010            BRA    RETOUR         ; Retour à BOUCLE
001A 96   80            PARITE    LDA    CPTEUR         ; Charge A avec le contenue de VALEUR
001C 84   11                      ANDA   #$11           ; ET logique entre A et $11
                                                      ;
001E 81   00                      CMPA   #$00           ; Comparaison de A avec $00
0020 26   02      0024            BNE    IMPAIR         ; Si A est différent de 0 -> IMPAIR
0022 20   E2      0006            BRA    DATA           ; Sinon retour à DATA
0024 A6   82            IMPAIR    LDA    ,-X            ; Décrémentation de X et chargement de A
0026 8A   80                      ORA    #$80           ; OU logique entre A et $08
0028 A7   80                      STA    ,X+            ; Mémorisation de A
002A 0C   81                      INC    CPTEURT        ; Incrémentation de CPTEURT
002C 0F   80                      CLR    CPTEUR         ; RAZ de CPTEUR
002E 20   D6      0006            BRA    DATA           ; Retour à DATA
0030 96   81            TRANS     LDA    CPTEURT        ; Chargement de A avec CPTEUR
0032 84   11                      ANDA   #$11           ; ET logique entre A et $11
0034 81   00                      CMPA   #$00           ; Comparaison entre A et $00
0036 26   06      003E            BNE    IMPAIRT        ; Si A est différent de $00 -> IMPAIR
0038 86   00                      LDA    #$00           ; Chargement de A avec $00
003A 97   AA                      STA    TABLE+10       ; Mémorisation de A en TABLE+10
003C 20   04      0042            BRA    FIN            ; -> Fin du programme
003E 86   01            IMPAIRT   LDA    #$01           ; Chargement de la valeur $01 dans A
0040 97   AA                      STA    TABLE+10       ; Mémorisation de A en TABLE+10
0042 3F                 FIN       SWI                   ; Fin du programme
                                                      ;
00A0                              ORG    $00A0          ;  Début de la table TABLE
00A0 61 63 61                     FCB    'a','c','a'    ;
00A3 63 61 61                     FCB    'c','a','a'    ;
00A6 61 61 61 61                  FCB    'a','a','a','a' ;
```

Etat de la mémoire avant exécution du programme

Table de données

```
00A0 61 63 61 63 61 61 61 61 61 61 00 00 00 00 00 00
```

Etat de la mémoire après exécution du programme

Données + clés de parité

```
00A0 E1 63 E1 63 E1 E1 E1 E1 E1 E1 01 00 00 00 00 00
```

NB: Clé de parité longitudinale

Prog 07 : Tri des données d'une table

Prog 07 : Question

On dispose d'une table de 10 données de 8 bits rangées initialement dans un ordre quelconque. Mettre au point un programme effectuant un tri des données et permettant après exécution de les ranger dans l'ordre croissant à partir de la même adresse de base.

Prog 07 : Commentaire

En tout premier lieu, on fait une copie de la table de données, puis on charge la 1ère valeur de la table d'origine que l'on compare aux valeurs du tableau de recopie. A chaque fois que la valeur trouvée est plus grande on incrémente la valeur COMPTEUR (qui représente la position de la valeur dans le tableau final).

Quand la table de données est entièrement balayée, on vérifie dans la table de position (TABLED) si l'emplacement est libre (00 = libre ; 01 = occupé) ; cette méthode de table de position est utilisé pour résoudre le problème de valeur identique, le COMPTEUR ayant dans ce cas la même valeur, on évite ainsi de réécrire la même valeur au même endroit, en incrémentant de 1 le COMPTEUR.

Si on remplace l'instruction de branchement BLO par BLT, ce programme est valable pour les nombres signés.

Prog 07 : Programme

Tri des données d'une table dans l'ordre croissant

```
                    0080  TABLE  EQU    $0080       ; Déclaration de la table TABLE
                    00A0  TABLEC EQU    $00A0       ; Déclaration de la table TABLEC
                    00C0  TABLED EQU    $00C0       ; Déclaration de la table TABLED
                    00D0  CPTEUR EQU    $00D0       ; Déclaration de la variable COMPTEUR
                    00E0  VALEUR EQU    $00E0       ; Déclaration de la variable VALEUR
                                                    ;
0000                              ORG    $0000       ; Début du programme
0000 8E   0080                    LDX    #TABLE      ; Chargement du pointeur X
0003 108E 00A0                    LDY    #TABLEC     ; Chargement du pointeur Y
0007 A6   80            Copy      LDA    ,X+         ; Chargement et incrémentation du pointeur X
0009 A7   A0                      STA    ,Y+         ; Chargement et incrémentation du pointeur Y
000B 8C   008A                    CMPX   #TABLE+10   ; Si le pointeur dépasse la fin de la table
000E 26   F7      0007            BNE    Copy        ; alors Copy
0010 5F                           CLRB               ;
0011 D6   D0            Boucle    LDB    CPTEUR      ; Chargement dans B du contenue de COMPTEUR
0013 8E   00A0                    LDX    #TABLEC     ; Chargement du pointeur X
0016 A6   85                      LDA    B,X         ; Chargement de A avec le contenue de X+B
0018 97   E0                      STA    VALEUR      ; Mémorisation de A à l'adresse VALEUR
001A C1   0A                      CMPB   #$0A        ; Si B = nombre de donnée de la table
001C 27   45      0063            BEQ    FIN         ; alors FIN
001E 5F                           CLRB               ; RAZ de l'accumulateur B
001F 8C   00AA          Data      CMPX   #TABLEC+10  ; Si pointeur X est égal à fin de table
0022 27   0B      002F            BEQ    Mem         ; alors -> Mem, mémorisation de la donnée
0024 A6   80                      LDA    ,X+         ; Chargement et incrémentation du pointeur X
0026 91   E0                      CMPA   VALEUR      ; Si A est strictement plus petit que VALEUR
0028 25   02      002C            BLO    Compt       ; alors -> Compt
002A 20   F3      001F            BRA    Data        ; Sinon -> Data
002C 5C                 Compt     INCB               ; Incrémentation de B
002D 20   F0      001F            BRA    Data        ; Retour à Data
002F 8E   00C0          Mem       LDX    #TABLED     ; Chargement du pointeur X
0032 A6   85                      LDA    B,X         ; Chargement de A avec le contenue de X+B
0034 81   01                      CMPA   #$01        ; Si A = $01
0036 27   12      004A            BEQ    Egal        ; alors pointeur déjà utilisé -> Egal
0038 96   E0                      LDA    VALEUR      ; Chargement de A avec le contenue de VALEUR
003A 8E   0080                    LDX    #TABLE      ; Chargement du pointeur X
003D A7   85                      STA    B,X         ; Mémorisation de A à l'adresse X+B
003F 8E   00C0                    LDX    #TABLED     ; Chargement du pointeur X
```

```
0042 86   01                LDA    #$01       ; Chargement de A avec $01
0044 A7   85                STA    B,X        ; Case mémoire utilisée -> $01
0046 0C   D0                INC    CPTEUR     ; Incrémentation de COMPTEUR
0048 20   C7      0011      BRA    Boucle     ; Retour à Boucle
004A 5C                Egal INCB              ; Incrémentation de B
004B A6   85                LDA    B,X        ; Chargement de A avec le contenue de X+B
004D 81   01                CMPA   #$01       ; Si A = $01
004F 27   F9      004A      BEQ    Egal       ; alors pointeur déjà utilisé -> Egal
0051 96   E0                LDA    VALEUR     ; Chargement de A avec le contenue de VALEUR
0053 8E   0080              LDX    #TABLE     ; Chargement du pointeur X
0056 A7   85                STA    B,X        ; Mémorisation de A à l'adresse X+B
0058 8E   00C0              LDX    #TABLED    ; Chargement du pointeur X
005B 86   01                LDA    #$01       ; Chargement de A avec $01
005D A7   85                STA    B,X        ; Case mémoire utilisée -> $01
005F 0C   D0                INC    CPTEUR     ; Incrémentation de COMPTEUR
0061 20   AE      0011      BRA    Boucle     ; Retour à Boucle
0063 3F                FIN  SWI                ; Fin du programme
                                               ;
0080                        ORG    $0080      ; Début de TABLE
0080 01 06 00 01            FCB    1,6,0,1,1  ;
0084 01                                        ;
0085 02 01 04 01            FCB    2,1,4,1,5  ;
0089 05                                        ;
```

Etat de la mémoire avant exécution du programme
Table de données

```
0080 01 06 00 01 01 02 01 04 01 05 00 00 00 00 00 00
```

Etat de la mémoire après exécution du programme Table de données après le tri

```
0080 00 01 01 01 01 01 02 04 05 06 00 00 00 00 00 00
```

Prog 08 : Détection et correction d'erreurs

Prog 08 : Sujet

Proposer un programme permettant après addition de deux données de 8 bits de vérifier la validité du résultat obtenu et, le cas échéant, de le corriger.

Prog 08 : Commentaires

Si A et B sont des données non signées de 8 bits elles peuvent prendre des valeurs allant de 0 à 255 et leur somme peut aller de 0 à 510. Ainsi, si l'on exprime le résultat sur 8 bits on court le risque de provoquer des dépassements de capacité.

Néanmoins, il est possible de vérifier la validité du calcul en testant la valeur de la retenue finale dans le registre CCR

-- si C = 0 le résultat non signé est correct sur 8 bits ;
-- si C = 1 il y a dépassement de capacité sur 8 bits (résultat correct sur 9 bits avec C = 9ème bit du résultat).

Dans ce dernier cas, il existe plusieurs possibilités pour corriger l'erreur commise, la meilleure consistant à exprimer le résultat avec un octet supplémentaire en considérant C comme le 9ème bit.
C'est cette méthode que nous retiendrons puisque la possibilité de travailler sur 16 bits par le biais de l'accumulateur D nous est offerte sur le 6809.

Si A et B sont des données signées de 8 bits elles peuvent prendre des valeurs allant de −128 à +127 en représentationCà2, et leur somme peut aller de −256 à +254.
Dans ce cas, l'addition de deux nombres de même signe et de valeurs élevées provoquera des dépassements de capacité sur 8 bits qui seront cette fois indiqués par le bit de débordement V du registre CCR

-- si V = 0 le résultat signé en Cà2 est correct sur 8 bits ;
-- si V = 1 dépassement de capacité en Cà2 sur 8 bits (résultat correct sur 9 bits avec C = bit de signe).

Pour corriger l'erreur on choisira également d'exprimer le résultat sur 16 bits ce qui nécessite d'effectuer une extension de signe qui consiste à recopier le bit C sur l'ensemble de l'octet de poids fort. Considérons par exemple l'addition suivante

```
11100011         (-29)
10000001       +(-127)
---------------------
101100100       (-156)
```

Résultat faux sur 8 bits (+100), correct sur 9 bits avec C = bit de signe (−156)

Pour exprimer ce résultat sur 16 bits il faut étendre le signe :

```
R = 11111111 01100100 = (-156).
```

Dans le programme proposé, les opérandes A et B à additionner sont placées aux adresses $0050 et $0051, le résultat de l'addition en non signé à l'adresse $0052 (et $0053 si la somme s'exprime sur l 6 bits) et le résultat de l'addition signée à l'adresse $0054 (et $0055 pour une somme sur 16 bits).

Par ailleurs, on choisit de placer le résultat de l'addition dans l'accumulateur B dans la mesure où celui−ci correspond à l'octet de poids faible de D (en cas d'erreur on rectifiera le contenu de A de manière à avoir le résultat correct dans D).

Pour tester la validité de la somme non signée on utilise un branchement conditionnel BCS qui aiguille le µP vers l'étiquette CORRUNS lorsque la retenue est égale à 1. La correction proprement dite consiste à faire pénétrer la retenue C dans l'accumulateur A par le biais de l'instruction ROLA.

Le résultat corrigé est ensuite placé en mémoire par stockage du contenu de D à l'adresse $0052.
Pour tester la validité de la somme signée on utilise un branchement BVS qui aiguille le µP vers l'étiquette CORRSIG lorsque le bit V est à 1.

Comme indiqué précédemment, la correction nécessite ici de faire une extension de signe par recopie du bit C.
L'instruction SEX permet de réaliser une extension de signe en transformant un nombre signé enCà2 de 8 bits en un nombre signé en Cà2 de 16 bits par recopie du bit de signe de B dans l'accumulateur A.

Autrement dit, il faut au préalable faire pénétrer la retenue C à la place du bit de signe dans B avant d'utiliser l'instruction SEX : C'est le rôle de l'instruction RORB (C−>b7 et b0 −>C).

Après l'extension, on veillera à récupérer le bit de poids faible de B par une instruction ROLB (C−>b0).
Enfin, le résultat corrigé est stocké à l'adresse $0054.

Prog 08 : Programme

```
0000                        ORG    $0000      ;
                 0010  Val  EQU    $10        ;
0000 4F                     CLRA              ;
0001 D6   50                LDB    $0050      ;
0003 DB   51                ADDB   $0051      ;
0005 25   0B      0012      BCS    CORRUNS    ;
0007 D7   52                STB    $0052      ;
0009 D6   50           SUITE LDB    $0050      ;
000B DB   51                ADDB   $0051      ;
000D 29   08      0017      BVS    CORRSIG    ;
000F D7   54                STB    $0054      ;
0011 3F                FIN  SWI               ;
                                              ;
                      ;-----Correction de l'addition non signée
0012 49               CORRUNS ROLA            ;
0013 DD   52                STD    $0052      ;
0015 20   F2      0009      BRA    SUITE      ;
                                              ;
                      ;-----Correction de l'addition signée
0017 56               CORRSIG RORB            ;
0018 1D                     SEX               ;
0019 59                     ROLB              ;
001A DD   54                STD    $0054      ;
001C 20   F3      0011      BRA    FIN        ;
                                              ;
0050                        ORG    $0050      ;
0050 BC 23             DATA FCB    $BC,$23    ;
```

Prog 09 : Table de correspondance hexadécimal décimal

Prog 09 : Question A

Analyser précisément le fonctionnement du programme proposé. Vous indiquerez ensuite la raison pour laquelle il ne fonctionne pas si, dans l'instruction de la ligne 3, on initialise le pointeur X par l'adresse $0150 par exemple.

Prog 09 : Programme A

```
0000                        ORG    $0000      ;
                 0010  Val  EQU    $10        ;
```

```
0000 4F                          CLRA            ;
0001 CE   0050                   LDU    #$0050   ;
0004 8E   0100                   LDX    #$0100   ;
0007 1F   10                     TFR    X,D      ;
0009 1E   89                     EXG    A,B      ;
000B A7   80            BOUCLE   STA    ,X+      ;
000D 8C   0164                   CMPX   #$0164   ;
0010 27   0E      0020           BEQ    FIN      ;
0012 4C                          INCA            ;
0013 36   02                     PSHU   A        ;
0015 84   0A                     ANDA   #$0A     ;
0017 81   0A                     CMPA   #$0A     ;
0019 37   02                     PULU   A        ;
001B 26   EE      000B           BNE    BOUCLE   ;
                                                 ;
001D 19                          DAA             ;
001E 20   EB      000B           BRA    BOUCLE   ;
0020 3F                 FIN      SWI             ;
                                 END             ;
```

Prog 09 : Commentaires A

Pour établir la table de correspondance, la méthode générale mise en œuvre dans le programme consiste à récupérer l'octet de poids faible de l'adresse de base de la table, puis de le placer dans l'accumulateur A et enfin à le ranger en mémoire.

Pour passer à la donnée suivante il suffit d'incrémenter de 1 le contenu de A puis de le ranger en mémoire.

Le premier problème se présente dès lors que le contenu de A est égal à $0A auquel cas il est nécessaire d'effectuer un ajustement décimal (instruction DAA) pour obtenir la traduction correcte. En effet, on ne dispose que de 10caractères (0 à 9) pour coder une donnée en décimal alors qu'il y en a 16 (0 à F) pour coder une donnée en hexadécimal.

Ainsi, pour traduire $0A, il faut lui ajouter $06 afin d'obtenir la correspondance adéquate $10. On peut ensuite continuer à incrémenter de 1 le contenu de A pour passer à la donnée suivante.

L'ajustement décimal devra être effectué chaque fois qu'apparaît le caractère "A" dans le nombre hexadécimal !

Ce programme ne répondra au fonctionnement souhaité que dans le cas où l'adresse de base de la table présente un octet de poids faible compris entre $00 et $09 dans la mesure où la première traduction correspond à une simple recopie de cet octet.

Prog 09 : Question B

Proposer une nouvelle version du programme assurant un fonctionnement correct quelle que soit l'adresse de base de la table.

Prog 09 : Commentaires B

Le principe du programme proposé pour respecter la contrainte imposée de même que précédemment, on commence par récupérer l'octet de poids faible de l'adresse de base de la table.

Pour traduire ce nombre en décimal il est nécessaire de savoir s'il correspond aux unités, aux dizaines, aux vingtaines, etc.

En effet, si on a affaire aux unités la traduction consiste en une simple recopie de l'octet de poids faible de l'adresse, si on a affaire aux dizaines il faut ajouter $06 à cet octet pour avoir son équivalent décimal, si on a affaire aux vingtaines il faut ajouter 2 * $06, etc.

Prog 09 : Programme B

```
0000                             ORG    $0000    ;
                  0126   ADRESSE  EQU    $0126    ;  Adresse de base de la table
0000 8E   0126                   LDX    #ADRESSE ;
0003 1F   10            BOUCLE   TFR    X,D      ;
0005 C1   09                     CMPB   #$09     ;
0007 22   08      0011           BHI    CORRIGE  ; Si octet poids faible est > $09 -> CORRIGE
                                                 ;
0009 E7   80            NEXT     STB    ,X+      ;
000B 8C   00A4                   CMPX   #0164    ;
000E 26   F3      0003           BNE    BOUCLE   ;
0010 3F                          SWI             ;
                                                 ;
0011 C1   13            CORRIGE  CMPB   #$13     ;
```

```
0013 22   04      0019           BHI    CORRIGE1   ; Si octet poids faible est > $13 ->CORRIGE1
0015 CB   06                     ADDB   #$06       ; sinon ajustement décimal
0017 20   F0      0009           BRA    NEXT       ;
                                                    ;
0019 C1   1D            CORRIGE1 CMPB   #$1D       ;
001B 22   04      0021           BHI    CORRIGE2   ; Si octet poids faible est > $1D ->CORRIGE2
001D CB   0C                     ADDB   #$0C       ; sinon double ajustement décimal
001F 20   E8      0009           BRA    NEXT       ;
                                                    ;
0021 C1   27            CORRIGE2 CMPB   #$27       ;
0023 22   04      0029           BHI    CORRIGE3   ; Si octet poids faible est > $27 ->CORRIGE3
0025 CB   12                     ADDB   #$12       ; sinon triple ajustement décimal
0027 20   E0      0009           BRA    NEXT       ;
                                                    ;
0029 C1   31            CORRIGE3 CMPB   #$31       ;
002B 22   04      0031           BHI    CORRIGE4   ;
002D CB   18                     ADDB   #$18       ;
002F 20   D8      0009           BRA    NEXT       ;
                                                    ;
0031 C1   3B            CORRIGE4 CMPB   #$3B       ;
0033 22   04      0039           BHI    CORRIGE5   ;
0035 CB   1E                     ADDB   #$1E       ;
0037 20   D0      0009           BRA    NEXT       ;
                                                    ;
0039 C1   45            CORRIGE5 CMPB   #$45       ;
003B 22   04      0041           BHI    CORRIGE6   ;
003D CB   24                     ADDB   #$24       ;
003F 20   C8      0009           BRA    NEXT       ;
                                                    ;
0041 C1   4F            CORRIGE6 CMPB   #$4F       ;
0043 22   04      0049           BHI    CORRIGE7   ;
0045 CB   2A                     ADDB   #$2A       ;
0047 20   C0      0009           BRA    NEXT       ;
                                                    ;
0049 C1   59            CORRIGE7 CMPB   #$59       ;
004B 22   04      0051           BHI    CORRIGE8   ;
004D CB   30                     ADDB   #$30       ;
004F 20   B8      0009           BRA    NEXT       ;
                                                    ;
0051 CB   36            CORRIGE8 ADDB   #$36       ;
0053 20   B4      0009           BRA    NEXT       ;
                                 END                ;
```

Prog 09 : Question C

Rédiger un programme de recherche de la traduction hexadécimale d'un nombre XX stocké en mémoire à l'adresse$0060.

Prog 09 : Commentaires C

Le programme de recherche de la traduction hexadécimale d'un nombre décimal XX quelconque (allant de 00 à 99) doit au préalable effectuer un rangement de la table de correspondance en mémoire.

Pour ce faire, nous utiliserons le programme proposé à la question 1 transformé ici en sous−programme.

La recherche proprement dite consistera alors en une lecture de la table jusqu'à ce que la donnée XX considérée soit décelée ; il suffit ensuite de récupérer l'octet de poids faible de son adresse dans la table pour avoir la traduction hexadécimale souhaitée.

Prog 09 : Programme C

```
0000                             ORG    $0000      ;
                   0100  ADR     EQU    $0100      ; Déf adresse de base de la table
0000 9D   13                     JSR    TABLE      ; Stockage de la table en mémoire
0003 8E   0013                   LDX    #TABLE     ;
0006 A6   80            LECTURE  LDA    ,X+        ; Lecture de la table
0008 91   60                     CMPA   $0060      ;
000A 26   FA      0006           BNE    LECTURE    ;
000C 30   1F                     LEAX   -1,X       ;
                                                    ;
000E 1F   10                     TFR    X,D        ;
0010 D7   61                     STB    $0061      ;
0012 3F                          SWI               ;
                                                    ;
```

```
                        ;-----S/prog Rangement de la table de correspondance en mémoire
0013 4F                 TABLE   CLRA            ;
0014 CE   0050                  LDU     #$0050  ;
0017 8E   0100                  LDX     #ADR    ;
001A 1F   10                    TFR     X,D     ;
001C 1E   89                    EXG     A,B     ;
001E A7   80            BOUCLE  STA     ,X+     ;
0020 8C   0164                  CMPX    #$0164  ;
0023 27   0E      0033          BEQ     FIN     ;
0025 4C                         INCA            ;
0026 36   02                    PSHU    A       ;
0028 84   0A                    ANDA    #$0A    ;
002A 81   0A                    CMPA    #$0A    ;
002C 37   02                    PULU    A       ;
002E 26   EE      001E          BNE     BOUCLE  ;
0030 19                         DAA             ;
0031 20   EB      001E          BRA     BOUCLE  ;
0033 39                 FIN     RTS             ;
                                                ;
0060                            ORG     $0060   ;
0060 23                 DATA    FCB     $23     ;
                                END             ;
```

Prog 09 : Remarque C

L'adresse de la donnée $23 dans la table est $0117 ; donc suite à l'exécution de ce programme, c'est la traduction hexadécimale $17 qui sera placée à l'adresse $0061

Prog 10 : Conversion DCB−binaire

Prog 10 : Question A

A l'aide de la méthode présentée en cours, établir un algorithme permettant de convertir un nombre entier codé en DCB sur 8 bits en binaire.

Prog 10 : Commentaires A

La conversion nécessite l'emploi de deux registres de 8 bits : l'un contenant initialement la donnée DCB à convertir, le second recueillant la donnée traduite en binaire.

Prog 10 : Algorithme A

```
A <- donnée DCB B <-0 Compteur <- 8
    Tant que (Compteur > 0)
        Décalage vers la droite des contenus
        combinés des registres A et B
        Si(a[3]= 1) alors
            A <- A - $03
        Fin si
        Compteur <- Compteur - 1
    Fin tant que
```

Prog 10 : Remarque A

Lorsqu'on parle de décalage à droite des contenus combinés des registres A et B cela signifie que le bit de poids faible de A doit être introduit comme bit de poids fort dans B.

Prog 10 : Question B

En vous appuyant sur l'algorithme précédent, proposer un programme réalisant la conversion d'un nombre DCB rangé en mémoire à l'adresse $0050 et stockant sa traduction binaire à l'adresse $0051.

Prog 10 : Programme B

```
0000                            ORG     $0000   ;
                    0010  Val   EQU     $10     ;
0000 96   50                    LDA     $0050   ; Charge dans A la donnée à traduire
0002 5F                         CLRB            ;
0003 8E   0008                  LDX     #$0008  ; L'index X, utilisé comme pointeur, est
                                                ; initialisé à 8
0006 44                 DECALE  LSRA            ; Décalage droite de A (a[0] -> C)
0007 56                         RORB            ; Rotation droite de B (C -> b[7])
```

```
0008 85   08                    BITA    #$08        ; Test du bit a[3]
000A 27   02      000E          BEQ     SUITE       ; Si a[3] = 0 on continue en SUITE
000C 80   03                    SUBA    #$03        ; sinon on retranche $03
000E 30   1F            SUITE   LEAX    -1,X        ; Décrémentation du compteur
0010 26   F4      0006          BNE     DECALE      ; S'il est non nul on poursuit les décalages
0012 D7   51                    STB     $0051       ; sinon on stocke résultat en $005l
0014 3F                         SWI                 ;
                                                    ;
0050                            ORG     $0050       ;
0050 28                 DATA    FCB     $28         ; Donnée DCB à traduire
```

Prog 10 : Remarque B

Après exécution, on trouvera la donnée $1C à l'adresse $0051.

6809 Prog 11 : Multiplication

Prog 11 : Question A

Sur le modèle de l'algorithme proposé en cours, rédiger un programme réalisant la multiplication de deux données de8 bits non signées. Le multiplicande, le multiplicateur et le résultat obtenu seront placés aux adresses mémoire $0050, $0051 et $0052.

Prog 11 : Commentaires A

On rappelle que l'algorithme de multiplication de deux nombres non signés de 8 bits nécessite l'emploi de trois registres 8 bits. Le 6809 n'en possédant que deux, on travaillera directement sur le contenu de la case mémoire renfermant le multiplicande.

Prog 11 : Algorithme A

```
A <- 0     B <- multiplicateur     $0050 <- multiplicande    Compteur <- 8
    Tant que (Compteur > 0)
        Si (b[0] = 1) alors
            A <- A + multiplicande
        Fin Si
        Décalage vers la droite des contenus
        combinés des registres A et B
        Compteur = Compteur – 1
    Fin Tant que
```

Prog 11 : Remarque A

En fin d'exécution, le résultat de la multiplication est situé dans D.

Prog 11 : Programme A

```
0000                            ORG     $0000       ;
                  0010  Val     EQU     $10         ;
0000 8E   0008                  LDX     #$0008      ; Compteur
0003 4F                         CLRA                ;
0004 D6   51                    LDB     $0051       ; Le multiplicateur est placé dans B
0006 C5   01            DEBUT   BITB    #$01        ; Test b[0]
0008 27   0B      0015          BEQ     DECALE      ; Si b[0] = 0 on passe en DECALE
000A 9B   50                    ADDA    $0050       ; sinon on additionne le multiplicande
                                                    ;
                        ;-----Décalage à droite de A-B dans le cas où le bit b[0] = 1
000C 46                         RORA                ; Rotation droite de A C --> a[7]
                                                    ;                   et a[0] --> C
000D 56                         RORB                ; Rotation droite de B C = a[0] --> b[7]
000E 30   1F                    LEAX    -1,X        ;
0010 26   F4      0006          BNE     DEBUT       ;
0012 DD   52                    STD     $0052       ;
0014 3F                         SWI                 ; Décalage à droite de A-B dans le
                                                    ; cas où b[0] =0
0015 44                 DECALE  LSRA                ; Rotation droite de A 0 ?> a[7]
                                                    ;                   et a[0] --> C
0016 56                         RORB                ; Rotation droite de B C = a[0] --> b[7]
0017 30   1F                    LEAX    -1,X        ;
0019 26   EB      0006          BNE     DEBUT       ;
001B DD   52                    STD     $0052       ;
```

```
                                                ;
0050                            ORG     $0050   ;
0050 FF 48              DATA    FCB     $FF,$48 ;
                                END             ;
```

Prog 11 : Remarque A

On notera que le processus de décalage combiné des registres A et B diffère selon que b [0] = 1 ou que b[0] = 0.

En effet, dans un cas il faut introduire en a[7] la retenue issue de l'addition du multiplicande (RORA) et dans l'autre il faut introduire en a[7] un 0 (LSRA).

Pour l'exemple choisi, le résultat de la multiplication stocké à l'adresse $0052 (et $0053) sera $47B8.

Enfin, si l'on souhaite vérifier la validité du programme on peut ajouter la séquence suivante au début du programme

```
0000 96   50                    LDA     $0050           ;
0002 D6   51                    LDB     $0051           ;
0004 3D                         MUL                     ;
0005 DD   54                    STD     $0054           ;
```

Prog 11 : Question B

Modifier le programme précédent de sorte qu'il puisse travailler sur des données de 16 bits.

Prog 11 : Commentaires B

L'algorithme utilisé reste identique au précédent mais la difficulté réside ici dans le fait que seuls les accumulateurs A et B peuvent subir des opérations de décalage ou de rotation.

Autrement dit, ces décalages doivent être effectués en deux temps : décalage de l'octet de poids fort de la donnée en premier lieu puis décalage de l'octet de poids faible en prenant soin de faire le report correctement.

Dans le programme présenté ci−dessous, le multiplicande est rangé aux adresses $0050 et $0051, le multiplicateur en$0052 et $0053 et le résultat de la multiplication en $0054, $0055, $0056 et $0057.

On travaillera donc sur ces quatre cases mémoire ; en particulier, le contenu de $0054 et $0055 est initialisé à 0 et le multiplicateur est placé en $0056 et $0057. Enfin, le compteur doit être initialisé par la valeur 16 soit $0010 en hexadécimal.

Prog 11 : Programme B

```
0000                            ORG     $0000       ;
                    0010  Val   EQU     $10         ;
0000 8E   0010                  LDX     #$0010      ; Initialise le compteur
0003 0F   54                    CLR     $0054       ;
0005 0F   55                    CLR     $0055       ;
0007 DC   52                    LDD     $0052       ;
0009 97   56                    STA     $0056       ;
000B D7   57                    STB     $0057       ; Place le multiplicateur
                                                    ;       en $0056 et $0057
000D C5   01            DEBUT   BITB    #$01        ; Test bit poids faible
                                                    ;       du multiplicateur
000F 27   17      0028          BEQ     DECALE      ; Si = à 0, on passe en DECALE
0011 DC   54                    LDD     $0054       ;
0013 D3   50                    ADDD    $0050       ; Si = à 1, on additionne
                                                    ;       le multiplicande
                                                    ;
                        ;------ au contenu des adresses $0054 et $0055
                        ;   Décalage à droite du contenu combiné des adresses $0054
                        ;   à $0057 dans le cas où le bit testé est égal à 1
0015 46                         RORA                ; Rotation à droite du contenu
                                                    ; des adresses $0054 et $0055
                                                    ; de manière à prendre en compte
                                                    ; la retenue issue de l'addition
0016 97   54                    STA     $0054       ;
0018 56                         RORB                ;
0019 D7   55                    STB     $0055       ;
001B DC   56                    LDD     $0056       ;
001D 46                         RORA                ; Rotation droite du contenu
                                                    ;       adrs $0056 $0057
001E 97   56                    STA     $0056       ;
```

```
0020 56                        RORB              ;
0021 D7   57                   STB    $0057      ;
0023 30   1F                   LEAX   -1,X       ;
0025 26   E6      000D         BNE    DEBUT      ;
0027 3F                        SWI               ;
                                                 ;
                         ;------ Décalage à droite du contenu combiné des
                         ;   adresses $0054 à $0057 dans le cas où le bit
                         ;   testé est égal à 0
0028 DC   54             DECALE LDD    $0054      ;
002A 44                        LSRA              ;
002B 97   54                   STA    $0054      ;
002D 56                        RORB              ;
002E D7   55                   STB    $0055      ;
0030 DC   56                   LDD    $0056      ;
0032 46                        RORA              ;
0033 97   56                   STA    $0056      ;
0035 56                        RORB              ;
0036 D7   57                   STB    $0057      ;
0038 30   1F                   LEAX   -1,X       ;
003A 26   D1      000D         BNE    DEBUT      ;
                                                 ;
0050                           ORG    $0050      ;
0050 04 FF 03 36         DATA  FDB    $04FF,$0336 ;
                               END               ;
```

Prog 11 : Remarque B

A l'issue de l'exécution, le résultat de la multiplication placé de $0054 à $0057 est égal à $00100ACA.

Prog 12 : Division

Prog 12 : Question A

Proposer un programme effectuant la division de X par Y, où X et Y sont deux nombres de 8 bits non signés tels que X supérieur ou égal à Y et Y différent de 0. Le dividende, le diviseur, le quotient et le reste seront rangés en mémoire à des adresses successives.

La division de deux entiers binaires non signés de 8 bits nécessite l'emploi de trois registres 8 bits.

Prog 12 : Algorithme A

```
A <- 0    B <- dividende      $0051 <- diviseur      Compteur <- 8
    Tant que (Compteur > 0)
        Décalage à gauche des registres
        combinés A et B

        Si (A < diviseur) alors
            b[0] <- 0
        sinon
            A <- A - diviseur
            b[0] <- 1
        Fin si
        Compteur <- Compteur - 1
    Fin Tant que
```

En fin d'exécution, le quotient est situé dans B et le reste dans A.

Prog 12 : Programme A

```
0020                           ORG    $0020      ;
                   0014  Val   EQU    $14        ;
0020 4F                        CLRA              ;
0021 D6   50                   LDB    $0050      ; Place le dividende dans B
0023 8E   0008                 LDX    #$0008     ; Compteur
0026 58                  DEBUT ASLB               ; Décalage gauche des
                                                 ; registres A et B
                         ;------ attention ici il faut commencer par
                         ;              l'octet de poids faible
                         ;  afin de faire le report correctement sur
                         ;              l'octet de poids fort
0027 49                        ROLA              ;
0028 91   51                   CMPA   $0051      ;
```

```
002A 25   04      0030          BLO    SUITE      ; Si A < diviseur --> SUITE
002C 90   51                    SUBA   $0051      ; sinon on retranche le diviseur
002E CA   01                    ORB    #$01       ; et on force b0 à 1
0030 30   1F            SUITE   LEAX   -1,X       ;
0032 26   F2      0026          BNE    DEBUT      ;
0034 D7   52                    STB    $0052      ; Range le quotient en $0052
0036 97   53                    STA    $0053      ; Range le reste en $0053
0038 3F                         SWI               ;
                                                  ;
0050                            ORG    $0050      ;
0050 FC 26              DATA    FCB    $FC,$26    ;
                                END               ;
```

Prog 12 : Question B

Comme application, écrire un programme permettant de réaliser la conversion inverse de celle qui est traitée dans le problème P3.

Le nombre entier binaire de 8 bits à convertir sera stocké à l'adresse $0050 et sa traduction en DCB à l'adresse $0051. On utilisera un sous programme pour effectuer les divisions nécessaires.

Prog 12 : Commentaires B

Rappelons que la méthode de conversion binaire−DCB consiste à diviser le nombre binaire par 10 successivement jusqu'à obtenir un résultat nul ; les restes des divisions écrits sur 4 bits correspondent à chaque chiffre DCB, le premier reste traduisant le chiffre des unités.

Prog 12 : Programme B

```
0020                            ORG    $0020      ;
                  0014  Val     EQU    $14        ;
0020 108E 0051                  LDY    #$0051     ;
0024 9D   2E            BOUCLE  JSR    DIVISE     ;
0027 A7   A0                    STA    ,Y+        ;
0029 C1   00                    CMPB   #$00       ;
002B 26   F7      0024          BNE    BOUCLE     ;
002D 3F                         SWI               ;
                                                  ;
002E 4F                 DIVISE  CLRA              ;
002F D6   50                    LDB    $0050      ;
0031 8E   0008                  LDX    #$0008     ;
0034 58                 DEBUT   ASLB              ;
0035 49                         ROLA              ;
0036 81   0A                    CMPA   #$0A       ;
0038 25   04      003E          BLO    SUITE      ;
003A 80   0A                    SUBA   #$0A       ;
003C CA   01                    ORB    #$01       ;
003E 30   1F            SUITE   LEAX   -1,X       ;
0040 26   F2      0034          BNE    DEBUT      ;
0042 D7   50                    STB    $0050      ;
0044 39                         RTS               ;
                                                  ;
0050                            ORG    $0050      ;
0050 DC                 DATA    FCB    $DC        ;
```

Prog 12 : Remarque B

La donnée binaire $DC = 1101 1100 doit être traduite en BCD par 0010 0010 000 (soit $0220 en hexadécimal). A l'issue de l'exécution de ce programme, on trouve $00, $02, $02 aux adresses $0051, $0052 et $0053 ce qui traduit le fait que les restes sont ici calculés sur 8 bits. Ainsi, le résultat souhaité est obtenu en examinant les 4 bits de poids faibles de ces 3 données.

Prog 12 : Question C

Proposer un programme de division travaillant sur des données de 8 bits signées. On rappelle que l'algorithme de division de nombres signés se présente sous la forme suivante

Prog 12 : Algorithme C

```
M <- diviseur      R-Q <- dividende      Compteur <- n S <- R[n-1]
    Tant que (Compteur > 0)
        Décalage gauche des registres combinés R et Q
        T <- R
        Si ( R[n-1] = M[n-1]) alors
            R <- R-M
        sinon
```

```
            R <- R+M
        Finsi

        Si ( R[n-1] = T[n-1]) OU ( R = 0 ET Q = 0 ) alors
            Q[0] <- 1
        sinon
            Q[0] <- 0
            R <- T (restaure le contenu de R)
        Finsi
        Compteur <- Compteur - 1
    Fin Tant que

    Si (S <> M[n-1]) alors
        Q <- Cà2 de Q
    Fin Si
```

Dans le programme propose ci-dessous, les cases mémoire utilisées se présentent comme suit :

```
ADRESSE     CONTENU
$0001       dividende
$0002       signe du dividende (S)
$0003       diviseur (M)
$0004       quotient (Q)
$0005       reste (R)
$0006       registre temporaire (T)
```

Prog 12 : Programme C

```
0020                            ORG     $0020       ;
                0014    Val     EQU     $14         ;
0020 96   01                    LDA     $0001       ; Charge le dividende dans A
0022 2B   45    0069            BMI     SIGNE       ; S'il est négatif on passe en SIGNE
0024 4F                         CLRA                ; S'il est positif, on fixe S=$00
0025 97   02                    STA     $0002       ;
0027 D6   01            NEXT    LDB     $0001       ; Charge le dividende dans B
0029 1D                         SEX                 ; Etend le signe du dividende dans l'accu A
002A D7   04                    STB     $0004       ; Initialise Q avec le dividende
002C 97   05                    STA     $0005       ; Initialise R avec le signe du dividende
002E C6   08                    LDB     #$08        ; Initialise le compteur
0030 D7   0A                    STB     $000A       ; Compteur placé à l'adresse $000A
                        ;------ Décalage gauche des registres combinés R-Q
0032 96   04            DEBUT   LDA     $0004       ;
0034 48                         ASLA                ;
0035 97   04                    STA     $0004       ;
0037 96   05                    LDA     $0005       ;
0039 49                         ROLA                ;
003A 97   05                    STA     $0005       ;
003C 97   06                    STA     $0006       ; Sauvegarde temporaire
003E 96   03                    LDA     $0003       ; Charge M dans A
0040 2B   2E    0070            BMI     NEGATIF     ; S'il est négatif on passe en NEGATIF
0042 96   05                    LDA     $0005       ; S'il est positif on charge le reste dans R
0044 2B   31    0077            BMI     ADDITION    ; Si M > 0 et R < 0 an passe en ADDITION
0046 20   38    0080            BRA     SOUSTRAI    ; On passe en SOUSTRAIT car M > 0 et R > 0
                                                    ;
                        ;------ Vérification de la condition R[n-1] = T[n-1]
0048 96   06            TEST2   LDA     $0006       ;
004A 2B   44    0090            BMI     NEGATIF2    ; Si T < 0 on passe en NEGATIF2
004C 96   05                    LDA     $0005       ; Si T > 0 on charge R dans A
004E 2B   47    0097            BMI     RESTAUR     ; Si T > 0 et R < 0 on passe en RESTAURE
0050 96   04            QUN     LDA     $0004       ; Cas T > 0 et R > 0
0052 8A   01                    ORA     #$01        ; On force Q[0] à 1
0054 97   04                    STA     $0004       ;
0056 0A   0A            COMPT   DEC     $000A       ; Décrémente le compteur
0058 26   D8    0032            BNE     DEBUT       ; Tant que compteur > 0 on retourne en DEBUT
005A 96   03                    LDA     $0003       ;
005C 2B   40    009E            BMI     NEG3        ; Si M < 0 on passe en NEG3
005E 4F                         CLRA                ; Cas M > 0 : on teste ta condition M[n-1] = S
005F 91   02                    CMPA    $0002       ;
0061 27   05    0068            BEQ     FIN         ; Si M[n-1] = S on passe en FIN
0063 96   04            COMPLEM LDA     $0004       ; Cas M[n-1] <> S, on prend le Cà2 du quotient
0065 40                         NEGA                ;
0066 97   04                    STA     $0004       ;
0068 3F                 FIN     SWI                 ;
                                                    ;
                        ;------ Lorsque le dividende est négatif, on fixe S = $01
```

```
0069 86   01           SIGNE    LDA    #$01       ;
006B 97   02                    STA    $0002      ;
006D 20   B8      0027          BRA    NEXT       ;
006F 3F                         SWI               ;
                                                  ;
                        ;------ Cas M < 0, étude du signe de R
0070 96   05            NEGATIF  LDA    $0005      ;
0072 2B   0C      0080          BMI    SOUSTRAI   ; Si R < 0 on passe en SOUSTRAIT
0074 20   01      0077          BRA    ADDITION   ; Cas R > 0 on passe en ADDITION
0076 3F                         SWI               ;
0077 9B   03            ADDITION ADDA   $0003      ;
0079 97   05                    STA    $0005      ;
007B 27   0C      0089          BEQ    TEST       ; Si R = 0 on passe en TEST
007D 20   C9      0048          BRA    TEST2      ; Cas R <> 0, on passe en TEST2
007F 3F                         SWI               ;
0080 90   03            SOUSTRAI SUBA   $0003      ;
0082 97   05                    STA    $0005      ;
0084 27   03      0089          BEQ    TEST       ; Si R = 0 on passe en TEST
0086 20   C0      0048          BRA    TEST2      ; Cas R <> 0, on passe en TEST2
0088 3F                         SWI               ;
                                                  ;
                        ;------ Cas R = 0, teste si Q = 0
0089 96   04            TEST     LDA    $0004      ;
008B 27   C3      0050          BEQ    QUN        ; Si Q = 0 on passe en QUN
008D 20   B9      0048          BRA    TEST2      ; Cas Q <> 0, on passe en TEST2
008F 3F                         SWI               ;
                        ;------ Cas T < 0. étude du signe de R
0090 96   05            NEGATIF2 LDA    $0005      ;
0092 2B   BC      0050          BMI    QUN        ; Si R < 0 on passe en QUN
0094 20   01      0097          BRA    RESTAUR    ; Cas R > 0, on passe en RESTAURE
0096 3F                         SWI               ;
0097 96   06            RESTAUR  LDA    $0006      ;
0099 97   05                    STA    $0005      ;
009B 20   B9      0056          BRA    COMPT      ;
009D 3F                         SWI               ;
                                                  ;
                        ;------ Cas M < 0, teste la condition M[n-1] = S
009E 86   01            NEG3     LDA    #$01       ;
00A0 91   02                    CMPA   $0002      ;
00A2 27   C4      0068          BEQ    FIN        ; Si M[n-1] = S on passe en FIN
00A4 20   BD      0063          BRA    COMPLEM    ; Cas M[n-1] <> S, on passe en COMPLEM
00A6 3F                         SWI               ;
                                                  ;
0001                            ORG    $0001      ;
0001 8F                 DIVDEND  FCB    $8F        ;
                                                  ;
0003                            ORG    $0003      ;
0003 FD                 DIVSEUR  FCB    $FD        ;
                                END               ;
```

Prog 13 : Kit MC09-B Sté DATA RD : Interface Parallèle

Voir également Prog 14, 15, 16 ci-après.

Prog 13 MC09-B : Introduction

L'étude du fonctionnement de l'interface parallèle peut être réalisée à l'aide d'un kit du type DATA RD ou MC09−B sur lesquels sont implantés un microprocesseur 6809 ainsi qu'un PIA. L'adresse de base de ce dernier est $7090 pour le 1" kit et $8000 pour le second.

Les divers registres du PIA ont donc pour adresse :

```
ORA/DDRA  $7090 (ou $8000)
CRA       $7091 (ou $8001)
ORB/DDRB  $7092 (ou $8002)
CRB       $7093 (ou $8003)
```

Les lignes d'interruption IRQA et IRQB des ports A et B du PIA sont reliées à la broche IRQ du microprocesseur.
Le vecteur d'adresse de l'interruption IRQ est $3F40 pour le kit DATA RD et $3FF8 pour le MC09−B.

Les leds permettent de visualiser l'état des diverses lignes. Les interrupteurs, munis d'anti-rebond, sont reliés à des fiches femelles et permettent d'imposer un niveau logique 0ou 1 sur ces fiches. La masse du microprocesseur est ramenée sur la fiche « Masse » de la platine d'étude !

Prog 14 : Kit MC09-B Sté DATA RD : Etude des Ports Entrée / Sortie

Prog 14 MC09-B : Sujet A

Port en entrée
Afficher, par le biais des interrupteurs, le mot $C4 sur les entrées A; du port A. Écrire en langage assembleur un programme permettant de stocker ce mot à l'adresse $0100. Exécuter ce programme et vérifier son bon fonctionnement.

Prog 14 MC09-B : Commentaires A

$C4 = %11000100
On initialise le port A en entrée, puis on vient le lire et on mémorise la donnée lue en $0100

Prog 14 MC09-B : Programme A

```
                  8000  DDRA   EQU    $8000      ; |
                  8000  ORA    EQU    $8000      ; | Définition adresses de port DDRA, ORA, CRA
                  8001  CRA    EQU    $8001      ; |
                                                 ;
0000                            ORG    $0000      ;
0000 4F                         CLRA              ; Effacement de A
0001 B7   8001                  STA    CRA        ; Stock A dans CRA pour demande d'accès à DDRA
0004 B7   8000                  STA    DDRA       ; Déclaration du port A en entrée
0007 86   04                    LDA    #$04       ;  |
0009 B7   8001                  STA    CRA        ;  | Demande d'accès à ORA
000C B6   8000                  LDA    ORA        ; Chargement du registre ORA
000F B7   0100                  STA    $0100      ; Stockage de la valeur à l'adresse $0100
0012 3F                         SWI               ; Fin du programme.
```

Prog 14 MC09-B : Sujet B

Port en sortie On utilise le port B en sortie pour commander, par microprocesseur, un moteur pas à pas.
Ce moteur possède 4 entrées notées **I1** à **I4** l'activation de ces entrées suivant la séquence décrite ci−dessous fait tourner le moteur par pas de 7,5° (le fait d'inverser la séquence de l'étape 4 à l'étape 1 fait tourner le moteur en sens inverse).

PAS	I1	I2	I3	I4
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1
4	1	0	0	1
5 Répétition du pas I1				

Prog 14 MC09-B : Question B :

Ecrire un programme en langage assembleur permettant d'assurer une rotation de 360° par pas de 3,75° dans un sens puis dans l'autre (utiliser entre chaque pas une temporisation, correspondant au décomptage de $FFFF, dont vous préciserez le rôle).

Pour vérifier le bon fonctionnement du programme, on prendra soin de connecter le moteur, par l'intermédiaire de ses entrées I1, à I4 au port B du PIA suivant le brochage : B0−I1, B1−I2, B2−I3 et B3−I4.

Remarque : la broche « 0 V » du moteur doit être reliée à la masse du microprocesseur.

Proposer une méthode permettant de déterminer la vitesse de rotation du moteur (on rappelle que le fonctionnement du 6809 est rythmé par une horloge de Fréquence égale à 1 MHz).

Prog 14 MC09-B : Commentaires B

Il faut créer une table de quatre donnés, correspondant en ordre et en valeur à l'enchaînement de la séquence moteur, cette séquence comporte quatre phases qui permettent chacune un déplacement de 3,75° du moteur, pour réaliser un tour complet soit 360° il faut faire 24 fois la boucle de séquence des phases.

Soit 24 x 4 x 3,75 = 360°

Entre chaque phase est imposée une temporisation de $FFFF permettant au moteur d'effectuer l'enchaînement de ses transitions. Le temps peut être réduit et est fonction de l'accélération, et donc de ce fait du couple moteur.

Pour calculer la vitesse moteur on commence par négliger les cycles machines or temporisation auquel cas on pourrait dire que le moteur met 24 x 4 temporisations pour faire un tour.

Calcul de la vitesse à 1MHz

VT environ égal à (3 NC + 4 NC+ 3 NC) x $FFFF x 24 x 4/360 = 0.017 tr/s soit environ 1 tr/min.

Prog 14 MC09-B : Programme B Sens Antihoraire

```
                8002  DDRB    EQU    $8002     ;  |
                8002  ORB     EQU    $8002     ;  | Déf. adresses de port DDRB, ORB, CRB
                8003  CRB     EQU    $8003     ;  |
                                                ;
0000                          ORG    $0000     ; Début du programme à l'adresse $0000
                        ;------ INITIALISATION DU PIA
0000 4F                       CLRA             ; Effacement du registre A
0001 B7   8003                STA    CRB       ; Stock A dans CRB pour demande d'accès à DDRB
0004 86   FF                  LDA    #$FF      ;   |
0006 B7   8002                STA    DDRB      ;   | Déclaration du port B en sortie
0009 86   04                  LDA    #$04      ;   |
000B B7   8003                STA    CRB       ;   | Demande d'accès à ORB
000E 5F                       CLRB             ; RAZ du registre B qui va compter le Nbre
                                                ;                               de séquence
000F 108E 0200          DEBUT LDY    #DATA     ; Chargement de l'adresse de début des données
0013 A6   A0            L1    LDA    ,Y+       ; Chargement de la donnée.
0015 B7   8002                STA    ORB       ; Envoi du mot de commande
0018 BD   0250                JSR    TEMPO     ; Appel du sous?programme TEMPO
001B 108C 0204                CMPY   #DATA+4   ; Compare Y à l'adresse de Fin des données
001F 26   F2      0013        BNE    L1        ; Si pas égal on boucle sur L1
0021 5C                       INCB             ; Sinon, Incrémente B
0022 C1   19                  CMPB   #25       ; Compare B à la valeur 25
0024 26   E9      000F        BNE    DEBUT     ; Si pas égale on boucle sur DEBUT
0026 3F                       SWI              ; Sinon, Fin du programme.
                                                ;
                        ;------ DONNEES POUR UNE ROTATION AU PAS DE 3.75°
0200                          ORG    $0200     ;
0200 03 06 0C 09        DATA  FCB    $03,$06,$0C,$09 ;
                                                ;
                        ;------ SOUS-PROGRAMME TEMPO
0250                          ORG    $0250     ;
0250 8E   FFFF          TEMPO LDX    #$FFFF    ; Chargement de X par $FFFF
0253 30   82            T2    LEAX   ,-X       ; X=X?1
0255 26   FC      0253        BNE    T2        ; Si X<>0 --> boucle sur T2
0257 39                       RTS              ; Sinon --> Retour au programme appelant.
```

Prog 14 MC09-B : Programme B Sens horaire

Pour la rotation en sens inverse on changera seulement la séquence des données pour la rotation

```
DATA    FCB     $09, $0C, $06, $03
```

Prog 15 : Kit MC09-B Sté DATA RD : Etude des Interruptions

Prog 15 MC09-B : Sujet

Programme chenillard, Un « chenillard » consiste à allumer une seule lampe à la fois parmi les huit et à la faire se déplacer dans l'ordre A0, A1, A2,....., A7, A0,..... À une vitesse donnée.

Prog 15 MC09-B : Question A

Proposer un programme en langage assembleur permettant de réaliser un tel chenillard sur le port A (on conservera la même temporisation que dans le problème précédent).

Prog 15 MC09-B : Commentaires A

On commence par établir une table de données correspondant en nombre et en valeurs à l'enchaînement du chenillard. Il suffit ensuite d'envoyer les données les unes après les autres sur le port A en intercalant la temporisation entre chaque séquences, lorsque l'on arrive à la fin de la table, on reboucle alors sur son début et ainsi de suite.

Prog 15 MC09-B : Programme A

```
                8000  DDRA    EQU    $8000          ; |
                8000  ORA     EQU    $8000          ; | Déf adresses de port DDRA, ORA, CRA
                8001  CRA     EQU    $8001          ; |
                                                    ;
0000                          ORG    $0000          ; Début du programme
```

```
                        ;------ INITIALISATION DU PIA
0000 4F                         CLRA                    ; Effacement de A
0001 B7   8001                  STA     CRA             ; Stock A dans CRA pour accès à DDRA
0004 86   FF                    LDA     #$FF            ;  |
0006 B7   8000                  STA     DDRA            ;  | Place le port A en sortie.
0009 86   04                    LDA     #$04            ;  |
000B B7   8001                  STA     CRA             ;  | Demande d'accès à ORA
                                                        ;
                        ;------ PROGRAMME PRINCIPAL
000E 108E 0200          DEBUT   LDY     #DATA           ; Charg adresse de début des données
0012 A6   A0            L1      LDA     ,Y+             ; Chargement des données
0014 B7   8000                  STA     ORA             ; Stockage de A sur les sorties ORA
0017 BD   0250                  JSR     TEMPO           ; Temporisation
001A 108C 0208                  CMPY    #DATA+8         ; Compare Y a l'adresse de fin des données
001E 26   F2      0012          BNE     L1              ; Si pas égale --> boucle sur L1
0020 20   EC      000E          BRA     DEBUT           ; Sinon --> boucle sur DEBUT
                                                        ;
                        ;------ DONNEES pour un défilement de b0 à b7.
0200                            ORG     $0200           ;
0200 01 02 04 08        DATA    FCB     $01,$02,$04,$08 ;
0204 10 20 40 80                FCB     $10,$20,$40,$80 ;
                                                        ;
                        ;------ Sous Programme TEMPO
0250                            ORG     $0250           ;
0250 8E   FFFF          TEMPO   LDX     #$FFFF          ; Chargement de X par $FFFF
0253 30   82            T2      LEAX    ,-X             ; X=X-1
0255 26   FC      0253          BNE     T2              ; Si X<>0 --> boucle sur T2
0257 39                         RTS                     ; Sinon --> Retour au programme appelant.
                                                        ;
```

Prog 15 MC09-B : Question B (Interruption par test d'état)

Modifier le programme précédent de façon à ce qu'il soit susceptible d'être interrompu par test d'état après avoir allumé AN et qu'il fonctionne selon le principe suivant

-- défilement normal du chenillard A0, A1,....., A7
-- test d'état du registre CRA ;
-- s'il n'y a pas eu de demande d'interruption, poursuite du défilement normal du chenillard avec nouveau test d'état après l'allumage de A7 ;
-- s'il y a eu demande d'interruption, extinction pendant 5 s environ de toutes les lampes du port A (sous-programme d'interruption) puis reprise du défilement du chenillard.

La demande d'interruption sera réalisée à l'aide d'un front descendant envoyée sur CA1, par exemple. Pour tester le programme, on générera manuellement le front descendant à l'aide de l'un des interrupteurs préalablement relié àCA1.

Proposer un programme permettant d'effectuer le comptage du nombre de données paires et impaires d'une table.

Prog 15 MC09-B : Commentaires B

On exécute le chenillard pour toute la table, à la fin de celle ci on vient scruter le bit 7 de CRA correspondant à une interruption d'état du PIA si ce bit est à 1, une interruption a eu lieu, on éteint donc alors toute les leds pendant une valeur de huit tempo soit 5s environs puis on recommence.

Si aucune interruption n'est demandée, le chenillard recommence en début de table.

Prog 15 MC09-B : Programme B

```
                8000    DDRA    EQU     $8000   ;  |
                8000    ORA     EQU     $8000   ;  | Déf adresses de port DDRA, ORA, CRA
                8001    CRA     EQU     $8001   ;  |
                                                ;
0000                            ORG     $0000   ; Début du programme à l'adresse $0000
                                                ;------ INITIALISATION DU PIA
0000 4F                         CLRA            ; Effacement de A
0001 B7   8001                  STA     CRA     ; Stock A dans CRA pour accès à DDRA
0004 86   FF                    LDA     #$FF    ;   |
0006 B7   8000                  STA     DDRA    ;   | Place le port A en sortie.
0009 86   06                    LDA     #$06    ;   | Demande d'accès à ORA et validation des
000B B7   8001                  STA     CRA     ;   | interruptions.
                                                ;
                        ;------ PROGRAMME PRINCIPAL
000E 108E 0200          DEBUT   LDY     #DATA   ; Chargement de l'adresse de début des données
0012 A6   A0            L1      LDA     ,Y+     ; Chargement des données
```

```
0014 B7   8000                STA    ORA      ; Stockage de A sur les sorties ORA
0017 BD   0250                JSR    TEMPO    ; Temporisation
001A 108C 0208                CMPY   #DATA+8  ; Compare Y a l'adresse de fin des données
001E 26   F2      0012        BNE    L1       ; Si pas égale --> boucle sur L1
0020 B6   8001                LDA    CRA      ; Chargement du registre d'état CRA
0023 84   80                  ANDA   #$80     ; Masque pour récupérer le bit b7 de CRA
0025 81   00                  CMPA   #$00     ; Compare à 0
0027 27   E5      000E        BEQ    DEBUT    ; Si pas d'interruption, boucle sur DEBUT, sinon
                                              ;
                        ;------ SOUS PROGRAMME D'INTERRUPTION.
0029 4F                       CLRA           ;   |
002A B7   8000                STA    ORA      ;   | Extinction des sorties
002D 86   08                  LDA    #$08     ;
002F BD   0250          L2    JSR    TEMPO    ; Appel du sous programme TEMPO
0032 4A                       DECA           ; Décrémente A de 1
0033 26   FA      002F        BNE    L2       ; Si A n'est pas nul on boucle sur L2
0035 B6   8000                LDA    ORA      ; Sinon, lecture de ORA pour RAZ du bit b7 de CRA
0038 20   D4      000E        BRA    DEBUT    ; Sinon --> boucle sur DEBUT
                                              ;
                        ;------ DONNEES
0200                          ORG    $0200    ;
0200 01 02 04 08        DATA  FCB    $01,$02,$04,$08 ;
0204 10 20 40 80              FCB    $10,$20,$40,$80 ;
                                              ;
                        ;------ Sous Programme TEMPO
0250                          ORG    $0250    ;
0250 8E   FFFF          TEMPO LDX    #$FFFF   ; Chargement de X par $FFFF
0253 30   82            T2    LEAX   ,-X      ; X=X-1
0255 26   FC      0253        BNE    T2       ; Si X<>0 --> boucle sur T2
0257 39                       RTS            ; Sinon --> Retour au programme appelant.
```

Prog 15 MC09-B : Question C

Interruption vectorisée

Modifier le programme du chenillard de sorte qu'il soit susceptible d'être interrompu par une interruption vectorisée et de réaliser les séquences suivantes

-- défilement normal du chenillard

-- demande d'interruption vectorisée déclenchée par front montant sur CA2 réalisé manuellement comme précédemment

-- exécution, le cas échéant, du programme d'interruption qui consiste à allumer toutes les lampes du port Adurant 5s environ, puis retour au programme principal.

Prog 15 MC09-B : Tester le programme et conclure C

Proposer un programme global, c'est à dire incluant les deux types d'interruption. Exécuter ce programme et lancer une interruption par test d'état ; pendant le déroulement de cette interruption, lorsque toutes les lampes sont éteintes, lancer une interruption vectorisée. Que se passe-t-il ? Recommencer la même expérience en lançant d'abord l'interruption vectorisée. Conclure.

Prog 15 MC09-B : Commentaires C

On exécute le chenillard normalement. L'interruption vectorisée sur front montant est déclarée sur CA2 par conséquent si pendant l'exécution du chenillard une interruption apparaît sur CA2 le programme se positionne automatiquement à l'adresse d'interruption du kit soit $3FF8 à laquelle est stockée l'adresse du programme d'interruption, on exécute alors ce programme qui consiste à allumer toutes les leds durant 5s, puis le chenillard reprends la ou il s'est arrêté.

Prog 15 MC09-B : Programme C

```
                  8000  DDRA    EQU    $8000    ;  |
                  8000  ORA     EQU    $8000    ;  | Déf adresses de port DDRA, ORA, CRA
                  8001  CRA     EQU    $8001    ;  |
                                              ;
0000                            ORG    $0000    ; Début du programme à l'adresse $0000
                        ;------ INITIALISATION DU PIA
0000 4F                         CLRA           ; Effacement de A
0001 B7   8001                  STA    CRA      ; Stock A dans CRA pour accès à DDRA
0004 86   FF                    LDA    #$FF     ;   |
0006 B7   8000                  STA    DDRA     ;   | Place le port A en sortie.
0009 86   1C                    LDA    #$1C     ;   | Demande d'accès à ORA et validation des
000B B7   8001                  STA    CRA      ;   | interruptions sur front montant.
000E 108E 0150                  LDY    #INTVECT ;   | Chargement de l'adresse du sous programme
0012 10BF 3FF8                  STY    $3FF8    ;   | d'interruption.
0016 1C   EF                    ANDCC  #$EF     ; Forçage du bit b4 du CCR à 0
```

```
                                        ;
                      ;------ PROGRAMME PRINCIPAL
0018 108E 0200        DEBUT   LDY    #DATA    ; Chargement de l'adresse de début des données
001C A6   A0          L1      LDA    ,Y+      ; Chargement des données
001E B7   8000                STA    ORA      ; Stockage de A sur les sorties ORA
0021 BD   0250                JSR    TEMPO    ; Temporisation
0024 108C 0208                CMPY   #DATA+8  ; Compare Y a l'adresse de fin des données
0028 26   F2      001C        BNE    L1       ; Si pas égale --> boucle sur L1
002A 20   EC      0018        BRA    DEBUT    ; Retour à DEBUT
                                        ;
                      ;------ SOUS PROGRAMME D'INTERRUPTION.
0150                          ORG    $0150    ;
0150 86   FF          INTVECT LDA    #$FF     ;   |
0152 B7   8000                STA    ORA      ;   | Allumage des sorties
0155 86   08                  LDA    #$08     ; Temporisation de 5s
0157 BD   0250        L2      JSR    TEMPO    ; Appel du sous programme TEMPO
015A 4A                       DECA            ; Décrémente A de 1
015B 26   FA      0157        BNE    L2       ; Si A n'est pas nul on boucle sur L2
015D B6   8000                LDA    ORA      ; Sinon, lecture de ORA pour RAZ bit b7 de CRA
0160 3B                       RTI             ; Retour au programme principal
                                        ;
                      ;------ DONNEES
0200                          ORG    $0200    ;
0200 01 02 04 08      DATA    FCB    $01,$02,$04,$08 ;
0204 10 20 40 80              FCB    $10,$20,$40,$80 ;
                                        ;
                      ;------ Sous Programme TEMPO
0250                          ORG    $0250    ;
0250 8E   FFFF        TEMPO   LDX    #$FFFF   ; Chargement de X par $FFFF
0253 30   82          T2      LEAX   ,-X      ; X=X-1
0255 26   FC      0253        BNE    T2       ; Si X<>0 --> boucle sur T2
0257 39                       RTS             ; Sinon --> Retour au programme appelant.
                                        ;
```

Prog 15 MC09-B : Programme Global D

Interruption vectorisée + Interruption d'état

```
                  8000  DDRA    EQU    $8000    ; |
                  8000  ORA     EQU    $8000    ; | Déf adresses de port DDRA, ORA, CRA
                  8001  CRA     EQU    $8001    ; |
                                        ;
0000                          ORG    $0000    ; Début du programme à l'adresse $0000
                      ;------ INITIALISATION DU PIA
0000 4F                       CLRA            ; Effacement de A
0001 B7   8001                STA    CRA      ; Stock A dans CRA pour accès à DDRA
0004 86   FF                  LDA    #$FF     ;  |
0006 B7   8000                STA    DDRA     ;  | Place le port A en sortie.
0009 86   1E                  LDA    #$1E     ;  | Demande d'accès à ORA et validation des
000B B7   8001                STA    CRA      ;  | interruptions.
000E 108E 0150                LDY    #INTVECT ;  | Chargement de l'adresse du sous programme
0012 10BF 3FF8                STY    $3FF8    ;  | d'interruption.
0016 1C   EF                  ANDCC  #$EF     ; Forçage du bit b4 du CCR à 0
                                        ;
                      ;------ PROGRAMME PRINCIPAL
0018 108E 0200        DEBUT   LDY    #DATA    ; Chargement de l'adresse de début des données
001C A6   A0          L1      LDA    ,Y+      ; Chargement des données
001E B7   8000                STA    ORA      ; Stockage de A sur les sorties ORA
0021 BD   0250                JSR    TEMPO    ; Temporisation
0024 108C 0208                CMPY   #DATA+8  ; Compare Y a l'adresse de fin des données
0028 26   F2      001C        BNE    L1       ; Si pas égale --> boucle sur L1
002A B6   8001                LDA    CRA      ; Chargement du registre d'état CRA
002D 84   80                  ANDA   #$80     ; Masque pour récupérer le bit b7 de CRA
002F 81   00                  CMPA   #$00     ; Compare à 0
0031 27   E5      0018        BEQ    DEBUT    ; Si pas d'interruption, boucle sur DEBUT, sinon
                                        ;
                      ;------ SOUS PROGRAMME D'INTERRUPTION par TEST D'ETAT
0033 4F                       CLRA            ;  |
0034 B7   8000                STA    ORA      ;  | Extinction des sorties
0037 86   08                  LDA    #$08     ; Temporisation de 5s
0039 BD   0250        L2      JSR    TEMPO    ; Appel du sous programme TEMPO
003C 4A                       DECA            ; Décrémente A de 1
003D 26   FA      0039        BNE    L2       ; Si A n'est pas nul on boucle sur L2
003F B6   8000                LDA    ORA      ; Sinon, lecture ORA pour RAZ du bit b7 de CRA
0042 20   D4      0018        BRA    DEBUT    ; Sinon --> boucle sur DEBUT
```

```
                                                ;
                        ;------ SOUS PROGRAMME D'INTERRUPTION VECTORISEE
0150                            ORG     $0150   ;
0150 86   FF            INTVECT LDA     #$FF    ;  |
0152 B7   8000                  STA     ORA     ;  | Allumage des sorties
0155 C6   08                    LDB     #$08    ; Temporisation de 5s
0157 BD   0250          L3      JSR     TEMPO   ; Appel du sous programme TEMPO
015A 5A                         DECB            ; Décrémente B de 1
015B 26   FA      0157          BNE     L3      ; Si B n'est pas nul on boucle sur L3
015D B6   8000                  LDA     ORA     ; Sinon, lecture ORA pour RAZ du bit b7 de CRA
0160 3B                         RTI             ; Sinon --> Retour au programme principal.
                                                ;
                        ;------ DONNEES
0200                            ORG     $0200           ;
0200 01 02 04 08        DATA    FCB     $01,$02,$04,$08 ;
0204 10 20 40 80                FCB     $10,$20,$40,$80 ;
                                                        ;
                        ;------ Sous Programme TEMPO
0250                            ORG     $0250   ;
0250 8E   FFFF          TEMPO   LDX     #$FFFF  ; Chargement de X par $FFFF
0253 30   82            T2      LEAX    ,-X     ; X=X-1
0255 26   FC      0253          BNE     T2      ; Si X<>0 --> boucle sur T2
0257 39                         RTS             ; Sinon --> Retour au programme appelant.
```

Prog 15 MC09-B : Commentaires D

Par test il s'avère que l'interruption vectorisée est prioritaire devant l'interruption d'état c'est−à−dire que si une interruption d'état est demandée et si en même on a une requête d'interruption vectorisée, alors c'est l'interruption vectorisée qui est exécutée.

Prog 16 : Kit MC09-B Sté DATA RD : Etude des Lignes de Dialogues

Prog 16 MC09-B : Question A

Mode programmé (Set−Reset)

On souhaite montrer l'action du bit b3 de CRA sur CA2, lorsque l'on travaille en mode programmé.

Comme application, proposer un programme permettant de faire clignoter la led associée à CA2 avec une période correspondant au décomptage de $FFFF.

Prog 16 MC09-B : Commentaire A

Il faut impérativement positionner le bit 4 de CRA à 1 pour utiliser le mode programmé, puis faire varier le bit 3 de 0à 1 entre chaque temporisation pour faire clignoter la led.

Prog 16 MC09-B : Programme A

```
                  8000  DDRA    EQU     $8000       ; |
                  8000  ORA     EQU     $8000       ; | Déf adresses de port DDRA, ORA, CRA
                  8001  CRA     EQU     $8001       ; |
                                                    ;
0000                            ORG     $0000       ; Début du programme
                        ;------ PROGRAMME PRINCIPAL
0000 86   38            DEBUT   LDA     #$38        ;  |
0002 B7   8001                  STA     CRA         ;  | Allumage de la LED
0005 BD   0250                  JSR     TEMPO       ; Temporisation
0008 86   30                    LDA     #$30        ;  |
000A B7   8001                  STA     CRA         ;  | Extinction de la LED
000D BD   0250                  JSR     TEMPO       ; Temporisation
0010 20   EE      0000          BRA     DEBUT       ; Retour au DEBUT du programme
                                                    ;
                        ;------ Sous Programme TEMPO
0250                            ORG     $0250       ;
0250 8E   FFFF          TEMPO   LDX     #$FFFF      ; Chargement de X par $FFFF
0253 30   82            T2      LEAX    ,-X         ; X=X-1
0255 26   FC      0253          BNE     T2          ; Si X<>0 --> boucle sur T2
0257 39                         RTS                 ; Sinon --> Retour au programme appelant.
```

Prog 16 MC09-B : Question B

Mode impulsion (Pulse Strobe)

Ce mode de fonctionnement va être étudié pour effectuer une conversion analogique numérique commandée par le microprocesseur.

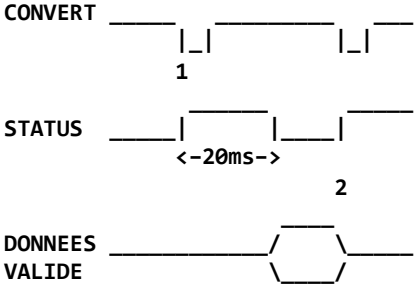
On utilisera pour cela un convertisseur (platine AD CONVERTER) dont les principales caractéristiques sont : tension analogique à convertir comprise entre 0 et 5 V, résolution égale à 20 mV et temps de conversion maximum égal à 30 ms.

L'entrée du signal analogique s'effectue sur la broche ANALOGUE I/P, la sortie numérique sur les broches D0, D1,....., D7.

L'entrée SELECT doit être positionnée à 1 afin de sélectionner le mode de conversion analogique −numérique.

La masse de la platine de conversion doit être reliée à celle du microprocesseur via la borne MASSE de la platine PIA.

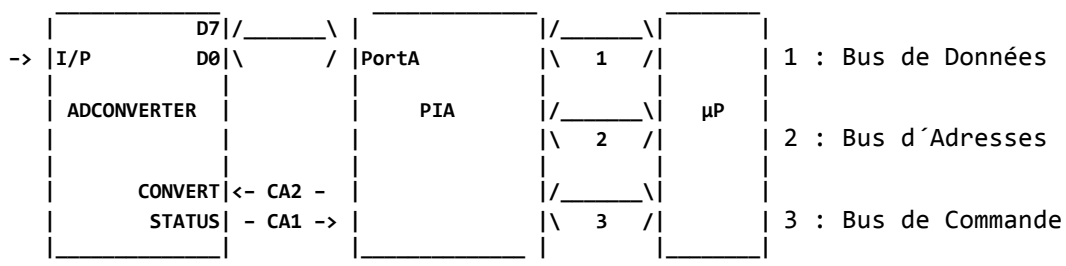
La séquence de conversion est la suivante :



1 : Demande de conversion envoyée par le µ−P

2 : Fin de conversion signalée au µ−P par le convertisseur

On propose de travailler avec le port A du PIA en utilisant la ligne CA1, pour le signal STATUS et la ligne CA2 pour le signal CONVERT suivant le schéma



Ecrire en langage assembleur un programme permettant d'effectuer l'acquisition numérique d'une donnée analogique en mode impulsion.

Effectuer ces conversions et ces acquisitions pour des tensions analogiques comprises entre 0 et 5 V par pas de 0,5 V. Analyser les résultats obtenus et conclure.

Prog 16 MC09-B : Commentaire B

On provoque une impulsion sur CA2, en utilisant, le mode impulsion, cette impulsion permettra d'activer le CAN, il faudra ensuite, tester le bit 7 d'interruption qui sera positionné à 1 par l'intermédiaire de CA1 lorsque la conversion sera terminée. Il ne reste plus qu'à lire la donnée sur le port A et enfin la mémoriser en $0100.

Prog 16 MC09-B : Programme B

```
                  8000  DDRA    EQU    $8000    ; |
                  8000  ORA     EQU    $8000    ; | Déf adresses de port DDRA, ORA, CRA
                  8001  CRA     EQU    $8001    ; |
                                                ;
0000                            ORG    $0000    ; Début du programme
                        ;------ PROGRAMME PRINCIPAL
0000 4F                         CLRA            ;  |
0001 B7   8001                  STA    CRA      ;  | Demande d'accès à DDRA
0004 B7   8000                  STA    DDRA     ;  | Configuration du port A en entrée
0007 86   3C            CONVERT LDA    #$3C     ;  |
0009 B7   8001                  STA    CRA      ;  | Chargement du mot de commande dans CRA.
000C B6   8001          ETAT    LDA    CRA      ; Chargement du registre d'état CRA
000F 84   80                    ANDA   #$80     ; Masquage du bit b7
0011 27   F9      000C          BEQ    ETAT     ; Si pas d'interruption -> boucle sur ETAT
0013 B6   8000                  LDA    ORA      ; sinon -> Conversion terminée
                                                ;             => Lecture conversion
0016 B7   0100                  STA    >$0100   ; Stockage de la conversion en $0100
```

```
0019 3F                        SWI             ;
```

Prog 17 : Ouvrage 06 : Mouvements de données 8 et 16 bits par LOAD et STORE

L'exercice suivant permet de voir les modes d'adressages associés aux instructions LD... et ST....
(Description incomplète → voir page IV.06 de l'ouvrage n°06)

```
8040                           ORG     $8040       ; positionne l'adresse de chargement
                                                   ; des directives FCB et FDB
                    0017  Val  EQU     $17         ;
8040 38 4E                     FCB     56,78       ; Charge 2 positions mémoires
                                                   ; ($8040) = $38  et  ($8041) = $4E
8042 16 2E                     FDB     5678        ; attire l'attention sur l'emploi de
                                                   ; ---section programme
8000                           ORG     $8000       ; si ce ORG est absent le programme
                                                   ; débutera à $8044 au lieu de $8000
8000 86   0C                   LDA     #12         ;
8002 C6   E0                   LDB     #224        ;
8004 8E   8060                 LDX     #$8060      ; adresce début zone stockage
8007 ED   81                   STD     ,X++        ; mise en mémoire données fixes
8009 FC   8040                 LDD     $8040       ; transfert donnée tableau
800C ED   81                   STD     ,X++        ;
800E AF   84                   STX     ,X          ; stockage index final
8010 3F                        SWI                 ;
```

Prog 18 : Programme capable de décrémenter le nombre $0E cinq fois et de stocker le résultat dans la case mémoire $0800

```
0000                           ORG     $0000       ;
                    0018  Val  EQU     $18         ;
0000 86   0E                   LDA     #$0E        ; pour décrémenter $0E on utilisera l'accu A
0002 5F                        CLRB                ; quant à l'accu B il servira de compteur
0003 4A                DEBUT1  DECA                ;
0004 5C                        INCB                ;
0005 C1   05                   CMPB    #$05        ;
0007 24   02      000B         BCC     FIN         ;
0009 20   F8      0003         BRA     DEBUT1      ;
000B B7   0800          FIN    STA     $0800       ;
                               END                 ;
                                                   ;
                       ;-----Programme identique à celui du dessus mais
                       ;     en utilisant un autre algorithme.
0100                           ORG     $0100       ;
0100 86   0E                   LDA     #$0E        ; pour décrémenter $0E on utilisera l'accu A
0102 5F                        CLRB                ; quant à l'accu B il servira de compteur
0103 4A                DEBUT2  DECA                ;
0104 5C                        INCB                ; B=1,2,3,4 et 5
0105 C1   05                   CMPB    #$05        ;
0107 25   FA      0103         BCS     DEBUT2      ;
0109 B7   0800                 STA     $0800       ;
                               END                 ;
```

Prog 19 : Programme capable de calculer la somme des 10 premiers entiers, le résultat doit être stocké à l'adresse $4000

```
0000                           ORG     $0000       ;
                    0019  Val  EQU     $19         ;
0000 C6   0A                   LDB     #$0A        ; l'accu B il servira de compteur
0002 4F                        CLRA                ;
0003 7F   0200                 CLR     $0200       ;
0006 7C   0200          DEBUT  INC     $0200       ;
0009 BB   0200                 ADDA    $0200       ;
000C 5A                        DECB                ;
```

```
000D 27   02      0011          BEQ    FIN        ;
000F 20   F5      0006          BRA    DEBUT      ;
0011 B7   4000          FIN     STA    $4000      ;
                                END                ;
```

Prog 20 : Programme capable de stocker les 100 premiers nombres entiers dans le bloc mémoire dont la première adresse est $1200

```
0000                            ORG    $0000      ;
                  0020  Val     EQU    $20        ;
0000 8E   1200                  LDX    #$1200     ; Le nombre à stocker est dans
                                                  ; l'accu A, il sert aussi de compteur
0003 4F                         CLRA               ;
0004 A7   84            DEBUT   STA    ,X         ;
0006 4C                         INCA               ;
0007 81   64                    CMPA   #100       ; 100 en décimal
0009 26   F9      0004          BNE    DEBUT      ;
                                END                ;
```

Prog 21 : Addition sur 8 bits

Supposons les contenus des mémoires suisvantes :

```
{$1000} = $04
{$1100} = $08
```

{.....} = contenu de

Soit ces quelques lignes

```
                        ;--Prog 21 : Addition sur 8 bits
0000                            ORG    $0000    ;
0000      0020          Val     EQU    $20      ;
0000                                            ;
0000 B6   1000                  LDA    $1000    ; Charger {$1000} dans A
0003 BB   1100                  ADDA   $1100    ; Addition A + {$1100} et le résultat dans A
0006 B7   1200                  STA    $1200    ; Sauvegarde de A dans {$1200}
```

Après exécution `{$1200}=$0C`

Prog 22 : Addition sur 16 bits

Supposons les contenus des mémoires suisvantes : {.....} = contenu de

```
{$1000}=$05
{$1001}=$F4              M = $05F4
                                          M   +   N   =   R
{$2000}=$09                              $05F4 + $09FC = $0FF0
{$2001}=$FC              N = $09FC
```

Soit ces quelques lignes

```
                        ;---Prog 22 : Addition sur 16 bits
0000                          ORG    $0000     ;
0000      0020          Val   EQU    $20       ;
                                               ;
                        ;---Addition des poids faibles
0000 B6   1001                LDA    $1001     ; Charger {$1001} dans A
0003 BB   2001                ADDA   $2001     ; Addition A + {$2001} et le résultat dans A
0006 B7   3002                STA    $3002     ; Sauvegarde de A dans {$3002}
                                               ;       $F4 + $FC = $1F0
                                               ;               le   1   sera mis dans le flag C
                                               ;                              (carry retenue)
                                               ;
                        ;---Addition des poids forts
0009 B6   1000                LDA    $1001-1   ; Charger {$1000} dans A
```

```
000C B9   2000              ADCA   $2001-1   ; Addition A + {$2000} + C et résultat dans A
000F B7   3001              STA    $3002-1   ; Sauvegarde de A dans {$3001}
                                             ;           $05 + $09 + 1 = $0F
```

Après exécution :

```
{$3001}=$0F
{$3002}=$F0
```

A chaque addition toute retenue (0 ou 1) est sauvegardée automatiquement dans le bit C du registre CC.

Pour l'addition des poids forts, si il y a retenue :
-- Soit on prévoit un programme pour que le cas n'arrive pas !
-- Soit on place la retenue (bits C) dans un octet en $3000 pour cet exemple. Ce qui ferai un résultat sur 16 + 8 = 24 bits

Prog 23 : Addition sur 16 bits (variante avec l'accumulateur D)

{.....} = contenu de

```
                        ;---Prog 23 : Addition sur 16 bits (variante avec l'accumulateur D)
0000                            ORG    $0000  ;
0000      0100          Val_1   EQU    $0100  ;
0000      0200          Val_2   EQU    $0200  ;
0000      0300          Val_3   EQU    $0300  ;
                                             ;
0000 FC   0100                    LDD    Val_1  ; Charger {$0100} --> D
0003 F3   0200                    ADDD   Val_2  ; Additionner l'adrs1 et l'adrs2 --> D
0006 FD   0300                    STD    Val_3  ; Stocker le résultat de l'addition --> l'adrs3
```

Prog 24 : Addition sur 32 bits (avec l'accumulateur D)

```
                        ;---Prog 24 : Addition sur 32 bits (avec l'accumulateur D)
0000                            ORG    $0000      ;
                                                  ;
0000      1000          Adrs1   EQU    $1000      ;
0000      2000          Adrs2   EQU    $2000      ;           [Prog. du 6809 de RZ p066]
0000      3000          Adrs3   EQU    $3000      ;
                                                  ;
0000 FC   1002                    LDD    Adrs1+2    ; Chager la partie base de l'opérande 1
0003 F3   2002                    ADDD   Adrs2+2    ; Ajouter la partie base de l'opérande 2
0006 FD   3002                    STD    Adrs3+2    ; Stocker la partie base du résultat
                                                  ;
0009 FC   1000                    LDD    Adrs1      ; Chager la partie haute de l'opérande 1
000C C9   00                      ADCB   #0         ; Ajouter 0 et la retenue --> B
000E 89   00                      ADCA   #0         ; Ajouter 0 et la retenue --> A
0010 F3   2000                    ADDD   Adrs2      ; Ajouter la partie haute de l'opérande 2
0013 FD   3000                    STD    Adrs3      ; Stocker la partie haute du résultat
```

Prog 25 : Recherche de la valeur $40 ('@') dans un tableau de 100 éléments

La rechercher de la valeur $40 (code ASCII de @) dans un tableau de 100 éléments

L'adresse de départ de ce tableau BASE $0200.la taille du tableau dans COMPT $0100

Il écrit l'adresse de la case mémoire contenant le symbole @ dans ($0000 et $0001)

```
                        ;---Prog 25 : Recherche de la valeur $40 ('@')
                        ;             dans un tableau de 100 éléments
                        ;
0000                              ORG    $0000               ;
0000      0020          Val       EQU    $20                 ;
                                                             ;
0100                              ORG    $0100               ;
0100 0A 01 03 04                  FCB    $0A,$01,$03,$04,$01 ;
0104 01                                                      ;
0105 40                           FCB    $40                 ; code ascii de @
0106 01 07 C0 F1                  FCB    $01,$07,$C0,$F1,$B1 ;
```

```
010A B1                                                    ;
                                                           ;
FC00                                ORG     $FC00          ;
FC00      0200          BASE        EQU     $0200          ;
FC00      0100          COMPT       EQU     $0100          ;
                                                           ;
FC00 8E   0200                        LDX     #BASE          ;
FC03 86   40                          LDA     #$40           ; code ascii de @
FC05 F6   0100                        LDB     COMPT          ;
                                                           ;
FC08 A1   80            TEST        CMPA    ,X+            ; comparaison B avec A et +1 sur X
FC0A 27   03                          BEQ     TROUVE         ; le car recherché est trouvé
FC0C 5A                               DECB                   ; décrémenter B
FC0D 26   F9                          BNE     TEST           ; est ce le dernier élément ?
                                                           ;
FC0F 1F   10            TROUVE      TFR     X,D            ;
FC11 83   0001                        SUBD    #$0001         ;
FC14 DD   00                          STD     $0000          ; sauvegarder l'dresse de l'élément
                                                           ; trouvé dans $0000 et $0001
                                    END                    ;
```

Prog 26 : Addition en 16 bits

Addition élément par élément, 2 blocs qui débutent respectivement aux adresses Adrs1 et Adrs2.

Ces 2 blocs ayant le même nombre d'élément, le but est de stocker le résultat dans Adrs1

```
                    ;---Prog 26 : Addition en 16 bits
0000                        ORG     $0000   ;
0000      0020      Val     EQU     $20     ;
                                            ;
0100                        ORG     $0100   ;
0100 01 03 04 01            FCB     $01,$03,$04,$01,$40,$01,$07,$C0,$F1,$B1  ;
0104 40 01 07 C0                                        ;
0108 F1 B1                                              ;
                                                        ;
0400                        ORG     $0400   ;
0400 01 01 04 01            FCB     $01,$01,$04,$01,$40,$01,$07,$C0,$01,$01  ;
0404 40 01 07 C0                                        ;
0408 01 01                                              ;
                                                        ;
FC00                        ORG     $FC00   ;
FC00      0100      Adrs1   EQU     $0100   ;
FC00      0400      Adrs2   EQU     $0400   ;
FC00      0100      COMPT   EQU     $0100   ;
                                            ;
FC00 8E   0100              LDX     #Adrs1  ;
FC03 108E 0400              LDY     #Adrs2  ;
FC07 F6   0100              LDB     COMPT   ; Nb d'élément à additionner mis dans B
FC0A 4F                     CLRA            ; RAZ du bit de retenue en prévision de
                                            ;                 la première addition
                                            ;
FC0B A6   84        BOUCLE  LDA     ,X      ; premier élément mis dans A
FC0D AB   A0                ADDA    ,Y+     ; ajout du 2ième élément, puis +1 sur Y
FC0F A7   80                STA     ,X+     ; résult sauv dans case mém Adrs1,Adrs1 + 1 sur X
FC11 5A                     DECB            ; B décrémenté
FC12 26   F7                BNE     BOUCLE  ; aussi longtemps que B n'est pas = à 0
                            END             ;
```

Prog 27 : Comptage des données positives, négatives et nulles d'une table de nombres signés de 8 bits.

Proposer un programme permettant d'effectuer le comptage des données positives, négatives et nulles d'une table de nombres signés de 8 bits. Le programme devra permettre de stocker ces résultats aux adresses $0150, $0151,$0152

INDICATIONS

Après avoir chargé la valeur dans le registre A, qui automatiquement positionne les bits N et Z, on peut utiliser les instructions de branchements qui en découlent.

```
                        ;---Prog 27 : Comptage des données positives, négatives et
                        ;             nulles d'une table de nombres signés de 8 bits
                        ;
0000                            ORG     $0000       ;
0000      0020          Val     EQU     $20         ;
                                                    ;
0000                            ORG     $0000       ;
0000 02 F1 00 03                FCB     $02,$F1,$00,$03,$00  ;
0004 00                                             ;
0005 81 02 15 03                FCB     $81,$02,$15,$03,$F1  ;
0009 F1                                             ;
                                                    ;
000A      0000          TABLE   EQU     $0000       ;
000A      000A          FIN_TAB EQU     $000A       ;
                                                    ;
FC00                            ORG     $FC00       ;
FC00 8E   0000                  LDX     #TABLE      ; Chargement du pointeur
                                                    ;
FC03 8C   000A          Boucle  CMPX    #FIN_TAB    ; Si le pointeur dépasse la fin de la table
FC06 27   21                    BEQ     FIN         ; alors FIN
FC08 A6   80                    LDA     ,X+         ; Chargement et incrémentation du pointeur
FC0A 2B   0B                    BMI     Negatif     ; Si l'opération est négative -> Négatif
FC0C 27   12                    BEQ     Nul         ; Si A = 0 -> Nul
FC0E F6   0150                  LDB     $0150       ; Sinon la données est positive
FC11 5C                         INCB                ; Incrémente le compteur situé en $0050
FC12 F7   0150                  STB     $0150       ; On mémorise la valeur
FC15 20   EC                    BRA     Boucle      ;
                                                    ;
FC17 F6   0151          Negatif LDB     $0151       ; La données est négative
FC1A 5C                         INCB                ; Incrémente le compteur situé en $0051
FC1B F7   0151                  STB     $0151       ; On mémorise la valeur
FC1E 20   E3                    BRA     Boucle      ;
                                                    ;
FC20 F6   0152          Nul     LDB     $0152       ; La données est nul
FC23 5C                         INCB                ; Incrémente le compteur situé en $0052
FC24 F7   0152                  STB     $0152       ; On mémorise la valeur
FC27 20   DA                    BRA     Boucle      ;
                                                    ;
                        FIN     END                 ;
```

Prog 28 : Soustraction en 16 bits

```
                        ;---Prog 28 : Soustraction en 16 bits
                        ;
0000                            ORG     $0000       ;
0000      1000          Adrs1   EQU     $1000       ;
0000      2000          Adrs2   EQU     $2000       ;
0000      3000          Adrs3   EQU     $3000       ;
                                                    ;      Hi     Lo
0000 FC   1000                  LDD     Adrs1       ;   {Adrs1, Adrs1+1} --> D
0003 B3   2000                  SUBD    Adrs2       ;
0006 FD   3000                  STD     Adrs3       ;
                        FIN     END                 ;
```

Prog 29 : Multiplication de nombres de 16 bits

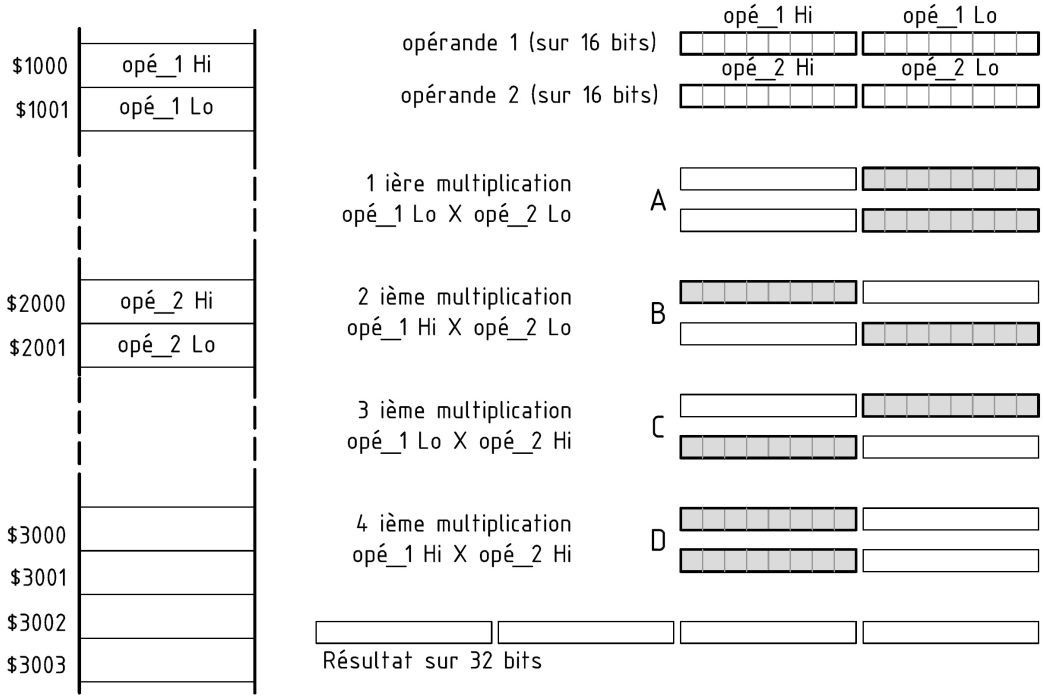
Le 6809 possède une instruction de multiplication `{A} x {B} --> D`

Cette instruction **MUL** met le bit de retenue C à 1 si le bit 7de l'accumulateur B est à 1

Dans le programme suivant ont multiplie deux opérande de 16 bits et le résultat sera sur 32 bits.

Pour cela on a recours à 4 multication de 8 bits.

Multiplication de deux opérandes de 16 bits (Résultat sur 32 bits)




```
                        ;---Prog 29 : Multiplication de nombres de 16 bits
                                                        ;
0000                                ORG     $0000       ;
                                                        ;
0000        1000        Adrs1   EQU     $1000       ;
0000        2000        Adrs2   EQU     $2000       ;           [Prog. du 6809 de RZ p076]
0000        3000        Adrs3   EQU     $3000       ;
                                                        ;
0000 7F     3000                CLR     Adrs3       ;
0003 7F     3001                CLR     Adrs3+1     ;
                                                        ;
0006 B6     1001                LDA     Adrs1+1     ;   opérande 1 Low --> A
0009 F6     2001                LDB     Adrs2+1     ;   opérande 2 Low --> B
000C 3D                         MUL                 ; A  Opé1_Lo x Opé2_Lo --> D
000D FD     3002                STD     Adrs3+2     ;   premier produit partiel
                                                        ;
0010 B6     1000                LDA     Adrs1       ;   opérande 1 Hight --> A
0013 F6     2001                LDB     Adrs2+1     ;   opérande 2 Low --> B
0016 3D                         MUL                 ; B  Opé1_Hi x Opé2_Lo --> D
0017 F3     3001                ADDD    Adrs3+1     ;   second octet le plus haut
001A FD     3001                STD     Adrs3+1     ;
                                                        ;
001D B6     1001                LDA     Adrs1+1     ;   opérande 1 Low --> A
0020 F6     2000                LDB     Adrs2       ;   opérande 2 Hight --> B
0023 3D                         MUL                 ; C  Opé1_Lo x Opé2_Hi --> D
0024 F3     3001                ADDD    Adrs3+1     ;   16 bits low effectué
0027 FD     3001                STD     Adrs3+1     ;
002A 24     03                  BCC     Suivant     ;   si pas de retenue alors allez au suivant
002C 7C     3000                INC     Adrs3       ;   additionner le bit de retenue
                                                        ;
002F B6     1000        Suivant LDA     Adrs1       ;   opérande 1 Hight --> A
0032 F6     2000                LDB     Adrs2       ;   opérande 2 Hight --> B
0035 3D                         MUL                 ; D  Opé1_Hi x Opé2_Hi --> D
0036 F3     3000                ADDD    Adrs3       ;   octet le plus haut
0039 FD     3000                STD     Adrs3       ;   valeur finale 16 bits hi
```

Circuits d'Interfaces de la famille 6800 et 6809

La plupart des circuits d'interface du 6800 sont compatibles avec le 6809

Réf.		Désignations	
6810	RAM	128 Ko 8 bits	
6830	ROM	1024 Ko 8 bits	
6821	**PIA**	Interface parallèle programmable	(Peripheral Interface Adaptor)
6828	PIC	Contrôleur de priorité d'interruption	(Priority Interrupt Controler)
6829	MMU	Interface de gestion mémoire	permet d'aborder un espace de mémoire de 2 Mo
6830	ROM	1 Ko par 8 bits	
6839	ROM	Mathématique	
6840	**PTM**	3 temporisateurs programmable	
6843	FDC	Contrôleur de disque souple simple densité	
6844	DMAC	Contrôleur d'accès mémoire	
6845	CRTC	Contrôleur de visualisation	
6846	ROM	Mémoire ROM 2 Ko, port parallèle 8 bits avec Temporisateur 16 bits	
6847	CRTC	Color Video Display Generator	
6850	**ACIA**	Interface série **asynchrone** (RS 232)	
6852	SSDA	Interface série **synchrone**	
6854	ADLC	Contrôleur de transmission avec protocole	
6855	DMA	Contrôleur d'accès mémoire (pas introduit sur le marché)	
68488	GPIA	Interface IEEE-488	
9365	GPIA	Contrôleur d'écran graphique	512 x 512 (entrelacé) THOMSON
9366	GPIA	Contrôleur d'écran graphique	512 x 256 (non entrelacé)
6829	**MMU**	Interface d'extension mémoire	
6839	ROM	Mémoire ROM mathématique	
6883		Synchrounous Adress Multiplexer (idem que 74 LS 783 et 74 LS 785)	

Table ASCII Description étendue de l'usage des caractères de contrôle (caractères 0 à 31)

$00	000	**NUL** (NULL) : caractère nul Typiquement (et spécialement en PureBasic) utilisé pour indiquer la fin d'une chaîne. Originellement une NOP, un caractère à ignorer. Lui donner le code 0 permettait de prévoir des réserves sur les bandes perforées en laissant des zones sans perforation pour insérer de nouveaux caractères a posteriori. Avec le développement du langage C il a pris une importance particulière quand il a été utilisé comme indicateur de fin de chaîne de caractères.
$01	001	**SOH** (Start Of Heading) : début de titre ou début d'en-tête Indique le début d'un bloc de données, ou la zone d'en-tête d'un bloc de données. Il est aujourd'hui souvent utilisé dans les communications séries pour permettre la synchronisation après erreur14.
$02	002	**STX** (Start of TeXt) : début de texte Typiquement envoyé comme premier caractère dans un bloc de texte, pendant les communications.
$03	003	**ETX** (End of TeXt) : fin de texte Typiquement envoyé comme dernier caractère dans un bloc de texte, pendant les communications.
$04	004	**EOT** (End Of Transmission) : fin de transmission Utilisé pour indiquer la fin d'une transmission.
$05	005	**ENQ** (ENQuiry) : requête - invitation à la transmission Envoyé à un récepteur afin d'obtenir une réponse.
$06	006	**ACK** (ACKnowledge) : accusé de réception Envoyé par un récepteur pour indiquer qu'il a reçu et/ou compris la requête.
$07	007	**BEL** (BELL) : cloche Produit un signal sonore (provoque l'émission d'un 'bip' par le haut-parleur du PC)
$08	008	**BS** (BackSpace) : retour arrière Déplace le curseur d'une position vers la gauche (pourrait également effacer le caractère à gauche du curseur avant d'effectuer le mouvement)
$09	009	**HT** (Horizontal Tab) : tabulation horizontale Typiquement utilisé pour la mise en forme de tableaux dans un texte.
$0A	010	**LF** (LineFeed) : saut de ligne Le caractère utilisé pour représenter l'action de passer une ligne sur une machine à écrire ou une imprimante en mode texte. Typiquement utilisé comme, ou partie des, caractères de fin de ligne.
$0B	011	**VT** (Vertical Tab) : tabulation verticale Même chose que la tabulation (horizontale), mais le déplacement s'effectue d'une rangée vers le bas au lieu d'une colonne vers la droite.
$0C	012	**FF** (Form Feed) : saut de page Caractère typiquement utilisé pour indiquer à une imprimante (en mode texte) de passer à la page (feuille) suivante.
$0D	013	**CR** (Carriage Return) : retour chariot Le caractère qui représente l'action de ramener la tête d'une machine à écrire ou d'une imprimante au début de la ligne. Typiquement utilisé comme, ou partie des, caractères de fin de ligne.
$0E	014	**SO** (Shift Out) : mouvement sortant Début d'un bloc de caractères dont la signification dépend de l'implémentation.

$0F	015	**SI** (Shift In) : mouvement entrant Ferme la transmission du type de bloc ci-dessus.
$10	016	**DLE** (Data Link Escape) : échappement de lien de donnée Utilisé pour indiquer que le caractère de contrôle suivant devrait être interprété comme donnée et non comme caractère de contrôle.
$11	017	**DC1** (Device Control 1) : contrôle de périphérique 1 Typiquement utilisé pour activer une partie d'un équipement. L'usage le plus courant aujourd'hui est en tant que caractère XON dans les communications série à contrôle de flux logiciel.
$12	018	**DC2** (Device Control 2) : contrôle de périphérique 2 Un autre caractère de contrôle de périphérique. Son usage dépend du contexte.
$13	019	**DC3** (Device Control 3) : contrôle de périphérique 3 Typiquement utilisé pour désactiver une partie d'un équipement. L'usage le plus courant aujourd'hui est en tant que caractère XOFF dans les communications série à contrôle de flux logiciel.
$14	020	**DC4** (Device Control 4) : contrôle de périphérique 4 Un autre caractère de contrôle de périphérique.
$15	021	**NAK** (Negative AcKnowledge) : accusé de réception négatif Typiquement utilisé pour signaler des données non-reçues ou non-comprises (erronée).
$16	022	**SYN** (SYNchronous idle) : attente synchronisée Comme son nom l'indique, il s'agit d'un signal envoyé à intervalle régulier pour indiquer que le canal de communication est en attente, mais toujours actif.
$17	023	**ETB** (End of Transmission Block) : fin de transmission de bloc Utilisé pour contrôler la transmission de donnée en indiquant la fin de bloc. A ne pas confondre avec EOT.
$18	024	**CAN** (CANcel) : annulation Signifie généralement que la donnée envoyée précédemment devrait être ignorée, bien que les détails dépendent de l'application.
$19	025	**EM** (End of Medium) : fin de média Utilisé pour indiquer la fin d'un média, par exemple la fin d'un lecteur de bande
$1A	026	**SUB** (SUBstitute) : substitution Un caractère utilisé pour indiquer qu'un caractère a été substitué.
$1B	027	**ESC** (ESCape) : échappement Le caractère produit habituellement en appuyant sur la touche 'ECHAP' de votre clavier, utilisé dans les "séquences d'échappement" pour fournir des informations de formattage aux afficheurs de texte (consoles, imprimantes, etc..)
$1C	028	**FS** (File Separator) : séparateur de fichier
$1D	029	**GS** (Group Separator) : séparateur de groupe
$1E	030	**RS** (Record separator) : séparateur d'enregistrement
$1F	031	**US** (Unit separator) : séparateur d'unité
$7F	127	**DEL** (Delete) : effacement. Lui donner le code 127 (1111111 en binaire) permettait de supprimer a posteriori un caractère sur les bandes perforées qui codaient les informations sur 7 bits. N'importe quel caractère pouvait être transformé en DEL en complétant la perforation des 7 bits qui le composaient.

Table ASCII (0 - 127)

Déci	Hexa	Octal	
000	$00	000	NUL
001	$01	001	SOH
002	$02	002	STX
003	$03	003	ETX
004	$04	004	EOT
005	$05	005	ENQ
006	$06	006	ACK
007	$07	007	BEL
008	$08	010	BS
009	$09	011	HT
010	$0A	012	LF
011	$0B	013	VT
012	$0C	014	FF
013	$0D	015	CR
014	$0E	016	SO
015	$0F	017	SI

Déci	Hexa	Octal	
032	$20	040	space
033	$21	041	!
034	$22	042	"
035	$23	043	#
036	$24	044	$
037	$25	045	%
038	$26	046	&
039	$27	047	'
040	$28	050	(
041	$29	051	)
042	$2A	052	*
043	$2B	053	+
044	$2C	054	,
045	$2D	055	-
046	$2E	056	.
047	$2F	057	/

Déci	Hexa	Octal	
064	$40	100	@
065	$41	101	A
066	$42	102	B
067	$43	103	C
068	$44	104	D
069	$45	105	E
070	$46	106	F
071	$47	107	G
072	$48	110	H
073	$49	111	I
074	$4A	112	J
075	$4B	113	K
076	$4C	114	L
077	$4D	115	M
078	$4E	116	N
079	$4F	117	O

Déci	Hexa	Octal	
096	$60	140	`
097	$61	141	a
098	$62	142	b
099	$63	143	c
100	$64	144	d
101	$65	145	e
102	$66	146	f
103	$67	147	g
104	$68	150	h
105	$69	151	i
106	$6A	152	j
107	$6B	153	k
108	$6C	154	l
109	$6D	155	m
110	$6E	156	n
111	$6F	157	o

Déci	Hexa	Octal	
016	$10	020	DLE
017	$11	021	DC1
018	$12	022	DC2
019	$13	023	DC3
020	$14	024	DC4
021	$15	025	NAK
022	$16	026	SYN
023	$17	027	ETB
024	$18	030	CAN
025	$19	031	EM
026	$1A	032	SUB
027	$1B	033	ESC
028	$1C	034	FS
029	$1D	035	GS
030	$1E	036	RS
031	$1F	037	US

Déci	Hexa	Octal	
048	$30	060	0
049	$31	061	1
050	$32	062	2
051	$33	063	3
052	$34	064	4
053	$35	065	5
054	$36	066	6
055	$37	067	7
056	$38	070	8
057	$39	071	9
058	$3A	072	:
059	$3B	073	;
060	$3C	074	<
061	$3D	075	=
062	$3E	076	>
063	$3F	077	?

Déci	Hexa	Octal	
080	$50	120	P
081	$51	121	Q
082	$52	122	R
083	$53	123	S
084	$54	124	T
085	$55	125	U
086	$56	126	V
087	$57	127	W
088	$58	130	X
089	$59	131	Y
090	$5A	132	Z
091	$5B	133	[
092	$5C	134	\
093	$5D	135	]
094	$5E	136	^
095	$5F	137	_

Déci	Hexa	Octal	
112	$70	160	p
113	$71	161	q
114	$72	162	r
115	$73	163	s
116	$74	164	t
117	$75	165	u
118	$76	166	v
119	$77	167	w
120	$78	170	x
121	$79	171	y
122	$7A	172	z
123	$7B	173	{
124	$7C	174	|
125	$7D	175	}
126	$7E	176	~
127	$7F	177	DEL

Table ASCII (128 - 255)

Déci	Hexa	Octal	
128	$80	200	Ç
129	$81	201	ü
130	$82	202	é
131	$83	203	â
132	$84	204	ä
133	$85	205	à
134	$86	206	å
135	$87	207	ç
136	$88	210	ê
137	$89	211	ë
138	$8A	212	è
139	$8B	213	ï
140	$8C	214	î
141	$8D	215	ì
142	$8E	216	Ä
143	$8F	217	Å

Déci	Hexa	Octal	
160	$A0	240	á
161	$A1	241	í
162	$A2	242	ó
163	$A3	243	ú
164	$A4	244	ñ
165	$A5	245	Ñ
166	$A6	246	ª
167	$A7	247	º
168	$A8	250	¿
169	$A9	251	⌐
170	$AA	252	¬
171	$AB	253	½
172	$AC	254	¼
173	$AD	255	¡
174	$AE	256	«
175	$AF	257	»

Déci	Hexa	Octal	
192	$C0	300	└
193	$C1	301	┴
194	$C2	302	┬
195	$C3	303	├
196	$C4	304	─
197	$C5	305	┼
198	$C6	306	╞
199	$C7	307	╟
200	$C8	310	╚
201	$C9	311	╔
202	$CA	312	╩
203	$CB	313	╦
204	$CC	314	╠
205	$CD	315	═
206	$CE	316	╬
207	$CF	317	╧

Déci	Hexa	Octal	
224	$E0	340	α
225	$E1	341	ß
226	$E2	342	Γ
227	$E3	343	π
228	$E4	344	Σ
229	$E5	345	σ
230	$E6	346	µ
231	$E7	347	τ
232	$E8	350	Φ
233	$E9	351	Θ
234	$EA	352	Ω
235	$EB	353	δ
236	$EC	354	∞
237	$ED	355	φ
238	$EE	356	ε
239	$EF	357	∩

Déci	Hexa	Octal	
144	$90	220	É
145	$91	221	æ
146	$92	222	Æ
147	$93	223	ô
148	$94	224	ö
149	$95	225	ò
150	$96	226	û
151	$97	227	ù
152	$98	230	ÿ
153	$99	231	Ö
154	$9A	232	Ü
155	$9B	233	¢
156	$9C	234	£
157	$9D	235	¥
158	$9E	236	Pts
159	$9F	237	ƒ

Déci	Hexa	Octal	
176	$B0	260	░
177	$B1	261	▒
178	$B2	262	▓
179	$B3	263	│
180	$B4	264	┤
181	$B5	265	╡
182	$B6	266	╢
183	$B7	267	╖
184	$B8	270	╕
185	$B9	271	╣
186	$BA	272	║
187	$BB	273	╗
188	$BC	274	╝
189	$BD	275	╜
190	$BE	276	╛
191	$BF	277	┐

Déci	Hexa	Octal	
208	$D0	320	╨
209	$D1	321	╤
210	$D2	322	╥
211	$D3	323	╙
212	$D4	324	Ô
213	$D5	325	╒
214	$D6	326	╓
215	$D7	327	╫
216	$D8	330	╪
217	$D9	331	┘
218	$DA	332	┌
219	$DB	333	█
220	$DC	334	▄
221	$DD	335	▌
222	$DE	336	▐
223	$DF	337	▀

Déci	Hexa	Octal	
240	$F0	360	≡
241	$F1	361	±
242	$F2	362	≥
243	$F3	363	≤
244	$F4	364	⌠
245	$F5	365	⌡
246	$F6	366	÷
247	$F7	367	≈
248	$F8	370	≈
249	$F9	371	∙
250	$FA	372	·
251	$FB	373	√
252	$FC	374	ⁿ
253	$FD	375	²
254	$FE	376	■
255	$FF	377	