Chapitre 3
Initiation à l'assembleur 68000

Version du 13/11/2020

Table des matières

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.

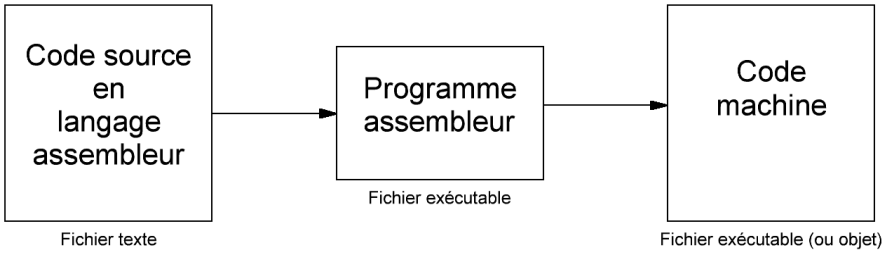
2.1. Le langage assembleur

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 :

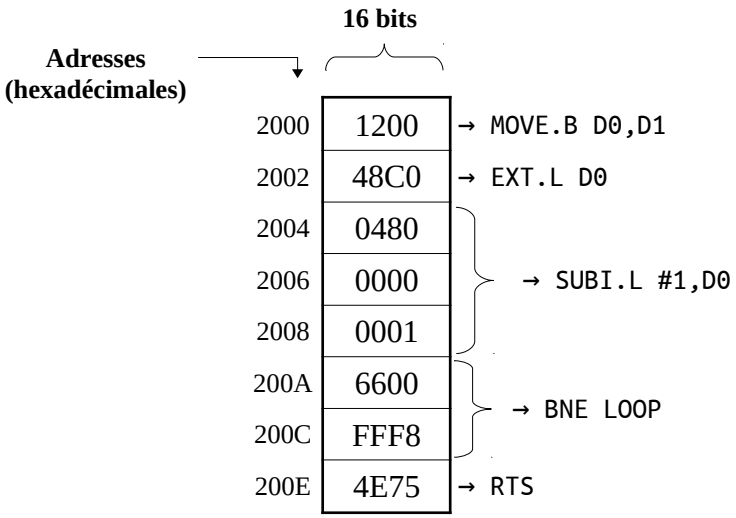
```
        ORG     $2000   ; Place le programme à l'adresse $2000.
START   MOVE.B  D0,D1   ; D0 → D1.
        EXT.L   D0      ; Extension de signe.
LOOP    SUBI.L  #1,D0   ; D0 - 1 → D0.
        BNE     LOOP    ; Saut à LOOP tant que D0 n'est pas nul.
        RTS             ; Retour de sous-programme.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

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}).

Certains 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

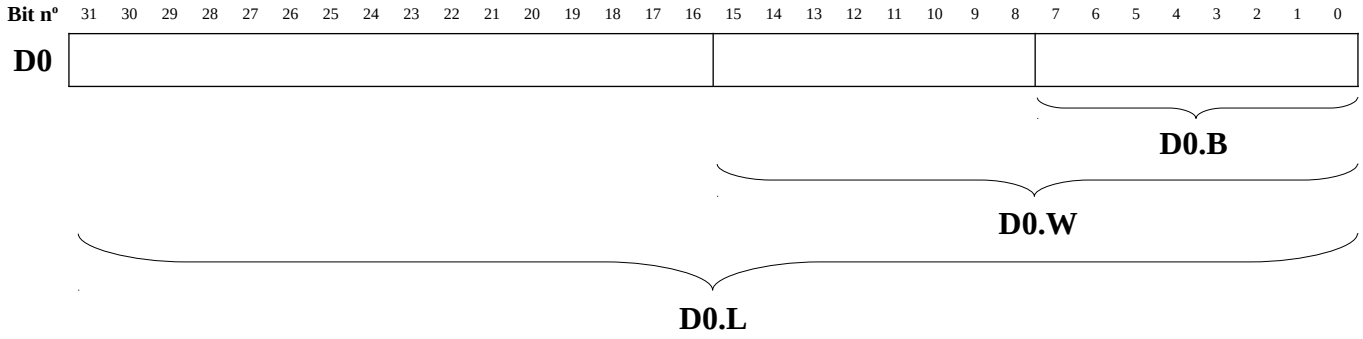
Un registre est un espace de stockage temporaire intégré dans un microprocesseur.

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** :

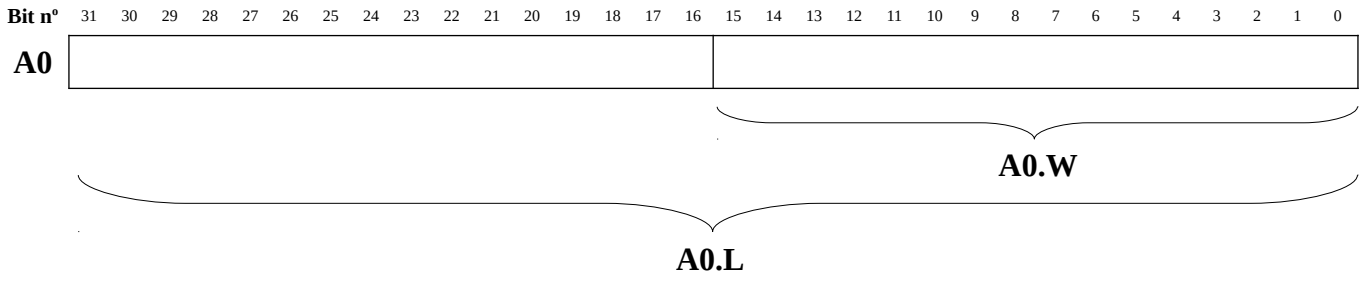


3.2. Les registres d'adresse et les pointeurs de pile

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

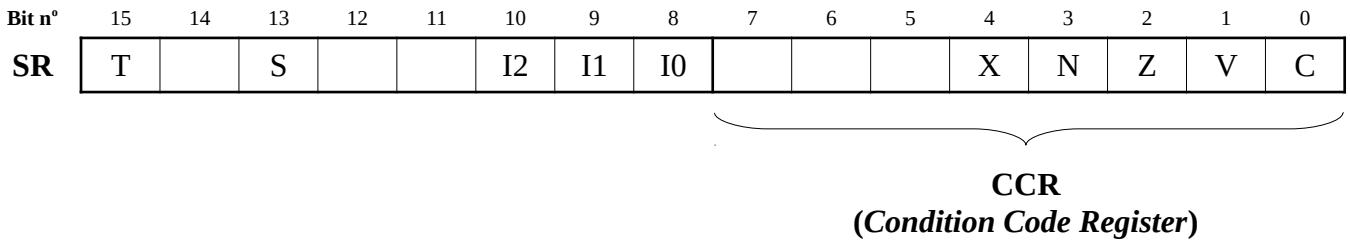
Le compteur programme **PC** (*Program Counter*) est un registre d'une taille de 32 bits qui contient l'adresse de la prochaine instruction à exécuter. Ce registre est modifié automatiquement par le 68000.

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 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 au 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)
```