Chapitre 1
Initiation à l'assembleur 68000

Version du 21/11/2017

Table des matières

I. Introduction....3
1. Le langage machine....3
2. L'assembleur....3
2.1. Le langage assembleur....3
2.2. Le programme assembleur....3
3. Structure d'un code source....4
3.1. Le champ étiquette....5
3.2. Le champ mnémonique....5
3.3. Le champ opérande....6
3.4. Le champ commentaire....7
II. Architecture du 68000....8
1. Les bus d'adresse et de donnée....8
2. Les modes de fonctionnement....8
2.1. Le mode superviseur....8
2.2. Le mode utilisateur....8
3. Les registres....9
3.1. Les registres de donnée....9
3.2. Les registres d'adresse et les pointeurs de pile....9
3.3. Le compteur programme....10
3.4. Le registre d'état....10
III. Les principales directives d'assemblage....12
1. La directive ORG....12
2. La directive EQU....12
3. La directive DC....13
4. La directive DS....13
IV. Les modes d'adressage....14
1. L'adresse effective....14
2. Les modes d'adressage qui ne spécifient pas d'emplacement mémoire....14
2.1. Direct par registre de donnée : Dn....14
2.2. Direct par registre d'adresse : An....15
2.3. Immédiat : #<data>....15
3. Les modes d'adressage qui spécifient un emplacement mémoire....16
3.1. Indirect par registre d'adresse : (An)....16
3.2. Indirect par registre d'adresse avec postincrémentation : (An)+....16
3.3. Indirect par registre d'adresse avec prédécrémentation : –(An)....17
3.4. Indirect par registre d'adresse avec déplacement : d16(An)....17

3.5. Indirect par registre d'adresse avec déplacement et index : d8(An,Xn)....18
3.6. Indirect relatif au compteur programme avec déplacement : d16(PC)....18
3.7. Indirect relatif au compteur programme avec déplacement et index : d8(PC,Xn)....19
3.8. Absolu long : (xxx).L....19
3.9. Absolu court : (xxx).W....19
4. Exemples....19
4.1. MOVE.W A1,D2....20
4.2. MOVE.W (A1),D2....20
4.3. MOVE.L #$100A,D2....20
4.4. MOVE.L $100A,D2....20
4.5. MOVE.W #36,(A0)....21
4.6. MOVE.B D1,(A1)+....21
4.7. MOVE.L $1004,-(A2)....21
4.8. MOVE.L -(A2),-(A2)....21
4.9. MOVE.B 5(A1),-1(A1,D0.W)....22
4.10. MOVE.W 2(A1,D1.L),-6(A2)....22
4.11. MOVE.W $1000(PC),$100A....22
V. Les branchements....23
1. Les branchements inconditionnels....23
2. Les branchements conditionnels....23
2.1. Les branchements à comparaison sur un flag....24
2.2. Les branchements à comparaison non signée et à comparaison signée....24
3. Exemples de boucle....25
4. L'instruction DBRA....26
VI. La pile....27
1. Définitions et principes de base....27
2. L'instruction MOVEM....29
VII. Les sous-programmes....31
VIII. Les principales instructions du 68000....33
1. Les mouvements de donnée....33
2. Les opérations arithmétiques....33
2.1. L'addition....33
2.2. La soustraction....34
2.3. La multiplication....34
2.4. La division....34
2.5. Autres....34
3. Les opérations logiques....35
4. Les décalages et les rotations....35
5. Les manipulations de bit....35
6. Les tests et les comparaisons....35

I. Introduction

1. Le langage machine

Le langage machine est le langage natif d'un microprocesseur. Il est constitué d'une suite de codes binaires : le code machine. Chaque code correspond à une opération que peut traiter le microprocesseur.

Le langage machine est le seul langage que peut exécuter un microprocesseur et chaque microprocesseur possède son propre langage machine. Néanmoins, certains microprocesseurs et plus particulièrement ceux appartenant à une même famille peuvent avoir des langages compatibles ou très proches.

Pour la suite de ce cours, nous travaillerons avec le **microprocesseur 68000**.

2. L'assembleur

Le terme *assembleur* désigne deux choses différentes :
- Un langage de programmation ;
- Un programme informatique.

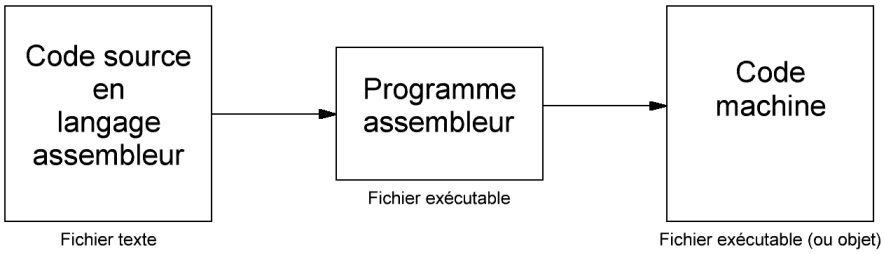
2.1. Le langage assembleur

Tout comme le langage machine, le langage assembleur (ou langage d'assemblage) est un langage de programmation propre à un microprocesseur. Il affecte un nom à chaque instruction que peut exécuter un microprocesseur. Ce nom est appelé *mnémonique*.

Le langage assembleur est donc très proche du langage machine. Il ne fait qu'effectuer une bijection entre des mnémoniques, compréhensibles par l'homme, et des codes binaires, compréhensibles par la machine.

2.2. Le programme assembleur

Un programme assembleur est un programme informatique qui traduit du code assembleur en code machine. Ce processus de traduction s'appelle l'assemblage. Il peut exister plusieurs programmes assembleurs différents pour un même langage assembleur (tout comme il existe plusieurs compilateurs C différents pour le langage C).



Une fois l'assemblage terminé, le code machine généré peut être chargé en mémoire et exécuté par un microprocesseur.

3. Structure d'un code source

Exemple d'un code source en langage assembleur 68000 :

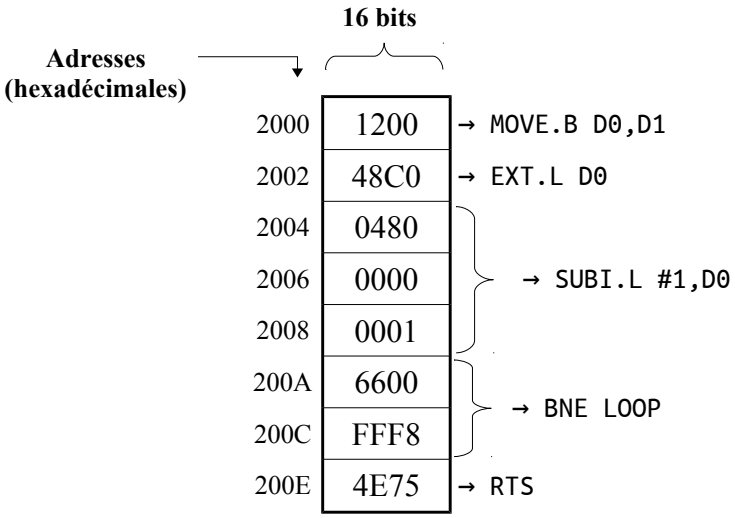
```
        ORG     $2000   ; Place le programme à l'adresse $2000.
START   MOVE.B  D0,D1   ; D0 → D1.
        EXT.L   D0      ; Extension de signe.
LOOP    SUBI.L  #1,D0   ; D0 - 1 → D0.
        BNE     LOOP    ; Saut à LOOP tant que D0 n'est pas nul.
        RTS             ; Retour de sous-programme.
```

Ce code source contient un certain nombre d'instructions qui sont nouvelles pour vous. N'essayez pas de les comprendre ou de trouver ce que fait ce programme ; il ne fait rien d'intéressant. Il nous sert simplement d'exemple pour identifier les différentes parties d'un code source.

Un code source contient une instruction par ligne. Une instruction est divisée en quatre champs distincts :

Étiquette	Mnémonique	Opérande	Commentaire
	ORG	$2000	; Place le programme à l'adresse $2000.
START	MOVE.B	D0,D1	; D0 → D1.
	EXT.L	D0	; Extension de signe.
LOOP	SUBI.L	#1,D0	; D0 - 1 → D0.
	BNE	LOOP	; Saut à LOOP tant que D0 n'est pas nul.
	RTS		; Retour de sous-programme.

Une fois assemblé et chargé en mémoire, voici le code machine obtenu :



Les instructions se retrouvent en mémoire sous la forme de mots de 16 bits. On constate que certaines instructions prennent plus de place en mémoire que d'autres.

Pour le 68000 :

- la taille minimale d'une instruction en mémoire est d'un mot de 16 bits (2 octets) ;
- la taille maximale d'une instruction en mémoire est de cinq mots de 16 bits (10 octets).

La première ligne (`ORG $2000`) n'a pas été convertie en langage machine. Nous verrons pourquoi un peu plus tard.

Représentation du code assembleur avec son code machine associé :

```
                                ORG     $2000  ; Place le programme à l'adresse $2000.
002000 1200           START     MOVE.B  D0,D1  ; D0 → D1.
002002 48C0                     EXT.L   D0     ; Extension de signe.
002004 0480 00000001  LOOP      SUBI.L  #1,D0  ; D0 - 1 → D0.
00200A 6600 FFF8                BNE     LOOP   ; Saut à LOOP tant que D0 n'est pas nul.
00200E 4E75                     RTS            ; Retour de sous-programme.
```

3.1. Le champ étiquette

Le champ étiquette est le premier champ d'une instruction en langage assembleur.

Une étiquette est facultative. Si la ligne contient une étiquette, cette dernière doit débuter sur le premier caractère de la ligne. Si la ligne ne contient pas d'étiquette, le premier caractère de la ligne doit être un espace ou une tabulation.

L'assembleur attribue à une étiquette la valeur de l'adresse à laquelle sera placée l'instruction qui suit l'étiquette dans le code source. Par exemple, dans notre cas :

- la valeur de l'étiquette `START` sera 2000_{16} ;
- la valeur de l'étiquette `LOOP` sera 2004_{16}.

Il est possible d'employer ces étiquettes en tant qu'opérandes dans le reste du programme. L'assembleur remplacera l'étiquette par la valeur qui lui a été affectée. Le programmeur n'a donc pas à se soucier de la valeur des adresses lors de la phase de développement.

Les étiquettes sont généralement utilisées pour les instructions de branchement.

3.2. Le champ mnémonique

Le champ mnémonique contient le nom d'une instruction ou d'une directive d'assemblage.

Une instruction peut être traduite en langage machine. Elle appartient au jeu d'instructions d'un microprocesseur.

Une directive d'assemblage n'est pas une instruction. Elle ne fait pas partie du jeu d'instructions d'un microprocesseur. Elle ne peut pas être traduite en langage machine.

Une directive appartient au programme assembleur et non pas au langage assembleur. Elle permet, comme son nom l'indique, de donner une directive au programme assembleur au moment de l'assemblage.

Voilà pourquoi la première ligne de notre programme n'a pas été traduite en langage machine. Le `ORG` n'est pas une instruction, mais une directive d'assemblage. Cette directive indique à l'assembleur l'adresse mémoire à laquelle devront être placées les instructions lors de leur chargement en mémoire (ici l'adresse 2000_{16}).

Certaines mnémoniques du 68000 acceptent une extension. Cette extension sert à préciser la taille de travail d'une instruction ou d'une directive d'assemblage. Les trois extensions principales sont :
- l'extension **`.B`** (*Byte*) pour l'octet (8 bits) ;
- l'extension **`.W`** (*Word*) pour le mot (16 bits) ;
- l'extension **`.L`** (*Long*) pour le mot long (32 bits).

L'extension **`.S`** (*Short*) peut également être acceptée par certaines instructions de branchement. Elle est alors équivalente à l'extension **`.B`** (*Byte*).

3.3. Le champ opérande

Certaines instructions ou directives d'assemblage manipulent des données. Les opérandes indiquent à une instruction où elle peut trouver les données dont elle a besoin.

Une instruction 68000 peut avoir zéro, un ou deux opérandes (jamais plus). Dans le cas de deux opérandes, l'opérande de gauche est l'opérande source, l'opérande de droite est l'opérande destination.

Par exemple, dans l'instruction suivante : `MOVE.B D0,D1`
- **D0** est l'opérande source (il ne sera pas modifié par l'instruction) ;
- **D1** est l'opérande destination (il sera modifié par l'instruction).

Nous verrons par la suite à quoi correspondent **D0** et **D1**.

Certains opérandes peuvent être des nombres. L'assembleur 68000 accepte une représentation des nombres dans les bases 2, 16 et 10.
- Un nombre en base 2 se représente à l'aide du préfixe **%** (par exemple : `%10001001`) ;
- Un nombre en base 16 se représente à l'aide du préfixe **$** (par exemple : `$2EF8`) ;
- Un nombre en base 10 ne prend aucun préfixe.

3.4. Le champ commentaire

Ce champ permet d'introduire des commentaires dans un programme.

Un commentaire débute par un point virgule. Tout ce qui suit ce point virgule jusqu'à la fin de la ligne ne sera pas pris en compte par l'assembleur lors de la phase d'assemblage.

Les commentaires n'ont aucun effet sur le code machine généré, mais ils procurent une aide fondamentale à la bonne compréhension d'un code source. De bons commentaires constituent une part essentielle de l'écriture de programmes en langage assembleur.

Remarque :
Un commentaire peut se placer n'importe où sur une ligne, y compris en début de ligne à la place d'une étiquette.

II. Architecture du 68000

1. Les bus d'adresse et de donnée

Le 68000 possède un bus d'adresse de 23 fils et un bus de donnée de 16 fils. Il est donc capable d'adresser 8 Mi mots de 16 bits.

Toutefois, le 68000 possède quelques signaux sur son bus de commande (notamment **UDS** et **LDS**) qui lui permettent d'accéder à la mémoire par mot de 8 bits et de se comporter comme si son bus d'adresse était de 24 fils.

Pour la suite du cours, nous considérerons donc que le 68000 possède **24 fils d'adresse** et qu'il est capable d'adresser 16 Mi mots de 8 bits, c'est-à-dire **16 Mio**.

Même si le 68000 est capable d'accéder à la mémoire octet par octet, son espace mémoire est physiquement organisé en mots de 16 bits. Cette organisation implique, entre autres, les caractéristiques suivantes :
- Une instruction est codée au minimum sur un mot de 16 bits ;
- Une instruction se trouve toujours à une adresse paire ;
- La lecture ou l'écriture d'un octet est possible à partir d'une adresse paire ou impaire ;
- La lecture ou l'écriture d'un mot de 16 bits est possible uniquement à partir d'une adresse paire ;
- La lecture ou l'écriture d'un mot de 32 bits est possible uniquement à partir d'une adresse paire.

2. Les modes de fonctionnement

Le 68000 possède deux modes de fonctionnement : le mode superviseur et le mode utilisateur.

2.1. Le mode superviseur

Dans ce mode, le 68000 ne fait l'objet d'aucune limitation. Il a la possibilité d'exécuter n'importe quelles instructions appartenant à son jeu d'instructions.

Ce mode de fonctionnement est principalement utilisé par les systèmes d'exploitation. Ces derniers nécessitent de posséder un contrôle total sur la machine.

2.2. Le mode utilisateur

Dans ce mode, le 68000 fait l'objet de quelques limitations. Il ne lui est pas permis d'exécuter un certain nombre d'instructions. Ces instructions sont dites privilégiées et ne peuvent être exécutées qu'en mode superviseur.

Ce mode de fonctionnement est principalement utilisé par les applications s'exécutant sur un système d'exploitation donné. Ces applications doivent avoir des droits limités afin de réduire les risques de corruption du système lorsqu'une application présente des dysfonctionnements.

3. Les registres

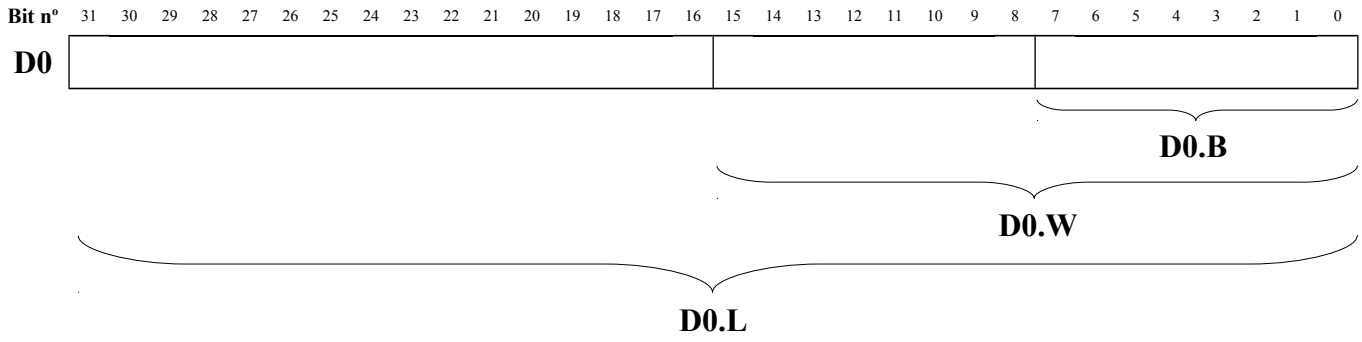
Un registre est un espace de stockage temporaire intégré dans un microprocesseur.

3.1. Les registres de donnée

Le 68000 possède huit registres de donnée d'une taille de 32 bits : **D0**, **D1**, **D2**, **D3**, **D4**, **D5**, **D6** et **D7**.

Ces registres n'ont aucune fonction prédéfinie et permettent de manipuler tout type de donnée. Ils sont accessibles sur 8, 16 et 32 bits.

Exemple du registre **D0** :

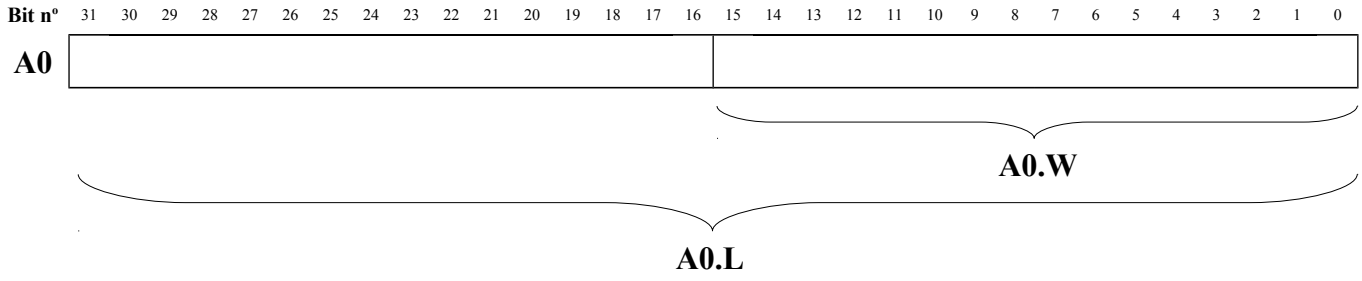


3.2. Les registres d'adresse et les pointeurs de pile

Le 68000 possède huit registres d'adresse d'une taille de 32 bits : **A0**, **A1**, **A2**, **A3**, **A4**, **A5**, **A6** et **A7**.

Ces registres servent à contenir des adresses. Ils sont accessibles sur 16 et 32 bits.

Exemple du registre **A0** :



Le 68000 possédant 24 lignes d'adresse, les bits 24 à 31 des registres d'adresse sont ignorés. La valeur de ces bits ne sera jamais positionnée sur le bus d'adresse. Par exemple, que l'on ait **A0** = \$67**9809AC**, ou bien **A0** = \$F5**9809AC**, ou encore **A0** = \$00**9809AC**, l'adresse à laquelle on accédera à partir de ce registre sera l'adresse **\$9809AC**. Pour la suite, nous laisserons toujours les huit bits de poids fort des registres d'adresse à zéro.

Le 68000 possède deux pointeurs de pile d'une taille de 32 bits :

- **SSP** (*Supervisor Stack Pointer*) qui est le pointeur de pile du mode superviseur ;
- **USP** (*User Stack Pointer*) qui est le pointeur de pile du mode utilisateur.

Le registre d'adresse **A7** tient un rôle particulier dans le 68000 puisque c'est à travers lui que se fait l'accès à ces deux pointeurs de pile :

- Lorsque le 68000 se trouve en mode superviseur, le registre **A7** est en fait le registre **SSP** ;
- Lorsque le 68000 se trouve en mode utilisateur, le registre **A7** est en fait le registre **USP**.

Ainsi, si l'on écrit en mode superviseur dans **A7**, et que l'on passe ensuite en mode utilisateur, on lira autre chose dans **A7** que ce que l'on vient d'écrire. Cette duplication de registre entre le mode superviseur et le mode utilisateur permet de gérer des piles distinctes pour chacun de ces modes.

Nous aborderons ultérieurement le fonctionnement de la pile.

3.3. Le compteur programme

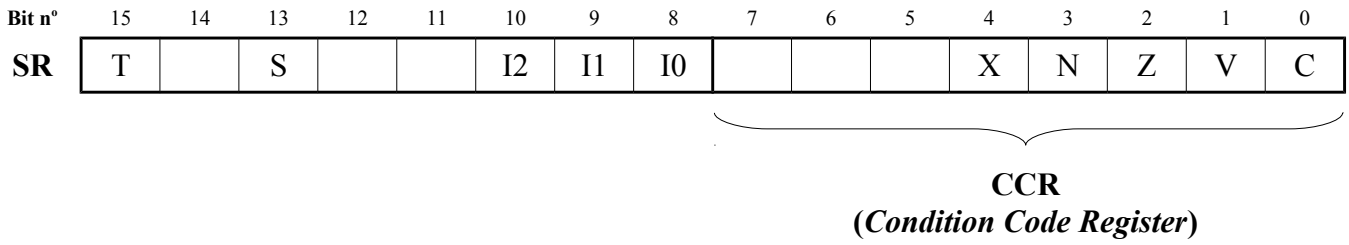
Le compteur programme **PC** (*Program Counter*) est un registre d'une taille de 32 bits qui contient l'adresse de la prochaine instruction à exécuter. Ce registre est modifié automatiquement par le 68000.

Tout comme les registres d'adresse, ses huit bits de poids fort sont ignorés et seuls ses 24 bits de poids faible sont positionnés sur le bus d'adresse.

3.4. Le registre d'état

Le registre d'état **SR** (*Status Register*) est un registre d'une taille de 16 bits qui contient l'état du microprocesseur.

L'intégralité des 16 bits du registre **SR** n'est accessible qu'en mode superviseur. Toutefois, le registre **CCR** (*Condition Code Register*) qui représente les 8 bits de poids faible du registre **SR** est, quant à lui, accessible en mode utilisateur. Il est donc possible d'accéder aux 8 bits de poids faible du registre **SR** en passant par le registre **CCR**.



Les différents bits constituant le registre d'état sont appelés *drapeaux* (ou *flags* en anglais). Ces drapeaux contiennent des informations complémentaires sur le résultat d'une opération ou sur l'état du 68000.

Le comportement général des *flags* est donné ci-dessous :

- **C** (*Carry*) : Retenue ou dépassement non signé (**C** = 1 si retenue, **C** = 0 si pas de retenue) ;
- **V** (*Overflow*) : Dépassement signé (**V** = 1 si dépassement, **V** = 0 si pas de dépassement) ;
- **Z** (*Zero*) : Zéro (**Z** = 1 si nul, **Z** = 0 si non nul) ;
- **N** (*Negative*) : Négatif (**N** = 1 si négatif, **N** = 0 si positif ou nul) ;
- **X** (*Extend*) : Extension (identique au *flag* **C** pour la plupart des instructions) ;
- **I0**, **I1**, **I2** (*Interrupt Priority Mask*) : Masque des priorités d'interruption ;
- **S** (*Supervisor State*) : Indicateur de mode (**S** = 0 si mode utilisateur, **S** = 1 si mode superviseur) ;
- **T** (*Trace Mode*) : Activation du mode *trace* (nous n'utiliserons pas le mode *trace* dans ce cours).

Toutefois, la gestion de certains *flags* peut légèrement varier d'une instruction à l'autre. Il est donc indispensable de consulter la documentation technique du 68000 afin de connaître précisément comment une instruction modifie les *flags*.

Voici quelques exemples du positionnement des flags **C**, **V**, **Z** et **N** pour les instructions d'additions :

- **Addition sur 8 bits (.B)**
  $7A + $86 = $100 (le résultat sur 8 bits est $00)
  C = 1, **V** = 0, **Z** = 1, **N** = 0

- **Addition sur 16 bits (.W)**
  $9F00 + $8E00 = $12D00 (le résultat sur 16 bits est $2D00)
  C = 1, **V** = 1, **Z** = 0, **N** = 0

- **Addition sur 32 bits (.L)**
  $70000000 + $10000000 = $80000000
  C = 0, **V** = 1, **Z** = 0, **N** = 1

- Le *flag* **N** est le bit de signe du résultat. Il prend donc la valeur du bit de poids fort du résultat.
- Le *flag* **Z** est à 1 si le résultat est nul.
- Le *flag* **C** est à 1 s'il y a une retenue.
- Le *flag* **V** se détermine en faisant la **supposition** que les nombres à additionner sont signés. Il est à 1 uniquement si l'une des deux conditions suivantes est vraie :
  - On additionne deux nombres positifs et le résultat est négatif.
  - On additionne deux nombres négatifs et le résultat est positif.

III. Les principales directives d'assemblage

1. La directive ORG

La directive `ORG` (*Origin*) sert à préciser à l'assembleur l'adresse à partir de laquelle seront assemblées les instructions en mémoire.

Un programme peut posséder plusieurs `ORG` (il faut faire attention au recouvrement).

Exemples :

```
        ORG     $1000

        ; Ces instructions seront assemblées
        ; à partir de l'adresse $1000.
        LEA     $2000,A0
        CLR.B   D1
        JSR     $5000

        ORG     $5000

        ; Ces instructions seront assemblées
        ; à partir de l'adresse $5000.
        MOVE.B  (A0)+,D0
        ADDQ.B  #1,D1
        RTS
```

2. La directive EQU

La directive `EQU` (*Equate*) permet d'attribuer explicitement une valeur à une étiquette. L'assembleur remplacera ensuite l'étiquette par la valeur qui lui a été attribuée.

Exemples :

```
START       EQU     $1000
PRINT       EQU     $5000
COUNT1      EQU     200
COUNT2      EQU     850

            ORG     START           ; ORG     $1000

            MOVE.L  #COUNT1,D0      ; MOVE.L  #200,D0
            JSR     PRINT           ; JSR     $5000

            MOVE.L  #COUNT2,D0      ; MOVE.L  #850,D0
            JSR     PRINT           ; JSR     $5000
```

3. La directive DC

La directive `DC` (*Define Constant*) permet de placer des données en mémoire. Ces données peuvent être des nombres en représentation décimale, hexadécimale ou binaire, mais aussi des chaînes de caractères. Dans ce dernier cas, ce sont les codes ASCII des caractères qui sont placés en mémoire.

Exemples :

```
        ORG     $1000

        DC.B    10,5,7,$7a,255,%11111001
        DC.B    "Hello World",13,10,0
        DC.W    5,6
        DC.L    5,6
```

Le contenu de la mémoire à partir de l'adresse $1000 sera donc le suivant :

```
1000 0A 05 07 7A FF F9                          DC.B    10,5,7,$7a,255,%11111001
1006 48 65 6C 6C 6F 20 57 6F 72 6C 64 0D 0A 00  DC.B    "Hello World",13,10,0
1014 00 05 00 06                                DC.W    5,6
1018 00 00 00 05 00 00 00 06                    DC.L    5,6
```

4. La directive DS

La directive `DS` (*Define Storage*) permet de réserver un espace de stockage qui pourra être utilisé lors de l'exécution d'un programme. Ceci permet d'éviter au programme d'écrire dans des emplacements qui ne lui seraient pas réservés et d'effacer des données ou du code qui ne doivent pas l'être.

Exemples :

```
        ORG     $5000

TAB1    DS.B    4  ; Réserve 4 octets en mémoire   (4 x  8 bits)  ; TAB1 = $5000
TAB2    DS.W    3  ; Réserve 3 mots en mémoire     (3 x 16 bits)  ; TAB2 = $5004
TAB3    DS.L    1  ; Réserve 1 mot long en mémoire (1 x 32 bits)  ; TAB3 = $500A
NEXT                                                              ; NEXT = $500E
```

IV. Les modes d'adressage

Un mode d'adressage permet de préciser à une instruction où se trouvent les opérandes dont elle a besoin.

À titre d'exemple, nous utiliserons l'instruction `MOVE` afin d'illustrer certains modes d'adressage. Cette instruction permet de copier le contenu d'un opérande source vers un opérande destination. La taille de l'instruction peut être de 8, 16 ou 32 bits :

```
MOVE.B  source,destination  ; source → destination (uniquement sur 8 bits)
MOVE.W  source,destination  ; source → destination (uniquement sur 16 bits)
MOVE.L  source,destination  ; source → destination (sur 32 bits)
```

1. L'adresse effective

L'adresse effective (*effective address*) est l'emplacement d'un opérande. **Attention ! une adresse effective n'est pas nécessairement une adresse mémoire !** Un opérande peut se trouver dans un registre ou dans la mémoire. Nous distinguerons deux types de modes d'adressage :

- **Les modes d'adressage qui ne spécifient pas d'emplacement mémoire** (l'adresse effective n'est pas une adresse mémoire). Il s'agit des modes d'adressage directs (l'adresse effective est un registre) et du mode d'adressage immédiat (l'adresse effective n'existe pas).

- **Les modes d'adressage qui spécifient un emplacement mémoire** (l'adresse effective est une adresse mémoire). Il s'agit des modes d'adressage indirects et des modes d'adressage absolus.

Dans la documentation technique du 68000, l'adresse effective est souvent notée **<ea>**. Nous utiliserons la même notation dans la suite de ce cours.

2. Les modes d'adressage qui ne spécifient pas d'emplacement mémoire

L'opérande spécifié par ces modes d'adressage se trouve soit directement dans un registre, soit immédiatement dans l'instruction. Il n'est pas utile de déterminer une adresse mémoire spécifique.

2.1. Direct par registre de donnée : Dn

L'adresse effective est un registre de donnée. L'opérande est lu ou écrit directement dans un registre de donnée. La taille de l'opérande dépend de la taille de l'instruction (8, 16 ou 32 bits).

Exemples :
Valeurs initiales : `D0 = $11223344` et `D1 = $AABBCCDD`

```
MOVE.B  D0,D1  ; D0.B → D1.B ; D1 = $AABBCC44 (copie sur  8 bits)
MOVE.W  D0,D1  ; D0.W → D1.W ; D1 = $AABB3344 (copie sur 16 bits)
MOVE.L  D0,D1  ; D0.L → D1.L ; D1 = $11223344 (copie sur 32 bits)
```

2.2. Direct par registre d'adresse : An

L'adresse effective est un registre d'adresse. L'opérande est lu ou écrit directement dans un registre d'adresse. La taille de l'opérande dépend de la taille de l'instruction (16 ou 32 bits).

Exemples :
Valeurs initiales : `D0 = $11223344` et `A0 = $005030B0`

```
MOVE.W  A0,D0  ; A0.W → D0.W ; D0 = $1122 30B0
MOVE.L  A0,D0  ; A0.L → D0.L ; D0 = $0050 30B0
```

Remarque :
L'instruction `MOVE` n'accepte pas de registre d'adresse comme opérande destination. Il faut utiliser les instructions `MOVEA` ou `LEA` pour assigner une valeur à un registre d'adresse (nous aborderons ce point ultérieurement).

2.3. Immédiat : #<data>

L'opérande est contenu dans le code machine de l'instruction. Il s'agit d'une donnée immédiate qui est précisée dans le programme assembleur. Cette donnée doit être précédée du caractère '**#**'.

Dans ce mode d'adressage, l'adresse effective n'existe pas, car la donnée est immédiatement traitée par l'instruction.

Exemples :
Valeur initiale : `D0 = $11223344`

```
MOVE.B  #$FF,D0   ; D0 = $112233FF (La donnée sur  8 bits est copiée dans D0.B)
MOVE.W  #$7A8,D0  ; D0 = $1122 07A8 (La donnée sur 16 bits est copiée dans D0.W)
MOVE.L  #$7A8,D0  ; D0 = $0000 07A8 (La donnée sur 32 bits est copiée dans D0.L)
```

Remarques :

- Le symbole '**$**' permet de préciser que le nombre qui suit est en représentation hexadécimale. Les trois instructions ci-dessous sont donc équivalentes aux trois instructions précédentes :

```
MOVE.B  #255,D0   ; D0 = $112233FF (La donnée sur  8 bits est copiée dans D0.B)
MOVE.W  #1960,D0  ; D0 = $1122 07A8 (La donnée sur 16 bits est copiée dans D0.W)
MOVE.L  #1960,D0  ; D0 = $0000 07A8 (La donnée sur 32 bits est copiée dans D0.L)
```

  Avec $255_{10} = FF_{16}$ et $1960_{10} = 7A8_{16}$

- Une donnée immédiate ne peut jamais se retrouver en opérande destination. Il n'est pas possible d'écrire dans une donnée immédiate.

3. Les modes d'adressage qui spécifient un emplacement mémoire

L'opérande spécifié par ces modes d'adressage se trouve en mémoire. Il est donc nécessaire de déterminer une adresse effective qui sera l'adresse mémoire à laquelle se trouve l'opérande.

Nous utiliserons les valeurs initiales ci-dessous pour les modes d'adressage qui nécessitent des exemples. **La mémoire et les registres seront réinitialisés pour chaque mode d'adressage**.

Valeurs initiales :

```
D0 = $11223344  A0 = $00001000  PC = $00002000
D1 = $AABBCCDD  A1 = $00001008
D2 = $0000FFFF  A2 = $00001010
D3 = $00000003  A3 = $00000006

$001000 21 45 87 AF B5 F3 3C 32
$001008 AD 45 39 98 9A 9B 9C 9D
$001010 03 69 01 00 12 0A 0D C9
```

3.1. Indirect par registre d'adresse : (An)

L'adresse effective est celle du registre d'adresse : **<ea> = An**

Exemples :

```
MOVE.B  (A0),D0   ; ($1000) → D0.B ; D0 = $11223321
MOVE.W  (A0),D0   ; ($1000) → D0.W ; D0 = $11222145
MOVE.L  (A0),D0   ; ($1000) → D0.L ; D0 = $214587AF

MOVE.B  D1,(A0)   ; D1.B → ($1000) ; $1000 DD 45 87 AF B5 F3 3C 32
MOVE.W  D1,(A0)   ; D1.W → ($1000) ; $1000 CC DD 87 AF B5 F3 3C 32
MOVE.L  D1,(A0)   ; D1.L → ($1000) ; $1000 AA BB CC DD B5 F3 3C 32

MOVE.B  (A1),(A2) ; ($1008) → ($1010) ; $1010 AD 69 01 00 12 0A 0D C9
MOVE.W  (A1),(A2) ; ($1008) → ($1010) ; $1010 AD 45 01 00 12 0A 0D C9
MOVE.L  (A1),(A2) ; ($1008) → ($1010) ; $1010 AD 45 39 98 12 0A 0D C9
```

3.2. Indirect par registre d'adresse avec postincrémentation : (An)+

L'adresse effective est celle du registre d'adresse : **<ea> = An**

Une fois que le 68000 a accédé à l'opérande, le registre d'adresse est incrémenté. La valeur de l'incrémentation dépend de la taille de l'instruction :

- **An = An + 1** si l'extension de l'instruction est **.B** ;
- **An = An + 2** si l'extension de l'instruction est **.W** ;
- **An = An + 4** si l'extension de l'instruction est **.L**.

Exemples :

```
MOVE.W  (A0)+,D0 ; ($1000) → D0.W ; D0 = $11222145 ; A0 = $1002
MOVE.L  (A0)+,D0 ; ($1002) → D0.L ; D0 = $87AFB5F3 ; A0 = $1006

MOVE.B  D1,(A1)+ ; D1.B → ($1008) ; $1008 DD 45 39 98 9A 9B 9C 9D ; A1 = $1009
MOVE.B  D1,(A1)+ ; D1.B → ($1009) ; $1008 DD DD 39 98 9A 9B 9C 9D ; A1 = $100A
MOVE.W  D1,(A1)+ ; D1.W → ($100A) ; $1008 DD DD CC DD 9A 9B 9C 9D ; A1 = $100C
MOVE.L  D1,(A1)+ ; D1.L → ($100C) ; $1008 DD DD CC DD AA BB CC DD ; A1 = $1010
```

3.3. Indirect par registre d'adresse avec prédécrémentation : –(An)

Le registre d'adresse est décrémenté avant même que le 68000 n'accède à l'opérande. La valeur de la décrémentation dépend de la taille de l'instruction :

- **An = An – 1** si l'extension de l'instruction est **.B** ;
- **An = An – 2** si l'extension de l'instruction est **.W** ;
- **An = An – 4** si l'extension de l'instruction est **.L**.

L'adresse effective est celle du registre d'adresse. **Attention ! il s'agit de la nouvelle valeur du registre qui vient d'être décrémenté : <ea> = An**

Exemples :

```
MOVE.B  -(A1),D0 ; A1 = $1007 ; ($1007) → D0.B ; D0 = $11223332
MOVE.B  -(A1),D0 ; A1 = $1006 ; ($1006) → D0.B ; D0 = $1122333C
MOVE.W  -(A1),D0 ; A1 = $1004 ; ($1004) → D0.W ; D0 = $1122B5F3
MOVE.L  -(A1),D0 ; A1 = $1000 ; ($1000) → D0.L ; D0 = $214587AF

MOVE.B  D1,-(A2) ; A2 = $100F ; D1.B → ($100F) ; $1008 AD 45 39 98 9A 9B 9C DD
MOVE.B  D1,-(A2) ; A2 = $100E ; D1.B → ($100E) ; $1008 AD 45 39 98 9A 9B DD DD
MOVE.W  D1,-(A2) ; A2 = $100C ; D1.W → ($100C) ; $1008 AD 45 39 98 CC DD DD DD
MOVE.L  D1,-(A2) ; A2 = $1008 ; D1.L → ($1008) ; $1008 AA BB CC DD CC DD DD DD
```

3.4. Indirect par registre d'adresse avec déplacement : d16(An)

L'adresse effective est la somme du registre d'adresse et d'un déplacement : **<ea> = An + d16**

d16 est un déplacement codé sur 16 bits signés : **–32768 ≤ d16 ≤ +32767.**

Remarque :
Il existe deux syntaxes équivalentes pour ce mode d'adressage : **d16(An)** et **(d16,An).**
Nous utiliserons la première : **d16(An).**

Exemples :

```
MOVE.B  -5(A1),D0 ; ($1003) → D0.B ; D0 = $112233AF
MOVE.W  4(A1),D0  ; ($100C) → D0.W ; D0 = $11229A9B

MOVE.B  D1,-1(A1) ; D1.B → ($1007) ; $1000 21 45 87 AF B5 F3 3C DD
MOVE.L  D1,-4(A1) ; D1.L → ($1004) ; $1000 21 45 87 AF AA BB CC DD
```

3.5. Indirect par registre d'adresse avec déplacement et index : d8(An,Xn)

L'adresse effective est la somme du registre d'adresse, d'un déplacement et d'un index :
<ea> = An + d8 + Xn

- **d8** est un déplacement codé sur 8 bits signés : **–128 ≤ d8 ≤ +127** ;
- **Xn** est un index codé sur 16 ou 32 bits signés. Cet index peut être un registre de donnée ou un registre d'adresse : **Dn.W, Dn.L, An.W ou An.L**.

Remarque :
Il existe deux syntaxes équivalentes pour ce mode d'adressage : **d8(An,Xn)** et **(d8,An,Xn).**
Nous utiliserons la première : **d8(An,Xn).**

Exemples :

```
MOVE.B  2(A1,A3.W),D0  ; ($1010) → D0.B ; D0 = $11223303
MOVE.W  1(A1,D3.L),D0  ; ($100C) → D0.W ; D0 = $11229A9B

MOVE.B  D1,0(A1,D2.W)  ; D1.B → ($1007) ; $1000 21 45 87 AF B5 F3 3C DD
MOVE.L  D1,-3(A1,D2.W) ; D1.L → ($1004) ; $1000 21 45 87 AF AA BB CC DD
```

3.6. Indirect relatif au compteur programme avec déplacement : d16(PC)

L'adresse effective est la somme du compteur programme et d'un déplacement : **<ea> = PC + d16**

d16 est un déplacement codé sur 16 bits signés : **–32768 ≤ d16 ≤ +32767.**

Remarques :
Il existe deux syntaxes équivalentes pour ce mode d'adressage : **d16(PC)** et **(d16,PC).**
Nous utiliserons la première : **d16(PC).**

Pour le programmeur, la valeur du **PC** est difficile, voire impossible à connaître lors de la phase de développement. Or, s'il ne connaît pas la valeur du **PC**, il ne peut pas non plus connaître l'adresse effective. Ce mode d'adressage serait donc inutilisable si l'on devait utiliser la syntaxe **d16(PC)** telle quelle. C'est pourquoi, dans un code source, on précise directement l'adresse effective sans se soucier ni de la valeur du **PC**, ni de la valeur du déplacement.

En pratique, la syntaxe utilisée est alors la suivante : <ea>(PC)

C'est l'assembleur, au moment de la phase d'assemblage, qui s'occupe de calculer le déplacement en fonction de la valeur du **PC**. Ceci simplifie énormément les choses puisque l'adresse effective apparaît directement dans le code source. Par conséquent, le programmeur n'a pas à réaliser lui-même l'addition (**PC + d16**).

Exemples :

```
MOVE.B  $1010(PC),D0  ; ($1010) → D0.B ; D0 = $11223303
MOVE.W  $100C(PC),D0  ; ($100C) → D0.W ; D0 = $11229A9B
MOVE.L  $1002(PC),D0  ; ($1002) → D0.L ; D0 = $87AFB5F3

; L'instruction MOVE n'accepte pas ce mode d'adressage en destination.
```

3.7. Indirect relatif au compteur programme avec déplacement et index : d8(PC,Xn)

L'adresse effective est la somme du compteur programme, d'un déplacement et d'un index :
<ea> = PC + d8 + Xn

- **d8** est un déplacement codé sur 8 bits signés : **–128 ≤ d8 ≤ +127** ;
- **Xn** est un index codé sur 16 ou 32 bits signés. Cet index peut être un registre de donnée ou un registre d'adresse : **Dn.W, Dn.L, An.W ou An.L**.

Nous n'utiliserons jamais ce mode d'adressage dans la suite de ce cours.

3.8. Absolu long : (xxx).L

L'adresse effective est directement spécifiée dans le code source. L'adresse est codée sur 32 bits.

Exemples :

```
MOVE.B  $1010,D0  ; ($1010) → D0.B ; D0 = $11223303
MOVE.W  $100C,D0  ; ($100C) → D0.W ; D0 = $11229A9B

MOVE.B  D1,$1007  ; D1.B → ($1007) ; $1000 21 45 87 AF B5 F3 3C DD
MOVE.L  D1,$1004  ; D1.L → ($1004) ; $1000 21 45 87 AF AA BB CC DD
```

3.9. Absolu court : (xxx).W

Ce mode d'adressage est équivalent au mode d'adressage absolu long, mais il ne fonctionne qu'avec des adresses que l'on peut coder sur 16 bits. L'exécution d'une instruction est alors plus rapide.

Nous n'utiliserons jamais ce mode d'adressage dans la suite de ce cours.

4. Exemples

Afin d'illustrer encore plus précisément le fonctionnement des modes d'adressage, vous trouverez ci-dessous une série d'instructions où le calcul des adresses effectives sera détaillé. Les contenus des registres (sauf le **PC**) et de la mémoire qui viennent d'être modifiés seront également précisés.

Pour chaque instruction, la mémoire et les registres seront réinitialisés aux valeurs ci-dessous :

Valeurs initiales :
```
D0 = $0000FFFF  A0 = $00001000  PC = $00002000
D1 = $00000004  A1 = $00001008
D2 = $FFFFF000  A2 = $00001010

$001000  21 45 87 AF B5 F3 3C 32
$001008  AD 45 39 98 9A 9B 9C 9D
$001010  03 69 01 00 12 0A 0D C9
```

4.1. MOVE.W A1,D2

Source	Destination
A1.W **#$1008**	**D2.W**

D2 = $FFFF**1008**

4.2. MOVE.W (A1),D2

Source	Destination
(A1) **#$AD45**	**D2.W**

D2 = $FFFF**AD45**

4.3. MOVE.L #$100A,D2

Source	Destination
#$100A **#$0000100A**	**D2.L**

D2 = $0000100A

4.4. MOVE.L $100A,D2

Source	Destination
($100A) **#$39989A9B**	**D2.L**

D2 = $39989A9B

4.5. `MOVE.W #36,(A0)`

Source	Destination
#36 **#$0024**	(A0) **($1000)**

```
$001000  00 24 87 AF B5 F3 3C 32
```

4.6. `MOVE.B D1,(A1)+`

Source	Destination
D1.B **#$04**	(A1) **($1008)**

```
$001008  04 45 39 98 9A 9B 9C 9D      A1 = $00001009
```

4.7. `MOVE.L $1004,-(A2)`

Source	Destination
($1004) **#$B5F33C32**	(A2) ($1010 – 4) **($100C)**

```
$001008  AD 45 39 98 B5 F3 3C 32      A2 = $0000100C
```

4.8. `MOVE.L -(A2),-(A2)`

Source	Destination
(A2) ($1010 – 4) ($100C) **#$9A9B9C9D**	(A2) ($100C – 4) **($1008)**

```
$001008  9A 9B 9C 9D 9A 9B 9C 9D      A2 = $00001008
```

4.9. `MOVE.B 5(A1),-1(A1,D0.W)`

Source	Destination
5(A1)	-1(A1,D0.W)
(A1 + 5)	(A1 + D0.W - 1)
($1008 + 5)	($1008 – 1 - 1)
($100D)	**($1006)**
#$9B	

```
$001000  21 45 87 AF B5 F3 9B 32
```

4.10. `MOVE.W 2(A1,D1.L),-6(A2)`

Source	Destination
2(A1,D1.L)	-6(A2)
(A1 + D1 + 2)	(A2 - 6)
($1008 + 4 + 2)	($1010 – 6)
($100E)	**($100A)**
#$9C9D	

```
$001008  AD 45 9C 9D 9A 9B 9C 9D
```

4.11. `MOVE.W $1000(PC),$100A`

Source	Destination
($1000)	**($100A)**
#$2145	

```
$001008  AD 45 21 45 9A 9B 9C 9D
```

V. Les branchements

1. Les branchements inconditionnels

Le 68000 possède deux instructions de branchement inconditionnel :

BRA	Branchement inconditionnel
JMP	Saut inconditionnel

Ces deux instructions sont similaires. Elles possèdent tout de même quelques différences au niveau des modes d'adressage et du codage machine. Nous n'aborderons pas ces différences. Dans le cadre de ce cours d'initiation, nous considérerons que si l'opérande est une adresse (ou une étiquette), alors ces deux instructions sont équivalentes et interchangeables.

Exemple :

```
        ORG     $1000

        BRA     NEXT    ; Branchement inconditionnel à l'étiquette NEXT.
        CLR.L   D1      ; Cette instruction ne sera pas exécutée.

NEXT    MOVE.L  #5,D0
        RTS
```

On peut remplacer le `BRA` par un `JMP` :

```
        ORG     $1000

        JMP     NEXT    ; Saut inconditionnel à l'étiquette NEXT.
        CLR.L   D1      ; Cette instruction ne sera pas exécutée.

NEXT    MOVE.L  #5,D0
        RTS
```

2. Les branchements conditionnels

Comme son nom l'indique, un branchement conditionnel comporte une condition. Ceci nous amène aux deux cas suivants :

- Soit la condition est vérifiée, auquel cas le branchement est effectué ;
- Soit la condition n'est pas vérifiée, auquel cas le branchement n'est pas effectué.

La condition d'un branchement conditionnel se fait sur un *flag* ou une combinaison de *flags*.

Les instructions de branchement conditionnel se trouvent dans le manuel à la mnémonique `Bcc` (*Branch condition code*). Les deux *c* représentent la condition et doivent être remplacés par deux lettres en fonction de la condition souhaitée (par exemple : `BNE`, `BEQ`, `BGE`, etc.).

2.1. Les branchements à comparaison sur un *flag*

Mnémonique	Condition	Branchement si
BPL	*Plus*	N = 0
BMI	*Minus*	N = 1
BNE	*Not Equal*	Z = 0
BEQ	*Equal*	Z = 1
BVC	*Overflow Clear*	V = 0
BVS	*Overflow Set*	V = 1
BCC	*Carry Clear*	C = 0
BCS	*Carry Set*	C = 1

L'instruction `TST` qui modifie les *flags* **N** et **Z** est très souvent utilisée avec les instructions `BPL`, `BMI`, `BNE` et `BEQ`.

Exemples :

```
        TST.L   D1
        BEQ     NEXT1   ; Saut si D1.L = 0 (si Z = 1)

        TST.W   D2
        BNE     NEXT2   ; Saut si D2.W ≠ 0 (si Z = 0)

        TST.B   D3
        BMI     NEXT3   ; Saut si D3.B < 0 (si N = 1)

        TST.L   D4
        BPL     NEXT4   ; Saut si D4.L ≥ 0 (si N = 0)

        TST.B   D5
        BMI     NEXT5   ; Saut si D5.B < 0 (si N = 1)
        BEQ     NEXT6   ; Saut si D5.B = 0 (si Z = 1)
```

2.2. Les branchements à comparaison non signée et à comparaison signée

Comparaison non signée		Comparaison signée		Branchement
Mnémonique	**Condition**	**Mnémonique**	**Condition**	
BHI	*Higher*	BGT	*Greater Than*	si supérieur
BHS	*Higher or Same*	BGE	*Greater or Equal*	si supérieur ou égal
BLO	*Lower*	BLT	*Less Than*	si inférieur
BLS	*Lower or Same*	BLE	*Less or Equal*	si inférieur ou égal

Remarques :

- L'instruction `BHS` est en fait un alias pour l'instruction `BCC` (ces deux instructions sont identiques).
- L'instruction `BLO` est en fait un alias pour l'instruction `BCS` (ces deux instructions sont identiques).

L'instruction `CMP` s'utilise principalement avec des branchements conditionnels selon le modèle suivant :

```
        CMP   source,destination
        Bcc   <étiquette>          ; Branchement si destination <condition> source
```

Exemples :

```
        CMP.L  D0,D1
        BHI    NEXT      ; Branchement si D1.L > D0.L (comparaison non signée)
```

```
        CMP.W  D0,D1
        BGT    NEXT      ; Branchement si D1.W > D0.W (comparaison signée)
```

```
        CMP.B  D0,D1
        BLS    NEXT      ; Branchement si D1.B ≤ D0.B (comparaison non signée)
```

```
        CMP.L  D0,D1
        BLE    NEXT      ; Branchement si D1.L ≤ D0.L (comparaison signée)
```

3. Exemples de boucle

La structure de boucle suivante est très utilisée en assembleur 68000 :

```
        MOVE.W  #n,D7        ; n → D7
                             ; (D7.W = compteur de boucle)

LOOP    ; ...                ; Corps de la boucle
        ; ...                ; exécuté n fois

        SUBQ.W  #1,D7        ; D7.W - 1 → D7.W (si D7.W est nul, Z passe à 1)
        BNE     LOOP         ; Branchement si D7.W ≠ 0 (si Z = 0)
```

Code assembleur équivalent à la structure d'une boucle *for* en langage C :

Langage C :

```c
#define MAX 17

int i;

for (i = 0; i < MAX; i++)
{
    /* Corps de la boucle */
}
```

Langage assembleur :

```
MAX     EQU     17          ; Définit la valeur de l'étiquette MAX.

        CLR.L   D0          ; 0 → D0
        MOVE.L  #MAX,D7     ; MAX → D7

LOOP    CMP.L   D7,D0
        BGE     DONE        ; Branchement à DONE si D0 ≥ D7

        ; ...               ; Corps de la boucle
        ; ...               ; (répété tant que D0 < D7)

        ADDQ.L  #1,D0       ; D0 + 1 → D0
        BRA     LOOP        ; Branchement à LOOP
DONE
```

4. L'instruction DBRA

L'instruction `DBRA` est une instruction optimisée pour les boucles. Elle réalise la décrémentation d'un registre **sur 16 bits** et effectue un branchement tant que la nouvelle valeur du registre est différente de –1.

Exemple :

```
        MOVE.W  #n,D7       ; n → D7
                            ; (D7.W = compteur de boucle)

LOOP    ; ...               ; Corps de la boucle
        ; ...               ; exécuté n + 1 fois

        DBRA    D7,LOOP     ; D7.W - 1 → D7.W ; Branchement à LOOP si D7.W ≠ -1
```

Remarque :
L'instruction `DBRA` est en fait un alias pour l'instruction `DBF` qui fait partie de la famille des instructions `DBcc`.

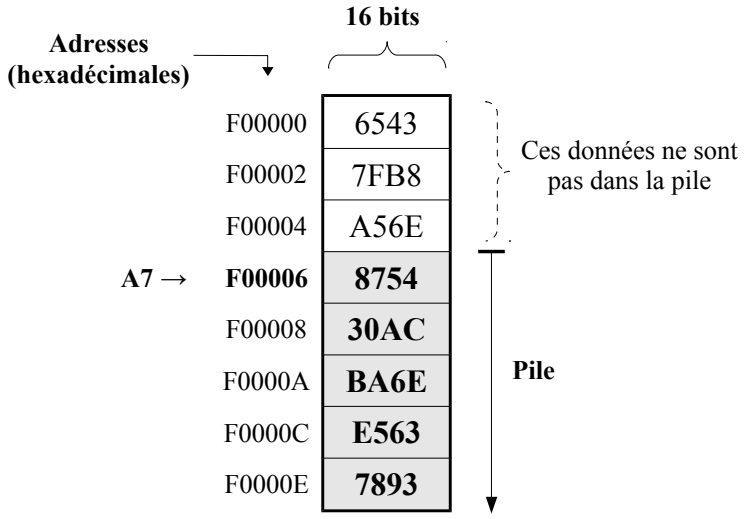
VI. La pile

1. Définitions et principes de base

La pile est une zone spéciale de la mémoire réservée à un stockage temporaire de données. Elle est de type LIFO (*Last In First Out*). Le dernier élément empilé sera le premier élément dépilé (exemples : pile de livres, pile d'assiettes, etc.).

Le registre **A7** est le pointeur de pile : il pointe le **sommet** de la pile.

Exemple pour **A7** = $F00006 :



Seuls des mots (16 bits) et des mots longs (32 bits) peuvent être empilés ou dépilés (jamais d'octet).

L'espace mémoire réservé à la pile doit être suffisant pour satisfaire les besoins du programme en cours d'exécution. Le programmeur ou le système d'exploitation devra donc initialiser le pointeur de pile en conséquence.

Pour la suite de ce cours, nous n'attacherons aucune importance à la valeur de l'adresse que contient ce pointeur. L'adresse en soi n'est pas significative et n'apporte rien à la bonne compréhension du fonctionnement de la pile. Ce qui compte, ce sont les variations du pointeur de pile (incrémentation, décrémentation, etc.). Nous considérerons donc que ce dernier pointe sur un emplacement mémoire suffisant et seules ses variations seront précisées.

Généralement, deux étapes sont nécessaires pour empiler ou pour dépiler une donnée :

Étape	Empiler une donnée	Dépiler une donnée
1	Décrémenter **A7**	Lire la mémoire pointée par **A7**
2	Écrire dans la mémoire pointée par **A7**	Incrémenter **A7**

Soit le programme ci-dessous où l'on numérote de un à huit les instructions qui modifient la pile :

```
     MOVE.L  #$11112222,D0  ; #$11112222 → D0.L
     MOVE.L  #$AAAABBBB,D1  ; #$AAAABBBB → D1.L

(1)  MOVE.W  #$42,-(A7)     ; Empile la donnée #$0042
(2)  MOVE.L  #$53,-(A7)     ; Empile la donnée #$00000053
(3)  MOVE.W  D0,-(A7)       ; Empile la donnée #$2222 (contenue dans D0.W)
(4)  MOVE.L  D1,-(A7)       ; Empile la donnée #$AAAABBBB (contenue dans D1.L)

     CLR.W   D0             ; 0 → D0.W ; Le registre D0.W est modifié.
     CLR.L   D1             ; 0 → D1.L ; Le registre D1.L est modifié.

(5)  MOVE.L  (A7)+,D1       ; Dépile la donnée #$AAAABBBB dans D1.L
(6)  MOVE.W  (A7)+,D0       ; Dépile la donnée #$2222 dans D0.W
(7)  MOVE.L  (A7)+,D2       ; Dépile la donnée #$00000053 dans D2.L
(8)  MOVE.W  (A7)+,D3       ; Dépile la donnée #$0042 dans D3.W
```

Les différents états de la pile seront les suivants :

État de la pile avant l'instruction 1

	2FA8
	3642
	EB89
	87AF
	2118
	11BC
	C948
	72C8
A7 →	**AF3D**
	CF4C
	7893

État de la pile après l'instruction 1

	2FA8
	3642
	EB89
	87AF
	2118
	11BC
	C948
A7 →	**0042**
	AF3D
	CF4C
	7893

État de la pile après l'instruction 2

	2FA8
	3642
	EB89
	87AF
	2118
A7→	**0000**
	0053
	0042
	AF3D
	CF4C
	7893

État de la pile après l'instruction 3

	2FA8
	3642
	EB89
	87AF
A7 →	**2222**
	0000
	0053
	0042
	AF3D
	CF4C
	7893

État de la pile après l'instruction 4

	2FA8
	3642
A7 →	**AAAA**
	BBBB
	2222
	0000
	0053
	0042
	AF3D
	CF4C
	7893

État de la pile après l'instruction 5

	2FA8
	3642
	AAAA
	BBBB
A7 →	**2222**
	0000
	0053
	0042
	AF3D
	CF4C
	7893

État de la pile après l'instruction 6

	2FA8
	3642
	AAAA
	BBBB
	2222
A7 →	**0000**
	0053
	0042
	AF3D
	CF4C
	7893

État de la pile après l'instruction 7

	2FA8
	3642
	AAAA
	BBBB
	2222
	0000
	0053
A7→	**0042**
	AF3D
	CF4C
	7893

État de la pile après l'instruction 8

	2FA8
	3642
	AAAA
	BBBB
	2222
	0000
	0053
	0042
A7 →	**AF3D**
	CF4C
	7893

2. L'instruction MOVEM

L'instruction `MOVEM` (*Move Multiple Registers*) est une instruction qui permet de lire ou d'écrire plusieurs registres en mémoire. Elle est très souvent utilisée pour empiler ou dépiler la valeur de plusieurs registres en une seule instruction.

Ses syntaxes sont les suivantes :
- `MOVEM <list>,<ea>`
- `MOVEM <ea>,<list>`

L'opérande `<list>` représente une liste de registres. **Seule l'instruction `MOVEM` est capable de manipuler une liste de registres**. Les adresses effectives `<ea>`, quant à elles, spécifient nécessairement un emplacement mémoire (mode d'adressage indirect ou absolu).

Une liste de registres peut contenir n'importe quels registres de donnée ou d'adresse. Les registres sont séparés par le caractère « `/` ». Par exemple : `D0/D1/A6/A4/D5`.

Quand plusieurs registres de donnée ou d'adresse se suivent, il est possible d'utiliser le caractère « - » afin d'indiquer un groupe de registres. Par exemple, ces deux listes sont équivalentes :
- `D0/D1/D2/D3/A2/A3/A4/A5/A6`
- `D0-D3/A2-A6`

L'ordre dans lequel apparaissent les registres dans la liste n'a aucune importance. Par exemple, ces trois listes sont équivalentes :
- `D6/A4/A2/A5/D1/A6/D2/D3`
- `D1/D2/D3/D6/A2/A4/A5/A6`
- `D1-D3/D6/A2/A4-A6`

L'instruction `MOVEM` traite les registres dans un ordre prédéfini qui lui est propre. Plus précisément, elle possède deux ordres bien distincts :
- l'ordre classique : **D0**, **D1**, **D2**, **D3**, **D4**, **D5**, **D6**, **D7**, **A0**, **A1**, **A2**, **A3**, **A4**, **A5**, **A6**, **A7**
- l'ordre inversé : **A7**, **A6**, **A5**, **A4**, **A3**, **A2**, **A1**, **A0**, **D7**, **D6**, **D5**, **D4**, **D3**, **D2**, **D1**, **D0**

Le 68000 utilise l'ordre inversé uniquement lorsque le mode d'adressage de l'adresse effective de destination est un mode d'adressage indirect avec prédécrémentation –(An). Dans tous les autres cas, c'est l'ordre classique qui est utilisé.

Par exemple :
- L'instruction `MOVEM.L D0/A4,(A0)` copiera d'abord le registre **D0** puis le registre **A4**.
- L'instruction `MOVEM.L D0/A4,-(A0)` copiera d'abord le registre **A4** puis le registre **D0**.

L'instruction `MOVEM` très souvent utilisée pour empiler ou dépiler la valeur de plusieurs registres en une seule instruction.

Par exemple, l'instruction suivante :

```
    MOVEM.L D1-D3/A4/A5,-(A7)  ; Empile A5, puis A4, puis D3, puis D2, puis D1.
```

est équivalente aux cinq instructions ci-dessous :

```
    MOVE.L  A5,-(A7)           ; Empile A5
    MOVE.L  A4,-(A7)           ; Empile A4
    MOVE.L  D3,-(A7)           ; Empile D3
    MOVE.L  D2,-(A7)           ; Empile D2
    MOVE.L  D1,-(A7)           ; Empile D1
```

Et l'instruction suivante :

```
    MOVEM.L (A7)+,D1-D3/A4/A5  ; Dépile D1, puis D2, puis D3, puis A4, puis A5.
```

est équivalente aux cinq instructions ci-dessous :

```
    MOVE.L  (A7)+,D1           ; Dépile D1
    MOVE.L  (A7)+,D2           ; Dépile D2
    MOVE.L  (A7)+,D3           ; Dépile D3
    MOVE.L  (A7)+,A4           ; Dépile A4
    MOVE.L  (A7)+,A5           ; Dépile A5
```

VII. Les sous-programmes

Un sous-programme est une séquence d'instructions relativement indépendante qui réalise une tâche précise. Il s'agit de l'équivalent d'une fonction dans des langages plus évolués ; par exemple le langage C. Un sous-programme peut être appelé à partir d'un programme principal, d'un autre sous-programme, ou encore de lui-même. On dira dans ce dernier cas qu'il est récursif.

Le 68000 possède trois instructions relatives aux sous-programmes :

BSR	Branchement à un sous-programme (*Branch to Subroutine*)
JSR	Saut à un sous-programme (*Jump to Subroutine*)
RTS	Retour de sous-programme (*Return from Subroutine*)

Les différences entre les instructions `BSR` et `JSR` sont les mêmes que pour les instructions de branchement inconditionnel `BRA` et `JMP`. Ces instructions sont similaires, mais possèdent quelques différences au niveau des modes d'adressage et du codage machine. Nous n'aborderons pas ces différences. Dans le cadre de ce cours d'initiation, nous considérerons que si l'opérande est une adresse (ou une étiquette), alors ces deux instructions sont équivalentes et interchangeables.

Prenons l'exemple d'un sous-programme que l'on appelle **`GetMin`** et qui renvoie dans le registre **D0** la plus petite valeur signée contenue dans les registres **D1** et **D2**. Il sera alors possible d'appeler ce sous-programme autant de fois que nécessaire comme le montre l'exemple ci-dessous :

```
Main    …
        …               ; Instructions quelconques
        …
        moveq.l #15,d1  ; 15 → D1
        moveq.l #25,d2  ; 25 → D2
        jsr     GetMin  ; Appel à GetMin (15 → D0)
Next1   …
        …               ; Instructions quelconques
        …
        moveq.l #30,d1  ; 30 → D1
        moveq.l #16,d2  ; 16 → D2
        jsr     GetMin  ; Appel à GetMin (16 → D0)
Next2   …
        …               ; Instructions quelconques
        …
GetMin  cmp.l   d1,d2   ; Compare D1 et D2.
        ble     d2min   ; Saut à d2min si D2 ≤ D1 (comparaison signée).

d1min   move.l  d1,d0   ; D1 est inférieur à D2, on copie D1 dans D0.
        rts             ; Sortie du sous-programme.

d2min   move.l  d2,d0   ; D2 est inférieur ou égal à D1, on copie D2 dans D0.
        rts             ; Sortie du sous-programme.
```

Ce programme charge une valeur dans chacun des registres **D1** et **D2** et fait appel à **`GetMin`**. L'appel se fait par l'instruction `JSR GetMin` (elle aurait pu également se faire par l'instruction `BSR GetMin`).

Les instructions du sous-programme sont ensuite exécutées jusqu'à rencontrer une instruction `RTS`. Cette dernière quitte le sous-programme et continue l'exécution du programme juste après l'instruction d'appel au sous-programme (`JSR` ou `BSR`) :

- Dans le cas du premier appel, le `RTS` quitte le sous-programme en sautant à l'adresse `Next1` ;
- Dans le cas du deuxième appel, le `RTS` quitte le sous-programme en sautant à l'adresse `Next2`.

La question qui se pose est alors la suivante : comment l'instruction `RTS` connaît-elle l'adresse de retour d'un sous-programme ? La réponse est simple : cette adresse est sauvegardée dans la pile au moment de l'appel au sous-programme.

Une instruction `JSR` ou `BSR` réalise les deux opérations suivantes :

- Sauvegarde dans la pile de l'adresse de retour. L'adresse de retour est l'adresse de l'instruction qui suit le `JSR` ou le `BSR`. Cette adresse est toujours codée sur une taille de 32 bits non signés ;
- Branchement au sous-programme.

Pour quitter un sous-programme, une instruction `RTS` dépile l'adresse de retour et la charge dans le registre **PC**. L'instruction `RTS` n'est donc qu'une simple instruction de saut dont l'adresse de branchement se trouve au sommet de la pile.

VIII. Les principales instructions du 68000

Cette partie présente une vue d'ensemble des principales instructions du 68000. Les instructions y sont classées par fonction et accompagnées d'une brève description. Il sera toutefois nécessaire de se reporter au manuel de référence du programmeur (*M68000 Family Programmer's Reference Manual*) afin d'obtenir une description complète et détaillée de chaque instruction.

La plupart des instructions possèdent plusieurs déclinaisons. Par exemple, il y a quatre suffixes que l'on retrouve sur certaines d'entre elles : **A** (*Address*), **I** (*Immediate*), **Q** (*Quick*) et **X** (*Extended*).

- Les instructions se terminant par le suffixe **A** (*Address*) sont généralement réservées à la manipulation des registres d'adresse ;
- Les instructions se terminant par le suffixe **I** (*Immediate*) utilisent un adressage immédiat pour l'opérande source ;
- Les instructions se terminant par le suffixe **Q** (*Quick*) utilisent un adressage immédiat pour l'opérande source avec une limitation de valeur. Cette limitation permet d'accélérer l'exécution de l'instruction ;
- Les instructions se terminant par le suffixe **X** (*Extended*) utilisent le *flag* **X** en tant qu'opérande source supplémentaire.

1. Les mouvements de donnée

Ce type d'instruction permet de copier le contenu d'un opérande source vers un opérande destination :
source → destination

MOVE	Transfert de donnée
MOVEA	Transfert d'adresse
MOVEQ	Transfert rapide
MOVEM	Transfert multiple de registres
LEA	Chargement d'une adresse effective

2. Les opérations arithmétiques

2.1. L'addition

Ce type d'instruction permet d'additionner un opérande source et un opérande destination :
source + destination → destination

ADD	Addition binaire
ADDA	Addition à une adresse
ADDI	Addition immédiate
ADDQ	Addition rapide
ADDX	Addition étendue (source + destination + **X** → destination)

2.2. La soustraction

Ce type d'instruction permet de soustraire un opérande source à un opérande destination :
destination – source → destination

SUB	Soustraction binaire
SUBA	Soustraction à une adresse
SUBI	Soustraction immédiate
SUBQ	Soustraction rapide
SUBX	Soustraction étendue (destination – source – **X** → destination)

2.3. La multiplication

Ce type d'instruction permet de multiplier un opérande source avec un opérande destination :
source × destination → destination

MULS	Multiplication signée
MULU	Multiplication non signée

2.4. La division

Ce type d'instruction permet de diviser un opérande destination par un opérande source :
destination ÷ source → destination

DIVS	Division signée
DIVU	Division non signée

Remarques :
- Le quotient est placé dans les 16 bits de poids faible de l'opérande destination.
- Le reste est placé dans les 16 bits de poids fort de l'opérande destination.

2.5. Autres

CLR	Mise à zéro d'un opérande (0 → destination)
EXT	Extension de signe (destination *sign-extended* → destination)
NEG	Négation (0 – destination → destination)
NEGX	Négation étendue (0 – destination – **X** → destination)

3. Les opérations logiques

AND	ET
ANDI	ET immédiat
OR	OU
ORI	OU immédiat
EOR	OU EXCLUSIF
EORI	OU EXCLUSIF immédiat
NOT	Complémentation logique

4. Les décalages et les rotations

ASL	Décalage arithmétique vers la gauche
ASR	Décalage arithmétique vers la droite
LSL	Décalage logique vers la gauche
LSR	Décalage logique vers la droite
ROL	Rotation vers la gauche
ROR	Rotation vers la droite
ROXL	Rotation étendue vers la gauche
ROXR	Rotation étendue vers la droite
SWAP	Échange les bits de poids faible et de poids fort d'un registre de donnée

5. Les manipulations de bit

BTST	Test d'un bit (seul le *flag* **Z** est modifié)
BCLR	Test d'un bit et mise à zéro
BSET	Test d'un bit et mise à un
BCHG	Test d'un bit et changement

6. Les tests et les comparaisons

TST	Test d'un opérande
CMP	Comparaison
CMPA	Comparaison à une adresse
CMPI	Comparaison immédiate
CMPM	Comparaison mémoire

Les instructions de test et de comparaison n'affectent pas les opérandes. Seuls les *flags* du registre **CCR** sont modifiés.

L'instruction `TST` ne fait que tester l'opérande en mettant à jour les *flags* **N** et **Z**.

Par exemple, l'instruction `TST.L D0` positionnera :
- **Z** à 0 si **D0** n'est pas nul ;
- **Z** à 1 si **D0** est nul ;
- **N** à 0 si **D0** est positif ou nul ;
- **N** à 1 si **D0** est négatif.

Une instruction de comparaison soustrait un opérande source à un opérande destination (Dst - Src). Les *flags* **N**, **Z**, **V** et **C** sont modifiés en fonction du résultat. Cependant, à la différence d'une instruction de soustraction, **le résultat n'est pas stocké dans l'opérande destination** (il est perdu).

Par exemple, l'instruction `CMP.L D0,D1` effectue l'opération **D1 – D0** et modifie les *flags* en fonction du résultat. Le registre **D1** reste inchangé.