Processeur 68000 (Motorola) et sa programmation : Documentations techniques

Table des matières

A - Introduction

A.1. Instructions courantes

Dans cette section A, le tableau contient seulement quelques instructions courantes pour les premiers TP en assembleur 68000.

Le **jeu d'instructions complet** du 68000 est donné en section L (dernières pages).

INDISPENSABLE :

Une instruction en assembleur a la structure suivante :

`OPER.lg  source,destination`

`source, destination` sont les deux opérandes sur lesquels l'opérateur `OPER` s'applique.

D'une manière générale, l'instruction réalisée donnée sous forme algorithmique est :

`Destination.lg := destination.lg OPER source.lg;`

`lg` indique au processeur la ***longueur*** à utiliser pour les opérandes et réaliser l'opération.

`B` (byte - octet) ou **`W`** (word - 2 octets) ou **`L`** (long - 4 octets).

Par exemple, `SUB.W #14,D4` indique l'opération : `D4.W := D4.W - 14;`

Dans le tableau suivant (issu de la documentation constructeur), l'opérande source est noté `Sour` et l'opérande destination est noté `Dest`. Le fonctionnement est explicité sous la forme (Dest) OPER (Sour) → Dest car l'emplacement (ici Dest) est distingué de son contenu (ici (Dest)). Pour ne pas prêter à confusion, on utilisera plutôt la forme algorithmique : Dest := Dest OPER Sour ;

Mnémonique	Description	Fonctionnement	X	N	Z	V	C
ADD	Addition binaire	(Dest) + (Sour) → Dest	*	*	*	*	*
ADDI	Addition immédiate	(Dest) + don. imm. → Dest	*	*	*	*	*
AND	ET logique	(Dest) ET (Sour) → Dest	—	*	*	0	0
B•• – Branchement conditionnel		si cond. alors PC+d → PC	—	—	—	—	—
BRA	Branchement inconditionnel	PC+d → PC	—	—	—	—	—
CMP	Comparaison	(Dest) - (Sour)	—	*	*	*	*
CMPI	Comparaison immédiate	(Dest) - données immédiates	—	*	*	*	*
EOR	OU exclusif logique	(Dest) ⊕ (Sour) → Dest	—	*	*	0	0
EXG	Echange de registres	Rx ↔ Ry	—	—	—	—	—
EXT	Extension de signe	(Dest) signe étendu → Dest	—	*	*	0	0
JMP	Saut inconditionnel	Dest → PC	—	—	—	—	—
MOVE	Transfert de données	(Sour) → Dest	—	*	*	0	0
MOVEA	Transfert vers An	(Sour) → An	—	—	—	—	—
NEG	Complément à 2	0 - (Dest) → Dest	*	*	*	*	*
NOP	Non opération		—	—	—	—	—
NOT	Complément logique	~(Dest) → Dest	—	*	*	0	0
OR	OU logique inclusif	(Dest) OU (Sour) → Dest	—	*	*	0	0
SUB	Soustraction	(Dest) - (Sour) → Dest	*	*	*	*	*

A l'exécution de chaque instruction, des **indicateurs** (X, N, Z, V, C) sont mis à jour par le processeur. Leur valeur est symbolisée par : « — » pour non modifié ; « * » pour mis à jour ; « u » pour indéfini ; « 0 » pour mise à 0 ; « 1 » pour mise à 1.

Dn et An sont des **registres** du processeur. PC (Program Counter) est le **compteur ordinal** du processeur (cf. section F). Les instructions usuelles sont détaillées dans la section C et les instructions de branchement sont détaillées dans la section E.

A.2. Exemples d'instructions

```
  ADD.W   #103,D2     ; signifie D2.W := D2.W + <103>dix ;
[ajout de la valeur immédiate <103>dix au contenu du registre de données D2 ;
résultat dans D2 ; opérandes et opération sur un mot (16 bits)]

  ADD.W   #$67,D2     ; signifie D2.W := D2.W + <$67>deux ;
  [ajout de la valeur immédiate <$67>deux ]

  ADD.W  #%01000011,D2  ; signifie D2.W := D2.W + <%01000011>deux ;
  [idem avec la valeur immédiate donnée par <01000011>deux ]

  EXG.L   D7,D3  ; échange des contenus des registres de données D7
                       et D3 (sur 32 bits)

  SUB.B   $0A00,D1     ; signifie D1.B := D1.B – MC[$0A00].B ;
  [soustraction de l'octet en mémoire centrale à l'adresse $000A00 (source) à
l'octet de poids faible du registre D1 (destination) ; résultat dans l'octet
de poids faible de D1 ; les autres octets ne sont pas modifiés]
```

• Attention, pour un opérateur donné, tous les **modes d'adressage** ne sont pas autorisés (cf. sections C et D).

• La mise à jour des indicateurs par les instructions, en particulier par les instructions de comparaison (CMP), est indispensable pour réaliser des branchements conditionnels (B••) (cf. section E).

A.3. Catégories d'instructions

Les instructions se répartissent en différentes catégories.

Opérations arithmétiques : ADD, ADDI, NEG, EXT...

Opérations logiques : OR, AND...

Mouvement (ou transfert) de données : MOVE, EXG, MOVEA...

Comparaisons et branchements : CMP, BRA, B••, JMP...

Autres instructions : NOP, SWAP, TAS...

B - Codage des instructions

B.1. Instruction en assembleur 68000

Instruction : **`OPER.lg source, destination`**

Pour le µprocesseur 68000, les mots sont de 2 octets (16 bits).

Le codage d'une instruction prend de 1 à 5 mots. Il y a toujours **un mot d'instruction**, complété suivant les cas de **0 à 4 mots d'extension**. Les mots (`.W`) sont placés à des **adresses paires** (on parle de « mémoire alignée »).

B.2. Tables pour le mot d'instruction

En général, le mot d'instruction fournit, sur les bits de poids fort l'opération à effectuer (add, jmp, etc) et sur les bits de poids faible des indications sur le (ou les) opérande(s) de l'instruction.

• Mot d'instruction - codes opérations

$b_{15}..b_{12}$	`OPER` (code-op)
0000	Opération sur bits, MOVEP, imm.
0001	MOVE.B
0010	MOVE.L
0011	MOVE.W
0100	JMP, autres instructions
0101	ADDQ, SUBQ, S••, DB••
0110	B•• (brcht conditionnel), BSR
0111	MOVEQ
1000	OR, DIV, SBCD
1001	SUB, SUBX
1010	*non utilisé*
1011	CMP, EOR
1100	AND, MUL•, ABCD, EXG
1101	ADD, ADDX
1110	Décalages et rotations
1111	*non utilisé*

• Mot d'instruction - modes d'adressage des opérandes source et destination
(voir section D pour une description complète des modes d'adressage)

Adressage	***Mode*** *(sur 3 bits)*	***suite mode*** *(sur 3 bits) (n°Registre ou autre)*
Dn	000	n° du registre Dn
An	001	n° du registre An
(An)	010	n° du registre An
(An)+	011	n° du registre An
-(An)	100	n° du registre An
d(An)	101	n° du registre An
d(An,Xi)	110	n° du registre An
Abs.W	111	000
Abs.L	111	001
d(PC)	111	010
d(PC,Xi)	111	011
Immmédiat	111	100

`n° du registre Dn/An` : le µ-processeur 68000 possède 8 registres de données (`Dn`) et 8 registres d'adresse (`An`) (cf. section F) ; donc le numéro de registre peut aller de 0 à 7, ce qui se code bien sur 3 bits.

Les champs ***Mode*** et ***suite mode*** sont placés dans le mot d'instruction suivant les indications du codage de l'instruction. **Bien respecter leur place.**

Attention, ils ne fournissent pas toujours toutes les informations nécessaires pour l'opérande, auquel cas il faut utiliser des **mots d'extension**.

• Mot d'instruction - codes conditions pour les branchements (bits 11 à 8)

$b_{11}..b_8$	B••	appellation
0111	BEQ	EQual
0110	BNE	Not Equal
0100	BCC	Carry Clear
0101	BCS	Carry Set
0010	BHI	HIgh
0011	BLS	Less or Same
1100	BGE	Greater or Equal
1101	BLT	Less Than
1110	BGT	Greater Than
1111	BLE	Less or Equal
1011	BMI	MInus
1010	BPL	PLus
1000	BVC	oVerflow Clear
1001	BVS	oVerflow Set
0000	BRA	True
0001	*BSR*	*Brct Subroutine*

Les trois tables précédentes sont des tables de correspondance, permettant de coder ou de décoder le mot d'instruction. Chaque instruction possède son format de codage et il est nécessaire d'en disposer pour utiliser ces tables, aussi bien pour coder que pour décoder (cf. exemples en section B.4). A l'exécution d'un programme, le processeur accède au codage du programme (données et instructions) placé en mémoire centrale.

B.3. Mots d'extension

Les informations complémentaires et nécessaires sur les opérandes source et/ou destination (adresse en mémoire, valeur immédiate, etc) et n'ayant pas pu être données dans le mot d'instruction sont précisées dans les mots d'extension. Ils sont placés après le mot d'instruction.

Adressage	Mots d'extension
Immédiat .B	**1 mot** d'extension : octet de poids faible
.W	**1 mot** d'extension
.L	**2 mots** d'extension (poids fort puis poids faible)
Abs.W	**1 mot** d'extension pour l'adresse
Abs.L	**2 mots** d'extension (poids fort puis poids faible)
d(An) ou d(PC)	**1 mot** d'extension pour le déplacement
d(An,Xi) ou d(PC,Xi)	**1 mot** d'extension pour indiquer le registre d'index et le déplacement (limité à 8 bits).

Pour une même donnée, les octets sont placés des poids forts aux poids faibles suivant les adresses croissantes en mémoire centrale : le processeur 68000 est *big endian*.
S'il y a des mots d'extension pour la source et pour la destination, ils sont placés (après le mot d'instruction) dans l'ordre : mots d'extension pour la source puis mots d'extension pour la destination.

B.4. Exemples

Exemple 1 : `décodage`

Soit en mémoire centrale à l'adresse `$0005A0 : $6900 puis $FFEE`. En supposant qu'à l'adresse `$0005A0` une instruction commence, quelle est l'instruction codée ?

Solution : `$6900` est donc le codage du mot d'instruction ; il sera peut-être suivi de mots d'extension, mais il faut commencer à décoder l'instruction pour le savoir.

Pour décoder ce mot d'instruction, écrivons-le en notation binaire :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0

$b_{15}..b_{12}$ = `0110` indique (cf. table des codes opérations) qu'il s'agit d'un branchement.
Pour continuer à décoder, il faut disposer du format de codage des branchements ; le voici :

```
BNE   déplacement  ; le déplacement donne le déplacement en octets
                   par rapport à la valeur du compteur ordinal à l'exécution.
```

2 cas sont possibles
<u>1er cas</u> : le déplacement est codé sur un octet (en complément à deux)

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Code-op.				*Code-cond.*				*Déplacement sur 1 octet*							

et il n'y a pas besoin de mot d'extension.

<u>2ème cas</u> : le déplacement est codé sur deux octets (en complément à deux)

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Code-op.				*Code-cond.*				*0000 0000*							

et le déplacement (codé en complément en deux) est codé dans le mot (appelé mot d'extension) qui suit le mot d'instruction.

Dans notre cas, $b_{11}..b_8$ = `1001`. Il s'agit donc d'un `BVS` (oVerflowSet) : branchement si V=1.

Ensuite, $b_7..b_0$ = `0000 0000`. Il s'agit donc du 2ème cas : il y a un mot d'extension qui fournit, codée en complément à deux, la valeur du déplacement.

Donc `déplacement = MC[$0005A2].W =` $<\$FFEE>_{deux} = -<\$0012>_{deux} = -<18>_{dix}$.
D'où l'instruction codée par $<\$6900FFEE>_{68000}$: `BVS #-18` (ou `BVS -18` car il n'y a pas d'ambiguïté : le déplacement doit forcément être donné pour un branchement).

Question complémentaire n°1 : si vous codez `BVS -18`, pouvez-vous aboutir à un autre codage ? Lequel le cas échéant ? Pourquoi ?

Question complémentaire n°2 : à l'exécution de cette instruction et si le branchement est pris, à quelle adresse en mémoire centrale se trouve la prochaine instruction exécutée ? Pourquoi ?

Exemple 2 : codage d'un `ADD`

Il s'agit de coder l'instruction : `ADD.W #100,D3 ; D3.W := D3.W + <100>dix`

Sachant que le codage du mot d'instruction d'un ADD est indiqué comme suit :

```
ADD.lg <AE>,Dn  ou  ADD.lg Dn,<AE>
      ; <AE> pour "adresse affective" de l'opérande
```

Remarque : l'un des deux opérandes d'un ADD est forcément un registre d'adresse.

`<AE> ("adresse affective")` est une notation générique pour indiquer qu'il s'agit d'un accès à un opérande (cf. section C.1 et section D.1) et qu'il y a une "adresse effective" à calculer.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Code-op.				n°registre			**lg**			mode			suite.mode		
										<- AE ->					

lg	octet(.B)	mot(.W)	mot long(.L)	
Dn destin.	000	001	010	ADD <AE>,Dn
Dn source	100	101	110	ADD Dn,<AE>

Note : s'il y a deux registres de données, c'est le format de la première ligne qui est utilisé (donc il n'y a pas de choix à ce niveau).

Pour les champs code-op, mode et suite-mode, il faut se référer aux tableaux de la section B.2. Pour la partie « adresse effective » (AE) il faudra en général compléter l'information par des mots d'extension (par exemple pour un adressage immédiat, absolu - ou direct mémoire -).

Codage :

Mot d'instruction :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	1	0	0	1	1	1	1	1	0	0

Mot d'extension, placé à la suite en MC :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	0

D'où le codage, noté en hexa, de cette instruction : `<$D67C 0064>`$_{68000}$

Exemple 3 : codage d'un `MOVE`

Il s'agit de coder l'instruction : `MOVE.W $04400F,D5 ; D5.W := MC[$04400F].W`

Sachant que le codage d'un MOVE se présente comme suit :

```
MOVE.lg  source,destination
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Code-op.				suite.mode			mode			mode			suite.mode		
				<- destination ->						<- source ->					

Compléter : (en étant attentif aux indications !)

Mot d'instruction :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				<- destination ->						<- source ->					

Mot d'extension, placé à la suite en MC :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

D'où le codage, noté en hexa, de cette instruction : <$ >$_{68000}$?

C - Instructions usuelles : mouvement de données, arithmétiques, logiques, ...

Il est **indispensable** de se souvenir que les instructions ont pour forme :

`OPER.lg source, destination`

`source, destination` sont les deux opérandes sur lesquels l'opérateur `OPER` s'applique.
D'une manière générale, l'instruction réalisée, donnée sous forme algorithmique, est :

`Destination.lg := destination.lg OPER source.lg;`

Dans les descriptifs ci-après, on utilise les notations des documentations du constructeur, c'est-à-dire : `(Dest) OPER (Sour)` → `Dest`. `Dest` désigne l'emplacement (contenant) et `(Dest)` désigne la valeur à cet emplacement (contenu). Le fonctionnement d'une instruction, par exemple une soustraction, étant décrit par : `(Dest) - (Sour)` → `Dest`
Pour être plus explicite par rapport à vos habitudes, notez en plus l'équivalent algorithmique.

Attention, pour chaque "`OPER` " il y a des contraintes sur les modes d'adressages qui sont autorisés pour les opérandes. Les contraintes sont précisées par instruction dans la suite de cette section, ainsi que des informations sur le rôle et l'utilisation des instructions.

C.1. Aperçu des modes d'adressage

Ce tableau indique les modes d'adressage de base. Voir la section D pour le tableau complet.

Nom	*Notation*	*Exemple*	*Remarques*
Implicite		`NOP`	
Immédiat	`Imm.`	`MOVE.B` **`#2,`**`D0`	`pas de calcul AE`
Direct (de registres)	`Dn, An`	`MOVE.B #2,`**`D0`**	`AE = Dn ou An`
Absolu court (direct mémoire)	`Abs.W`	`MOVE.L D4,`**`$2000`**	`AE = Adresse sur 16 bits`
Absolu long (direct mémoire)	`Abs.L`	`MOVE.L D4,`**`$020000`**	`AE = Adresse sur 24 bits`
Indirect (par registre d'adresse)	`(An)`	`NEG.B` **`(A0)`**	`AE = (An)`
Indirect avec déplacement	`d(An)`	`NEG.B` **`16(A0)`**	`AE = (An) + d` `Rmq: An non modifié`

La notation `<AE>` est souvent utilisée dans la désignation d'un opérande. Comme déjà mentionné, `<AE> ("adresse affective")` est une notation générique pour indiquer qu'il s'agit d'un accès à un opérande et qu'il y a une "adresse effective" à calculer (cf. section D.1).

C.2. Mouvement de données et chargement d'adresse

Mouvement de données

♦ **`MOVE`** `(transfert de données)` *Taille:* .B, .W, .L

`MOVE <AE>,<AE>` ; (source) → destination

Source Tout mode d'adressage autorisé

Destination An est interdit (utiliser `MOVEA`) (Bien sûr, le mode immédiat est interdit !)

Exemple `MOVE.B #54,D1` **`; algo : D1.B := <54>dix ;`**

♦ **MOVEA** (transfert vers An) *Taille:* .W, .L

`MOVEA <AE>,An` ; (source) → destination

Avec extension de signe de l'opérande source sur 4 octets et affectation de An sur 4 octets.

Source Tout mode d'adressage autorisé

Exemple `MOVEA.L D1,A0`

Voir l'instruction **LEA** pour affecter un registre d'adresse à l'adresse effective d'une donnée.

♦ **MOVEM** (transfert de registres) *Taille:* .W, .L

`MOVE <liste de registres>,<AE>` ; Registres → destination

`MOVE <AE>,<liste de registres>` ; source → Registres

(avec extension de signe pour tous les registres)

Exemple `MOVEM.L D3 D2 D1 D0 A1,(A0)`

Cette instruction permet une sauvegarde et restauration de registres avec la mémoire.

♦ **EXG** (échange de registres)

`EXG Rn,Rm` ; Rn ↔ Rm *Taille:* .L uniquement

Exemple `EXG A0,D1`

Chargement d'adresse

Cette instruction est essentielle et indispensable dès que l'on utilise des accès en mémoire en mode indirect (par exemple des accès aux éléments d'un tableau, etc).

♦ **LEA** (Load Effective Address) *Taille:* .L uniquement

`LEA <AE>,An` ; Adresse effective → An

L'adresse effective du premier opérande est calculée et chargée dans An (il n'y a pas recherche de l'opérande en mémoire : pour comparaison, l'instruction `MOVEA <AE>,An` effectue le même calcul d'adresse, mais va ensuite chercher l'opérande à l'adresse calculée).

Exemple `LEA 8(A4),A0`

C.3. Arithmétique

♦ **ADD** (addition binaire) *Taille:* .B, .W, .L

`ADD <AE>,Dn`

`ADD Dn,<AE>` ; (destination) + (source) → destination

Il faut obligatoirement un registre Dn, soit en source, soit en destination.

<AE> comme source ou destination Tout mode d'adressage autorisé.

Exemple `ADD.L (A0),D1`

♦ **ADDI** (addition immédiate) *Taille:* .B, .W, .L

`ADDI #donnée,<AE>` ; (destination) + <don. imm.> → destination

La donnée immédiate peut être sur 8, 16 ou 32 bits.

Exemple `ADDI.W #10000,(A1)`

♦ **ADDQ** (addition rapide) *Taille:* .B, .W, .L

`ADDQ #<donnée>,<AE>` ; <don. imm.> + (destination) → destination

La donnée peut prendre une valeur de 0 à 7.

Exemple `ADDQ.W #2,D1`

♦ **ADDX** (addition binaire avec bit d'extension) *Taille:* .B, .W, .L

`ADDX Dn,Dm` ; (source) + (destination) + X → destination

Permet de programmer une addition sur des mots de plus de 4 octets.

♦ **ADDA** (addition à un registre d'adresse) *Taille:* .W, .L

`ADDA <AE>,An` ; (destination) + (source) → destination

Exemple `ADDA.L D0,A0`

Il y a extension de signe de l'opérande source.

♦ **SUB** (soustraction binaire) *Taille:* .B, .W, .L

idem ADD avec : (destination) - (source) → destination

Les mêmes variantes que pour l'instruction ADD existent pour l'instruction SUB :
SUBX, SUBI, SUBQ, SUBA

♦ **MULS** (multiplication signée) *Taille:* .W uniquement

```
MULS <AE>,Dn          ; (source) x (destination) → destination
```

Les 2 opérandes signés sur 16 bits (Dn.W) sont multipliés, le résultat est signé sur 32 bits.
Source seul An est interdit
Exemple `MULS D2,D1`

♦ **MULU** (multiplication non signée) *Taille:* .W uniquement

```
MULU <AE>,Dn          ; (destination) x (source) → destination
```

Idem MULS, mais opérandes et résultat non signés.

♦ **DIVS** (division signée)

```
DIVS <AE>,Dn          ; (destination) / (source) → destination
```

Taille L'opérande destination est pris signé sur 32 bits et divisé par l'opérande source pris signé sur 16 bits. Le quotient est dans le mot de poids faible ; le reste est dans le mot de poids fort. Le signe du reste est le même que celui du dividende (attention, ceci est différent de la division euclidienne). La division par zéro génère une exception ; le débordement du quotient positionne à 1 l'indicateur V.
Source seul An est interdit

♦ **DIVU** (division non signée)

```
DIVU <AE>,Dn          ; (destination) / (source) → destination
```

Idem DIVS, mais opérandes et résultat non signés.

♦ **NEG** (négation) *Taille:* .B, .W, .L

```
NEG  <AE>             ; 0 - (destination) → destination
```

Négation calculée en codage en complément à 2.

♦ **EXT** (extension du signe)

```
EXT  Dn
```

.W : eb_7 est recopié sur eb_{15} à eb_8
.L : eb_{15} est recopié sur eb_{31} à eb_{16}

♦ **ABCD** (addition décimale avec bit d'extension)

```
ABCD  Dn,Dm     ; (destination)dix + (source)dix + X → destination
```

Taille .B uniquement (1 octet = ce qu'il faut pour coder un chiffre en base 10)
Indicateurs

X	N	Z	V	C
*	u	*	u	*

Z est mis à zéro si le résultat est non nul, et est inchangé si le résultat est nul
(positionner Z à 1 pour écrire une addition sur plusieurs octets).
C est mis à 1 si une retenue décimale est générée, et est mis à zéro sinon.
X identique à C.

C.4. Opérations logiques

♦ **AND** (ET logique) *Taille:* .B, .W, .L

```
AND  <AE>,Dn          ; (destination) ET (source) → destination
AND  Dn, <AE>
```

Comme pour le ADD, SUB, etc, l'un des 2 opérandes doit être un registre Dn.

♦ **ANDI** (ET logique immédiat) *Taille:* .B, .W, .L

```
ANDI  #<donnée>,<AE>        ; (destination) ET <don. imm.> → destination
```

♦ **OR** (OU logique inclusif) *Taille:* .B, .W, .L

```
OR  <AE>,Dn                 ; (destination) OU (source) → destination
OR  Dn, <AE>
```

Comme pour le `ADD`, `AND`, etc, l'un des 2 opérandes doit être un registre Dn.

♦ **ORI** (OU logique inclusif immédiat) *Taille:* .B, .W, .L

```
ORI   #<donnée>,<AE>        ; (destination) OU <don. imm.> → destination
```

♦ **EOR** (OU exclusif) *Taille:* .B, .W, .L

```
EOR  Dn,<AE>                ; (destination) ⊕ (source) → destination
```

♦ **EORI** (OU exclusif immédiat) *Taille:* .B, .W, .L

```
EORI  #<donnée>,<AE>        ; (destination) ⊕ <don. imm.> → destination
```

♦ **NOT** (complément à 1) *Taille:* .B, .W, .L

```
NOT  <AE>       ; (destination)‾ → destination
```

C.5. Instructions spéciales

♦ **NOP** (pas d'opération)

```
NOP
```

L'état du processeur est inchangé, seul PC est incrémenté (de 2) pour permettre l'exécution de l'instruction suivante.

♦ **CLR** (mise à zéro) *Taille:* .B, .W, .L

```
CLR  <AE>       ; 0 → destination
```

♦ **SWAP** (échange interne)

```
SWAP  Dn        ; Dn[31..16] ↔ Dn[15..0]
```

♦ **TAS** (test d'un octet et mise à 1 de eb_7) *Taille:* .B uniquement

```
TAS  <AE>
```

L'octet d'adresse <AE> est testé et son eb_7 est mis à 1 ; les indicateurs sont mis à jour par le test : si l'octet est nul alors Z=1 sinon Z=0 ; si eb_7=1 alors N=1 sinon N=0.

Cette instruction s'effectue durant un seul cycle, qui est indivisible.

♦ **LSL,LSR** (décalage logique gauche, droite) *Taille:* .B, .W, .L

```
LSx  Dn,Dm      ; nombre de décalages indiqué dans Dn - pris modulo 64
LSx  #<donnée>,Dm       ; valeur immédiate entre 0 et 7, attention 0 pour 8 décalages
LSx  <AE>       ; .W uniquement, décalage d'une seule position
```

♦ **ROL,ROR** (rotation) *Taille:* .B, .W, .L

```
ROx  Dn,Dm              ; nombre de rotations indiqué dans Dn, pris modulo 64
ROx  #<donnée>,Dm       ; valeur immédiate entre 0 et 7, attention 0 pour 8 rotations
ROx  <AE>               ; .W uniquement, rotation d'une seule position
```

D - Modes d'adressage

Les instructions ont pour forme

`OPER.lg source,destination`

Tableau récapitulatif de tous les modes d'adressage.

Nom	*Notation*	*Exemple*	*Remarques*
Implicite		`NOP`	
Immédiat	`Imm.`	`MOVE.B #2,D0`	`pas de calcul AE`
Direct (de registres)	`Dn, An`	`MOVE.B #2,D0`	`AE = Dn ou An`
Absolu court (direct mémoire)	`Abs.W`	`MOVE.L D4,$2000`	`AE = Adresse sur 16 bits`
Absolu long (direct mémoire)	`Abs.L`	`MOVE.L D4,$020000`	`AE = Adresse sur 24 bits`
Indirect (par registre d'adresse)	`(An)`	`NEG.B (A0)`	`AE = (An)`
Indirect avec déplacement	`d(An)`	`NEG.B 16(A0)`	`AE = (An) + d` `Rmq: An non modifié`
Indirect indexé avec déplacement	`d(An,Ri)`	`NEG.B 16(A0,D0)`	`Ri : Ai ou Di` `AE = (An)+(Ri)+d`
Indirect postincrémenté	`(An)+`	`NEG.B (A0)+`	`AE = (An)` `puis An incrémenté`
Indirect prédécrémenté	`-(An)`	`NEG.L -(A0)`	`An décrémenté` `AE = (An)`
Relatif au compteur ordinal	`d(PC)`	`BRA ad_boucle`	`AE = (PC) + d`
Relatif indexé au compteur ordinal	`d(PC,Ri)`	`BRA 2(PC,D0)`	`AE = (PC)+(Ri)+d`

D.1. Introduction

On indique la *longueur* des opérandes pour chaque instruction en assembleur (**`.lg`**) :

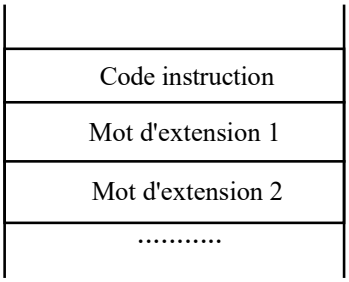
.B (Byte) opérande sur 1 octet
.W (Word) opérande sur 2 octets
.L (Long) opérande sur 4 octets

Par défaut (sans postfixe), c'est la longueur Word qui est prise.

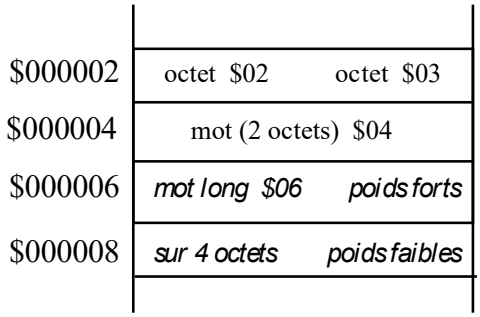
Lorsque l'opérande est sur 2 ou sur 4 octets et que cet opérande est accédé par son adresse en mémoire, cette adresse doit impérativement être paire.

Dans les schémas qui suivent, comme d'habitude, la mémoire est représentée comme une succession de mots (2 octets), avec des ***adresses croissantes de haut en bas***.

L'octet de poids fort est "à gauche" (adresse paire) et l'octet de poids faible est "à droite" (adresse impaire) : le processeur Motorola 68000 est dit « big-endian » (gros-boutiste).



Codage d'une instruction



Représentation de la mémoire et des adresses

Le µprocesseur 68000 autorise 14 types d'adressage distincts.
Du point de vue fonctionnel, ils se répartissent en **5 catégories** :
— il n'y a pas d'opérande (mode implicite)
— valeur de l'opérande donnée dans l'instruction (valeur immédiate)
— valeur de l'opérande dans un registre (direct registre)
— valeur de l'opérande en mémoire (direct mémoire, indirect, déplacement, index)
— la valeur de l'opérande est fonction de l'adresse de l'instruction en cours (adressage relatif au compteur ordinal dans le cas d'un branchement)

Pour les deux derniers cas, il faut calculer l'adresse d'accès à l'opérande. On dit souvent que le résultat de ce calcul est l'***adresse effective*** (ce qui assimile la notation `<AE>` à la valeur de l'adresse de l'opérande après calcul).

Les modes d'adressage pré-décrémenté et post-incrémenté (indispensables à l'appel et au retour de sous-programmes) sont présentés en cours.

Les registres du processeur 68000 (voir section F) ont pour longueur 32 bits (4 octets).

On présente ci-après des exemples à compléter pour les différents cas.
La notation '- - - -' signifie que les chiffres hexa correspondants ne sont pas modifiés.

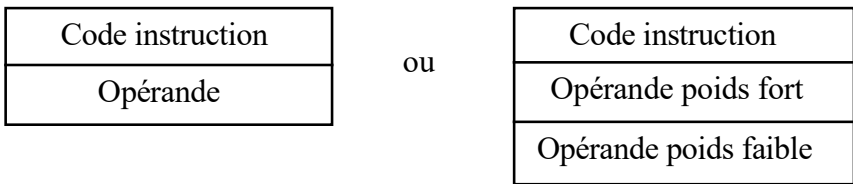
D.2. Mode implicite

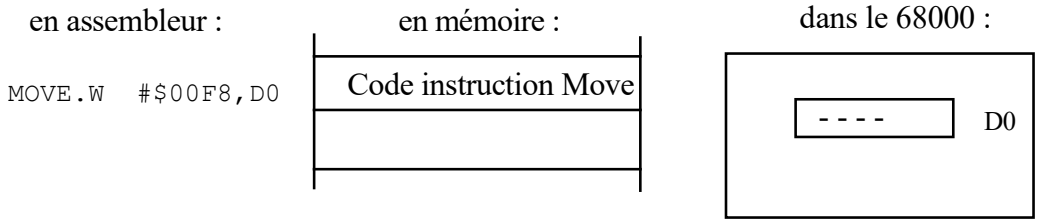
Il n'y a pas d'opérande à préciser : il n'existe pas ou bien il est toujours au même endroit.

D.3. Valeur immédiate

La valeur de l'opérande est donnée dans l'instruction. En langage d'assemblage du 68000, on spécifie qu'un opérande est immédiat en le précédant du caractère #.

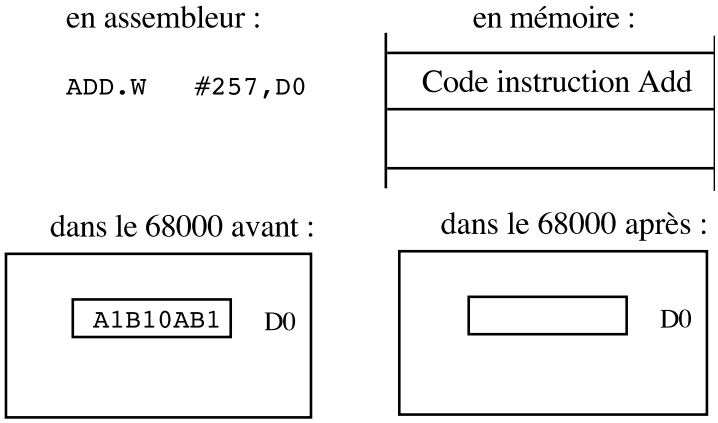
Le nombre d'octets utilisé pour coder la valeur immédiate dépend de la longueur indiquée dans l'instruction, suivant les cas, ce sera 1 mot d'extension (.W) ou 2 mots d'extension (.L) :





D.4. Adressage direct de registres

La valeur de l'opérande est dans un registre du processeur. AE = Dn ou An.
Exemple :



Le compteur ordinal n'est pas adressable directement, il n'est utilisé que dans les modes d'adressage relatifs au compteur ordinal.

Les registres d'adresse ne peuvent être utilisés qu'avec les longueurs `.W` et `.L`.

Si un registre d'adresse est utilisé en destination, il y a extension de signe de l'opérande même s'il porte sur un seul mot. Par exemple :

```
MOVEA.W   #$8F60,A1
```

a pour effet d'affecter A1 à `$FFFF8F60`.

D.5. Adressage direct mémoire (ou absolu court/long)

On indique dans l'instruction l'adresse en mémoire à laquelle l'opérande est placé. L'adresse effective est égale à l'adresse donnée dans l'instruction.

Généralement, l'adresse est exprimée en notation hexadécimale ; elle est alors précédée du caractère $.

Pour le processeur 68000, l'adressage direct mémoire peut être court ou long.

Si l'adressage est court ($ - - - -), l'adresse est étendue sur 32 bits avec *extension de signe* pour donner l'adresse de la donnée en mémoire, c'est-à-dire que :

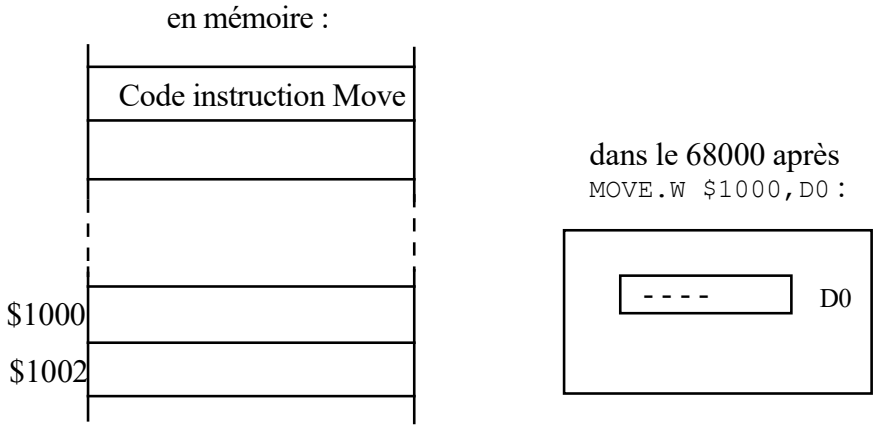
- si $b_{15} = 0$, l'adresse effective est $0000 - - - -
- si $b_{15} = 1$, l'adresse effective est $FFFF - - - -

L'adresse est donnée après le mot d'instruction. Il y a donc un mot d'extension pour un adressage absolu court et deux mots d'extension pour un adressage absolu long.

Exemple (adressage court) :

```
MOVE.W      $1000, D0
```

Le contenu du mot mémoire à l'adresse `$1000` est copié dans le registre D0. Comme il s'agit d'une instruction sur 2 octets (.W), les 2 octets d'adresse `$1000` et `$1001` sont copiés dans les 2 octets de poids faible de D0.



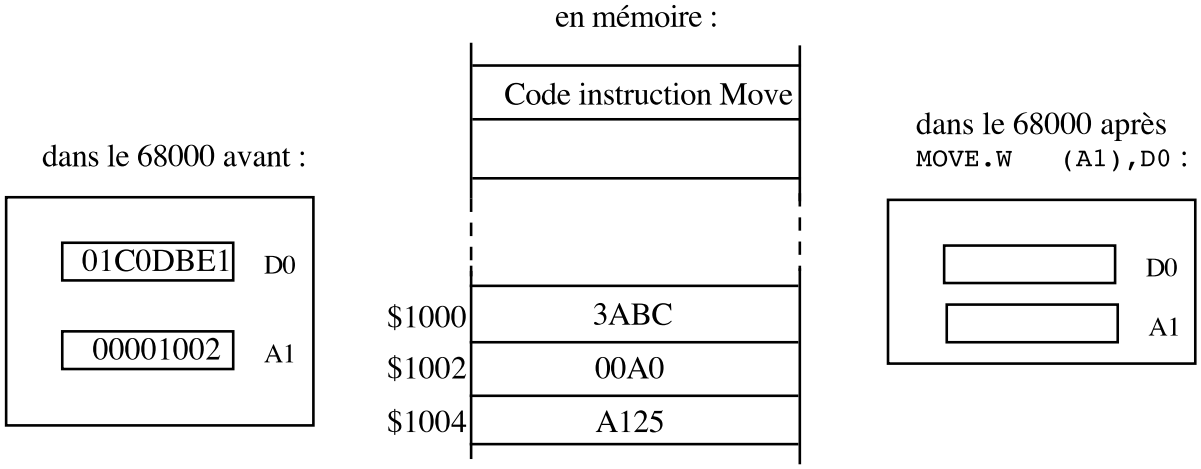
D.6. Adressage indirect (par registre d'adresse)

Un registre Ai du processeur contient l'adresse d'un mot en mémoire, Ai est donc un pointeur vers la donnée. L'adresse effective est le contenu du registre Ai, on désigne ce contenu par la notation (Ai). AE = (An). Le contenu du registre Ai n'est pas modifié.

Exemple :

```
MOVE.W      (A1), D0
```

Le registre A1 contient une adresse. Cette adresse est celle du mot (2 octets) qui doit être copié dans le registre D0.



D.7. Adressage indirect avec déplacement

On utilise un registre d'adresse (par exemple `A4`) comme "base" à l'adressage de données. L'accès aux données se fait en donnant le déplacement (en octets) par rapport à cette base (le contenu de A4).

Le déplacement est codé sur 16 bits dans un mot d'extension de l'instruction et il est étendu (avec extension de signe) sur 32 bits. AE = (Ai) + déplacement étendu.

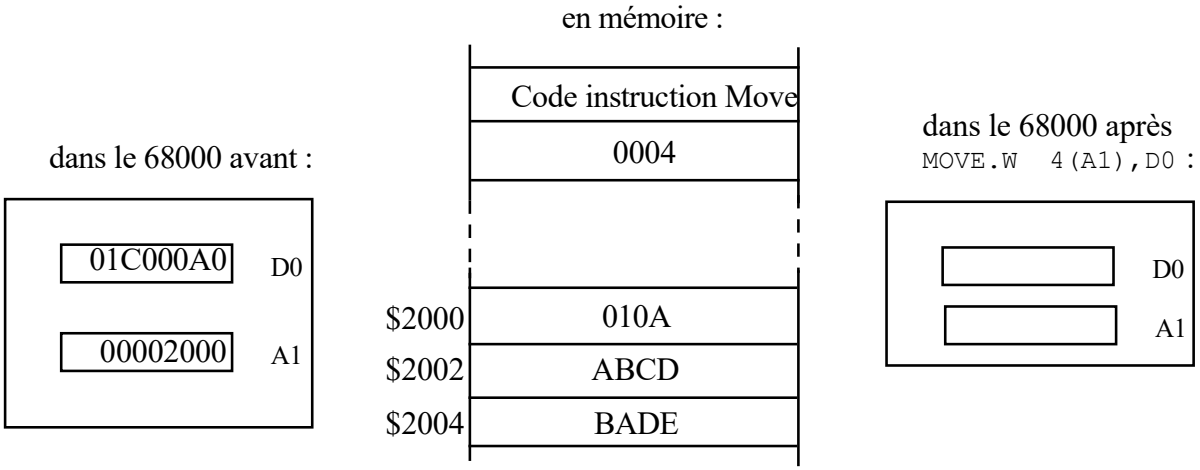
Le déplacement sur 16 bits est interprété comme une valeur signée (-32768 à +32767)

Ce type d'adressage est utilisé pour adresser des données en mémoire à partir d'une adresse fixe (mémorisée dans un registre Ai).

Attention, là non plus, le contenu du registre Ai n'est pas modifié.

Exemple :

```
MOVE.W      4(A1),D0        ; l'adresse en mémoire du 1er opérande est (A1)+4
```



D.8. Adressage indirect indexé avec déplacement

En plus de l'indirection et du déplacement, on indique un registre d'index.

Le contenu de ce registre d'index est ajouté au calcul de l'adresse effective :

AE = (Ai) + déplacement (étendu) + (registre d'index)

Sur le 68000, le déplacement est alors limité à 8 bits. Le contenu de Ai n'est pas modifié.

Le registre d'index (noté `Ri`) doit être l'un des registres de données ou d'adresse (certains processeurs possèdent un registre spécialisé). Il peut être utilisé sur 16 bits (`Ri` ou `Ri.W`) ou sur 32 bits (`Ri.L`).

Ce type d'adressage est utile pour traiter des tableaux : le déplacement donne le début du tableau par rapport à la base (registre d'adresse), et l'index permet de parcourir les éléments du tableau. On peut aussi mettre l'adresse de début du tableau dans un registre d'adresse, il suffit alors de manipuler l'index, et de fixer le déplacement à 0.

Exemple :

```
CLR.W     D1
MOVE.W    4(A0,D1),(A1,D1)
ADDQ.W    #1,D1
MOVE.W    4(A0,D1),(A1,D1)
```

D.9. Adressage pré-décrémenté et adressage post-incrémenté

Exemples :

```
MOVE.W   D1,-(A7) ; Pré-décrémenté (de 2 octets) par rapport à A7.
MOVE.W   (A7)+,D1 ; Post-décrémenté (de 2 octets) par rapport à A7.
```

Ces adressages sont indispensables pour le passage de paramètres par la pile (A7) lors d'appels de sous-programmes. Ils peuvent aussi être utilisés pour parcourir un tableau à partir d'une "base" stockée dans un registre d'adresse. Cf. cours et TP.

Attention, ici le contenu du registre d'adresse utilisé (ici A7) est modifié par l'instruction.

D.10. Adressage relatif au compteur ordinal

L'adresse effective est fonction du contenu du compteur ordinal et d'un déplacement. Au moment de l'exécution de l'instruction, le compteur ordinal contient l'adresse du mot suivant le mot d'instruction (c'est-à-dire adresse du mot d'instruction +2).

De même que pour les adressages indirects, il peut y avoir :

- soit simplement un déplacement : d(PC) $\rightarrow$ déplacement sur 16 bits
- soit un déplacement et un registre d'index : d(PC,Ri) $\rightarrow$ déplacement sur 8 bits

Le déplacement est étendu (avec son "signe") sur 32 bits.

Ces modes d'adressage sont utilisés principalement pour les branchements.

Remarque sur l'adresse effective (notée `AE`) :

La notion d'adresse effective nous a amenés à distinguer un registre de son contenu.

Ainsi, `AE = An` indique que la valeur de l'opérande est dans le registre `An`,

alors que `AE = (An)` indique que la valeur de l'opérande est en mémoire centrale à l'adresse donnée par le contenu de `An` ; cette indirection est notée `(An)`.

Donc `AE = (An)+d` indique que la valeur de l'opérande est en mémoire centrale à l'adresse donnée par le contenu de `An`, auquel on ajoute `d`.

Ce `(An)` correspond à la notion de pointeur des langages de programmation.

E - Branchements

E.1. Instructions de comparaison et branchement

♦ **BRA** (branchement inconditionnel)

`BRA <étiquette>` ; PC + d → PC

Le déplacement, qui sera codé en complément à deux, doit tenir sur 8 bits ou 16 bits.

La valeur du déplacement est calculée par l'assembleur, de manière à ajouter à PC la valeur correcte (à l'exécution du BRA, PC vaut l'adresse de l'instruction BRA+2).

`BRA` n'est pas utilisable si le déplacement est trop grand (non codable sur 16 bits).

♦ **CMP** (comparaison) *Taille:* .B, .W, .L

`CMP <AE>,Dn` ; (destination) - (source)

La destination est inchangée.

♦ **CMPI** (comparaison immédiate) *Taille:* .B, .W, .L

`CMPI #<donnée>,<AE>` ; (destination) - (source)

La destination est inchangée.

♦ **B••** (branchement conditionnel)

`B•• <étiquette>` ; si (condition vraie) alors PC + d → PC
sinon l'instruction exécutée est l'instruction suivante.

Le déplacement (d) est un entier relatif qui doit tenir sur un octet ou un mot. Sa valeur est calculée par l'assembleur (idem BRA).

♦ **JMP** (saut)

`JMP <AE>` ; destination → PC

L'instruction suivante sera celle d'adresse spécifiée dans l'instruction JMP.

Adresse effective (An), d(An), d(An,Ri), Abs.W, Abs.L, d(PC), d(PC,Ri).

Voir la section E.2 pour les possibilités de "••". L'exécution des instructions de branchement conditionnel dépend du positionnement (0 ou 1) des indicateurs par l'instruction qui précède le `B••`. Les conditions sont des expressions booléennes sur les valeurs des indicateurs N, Z, V et C.

E.2. Conditions de branchement

Une instruction B•• est généralement écrite après une instruction de comparaison (CMP ou CMPI).

`CMP source,destination` met à jour les **indicateurs** sur la base du résultat de la soustraction (`destination) - (source)`.

Dans la traduction d'un algorithme en langage d'assemblage, afin d'avoir les mêmes stratégies de programmation, on s'attachera à respecter l'ordre des instructions et des blocs d'instructions, même si ce n'est pas toujours le plus efficace en terme de rapidité d'exécution.

$b_{11}..b_8$	B••	appellation	interprétation (dest) ? (source)	contexte d'utilisation	test sur les indicateurs
0111	BEQ	EQual	égal	quelconque	Z = 1
0110	BNE	Not Equal	différent	quelconque	Z = 0
0100	BCC	Carry Clear	≥ ; non dépassement	base deux	C = 0
0101	BCS	Carry Set	< ; dépassement	base deux	C = 1
0010	BHI	HIgh	>	base deux	C + Z = 0
0011	BLS	Less or Same	≤	base deux	C + Z = 1
1100	BGE	Greater or Equal	≥	complément à 2	N ⊗ V = 1
1101	BLT	Less Than	<	complément à 2	N ⊕ V = 1
1110	BGT	Greater Than	>	complément à 2	Z + (N ⊕ V) = 0
1111	BLE	Less or Equal	≤	complément à 2	Z + (N ⊕ V) = 1
1011	BMI	MInus	négatif	cpt à 2 ; ± ||	N = 1
1010	BPL	PLus	positif	cpt à 2 ; ± ||	N = 0
1000	BVC	oVerflow Clear	pas de débordement	complément à 2	V = 0
1001	BVS	oVerflow Set	débordement	complément à 2	V = 1
0000	BRA	True	vrai	quelconque	1
0001	*F*	False	faux	-	0

• **IMPORTANT** : lorsque l'instruction de branchement conditionnel est précédée d'une instruction de comparaison (`CMP, CPMI...)`, plutôt que le tableau précédent, utilisez la liste ci-dessous pour **choisir la bonne condition**.
NB. Attention à l'ordre de la comparaison (à indiquer dans le commentaire) qui est induit par l'ordre des opérandes du `CMP` :

```
          CMP source,destination    ; compare (destination) ? (source)
          B.. ad_brcht              ; va à ad_brcht pour cond. du B.. vraie
          IIIIII                    ; instruction exécutée pour cond. fausse
          ...
ad_brcht  JJJJJJ                    ;
```

*Pour des **entiers naturels** (codage en base deux)*

BCC	branchement si (destination) ≥ (source)	*Carry Clear*
BCS	branchement si (destination) < (source)	*Carry Set*
BHI	branchement si (destination) > (source)	*High*
BLS	branchement si (destination) ≤ (source)	*Less or Same*

*Pour des **entiers relatifs** (codage en complément à deux)*

BGE	branchement si (destination) ≥ (source)	*Greater or Equal*
BLT	branchement si (destination) < (source)	*Less Than*
BGT	branchement si (destination) > (source)	*Greater Than*
BLE	branchement si (destination) ≤ (source)	*Less or Equal*

NB. L'instruction `CMP`, **préalable** au branchement conditionnel, n'est pas toujours indispensable. D'autres instructions positionnent les indicateurs (du registre d'état SR). A vous de l'utiliser à bon escient.

• Exemple pour des entiers codés sur des mots (2 octets)

Pour traduire la structure conditionnelle suivante :

```
si (D0 ≥ D2) ou (D0 < D1)
alors D1 := D1 + D0
sinon D1 := D1 - D2
finsi ;
```

On détermine, dans l'ordre des conditions à évaluer et pour chaque condition :
- dans quel cas un branchement doit être pris ;
- et quelle est l'instruction cible dans ce cas.
NB. Sinon, l'exécution du programme continue « en séquence ».
Il est essentiel de **respecter le même ordre des instructions que dans l'algorithme**.

Sur cet exemple, on note tout d'abord que la condition à évaluer est un `ou`.

On commence par la première clause (de même que le fait un compilateur).
Aussi, lorsque `(D0 ≥ D2)`, cela suffit pour décider d'aller au `alors`.
Par contre, lorsque `(D0 < D2)`, il faut aller évaluer la 2ème clause du `ou`.

On raisonne ensuite sur la 2ème clause.
Lorsque `(D0 < D1)`, il faut aller exécuter les instructions du `alors`.
Par contre, lorsque `(D0 ≥ D1)`, il faut aller exécuter les instructions du `sinon`.

On résume ce raisonnement sur l'algorithme, en ne notant que les cas de « rupture de séquence » (où aller et pour quelle condition) : à vous de compléter ci-dessous avec 2 ou 3 flèches :

```
si      (D0 ≥ D2)

        ou

        (D0 < D1)

alors D1 := D1 + D0

sinon D1 := D1 - D2

finsi ;
```

On passe ensuite à la traduction en assembleur.
On se place dans le cas où les données sont des nombres entiers naturels (codés en base deux).

Il est **indispensable d'écrire les commentaires** associés aux instructions `CMP` et `B..`

```
si      CMP.W   D0,D2   ; D2 ? D0
        BLS     alors   ; pour D2 ≤ D0 aller à alors (d'où BLS)

ou      CMP.W   D1,D0   ; D0 ? D1
        BCC     sinon   ; pour D0 ≥ D1 aller à sinon (d'où BCC)

alors   ADD.W   D0,D1
        BRA     finsi               indispensable

sinon   SUB.W   D2,D1
finsi   NOP
```

• Dans la pratique, l'instruction `CMP` est souvent essentielle, puisqu'elle permet de comparer deux opérandes et de mettre à jour les indicateurs en conséquence.

Cependant elle n'est pas indispensable dans tous les cas, puisque ce sont les valeurs des indicateurs qui sont importantes.

Exemple :

```
        ADD.W   D2,D1   ; met à jour entre autres l'indicateur N
        BNE     ad_toto ; branchement à ad_toto si Z=0, c-à-d D1≠0
```

... à vous de maîtriser ce que vous faites.

L'instruction `TST` (Test an Operande) met à jour les indicateurs N et Z en fonction de l'opérande (et positionne V et C à 0).

E.3. Utilisation d'étiquettes

Comme déjà dit, on prendra l'habitude de conserver l'ordre des blocs d'instructions de l'algorithme dans la traduction en assembleur des instructions conditionnelles, des boucles, etc. C'est bien ce que nous avons fait dans l'exemple précédent.

Voici récapitulé la structure d'un programme en assembleur pour un si - alors - sinon.

"si conditions alors instructions1 sinon instructions2" se traduit par :

```
ad_si     évaluation conditions (pouvant nécessiter plusieurs
          instructions dont des branchements conditionnels)
ad_alors  instructions1
          branchement à ad_finsi
ad_sinon  instructions2
ad_finsi  suite du programme
```

Pour ne pas avoir à manipuler explicitement les adresses des instructions, vous remarquez que l'on a utilisé des *étiquettes*. Dans notre exemple, `ad_si` et `ad_alors` aident à la lisibilité du programme, mais ne sont pas indispensables.

Le programme d'assemblage calcule lui-même les déplacements à effectuer. Soit d le déplacement calculé ; l'instruction de branchement réalise alors simplement une modification du compteur ordinal par : (PC)+d → PC (au lieu de l'incrémentation simple du compteur ordinal).

Un déplacement peut être positif (branchement en avant) ou négatif (branchement en arrière). Le déplacement est codé en complément à 2 sur 8 bits ou sur 16 bits.

E.4. Autres exemples de branchements

Vous pouvez réfléchir à la traduction en assembleur d'autres schémas algorithmiques, comme les boucles « tant que ». Voici quelques exemples.

while (condition simple) {instructions}

Par exemple, l'itération (avec n entier sur 16 bits) :

```
while (n!=10) {instructions}
```

s'écrit en assembleur :

```
ad_tq   CMPI.W       #10,n
        BEQ          ad_ftq
        {instructions}
        BRA          ad_tq
ad_ftq  {suite du programme}
```

while (condition avec ET) {instructions}

Par exemple, l'itération (avec n et m entiers sur 16 bits) :

```
while (n!=10) and (m!=20) {instructions}
```

s'écrit en assembleur :

```
ad_tq   CMPI.W      #10,n
        BEQ         ad_ftq
        CMPI.W      #20,m
        BEQ         ad_ftq
        {instructions}
        BRA         ad_tq
ad_ftq  {suite du programme}
```

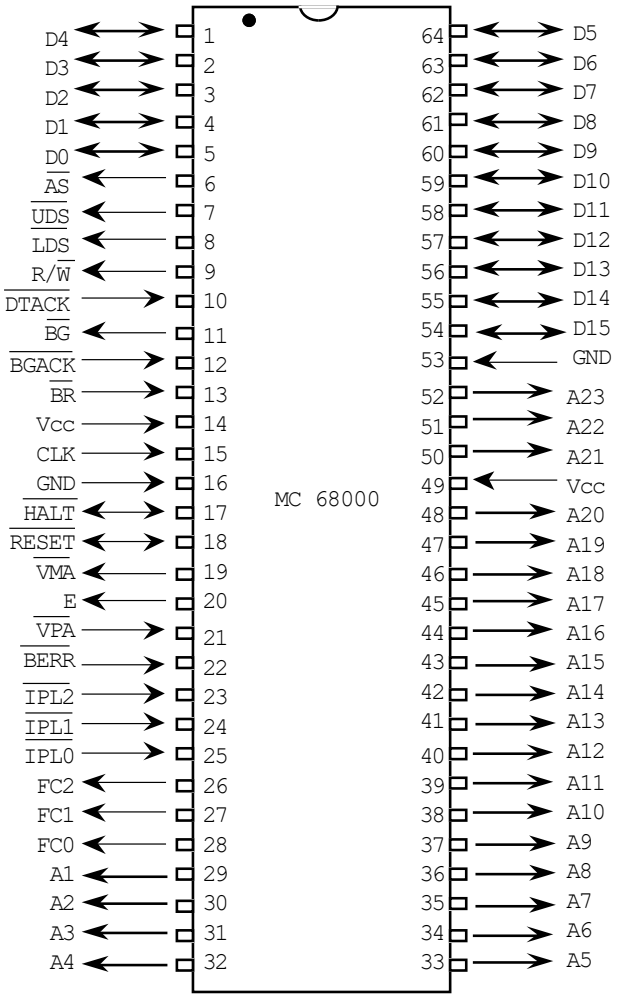
F - Le µprocesseur 68000

F.1. Brochage

Processeur à architecture CISC (Complex Instruction Set Computer).
Commercialisé en 1980.
Fréquence de 4 à 12,5 MHz.

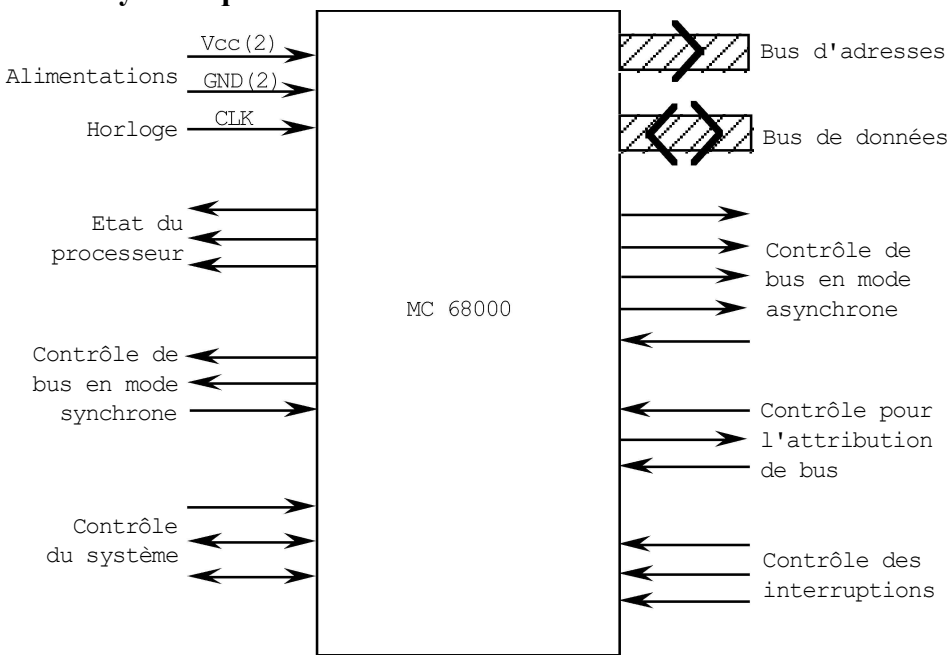
Le boîtier comprend 64 broches. Les lignes associées sont des entrées, des sorties, ou bidirectionnelles. Elles se répartissent en trois catégories :

- alimentations et horloge
- lignes de données et lignes d'adresse
- signaux de contrôle



Le bus d'adresse est bien sûr unidirectionnel, 3 états. La ligne A_0 est inexistante en externe ; mais les lignes $\overline{UDS}$ et $\overline{LDS}$ (UDS : Upper Data Strobe ; LDS : Lower Data Strobe) s'associent à la ligne $R/\overline{W}$ (lecture si 1, écriture si 0) pour définir l'octet de poids faible ou de poids fort *(non détaillé davantage ici)*.

Représentation symbolique du 68000 :



F.2. Registres

- 8 ***registres de données*** de 32 bits D0-D7 *(Data Register)*
  // ne pas confondre avec la numérotation des lignes de données !

- 8 ***registres d'adresse*** de 32 bits A0-A7 *(Address Register)*
  // ne pas confondre avec la numérotation des lignes d'adresse !
  // A7 a un rôle spécial :
  - ***pointeur de pile*** utilisateur de 32 bits A7 (ou USP : *User Stack Pointer*)
  - pointeur de pile superviseur de 32 bits A7'(ou SSP : *Supervisor Stack Pointer*)

- 1 ***compteur ordinal*** de 32 bits PC *(Program Counter)*

- 1 ***registre d'état*** de 16 bits SR *(Status Register)*

SR : Registre d'état (16 bits)

15		13			10	9	8				4	3	2	1	0
T	0	S	0	0	I2	I1	I0	0	0	0	X	N	Z	V	C
octet superviseur								octet utilisateur (CCR)							

Si **T=1**, le microprocesseur est en mode trace : après chaque instruction, une exception est levée permettant à un programme de mise au point (debugger de bas niveau) de contrôler l'exécution.

Si **T=0**, l'instruction suivante est exécutée.

Si **S=1**, le microprocesseur fonctionne en mode superviseur (sécurité système). On peut écrire dans tout SR.

Si **S=0**, le microprocesseur fonctionne en mode utilisateur. On ne peut écrire que dans l'octet utilisateur ; on ne peut pas accéder au pointeur de pile superviseur (A7'), ni adresser les boîtiers réservés au superutilisateur ; les instructions dites "privilégiées" sont interdites. Si l'utilisateur tente de le faire, une exception est levée.

I2, I1, I0 sont relatifs aux interruptions. Ils définissent le niveau de priorité à partir duquel une interruption demandée par un circuit externe sera effectivement prise en compte.

F.3. Rôle des registres

Registres de données (D0-D7) de 32 bits

Ils sont utilisés par le programmeur comme des tampons de stockage temporaires.
Ils sont exploités au mieux par le compilateur.
Le langage C permet au programmeur de forcer une variable à être "placée" dans un registre, en précisant *register* pour la classe d'allocation de cette variable (à employer avec maîtrise...). Cette utilisation entraîne que l'accès à l'adresse d'une variable de type register par l'opérateur & (langage C) est impossible.

Registres d'adresse (A0-A6) de 32 bits

Les registres d'adresse traitent seulement des opérandes de 2 ou 4 octets (ils n'acceptent pas les opérandes octet).
Lorsqu'ils sont utilisés en *destination* (2ème opérande), leurs 4 octets sont affectés avec extension de signe (cf. section D.4).

Exemples à compléter :

```
MOVEA.L  #$2000,A1    ; A1 est affecté à : ?
MOVEA.L  #$8000,A1    ; A1 est affecté à : ?
```

Compteur ordinal (PC) de 32 bits

Ce registre contient : l'adresse de l'instruction courante **+2** (c'est-à-dire l'adresse du mot qui suit le mot d'instruction qui a été stocké dans le registre d'instruction).
Il peut être encore modifié à la fin de chaque instruction, s'il y a des mots d'extension ou si l'instruction est un branchement qui est pris.

Registre d'instruction

Le registre d'instruction contient le "mot d'instruction" - code de l'instruction en cours d'exécution. Il est affecté à chaque cycle instruction. Il n'est pas accessible à la programmation.

Pointeurs de pile A7 et A7'.

Ce sont des registres d'adresse particuliers. Ils sont nécessaires pour les appels et retours de sous-programmes.

Registre d'état : indicateurs

L'octet utilisateur renseigne sur l'état du processeur après une instruction arithmétique ou logique. On le note aussi **CCR** (*Conditions Code Register*), il donne les valeurs des indicateurs (ou codes conditions).

C : Carry ; C=1 si Dépassement (significatif dans une opération en base deux).
V : Overflow ; V=1 si Débordement (significatif dans une opération en complément à 2).
Z : Zéro ; Z=1 si le résultat est nul.
N: Négatif ; N=1 si (bit de poids fort =1) après une instruction (significatif pour des nombres codés en complément à 2).
X : Extension ; c'est la copie de C sauf pour les instructions de transfert de données (il sert de bit de retenue pour les opérations sur opérandes étendus : ADDX, ROLXL, ROLXR).

Complétez les exemples de mises à jour d'indicateurs, avec comme notations :
— : pour non modifié * : pour mis à jour u : pour indéfini
0 : pour mise à 0 1 : pour mise à 1

	Indicateurs				
Instruction	X	N	Z	V	C
MOVE					
ADD					
AND					
B••					
JMP					
CMP					

Donnez, pour les instructions suivantes (exécutées en séquence), les valeurs des indicateurs et du registre D1 :

```
                        X  N  Z  V  C
MOVE.B    #$F0,D1                      D1.B =

ADD.B     #$02,D1                      D1.B =

ADD.B     #$0F,D1                      D1.B =

JMP       ad_toto
```

G - bsvc : simulateur de processeur 68000

G.1. Description

Le logiciel **bsvc** est un simulateur du microprocesseur 68000, c'est-à-dire qu'il permet d'exécuter des programmes spécifiés dans le langage d'assemblage du 68000.

Les différents "formats" d'un programme sont : format source, format exécutable, format désassemblé.

Le simulateur exécute des programmes (format exécutable en langage machine 68000) préalablement stockés dans des fichiers portant le suffixe **`.h68`**. Les programmes exécutables sont chargés dans le simulateur par la commande **`Load Program`** du menu **`File`**.

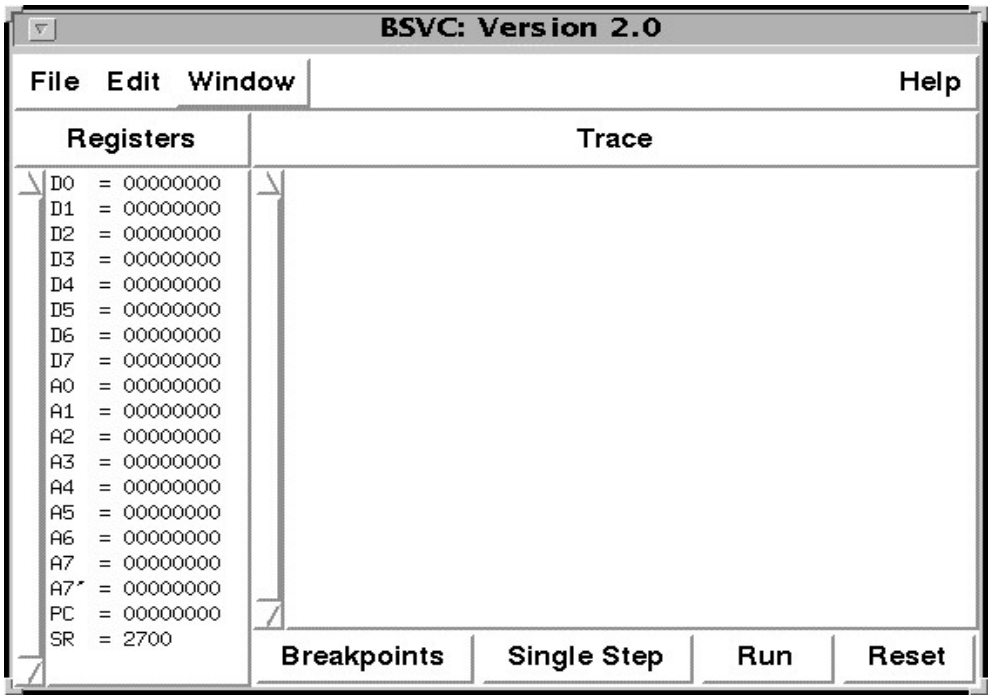
Le programme en langage machine **`toto.h68`** est obtenu à partir du programme source **`toto.s`** (forme textuelle écrite selon la syntaxe du langage assembleur 68000). Cette traduction est effectuée par un programme spécifique appelé **assembleur**. Le programme assembleur utilisé dans les travaux pratiques est : **`68kasm`**.

L'opération inverse est appelée **désassemblage** : elle produit la forme source à partir de la forme exécutable **`toto.h68`**. Avec le simulateur bsvc, il sera possible de visualiser (dans la fenêtre **`listing`**) un format désassemblé couplé avec des informations du format source (commentaires, noms des variables). Ce format est produit par le programme assembleur.

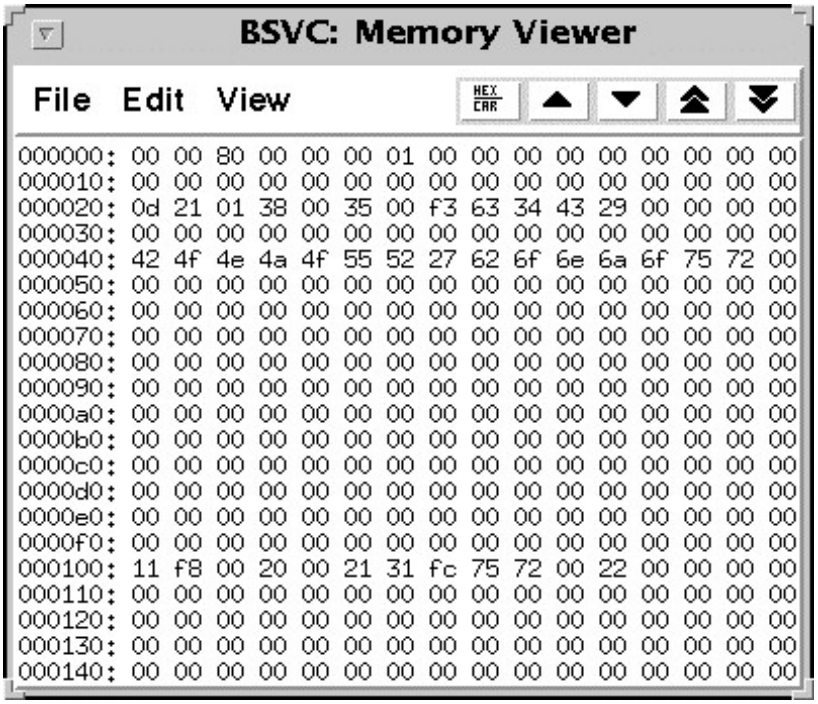
G.2. Les différentes fenêtres de bsvc

bsvc possède un environnement graphique avec plusieurs fenêtres pour faciliter l'observation de l'exécution d'un programme.

1. La **fenêtre principale**, affichée au lancement du logiciel bsvc. Elle visualise les registres et la trace d'exécution.



2. La **fenêtre de visualisation de la mémoire** du 68000 Memory Viewer. Cette fenêtre est ouverte lorsque l'on lance la commande **`Memory Viewer`** du menu **`Window`** de la fenêtre principale.



3. La **fenêtre de listing** qui visualise la version désassemblée du programme à exécuter. Cette fenêtre s'ouvre lorsque l'on lance la commande **`Program Listing`** du menu **`Window`**.

G.3. Chargement d'un programme et lancement du simulateur 68000

Ces consignes seront à appliquer pour différentes séances de TP. On note ici **xx** le n° du TP.

1. Après vous être connectés sur votre compte Linux :
• Placez-vous (ou créez si besoin) dans le répertoire **`Archi`**.
• Créez un répertoire que vous appellerez **`tpxx`**.
• Placez-vous dans ce répertoire.
• Recopiez dans ce répertoire les fichiers du répertoire
`/users/info/pub/commun/r1-03archi/tpxx/`
avec la commance **`cp /users/info/pub/commun/ r1-03archi/tpxx /* .`**

2. Lancez le simulateur 68000 avec la commande en arrière-plan **`bsvc &`**
(le **&** après une commande lance son exécution en " arrière-plan " et vous retrouvez le contrôle du shell dans la fenêtre de commande).

3. Chargez le programme exécutable (**tpx.h68**) avec la commande **`Load Program`** du menu **`File`**.

4. Ouvrez la fenêtre permettant de visualiser la mémoire avec la commande **`Memory Viewer`** du menu **`Window`**.

5. Ouvrez la fenêtre permettant de visualiser le programme désassemblé (**tpx.lis**) avec la commande **`Program Listing`** du menu **`Window`**.

G.4. Consultation de la mémoire

C'est la fenêtre **`Memory Viewer`** qui permet d'observer la mémoire (qui contient les données et les instructions du programme à exécuter).

Le contenu de la mémoire est affiché en mettant 16 octets par ligne (deux chiffres hexadécimaux par octet) (soit aussi $<10>_{seize}$ octets par ligne).

L'adresse du 1er octet de chaque ligne est donnée en début de ligne. Cette adresse est notée en base seize :

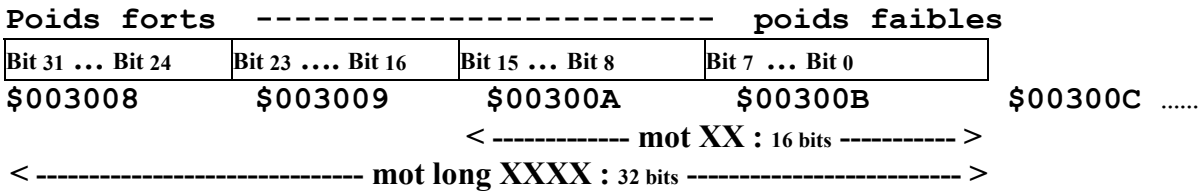
$000000 pour le premier octet de la première ligne,
$00000F pour le dernier octet de la première ligne,
$000010 pour le premier octet de la deuxième ligne, etc.

Les "boutons" en haut à droite dans la fenêtre d'affichage de la mémoire vous permettent de vous déplacer dans la mémoire.

On peut charger en mémoire des données de la taille :

- d'un **octet** (8 bits),
- d'un **mot** (16 bits = 2 octets),
- d'un **mot long** (32 bits = 4 octets)
- d'une **chaîne** de caractères (un octet par caractère codé en ASCII).

L'adresse d'une zone mémoire de plusieurs octets est par convention donnée par l'adresse de l'octet d'adresse la plus petite (ici $00300A pour le mot XX et $003008 pour le mot long XXXX) .



G.5. Exécution d'un programme

Il faut charger en mémoire centrale un programme (en format exécutable) avant qu'il soit exécuté par le processeur (ici le simulateur). La mémoire centrale contient alors les données et les instructions de ce programme.

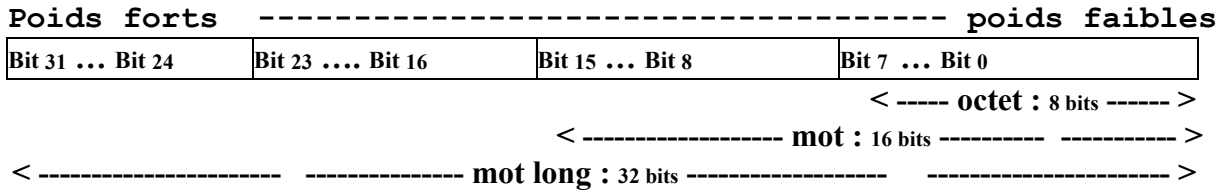
Les boutons en bas de la fenêtre principale permettent de lancer et de contrôler la simulation de l'exécution d'un programme (préalablement chargé en mémoire) :

- Le bouton **`Reset`** permet de se positionner au début du programme (mise à jour de PC par le 2ème mot long de la mémoire centrale et mise à 0 de SR).
- Le bouton **`Single Step`** permet d'exécuter un pas de programme (une seule instruction est exécutée) : **fonctionnement à utiliser en priorité**.
- Le bouton **`Run`** lance l'exécution du programme complet (ATTENTION, le programme peut ne jamais s'arrêter s'il comporte une boucle infinie … il faut alors « tuer » le programme simulateur) : **fonctionnement à éviter !**
- Le bouton **`Breakpoint`** (points d'arrêt) permet de spécifier des instructions sur lesquelles le simulateur devra stopper (lors de la commande **`Run`**).
- Il est possible d'exécuter une ligne particulière en modifiant la valeur du registre PC (**`Program Counter`**) en y mettant l'adresse de l'instruction à exécuter.

G.6. Utilisation des registres du 68000

Le 68000 possède un certain nombre de registres qui sont visualisés sous notation hexadécimale dans la fenêtre principale de bsvc. Parmi eux, on trouve :

• 8 registres de données (de **D0** à **D7**) pouvant contenir la valeur d'un opérande ou le résultat d'une opération. D'un point de vue algorithmique, ces registres sont des variables internes au processeur, aux noms prédéfinis (D0, D1, ...). En fonction du format des instructions, les données seront formatées dans ces registres sur 8, 16 ou 32 bits.



• **Le registre d'état** (**SR** pour **Status Register**). Les 4 bits de **poids faible** de SR correspondent à 4 indicateurs dont la valeur est mise à jour après la plupart des instructions. Ces 4 indicateurs sont :

Vous pouvez réinitialiser la valeur d'un registre en double-cliquant dessus et en saisissant des zéros (ou toute autre valeur).

H - Langage d'assemblage pour 68000

H.1. Présentation

Cette documentation est une introduction au langage de l'assembleur 68000 utilisé avec le simulateur bsvc.

La syntaxe présentée ici vous permet de créer des programmes source (stockés dans des fichiers source, par exemple toto.s) qui seront traduits par le programme traducteur appelé " assembleur 68000 " en instructions du processeur 68000 (programmes exécutables par le simulateur bsvc, par exemple toto.h68).

H.2. Appel de l'assembleur

```
Commande : 68kasm [-clna] prog.s

Options :   -l    Production d'un fichier de listing (toto.lis)
            -n    Pas de production de fichier objet (toto.h68)
            -a    génération des adresses en .L
```

H.3. Format des fichiers source

Format d'une ligne

L'entrée de l'assembleur est un fichier de texte contenant des **instructions**, des **directives d'assemblage** et des commentaires. Chaque ligne comporte les champs suivants :

```
ETIQUETTE          OPERATION          OPERANDE          COMMENTAIRE
```

Exemple

```
LOOP               MOVE               #34,D0            ligne exemple
```

Les champs peuvent être séparés par une combinaison quelconque de caractères espace et tabulation. Il ne doit y avoir ni espace, ni tabulation à l'intérieur d'un champ (sauf dans le champ commentaire et dans les chaînes entre quotes).

Champ étiquette

Il permet de définir un symbole utilisable n'importe où dans le programme pour référer la location originale de l'étiquette. Il doit commencer par une lettre ou un point.

Champ opération

Il contient le code mnémonique d'une instruction à assembler ou d'une directive d'assemblage. Il **ne doit pas commencer en première colonne** sous peine d'être confondu avec une étiquette.

Champ opérande

Une opération peut être suivie d'un ou plusieurs opérandes qui sont séparés par une **virgule** (sans espace ni avant ni après). Dans une instruction, un opérande peut être un nom de registre, une étiquette, une valeur immédiate (précédée du caractère #) ou une expression. Dans une directive, c'est une valeur qui peut être entrée sous la forme suivante :

• Nombre décimal : Séquence de chiffres (0-9) de longueur quelconque.

• Nombre hexadécimal : Séquence commençant par un dollar ($) et suivie d'une séquence de chiffres hexadécimaux (0-9, A-F).

• Nombre binaire : Séquence commençant par un caractère pour-cent (%) et suivie d'une séquence de 1 et de 0.

• Caractères ASCII : Séquence de caractères entre des marques de quote simples (').

Champ commentaire

Un commentaire de la ligne concernée. Un commentaire peut aussi être inséré en étant précédé du caractère * au début d'une ligne ou après une étiquette.

H.4. Opérateurs dans les expressions

Les opérations permises dans les expressions sont décrites ci-après dans l'ordre de priorité décroissantes. Au sein d'un groupe, les opérateurs ont la même priorité et sont évalués de gauche à droite (sauf pour le groupe 2 qui est évalué de droite à gauche).

1.	***()***	***sous-expression parenthésée***
2.	-	moins unaire (complément à 2)
	~	non bit-à-bit (complément à 1)
3.	<<	décalage à gauche (x<<y produit un décalage de y bits et y 0 sont insérés)
	>>	décalage à droite
4.	&	et bit-à-bit
	!	ou bit-à-bit
5.	*	multiplication
	/	division
	\	modulo
6.	+	addition
	-	soustraction

H.5. Spécification des modes d'adressage

Le 68000 offre 14 modes d'adressage généraux dont seulement 4 sont donnés ci-dessous. Les symboles suivants sont utilisés pour décrire le format des opérandes :

```
Dn    = registre de données
An    = registre d'adresse
.lg   = code de taille du registre d'index
        (.W ou .L, si rien de précisé ⇒.W)
<ex>  = n'importe quelle expression
```

Mode	***Format d'assemblage***
Direct registre de données	Dn
Direct registre d'adresse	An
Absolu	<ex>
Immédiat	#<ex>

H.6. Directives d'assemblage

ORG - Set Origin

L'assembleur tient à jour un **compteur d'emplacement** de 32 bits dont la valeur est initialement nulle et qui est incrémenté chaque fois qu'une nouvelle instruction est assemblée ou qu'une directive de stockage est exécutée. La valeur de ce compteur d'emplacement peut être fixée avec la directive **ORG**. Cela est typiquement fait au début d'un programme et à des endroits appropriés à l'intérieur de celui-ci. Le format de la directive **ORG** est le suivant :

```
<étiquette>        ORG              <expression>
```

où **`expression`** est la valeur à affecter au compteur d'emplacement. L'étiquette est optionnelle.

DC - Define Constant

Cette directive est utilisée pour stocker des chaînes et des listes de constantes dans la mémoire ou pour définir des **variables initialisées**. Le format de cette directive est :

```
<étiquette>        DC.<longueur>      <élément>,<élément>, …
```

L'étiquette sera égale à l'adresse du début de la liste des éléments. `Longueur` indique qu'une liste d'octets (**.B**) de mots (**.W**) ou de mots longs (**.L**) est en train d'être définie. Si la longueur n'est pas précisée alors la longueur **mot** est utilisée par défaut.

Une liste d'éléments suit la directive ; chaque élément peut être une expression numérique ou une chaîne.

DS - Define Storage

Cette directive génère un bloc d'octets, de mots ou de mots longs **non initialisés**. Le format de cette directive est :

```
<étiquette>        DS.<longueur>      <nombre>
```

L'étiquette sera égale à l'adresse du début du bloc. `longueur` spécifie qu'un bloc de **<nombre>** octets, mots ou mots longs est réservé. Si la longueur n'est pas spécifiée, la longueur **mot** est utilisée par défaut.

nombre est défini par une expression numérique.

END - Fin du fichier source

Cette directive est utilisée pour indiquer la fin du fichier source. Cette directive est **obligatoire**. L'assembleur ignorera tout ce qui suit cette directive. Le format de cette directive est simplement :

```
END
```

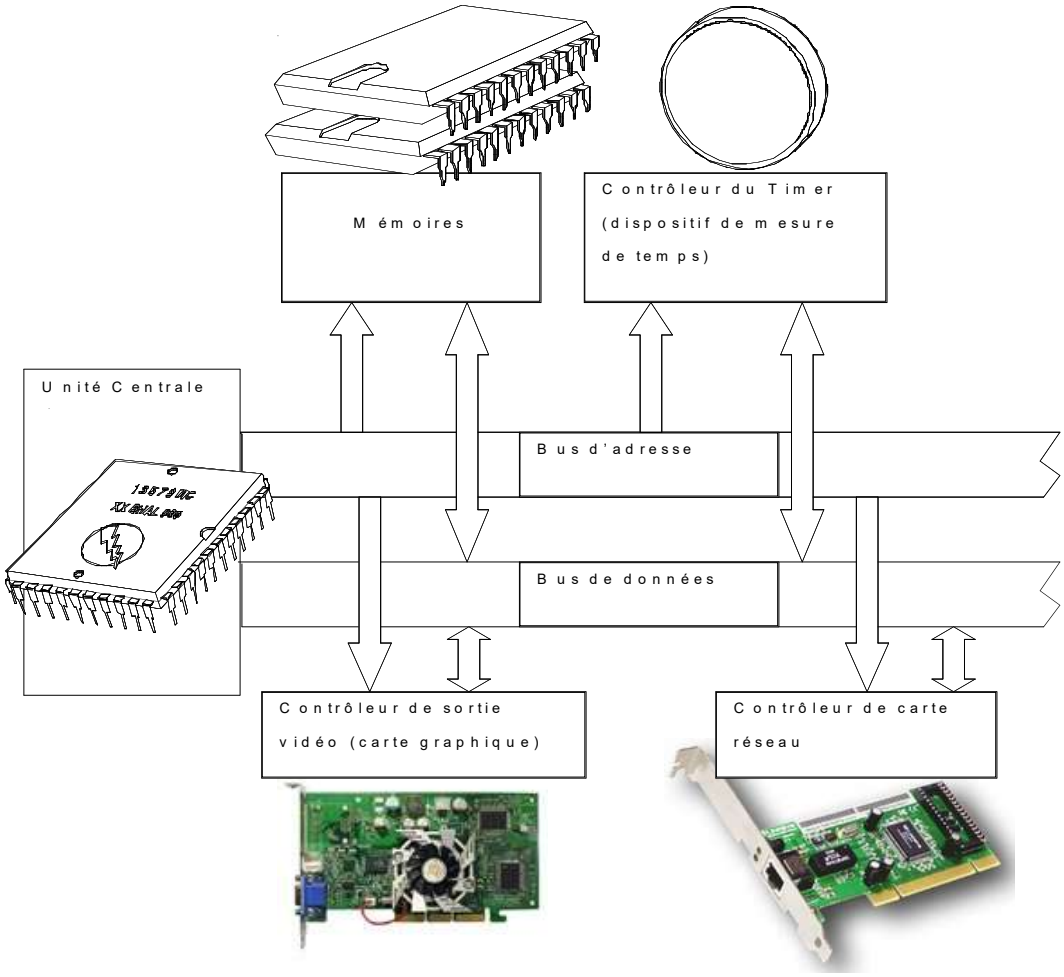
I - Circuits de contrôle de périphériques

I.1. Organisation globale et interruptions

Les ordinateurs sont composés d'une unité centrale, de mémoires et de différents périphériques. Ces périphériques sont "attachés" au processeur via les bus de données et d'adresse.

Chaque périphérique possède une adresse (comme une mémoire) et peut échanger des données avec l'unité centrale. Chaque périphérique a des registres qui permettent au processeur de piloter ce périphérique en écrivant ou lisant des données dans ces registres via le bus de données.

L'adresse d'un registre R d'un périphérique P s'obtient par l'addition de l'adresse de P avec l'adresse de R interne à P (vue comme un déplacement pour accéder au registre). Le programme qui gère les accès à ces registres et de ce fait le comportement du périphérique est un pilote de périphérique (ou Driver).



Organisation globale d'un ordinateur

Les périphériques augmentent les possibilités d'action d'un processeur. Les périphériques d'entrée (comme un clavier) permettent d'entrer des données dans l'ordinateur, les périphériques de sortie (comme un écran) permettent d'observer les données de l'ordinateur. Certains périphériques servent à la fois aux entrées et sorties. Les périphériques peuvent fonctionner indépendamment du processeur. Ils génèrent des événements. Un événement correspond à un changement d'état d'un périphérique. Ce changement d'état est observable dans les registres du périphérique.

Pour traiter ces événements, le programmeur (c'est-à-dire vous) a deux possibilités :

La scrutation (attente active) : le processeur teste régulièrement l'état du périphérique, c'est-à-dire les octets stockés dans les registres du périphériques. Tant que l'état souhaité du périphérique n'est pas atteint, le processeur recommence le test. Pendant cette scrutation, le processeur ne peut faire aucune autre instruction que celles écrites dans la boucle de scrutation.

L'interruption (attente passive) : le processeur continue son activité (exécute les instructions en mémoire) jusqu'à l'arrivée d'une interruption. Une interruption est un signal émis par un périphérique qui annonce un changement d'état. Le traitement des interruptions est expliqué dans la section suivante.

I.2. Traitement des interruptions pour le 68000

Déroulement d'une interruption

Lorsque le processeur reçoit une interruption, il interrompt son fonctionnement normal.

Tout d'abord, le contexte d'exécution (le compteur ordinal PC et registre d'état SR) est alors sauvegardé dans la pile. Le pointeur de pile est alors modifié de 6 octets : 4 octets pour l'adresse de l'instruction de retour (PC) et 2 octets pour le registre d'état. L'adresse de retour est empilée en premier.

Ensuite, le registre d'état est modifié en fonction de l'interruption (voir le paragraphe sur les niveaux d'interruption pour plus de détails). Le compteur ordinal est alors mis à l'adresse du sous-programme de traitement d'interruption. Cette adresse est définie par la phase d'initialisation décrite au paragraphe suivant.

Puis le processeur exécute ce sous-programme.

Finalement, le sous-programme de traitement d'interruption est terminé par l'instruction RTE. Cette instruction remet le processeur dans le contexte d'exécution précédent l'interruption, contexte qui a été sauvegardé dans la pile. Ainsi, le registre d'état SR est restauré et l'adresse de l'instruction de retour est mise dans le compteur ordinal PC.

Initialisation d'une interruption

Le sous-programme de traitement d'une interruption est trouvé à partir d'un vecteur d'interruption. Ce vecteur d'interruption est un numéro compris entre 0 et 255 mémorisé dans le périphérique. A ce numéro correspond une adresse en mémoire où doit être placée l'adresse du sous-programme traitant l'interruption.

Certains vecteurs d'interruption sont réservés au fonctionnement du processeur 68000. Il existe cependant des numéros pour des interruptions programmées par les utilisateurs. Ce sont les numéros de 64 à 255 ($40 à $FF). L'emplacement de l'adresse du sous-programme traitant l'interruption est donnée par la multiplication du vecteur d'interruption utilisateur par 4.

Les octets de la mémoire centrale d'adresses $008 à $3FF sont donc réservées aux vecteurs d'interruption. C'est pour cela que l'on utilisait pas des adresses jusqu'à présent.

Par exemple, si le programmeur donne le vecteur d'interruption $46 (%0100 0110) (70 en base dix) à un contrôleur de périphérique, les interruptions levées par ce périphérique seront traitées par le sous-programme dont l'adresse a été placée en $118 (%01 0001 1000). Le sous-programme peut être n'importe où en mémoire, simplement son adresse doit être écrite en mémoire centrale des adresses $118 à $11A. Cette opération est réalisée par les instructions suivantes :

```
Numero_IT  EQU    70          * définition du numéro d'interruption
Adresse_IT EQU    numero_IT*4 * définition de l'adresse où il faut écrire
                              * l'adresse du sous-programme
* mise en mémoire à l'adresse Adresse_IT de l'adresse du sous-programme
* traitant l'interruption. <étiquette_du_sous_programme> est l'étiquette
* définissant ce sous-programme.
        MOVE.L    #<étiquette_du_sous_programme>, adresse_IT
```

Niveau d'interruption

L'unité centrale n'accepte que certains niveaux d'interruption en fixant un masque d'interruption. Pour le 68000, ce masque est défini par les bits 10, 9 et 8 du registre d'état SR. Ces trois bits s'interprètent en un nombre (compris entre 0 et 7). Le bit 11 de SR n'étant pas utilisé, ce niveau d'interruption correspond au deuxième chiffre hexadécimal de SR.

Plus le niveau de l'interruption est grand, plus l'interruption est prioritaire. Mais son niveau doit cependant être supérieur à celui du travail en cours par le processeur.

Par exemple, si SR vaut $2500, alors le processeur n'accepte que les interruptions de niveau 6 ou 7. Et une interruption de niveau 6 peut interrompre une interruption de niveau inférieur (comme 4), mais l'inverse n'est pas possible.

Pour cela, chaque interruption a un niveau d'interruption NI défini matériellement par le périphérique. Si le niveau d'interruption NI est inférieur au masque d'interruption, alors l'interruption n'est pas prise en compte. En revanche, si NI est supérieur ou égal au masque d'interruption, l'unité centrale s'interrompt pour traiter l'interruption.

Lorsqu'une interruption est levée et qu'elle est acceptée par l'unité centrale, le registre d'état SR est empilé. Cette modification consiste à mettre les bits 10, 9 et 8 du registre d'état SR au niveau d'interruption.

Par exemple, si SR vaut $2400 avant qu'une interruption de niveau 6 ne soit levée, alors SR prendra la valeur $2600 au moment du traitement de l'interruption. A la fin du sous-programme (instruction RTE), SR sera restauré à la valeur $2400.

Notons qu'à l'initialisation, `bscv` positionne le travail du processeur au niveau 7. Pour rendre le processeur interruptible, il faudra donc modifier de manière adaptée SR.

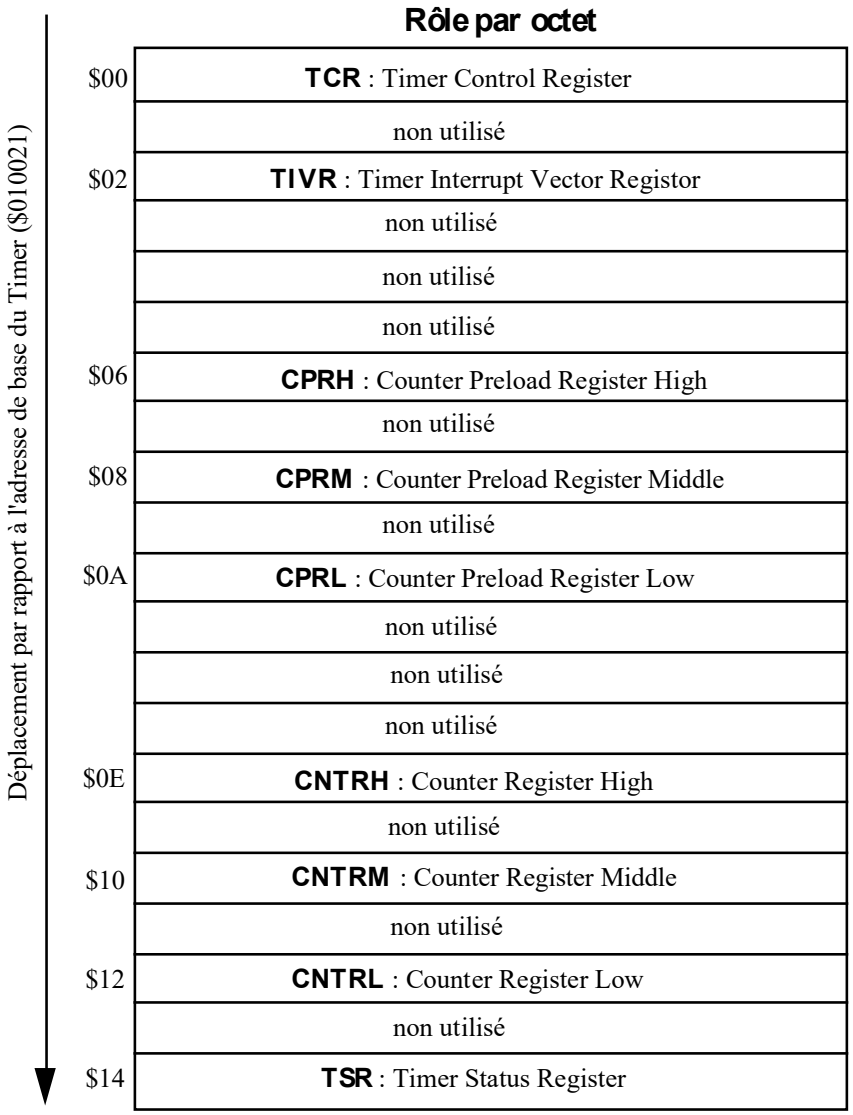
J - Périphérique “Timer“

J.1. Présentation générale

Le Timer est un dispositif, qui sera positionné à l’adresse $010021 à l’initialisation sous bsvc. Ce périphérique permet notamment de mesurer une durée ou de fonctionner sous forme de compte à rebours.

La programmation en assembleur se fait par affectation de valeurs dans des registres de ce dispositif. Le Timer simulé par `bsvc` est une sous-partie du contrôleur de périphérique de Motorola M68230 qui gère une interface parallèle.

L’organisation du Timer est décrite ci-après : le rôle est donné par octet utilisé.



Organisation des registres du Timer

J.2. Détails des registres du Timer

L'accès à un registre se fait à travers l'adresse de base du Timer ($10021) plus l'adresse du registre auquel il faut accéder. Par exemple, l'adresse de TSR est $10021+$14=$10035.

La programmation du Timer en assembleur va se faire par des écritures et des lectures de données (des octets ou des bits) dans les registres du Timer. Pour cela, le mode d'adressage le plus pratique est l'adressage indirect avec déplacement *via* un registre d'adresse.

Les différents registres (implémentés pour `bsvc`) sont les suivants :

- TCR : Timer Control Register.

C'est le registre qui permet de définir le fonctionnement du Timer :
Déclaration du mode (bits TCR_7-TCR_1): Il existe 6 modes d'utilisation. Nous n'utiliserons que le mode "horloge temps-réel" (Real-time clock). Ce mode se traduit par la valeur 1010000 pour les 7 bits TCR_7-TCR_1.
Lancement/arrêt (bit TCR_0 = 1 : lancement, bit TCR_0 = 0 : arrêt).

- TIVR : Timer Interrupt Vector Register.

Ce registre est utilisé lorsque le Timer fonctionne en mode interruption. Il contient le numéro du vecteur d'interruption (valeur entre 64 et 255) qui permet d'associer à cette interruption l'adresse de la routine (programme de traitement de l'interruption) correspondante : l'adresse de la routine d'interruption doit être placée en mémoire à l'adresse définie par ce numéro*4.
On notera que le niveau d'interruption du Timer est défini par hardware à 5.

- CPR : Counter Preload Register.

Il définit la valeur initiale du Timer sur 3 octets d'adresses non consécutives ($6, $8 et $A).

- CNTR : Counter Register.

Ce registre contient la valeur temporaire du Timer (registre de décrémentation). Il est initialisé par le Timer au moment de son lancement (quand TCR_0 passe à 1) par la valeur mémorisée dans CPR, et est décrémenté à chaque tick d'horloge de la valeur $FA (250 en base dix). Lecture seule.

- TSR : Timer Status Register.

C'est le registre d'état du Timer. Le bit TSR_0 passe à 1 lorsque le registre CNTR passe à zéro. Ce bit doit être mis à 0 par programme.

Note : la programmation du Timer en assembleur va se faire par des écritures et des lectures de données (des mots, des octets ou des bits) dans les registres du Timer. Pour cela, le mode d'adressage le plus pratique est l'adressage indirect *via* un registre d'adresse.

K - Table des codes ASCII

K.1. Table des codes ASCII (7 bits significatifs)

Hex	*Car*	*Hex*	*Car*	*Hex*	*Car*	*Hex*	*Car*	*Hex*	*Car*	*Hex*	*Car*	*Hex*	*Car*	*Hex*	*Car*
00	NUL	10	DLE	20	SP	**30**	**0**	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	**31**	**1**	**41**	**A**	**51**	**Q**	**61**	**a**	**71**	**q**
02	STX	12	DC2	22	"	**32**	**2**	**42**	**B**	**52**	**R**	**62**	**b**	**72**	**r**
03	ETX	13	DC3	23	#	**33**	**3**	**43**	**C**	**53**	**S**	**63**	**c**	**73**	**s**
04	EOT	14	DC4	24	$	**34**	**4**	**44**	**D**	**54**	**T**	**64**	**d**	**74**	**t**
05	ENQ	15	NAK	25	%	**35**	**5**	**45**	**E**	**55**	**U**	**65**	**e**	**75**	**u**
06	ACK	16	SYN	26	&	**36**	**6**	**46**	**F**	**56**	**V**	**66**	**f**	**76**	**v**
07	BEL	17	ETB	27	'	**37**	**7**	**47**	**G**	**57**	**W**	**67**	**g**	**77**	**w**
08	BS	18	CAN	28	(	**38**	**8**	**48**	**H**	**58**	**X**	**68**	**h**	**78**	**x**
09	HT	19	EM	29	)	**39**	**9**	**49**	**I**	**59**	**Y**	**69**	**i**	**79**	**y**
0A	LF	1A	SUB	2A	*	3A	:	**4A**	**J**	**5A**	**Z**	**6A**	**j**	**7A**	**z**
0B	VT	1B	ESC	2B	+	3B	;	**4B**	**K**	5B	[	**6B**	**k**	7B	{
0C	FF	1C	FS	2C	,	3C	<	**4C**	**L**	5C	\	**6C**	**l**	7C	|
0D	CR	1D	GS	2D	-	3D	=	**4D**	**M**	5D	]	**6D**	**m**	7D	}
0E	SO	1E	RS	2E	.	3E	>	**4E**	**N**	5E	^	**6E**	**n**	7E	~
0F	SI	1F	US	2F	/	3F	?	**4F**	**O**	5F	_	**6F**	**o**	7F	DEL

K.2. Extension ISO-8859-1 (ISO Latin 1) sur un octet

Hex	*Car*	*Hex*	*Car*	*Hex*	*Car*	*Hex*	*Car*	*Hex*	*Car*	*Hex*	*Car*	*Hex*	*Car*	*Hex*	*Car*
80	NUL	90	DLE	A0		B0	°	C0	À	D0	_	E0	à	F0	_
81	SOH	91	DC1	A1	¡	B1	±	C1	Á	D1	Ñ	E1	á	F1	ñ
82	STX	92	DC2	A2	¢	B2	_	C2	Â	D2	Ò	E2	â	F2	ò
83	ETX	93	DC3	A3	£	B3	_	C3	Ã	D3	Ó	E3	ã	F3	ó
84	EOT	94	DC4	A4	€	B4	´	C4	Ä	D4	Ô	E4	ä	F4	ô
85	ENQ	95	NAK	A5	¥	B5	µ	C5	Å	D5	Õ	E5	å	F5	õ
86	ACK	96	SYN	A6	_	B6	¶	C6	Æ	D6	Ö	E6	æ	F6	ö
87	BEL	97	ETB	A7	§	B7	·	C7	Ç	D7	_	E7	ç	F7	÷
88	BS	98	CAN	A8	¨	B8	¸	C8	È	D8	Ø	E8	è	F8	ø
89	HT	99	EM	A9	©	B9	_	C9	É	D9	Ù	E9	é	F9	ù
8A	LF	9A	SUB	AA	ª	BA	º	CA	Ê	DA	Ú	EA	ê	FA	ú
8B	VT	9B	ESC	AB	“	BB	”	CB	Ë	DB	Û	EB	ë	FB	û
8C	FF	9C	FS	AC	¬	BC	_	CC	Ì	DC	Ü	EC	ì	FC	ü
8D	CR	9D	GS	AD		BD	_	CD	Í	DD	_	ED	í	FD	_
8E	SO	9E	RS	AE	®	BE	_	CE	Î	DE	_	EE	î	FE	_
8F	SI	9F	US	AF	¯	BF	¿	CF	Ï	DF	ß	EF	ï	FF	ÿ

K.3. Notation hexadécimale (rappel)

< >dix	*< >seize*	*< >deux*
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111

< >dix	*< >seize*	*< >deux*
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

K.4. UTF-32 & UNICODE

UTF : Unicode Transformation Format
Sur 32 bits : 8 chiffres hexa : $\$h_7h_6h_5h_4h_3h_2h_1h_0$
avec h_7h_6 = \$00 d'où $\$00h_5h_4h_3h_2h_1h_0$

Ensuite, h_5h_4 définissent les "plans", d'où 65536 cellules (ou caractères) possibles par plan.
17 plans sont utilisés (pour h_5h_4 = \$00 .. \$10).
Les caractères ASCII et Iso-Latin 1 se retrouvent dans le premier plan ((h_5h_4 = \$00), appelé PMB (Plan Multilingue de Base).

Exemple de contenu du PMB :

$h_7h_6h_5h_4h_3h_2$	h_1h_0 = \$00 .. \$FF					
0 0 0 0 0 0		Latin de base		Supplément Latin-1		
0 0 0 0 0 1	Latin étendu A				Latin étendu B	
0 0 0 0 0 2	Latin étendu B		Alph. phon. internat.		Modificateurs	
0 0 0 0 0 3	Signes combinatoires			Grec et copte		
0 0 0 0 0 4	Cyrillique					
0 0 0 0 0 5		Arménien			Hébreu	
0 0 0 0 0 6	Arabe					
0 0 0 0 0 7	Svriaque			Thâna		
0 0 0 0 0 8						
0 0 0 0 0 9	Dévanâgarï			Bengali		
0 0 0 0 2 8	Combinaisons braille					
0 0 0 0 2 9	Supplément B de flèches			Divers symboles math. B		
0 0 0 0 2 A	Opérateurs mathématiques supplémentaires					
0 0 0 0 3 1	Bopomofo	Jamos de compatibilité		Kanbun	Booo.2	*(CJC)4.0?*
0 0 0 0 F F	....					

L - Récapitulatif du jeu d'instructions

Le jeu d'instructions complet du 68000 est récapitulé ci-après par ordre alphabétique.
Les instructions détaillées dans ce document sont précédées de la page où elles sont traitées.
Les instructions ont pour forme en assembleur : `OPER.lg source, destination`
L'instruction réalisée est : `destination.lg := destination.lg OPER source.lg`
Dans ce tableau, l'opérande destination est noté `dest`, l'opérande source est noté `sour`.
`dest` désigne l'emplacement (contenant) et `(dest)` la valeur à cet emplacement (contenu).
Le fonctionnement par ex. d'une soustraction, est décrit par : `(dest) - (sour)` → `dest`.

Pour chaque instruction, chaque indicateur porte une marque :

— : non modifié * : mise à jour u : indéfini
0 : mise à 0 1 : mise à 1

page	Mnémonique	Description	Fonctionnement	X	N	Z	V	C
11	ABCD	Addition décimale étendue	$(Dest)_{dix}$ + $(Sour)_{dix}$ + X → Dest	*	u	*	u	*
10	ADD	Addition binaire	(Dest) + (Sour) → Dest	*	*	*	*	*
10	ADDA	Addition d'adresse	(Dest) + (Sour) → Dest	—	—	—	—	—
10	ADDI	Addition immédiate	(Dest) + don. imm. → Dest	*	*	*	*	*
10	ADDQ	Addition rapide	(Dest) + don. imm. → Dest	*	*	*	*	*
10	ADDX	Addition étendue	(Dest) + (Sour) + (X) → Dest	*	*	*	*	*
11	AND	ET logique	(Dest) ET (Sour) → Dest	—	*	*	0	0
12	ANDI	ET immédiat	(Dest) ET don. imm. → Dest	—	*	*	0	0
.	ASL, ASR	Décalage arithmétique	(Dest) décalée <compte> → Dest	*	*	*	*	*
19	B••	Branchement conditionnel	si cond. alors PC+d → PC	—	—	—	—	—
.	BCHG	Test de bit et changement	*non détaillé ici*	—	—	*	—	—
.	BCLR	Test de bit et mise à zéro	*non détaillé ici*	—	—	*	—	—
19	BRA	Branchement inconditionnel	PC+d → PC	—	—	—	—	—
.	BSET	Test de bit et mise à un	*non détaillé ici*	—	—	*	—	—
.	BSR	Branchement à un sous-pg	PC → -(SP) ; PC+d → PC	—	—	—	—	—
.	BTST	Test de bit	*non détaillé ici*	—	—	*	—	—
.	CHK	Test registre aux limites	*non détaillé ici*	—	*	u	u	u
12	CLR	Mise à zéro de l'opérande	0 → Dest	—	0	1	0	0
19	CMP	Comparaison	(Dest) - (Sour)	—	*	*	*	*
.	CMPA	Comparaison d'adresse	(Dest) - (Sour)	—	*	*	*	*
19	CMPI	Comparaison immédiate	(Dest) - données immédiates	—	*	*	*	*
.	CMPM	Comparaison mémoire	(Dest) - (Sour)	—	*	*	*	*
.	DB••	Test cond., décrémentation et bra.	si ~•• alors Dn-1 → Dn ; si Dn=-1 alors PC + d → PC	—	—	—	—	—
11	DIVS	Division signée	(Dest) / (Sour) → Dest	—	*	*	*	0
11	DIVU	Division non signée	(Dest) / (Sour) → Dest	—	*	*	*	0
12	EOR	OU exclusif logique	(Dest) ⊕ (Sour) → Dest	—	*	*	0	0
12	EORI	OU exclusif immédiat	(Dest) ⊕ don. imm. → Dest	—	*	*	0	0
10	EXG	Echange de registres	Rx ↔ Ry	—	—	—	—	—
11	EXT	Extension de signe	(Dest) signe étendu → Dest	—	*	*	0	0
19	JMP	Saut inconditionnel	Dest → PC	—	—	—	—	—
.	JSR	Saut à un sous-pg	PC → -(SP) ; Dest → PC	—	—	—	—	—
10	LEA	Chargement adresse effective	@Sour → An	—	—	—	—	—

				X	N	Z	V	C
.	LINK	Chaînage de pile	*non détaillé ici*	—	—	—	—	—
12	LSL, LSR	Décalage logique gauche/droite	(Dest) décalée <compte> → Dest	*	*	*	0	*
9	MOVE	Transfert de données	(Sour) → Dest	—	*	*	0	0
.	MOVE → CCR	Transfert vers CCR	(Sour) → CCR	*	*	*	*	*
.	MOVE → SR	Transfert vers registre d'état	(Sour) → SR	*	*	*	*	*
.	MOVE from SR	Transfert du registre d'état	SR → Dest	—	—	—	—	—
.	MOVE USP	Transfert pteur pile utilisateur	USP → An ; An → USP	—	—	—	—	—
10	MOVEA	Transfert vers An	(Sour) → An	—	—	—	—	—
10	MOVEM	Transfert registres multiples	Registres → Dest ou (Sour) → registres	—	—	—	—	—
.	MOVEP	Transfert données périphériques	(Sour) → Dest	—	—	—	—	—
.	MOVEQ	Transfert rapide	Don. imm. → Dest	—	*	*	0	0
11	MULS	Multiplication signée	(Dest) * (Sour) → Dest	—	*	*	0	0
11	MULU	Multiplication non signée	(Dest) * (Sour) → Dest	—	*	*	0	0
.	NBCD	Opposé décimal étendu	0 - $(Dest)_{dix}$ - X → Dest	*	u	*	u	*
11	NEG	Complément à 2	0 - (Dest) → Dest	*	*	*	*	*
.	NEGX	Complément à 2 étendu	0 - (Dest) - X → Dest	*	*	*	*	*
12	NOP	Pas d'opération		—	—	—	—	—
12	NOT	Complément logique	~(Dest) → Dest	—	*	*	0	0
12	OR	OU logique inclusif	(Dest) OU (Sour) → Dest	—	*	*	0	0
12	ORI	OU logique immédiat	(Dest) OU don. imm. → Dest	—	*	*	0	0
.	PEA	Empilement adresse effective	Dest → -(SP)	—	—	—	—	—
.	RESET	Mise à zéro des circuits externes		—	—	—	—	—
12	ROL, ROR	Décalage circulaire non étendu	(Dest) décalée <compte> → Dest	—	*	*	0	*
.	ROXL, ROXR	Décalage circulaire étendu	(Dest) décalée <compte> → Dest	*	*	*	0	*
.	RTE	Retour d'exception	(SP)+ → SR ; (SP)+ → PC	*	*	*	*	*
.	RTR	Retour et restaure CCR	(SP)+ → CC ; (SP)+ → PC	*	*	*	*	*
.	RTS	Retour de sous-programme	(SP)+ → PC	—	—	—	—	—
.	SBCD	Soustraction décimale étendue	$(Dest)_{dix}$ - $(Sour)_{dix}$ - X → Dest	*	u	*	u	*
.	S••	Mise à 1 conditionnelle	si •• vrai alors 1 → Dest sinon 0 → Dest	—	—	—	—	—
.	STOP	Chgt SR et suspension pg	*non détaillé ici*	*	*	*	*	*
11	SUB	Soustraction	(Dest) - (Sour) → Dest	*	*	*	*	*
11	SUBA	Soustraction d'adresse	(Dest) - (Sour) → Dest	—	—	—	—	—
11	SUBI	Soustraction immédiate	(Dest) - don. imm. → Dest	*	*	*	*	*
11	SUBQ	Soustraction rapide	(Dest) - don. imm. → Dest	*	*	*	*	*
11	SUBX	Soustraction étendue	(Dest) - (Sour) - X → Dest	*	*	*	*	*
12	SWAP	Echange des moitiés de registre	R[31..16] ↔ R[15..0]	—	*	*	0	0
12	TAS	Test opér. et indicateur à 1	*non détaillé ici*	—	*	*	0	0
.	TRAP	Trappe	PC → -(SSP) ; SR→ -(SSP) ; (Pointeur) → PC	—	—	—	—	—
.	TRAPV	Trappe sur dépassement	si V alors TRAP	—	—	—	—	—
.	TST	Test d'un opérande	(Dest) testé → cc	—	*	*	0	0
.	UNLK	Rupture du lien	*non détaillé ici*	—	—	—	—	—