ENSIL

Première année Electronique et Télécom

Introduction aux Microprocesseurs

Vahid Meghdadi

2008-2009

Table de Matières

1. RAPPELS

Ce chapitre survole les pré-requis à l'étude de fonctionnement des systèmes à microprocesseur. Ces pré-requis sont principalement les suivants :

- codage
- numération
- circuits combinatoires
- circuits séquentiels

1.1. Codage

1.1.1. Notation positionnelle

La représentation d'un nombre est un codage. La représentation d'un nombre dans une base *b* donnée :

$$a_n a_{n-1} ... a_1 a_0 = \sum_{i=0}^{n} a_i b^i$$

1.1.2. Code binaire naturel

Alphabet de 2 caractères : 0 et 1 (caractères binaires appelés bits). Correspondance basée sur la représentation des nombres en base 2. Par exemple les représentations de 0 à 7 sur 3 bits sont les suivantes :

0 000
1 001
2 010
3 011
4 100
5 101
6 110
7 111

Généralement, les bits les plus à gauche sont appelés poids forts ou MSB en anglais (*most significant bit*).
Les bits les plus à droite sont appelés poids faible ou LSB en anglais (*least significant bit*).

1.1.3. Code *BCD*/DCB

Décimal Codé Binaire : chaque chiffre d'un nombre est codé sur 4 bits

0 0000
1 0001
2 0010

10 0001 0000

11 0001 0001

Ce code simplifie la conversion décimale binaire.

1.1.4. Codage des données alphanumériques

Face aux multiples possibilités de codage, des organismes de normalisation ont vu le jour (le plus célèbre étant l'ISO).
Code ASCII = code employé pour les caractères alphanumériques (a,b,1,?, …). Code sur 7 bits (*American Standard Code for Information Interchange*).

A est codé par 1000001,
e est codé par 1100101,
7 est codé par 0110111,
! est codé par 0100001,
etc...

1.2. Système de numération

L'homme a 10 doigts. Le système de numération humain est le système décimal. L'ordinateur a 2 états significatifs (impulsion électrique). Le système de numération qu'il emploie est donc le système binaire.

1.2.1. Conversion binaire-décimale

Base *b* vers base 10

$a_n a_{n-1} a_1 a_0, f_1 f_2 \ldots$ exprimé en base b vers une représentation en base 10 :

$$a_n b^n + a_{n-1} b^{n-1} + \ldots + a_1 b^1 + a_0 b^0 + f_1 b^{-1} + f_2 b^{-2} + \ldots$$

Base 10 vers base b

$$A_{10} = a_n b^n + a_{n-1} b^{n-1} + \ldots + a_1 b^1 + a_0 =$$
$$((\ldots(a_n b + a_{n-1})b + \ldots)b + a_1)b + a_0$$

a_0 est le reste de la division entière du nombre par la base b. Des divisions entières successives par la base donnent donc tous les a_i.
Cas de la partie fractionnaire : sur un principe similaire, des multiplications successives par la base donnent tous les a_i.

1.2.2. Représentation des nombres négatifs

Signe et valeur absolue

Sur n bits : signe = bit de poids fort (0 : positif, 1 : négatif), n-1 bits = valeur absolue. Intervalle de valeurs représentées pour n bits : $[-2^{n-1} + 1, 2^{n-1} - 1]$

Quelles sont les 2 représentations possibles pour zéro ?

Notations complémentés

Complément à 1: inversion de chaque bit de la valeur absolue du nombre à représenter.

-1 110
-2 101
-3 100

Intervalle de valeurs représentées pour n bits : $[-2^{n-1} + 1, 2^{n-1} - 1]$

Quelles sont les 2 représentations possibles pour zéro ?

Complément à 2 = complément à 1 + 1

-1 111
-2 110
-3 101

Intervalle de valeurs représentées pour n bits : $[-2^{n-1} , 2^{n-1} - 1]$

Combien de représentation possible pour zéro ?

Notation excédentaire

Ajout au nombre de la valeur d'un excès (souvent translation de 2^{n-1}, ainsi le bit de poids fort fait office de bit de signe).
Intervalle de valeurs représentées pour n bits avec un excès de 2^{n-1} : $[-2^{n-1} , 2^{n-1} - 1]$
Intérêt : simplifie toutes les opérations ou les comparaisons (qui se font uniquement sur des nombres positifs).

1.2.3. Représentation des nombres fractionnés en virgule fixe

Dans cette présentation, on fixe la position de la virgule (un entier peut être représentatif d'un nombre fractionnaire si on connaît la place de la virgule). Par exemple, si on nous dit parmi les nombres ci-dessous, nous avons toujours deux digits entiers et 4 digits fractionnaires, les nombres ci-dessous seront interprétés comme présentés :

- 451122 => 45,1122
- 12401 => 012401 => 1,2401
- 2 => 000002 => 00,0002 => 0,0002
- 20000 => 2

En binaire nous avons les mêmes principes. On parle de format. Par exemple le format 2,4 veut dire qu'il y a 2 digits binaires avant la virgule et 4 chiffres après la virgule. Ainsi, nous interprétons les valeurs données ci-dessous en considérant ce format :

011001 => 01,1001 => $0*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 0*2^{-3} + 1*2^{-4}$ => 1,5625
001100 => 0,75

001000 => 0,5
Pour les nombres négatifs, on considère le complément à 2 :
110010 => -(001110) => -(0,875) => -0,875
111000 => -(001000) => -(0,5) => -0,5

Une astuce pour calculer les nombres ci-dessus :
011001, format=2,4 : on considère le nombre comme un entier : 011001=16+8+1=25, on divise le résultat par 2^4 => $25/2^4 = 1,5625$
110010, format 2,4 : d'abord entier : -32+16+2=-14, le résultat divisé par 2^4 => -14/16= -0,875

Pour aller dans l'autre sens :
Exemple 1 : quelle est la représentation en virgule fixe de 1,125 au format 3,5 ?
On multiplie la valeur par 2^5, et on représente en binaire le résultat :
$1,125*2^5$ => 00100100
Exemple 2 : quelle est la représentation en virgule fixe de -3,25 au format 3,6 ?
$3,25*2^6$=208 => 011010000 => -3,25 => 100110000

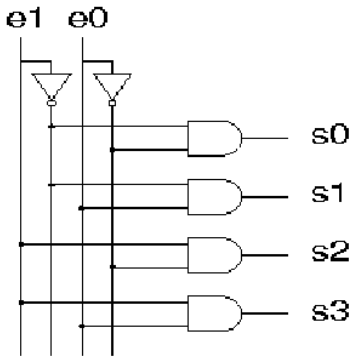
1.3.Logique combinatoire

1.3.1. Décodeur

Circuit permettant d'envoyer un signal à une sortie choisie. Il dispose de n lignes d'entrées et 2^n lignes de sortie. La table de vérité d'un décodeur "2 vers 4" (n = 2) est la suivante :

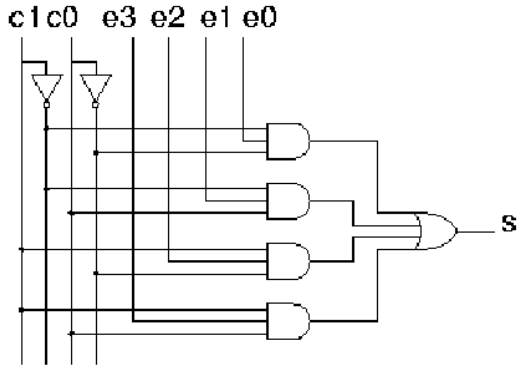
e1	e0	s0	s1	s2	s3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Et voici la réalisation d'un tel décodeur :



1.3.2. Multiplexeur

Circuit permettant de sélectionner une entrée parmi plusieurs. Il dispose de 2^n entrées, 1 sortie et n lignes de sélection.
Voici la réalisation d'un multiplexeur "4 voies" (n = 2) :



1.4.Logique séquentielle

Appelés aussi circuits de mémorisation car leurs sorties dépendent de l'état des variables d'entrée et de l'état antérieur de certaines variables de sortie.
Un circuit séquentiel possède des entrées E, des sorties S et un état interne Q. Il est défini par deux fonctions S = f(E,Q) et Q'=g(E,Q)indiquant respectivement la nouvelle sortie et le nouvel état.

1.4.1. Bascule RS

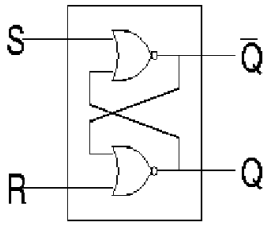
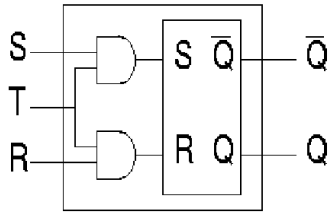


Table de vérité (Q_i représente la sortie Q à l'instant i, x un état quelconque) :

R	S	Q_i	Q_{i+1}
0	0	x	x
0	1	x	1
1	0	x	0
1	1	x	interdit

1.4.2. Bascule RST

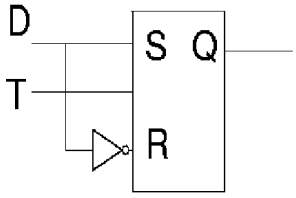
L'horloge peut être vue comme la synchronisation de l'odre d'apparition des variables logiques.



si T = 1, fonctionnement comme une bascule RS
si T = 0, conservation de l'état courant

1.4.3. Bascule D

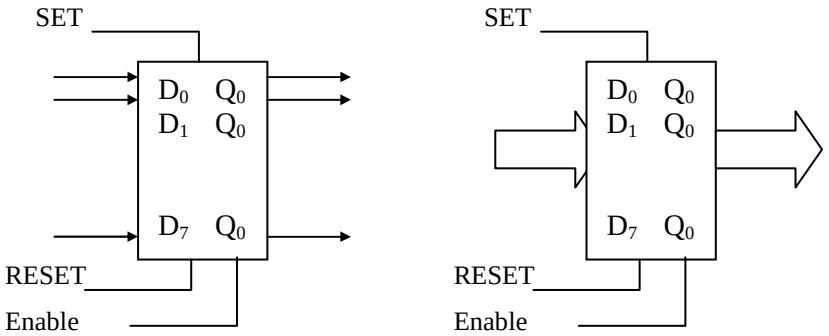
Intérêt : pas d'état instable; mémorisation de l'entrée. La sortie est égale à l'entrée tant que le signal de contrôle est à 1. Dès que le signal de contrôle passe à zéro, la sortie est verrouillée indépendant du D.



La bascule D est utilisée pour la conception des mémoires.

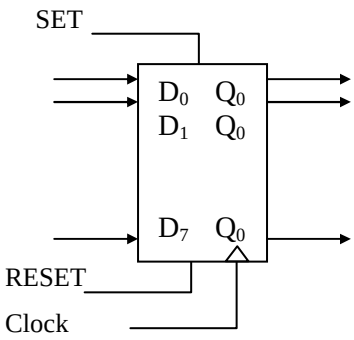
1.4.4. Verrou (latch)

Celui-ci est équivalent à 8 bascules D en parallèle. On peut y ajouter aussi deux entrées pour la mise à zéro ou à 1 de la sortie, comme présenté ci-dessous :



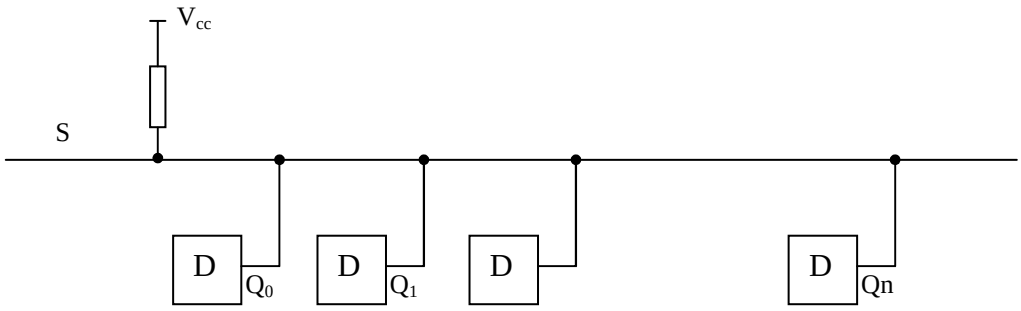
1.4.5. Registres

Quand les sorties prennent les valeurs d'entrée uniquement sur un front montant (ou descendant) du signal de contrôle, le verrou s'appellera registre. Ceci est schématisé par un petit triangle sur le signal de contrôle :



1.4.6. Sortie à 3 états

La sortie d'une bascule peut être un des trois états possibles, 0 volt, 5 volts ou haute impédance. Dans beaucoup de cas, la tension 5 volts est remplacée par l'état haute impédance. Pour mettre la sortie en état haute impédance, il faut une entrée supplémentaire souvent appelée CS (*Chip Select*) ou OE (*Output Enable*). Ceci permet de connecter plusieurs sorties sans qu'il y ait une casse de circuit. Exemple ci-dessous montre une application pour cette possibilité.



La tension de ligne S est à 1 si les sorties de toutes les bascules sont en haute impédance. Si au moins, une des sorties passe à zéro volt, la tension de cette ligne passe à zéro. Avec ce circuit, quoi qu'il arrive, les bascules ne vont pas griller ! Ce câblage est aussi appelé "et câblé" car il se comporte comme une porte "et" logique.

1.4.7. Mémoire

Une mémoire est un ensemble de registres précédé par un décodeur qui permet de sélectionner un des registres. On distingue deux types de mémoire : RAM (mémoire vive) (*Random Acces Memory*) et ROM (mémoire morte) (*Read Only Memory*).
Les ROM se programme une fois et leur contenu n'est pas modifié dans le circuit tandis que les RAM sont fait pour sauvegarder les données au moment de fonctionnement normal du circuit.

Le choix d'écriture ou lecture se fait par une entrée souvent appelée $R/\overline{W}$. Si le circuit qui se sert de la RAM veut écrire dans la RAM, il met la tension zéro sur cette broche et pour écrire 5 volts (1 logique).

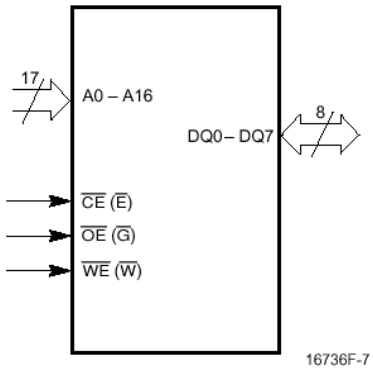
De l'autre côté, il faut aussi des lignes d'adresse qui sont utilisées pour sélectionner un des registres. Il y souvent une autre entrée (CS) qui sert à sélectionner le boîtier : si cette entrée n'est pas validée, le boîtier ne réagit pas. Les mémoires sont souvent synchrones car ce sont un ensemble de registre. On parle des mémoires asynchrones quand un ensemble de verrous se trouve à l'intérieur de mémoire.

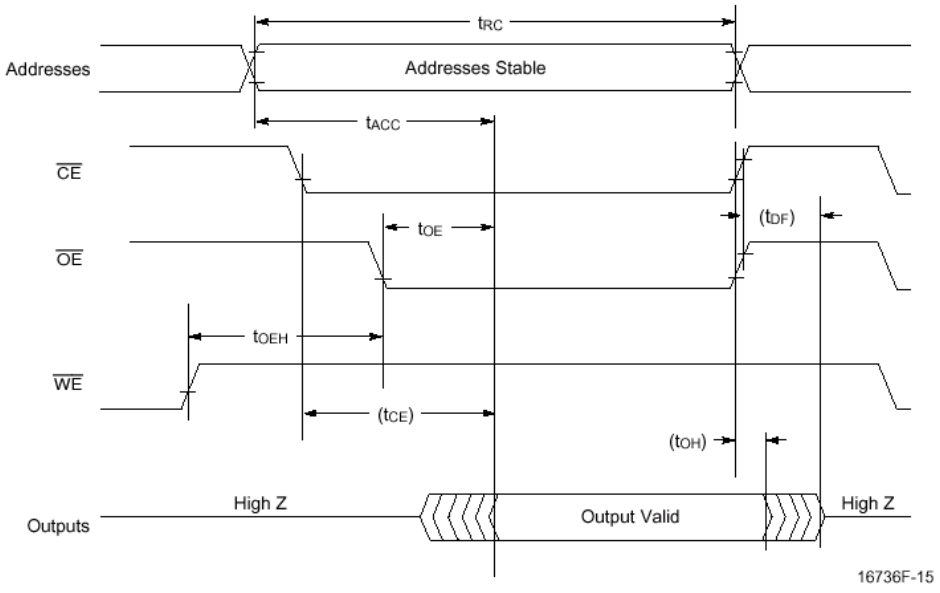
Ci-dessous, un exemple d'un EPROM commercialisé (Am29F010, 128 MO) est présenté.

PIN CONFIGURATION

A0–A16	=	17 Addresses
$\overline{CE}$	=	Chip Enable
DQ0–DQ7	=	8 Data Inputs/Outputs
NC	=	Pin Not Connected Internally
$\overline{OE}$	=	Output Enable
V_{CC}	=	+5.0 Volt Single-Power Supply (±10% for –55, –70, –90, –120) or (±5% for –45)
V_{SS}	=	Device Ground
$\overline{WE}$	=	Write Enable

LOGIC SYMBOL





Chronogramme pour l'opération de lecture

1.5.Exercices

1- Conversion nombre entier signé vers binaire complément à deux. Pour les valeurs signées ci-dessous donner si c'est possible la représentation en C'2 sur 8 bits.
100, -18, 128, -128, 18

2- Quel est équivalent décimal des valeurs suivantes :
   $(0111\ 0000)_2$ c'est le format binaire signé C'2
   $(0111\ 0000)_2$ c'est le format binaire non signé
   $(1111\ 0000)_2$ c'est le format binaire signé C'2
   $(1111\ 0000)_2$ c'est le format binaire non signé
   $(0111\ 0000)_2$ c'est le format (1,7) signé C'2
   $(0111\ 0000)_2$ c'est le format (1,7) non signé
   $(1100)_2$ c'est le format (1,3) signé C'2
   $(1111\ 1100)_2$ c'est le format (1,7) signé C'2
   $(0000\ 1100)_2$ c'est le format (1,7) signé C'2

3- Pour les valeurs décimales ci-dessous donner la représentation binaire à virgule fixe au format (2,7) (arrondir là où il faut).
   0.25, -0.25, 0.27, -0.27, 1.55, -2, -1, 1, 1.375, -1.375.
   Quelle sont les valeurs maximale et minimal ?

4- Les valeurs suivantes sont-elles équivalentes ?
   1110 signé format 1.3
   11 1110 signé format 1.5
   11 1110 signé format 3.3

5- Effectuer l'addition en base 2 et vérifier si le résultat est correct (dans un premier temps prenez les valeurs signées C'2 et ensuit refaites le même exercice pour les valeurs non-signée).
   1100 + 1010
   0100 + 0010
   1010 + 0110

2. CONCEPTION D'UN MICROPROCESSEUR

2.1. Introduction

On entend par un processeur, un dispositif capable d'exécuter un certain nombre d'instructions une par une.

- Il nous faut donc un endroit pour stocker les instructions.
- Il faut pouvoir comprendre et exécuter les instructions.
- Il faut pouvoir stocker des données dans un endroit.

Puisque nous ne travaillons qu'avec des circuits électroniques, chaque instruction peut être codée en tension électrique. Pour simplifier des choses, on prend uniquement deux niveaux de tension. Ainsi chaque instruction est codée sur une combinaison de ces deux niveaux de tension (bit). Par exemple l'opération mathématique d'addition peut être codée sur "00100110" (pourquoi pas !). Par convention, le niveau haut de tension se présente par un "un" et le niveau bas par un "zéro".

Alors, une succession d'instructions n'est qu'une suite des données binaires sur n bits. Ensuite, il faudrait décoder l'instruction courante et l'exécuter, puis passer à l'instruction suivante. Nous allons détailler chaque tâche en essayant d'inventer l'architecture nécessaire pour l'accomplir.

2.2. Architecture

2.2.1. Bus d'adresses et de données

Dans un premier temps il faudrait dimensionner le système. Pour la taille de la mémoire, on prend assez large. Par exemple on peut avoir une mémoire de taille 8 millions de cases. Alors pour pouvoir avoir accès à toutes ces cases, il nous faut 23 bits (pourquoi ?). C'est-à-dire que la mémoire comporte 23 lignes d'entrée qui servent à pointer une case mémoire parmi les 8 millions disponibles. C'est 23 lignes d'adresse s'appelle le bus d'adresse puisqu'ils se servent d'indiquer une case particulière : ils donnent son adresse.

Ensuite, il faut penser à la taille de chaque case. Imaginez que l'on prend des cases à 16 bits (ou un mot). Ainsi, la mémoire peut être considérée comme un tableau de 8 millions lignes fois 16 colonnes. Chaque élément de ce tableau est un bit.

Avec ces informations, on peut déjà dimensionner le nombre de fils (broches) nécessaire de la mémoire. Nous pouvons donc prévoir le circuit et le câblage donné sur la Figure 1.

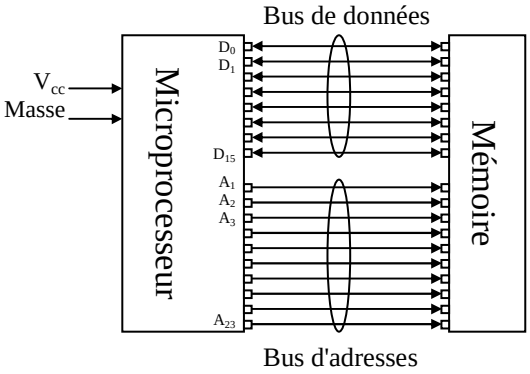


Figure 1- Câblage du µP et le mémoire

Remarquer les directions de bus d'adresses qui sont toujours du processeur vers la mémoire car c'est toujours le processeur qui décide quelle case mémoire doit être lue ou écrite. Ce qui n'est pas le cas pour le bus de données car les données peuvent circuler du processeur vers la mémoire ou à l'inverse.

Afin d'éviter un court circuit possible 0 volt 5 volts, quand les données sont mises sur le bus de données par le processeur, la mémoire applique une impédance importante sur ces broches. Par contre dans le cas contraire, c'est le processeur qui va mettre une impédance importante sur ces lignes. Ce qui signifie que les registres internes du processeur et de la mémoire sont des registres à trois états.

2.2.2. Bus de contrôle

Pour autoriser la mémoire à répondre ou pas, on prévoit une ligne supplémentaire car si plusieurs mémoires sont connectées au µP, il faut qu'un seul boîtier réponde. Sinon, on risque de griller la mémoire. On appelle cette ligne CS (*Chip select*) ou sélecteur du boîtier. Imaginons le cas d'utilisation des mémoires moins grandes, 2 mega mots par exemple. Pour obtenir notre mémoire de 8 mega mots, nous pouvons imaginer le circuit donné sur la Figure 2.

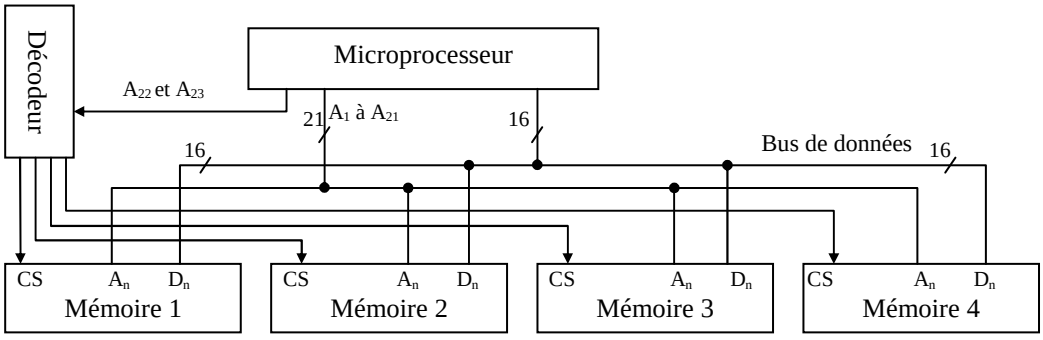


Figure 2- Utilisation de CS pour sélectionner les différents boîtiers

De plus, il faut que la mémoire sache c'est une écriture ou lecture. Ceci est indiqué par une broche (fils) supplémentaire qui s'appelle $R/\overline{W}$. Si la tension de cette broche est à 1 (5 volts),

la mémoire va mettre le contenu de la case mémoire pointée par le bus d'adresse sur le bus de données. Si par contre, la tension est de '0' logique (0 volts), la mémoire se met en haute impédance et va lire les données présentes sur le bus de données. Dans tous les cas, la mémoire non sélectionnée met en haute impédance ces broches de bus de données. Ainsi, ils n'interviendront pas sur les états de bus.

2.2.3. Horloge

Dans un système synchrone comme un microprocesseur, tout se synchronise avec une horloge. C'est à dire que les tâches sont exécutées à des moments réguliers cadencés par une horloge. Il nous faut donc une entrée supplémentaire pour appliquer cette horloge. Plus cette horloge est rapide, plus le µP exécute les instructions rapidement. Normalement les tâches commencent sur le front montant de l'horloge et se termine avant le front montant de la période suivante.

2.2.4. Unité arithmétique et logique

Il faut maintenant penser aux opérations à effectuées. On peut prévoir des opérations :
- logiques comme "ou", "et", "ou exclusif"…
- des opérations arithmétiques comme addition, soustraction et négation
- des opérations diverses comme décalage à gauche et à droite, complémenter à 1, …

Cette unité couramment appelée ALU (*Arithmetic logic Unit*) est donc à réaliser avec des circuits électroniques (Figure 3). Maintenant, il faut penser à ce que les opérandes soient véhiculés en entrée de cette unité. On ne peut pas faire des opérations sur les deux données qui se trouvent en mémoire car nous n'avons qu'un seul bus de données et d'adresse. Il faut donc récupérer la première donnée, la sauvegarder dans un registre, et puis effectuer l'opération avec l'autre opérande qui se trouve éventuellement dans la mémoire. Il nous faut donc des registres en interne du µP.

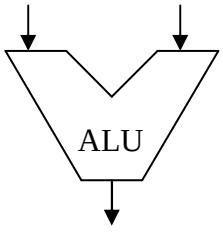


Figure 3- Schéma de l'ALU

Le résultat aussi peut être écrit dans la mémoire ou dans un registre. Il faut donc prévoir un registre pour sauvegarder la sortie de l'ALU.

2.2.5. Registres internes

Les registres internes permettent un accès rapide aux données au sein du processeur sans aller les chercher dans la mémoire. Par contre, il faut bien réfléchir sur le nombre et la connexion de ces registres au reste du µP. Bien sûr, plus ils sont nombreux, plus la tâche de la programmation se facilite et plus la vitesse d'exécution augmente. Choix de l'architecture à

mettre en place est un point critique pour la conception d'un µp et en même temps pour la compréhension d'un µP déjà commercialisé. Imaginons que nous prévoyons 8 registres que nous appelons D0 à D7.

Imaginons l'architecture présentée sur la Figure 4.

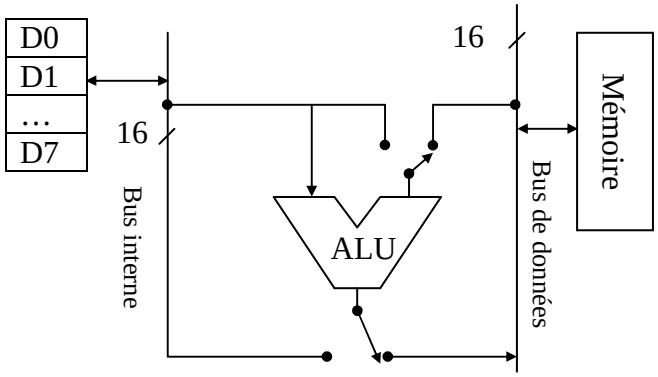


Figure 4- Architecture proposée du µP

Nous avons choisi d'avoir 16 registres de données de taille 16 bits pour accueillir les données. Remarques sur l'architecture :

- Les registres de données ne sont pas connectés directement au bus de donnée mais à un bus interne.
- Une des entrées de l'ALU est forcement une donnée qui provient des registres tandis que l'autre opérande peut venir de la mémoire ou un des registres internes. Par conséquent, une opération sur les deux registres (D0+D1 par exemple) est possible. Le résultat peut être sauvegardé dans la mémoire ou dans un registre interne.
- Une opération sur deux données de mémoire n'est pas possible.
- La sortie de l'ALU peut être sauvegardée soit dans un registre, soit en mémoire.

2.2.6. Génération d'adresse

Pour pouvoir adresser la mémoire, il nous faut mettre l'adresse voulue sur le bus d'adresse. Sachant que nous nous sommes fixés sur 8 méga mots de mémoire, la taille du bus d'adresse est de 23 bits. Nous supposons que l'adresse est dans un registre spécial (registre d'adresse) qui est connecté au bus d'adresse. Cependant, on ne met pas un nom particulier sur ce registre. A chaque fois que l'on effectue un accès mémoire, ce registre est chargé par l'adresse de l'endroit où la donnée doit être lue ou écrite.

Nous laissons de côté pour le moment l'architecture de notre µP mais nous y reviendrons une fois que nous rencontrons des nouveaux besoins.

2.3. Syntaxe d'instructions

Nous allons maintenant arrêter sur le jeu d'instructions. La syntaxe d'une instruction est à inventer. On va opter la syntaxe suivante :

Pour des instructions nécessitant deux opérandes :

opération opérande2, opérande 1

Dans ce cas, l'opérande 1 indique aussi la destination. Par exemple "ADD D1,47", signifie que le registre D1 sera additionné au contenu de la case mémoire 47 et le résultat sera sauvegardé dans la case mémoire 47. Par contre "ADD 47,D1" sauvegarde le résultat dans le registre D1.

Pour des instructions nécessitant un seul opérande la syntaxe est la suivante :

opération opérande

Dans ce cas, l'opérande indique la source aussi bien que la destination. Par exemple "NEG D6" signifie que l'inverse de D6 est sauvegardé (chargé) dans D6.

2.4. Jeu d'instructions

On peut considérer un jeu d'instruction comportant les opérations suivantes :
ADD (pour addition), SUB (pour soustraction), NEG (pour complémenter à deux), COM (pour complémenter à 1), MOVE (pour le transfert de donnée), AND (pour effectuer un "et" logique bit à bit), OR (pour effectuer un "ou" logique bit à bit), XOR (pour effectuer un "ou exclusif" bit à bit).

Comme nous allons voir dans les paragraphes suivants, ce jeu d'instructions n'est pas suffisant dans la plupart des cas. Nous y ajoutons quelques-unes par la suite.

2.5. Codes opératoires (op-code)

Les instructions à exécuter sont stockées dans la mémoire. Or la mémoire n'est qu'un tableau de taille 8méga mots. Il faut donc transformer (coder) chaque instruction en un nombre pouvant être codé sur 16 bits. L'attribution d'un code particulier à une instruction a des conséquences sur le circuit chargé de décoder cette instruction.
Imaginer par exemple une instruction de déplacement : MOVE. On peut déplacer une donnée (un mot) d'un registre vers un autre, de la mémoire vers un registre ou d'un registre vers la mémoire. Par exemple « MOVE D1, D5 » va faire copier le contenu de D1 dans D5. Il faut maintenant donner un code de 16 bits pour représenter cette opération. Prenons la convention suivante pour l'instruction MOVE pour construire le code opératoire de cette instruction :

Copier le registre R_S dans R_D.

15 12	11-9	8-6	5-3	2-0
0011	R_D	000	000	Rs

Exemple : MOVE D3,D5 : le code opératoire=00.11.101.000.000.011=$3A03

Copier le registre R_S dans la mémoire.

15 12	11-9	8-6	5-3	2-0
0011	001	111	000	Rs

Exemple : MOVE D2,$12A31F. Cette instruction va copier le contenu du registre D2 à l'adresse $12A31F de la mémoire. Le code opératoire contiendra donc 3 mots, le premier est le code opératoire et les deux suivant indiquent l'adresse de la case :
Le code : (0011 001 111 000 010) puis (0000 0000 0001 0010) puis (1010 0011 0001 1111). En hexadécimal il sera : $33C2, $0012, $A31F. Cette instruction prendra alors trois mots.

Copier la mémoire dans le registre R_D.

15 12	11-9	8-6	5-3	2-0
0011	R_D	000	111	001

Exemple : « MOVE $12A31F, D7 ». Cette instruction va copier le contenu de la mémoire à l'adresse $12A31F dans le registre D7. Le code opératoire contiendra donc 3 mots, le premier est le code opératoire et les deux suivant indiquent l'adresse de la case :
Le code : (0011 111 000 111 001) puis (0000 0000 0001 0010) puis (1010 0011 0001 1111). En hexadécimal il sera : $3E39, $0012, $A31F. Cette instruction prendra alors trois mots.

Ainsi, les quatre MSB vont sélectionner le circuit correspondant à un déplacement et les bits qui les suivent indiquent la source et la destination.

Il se peut que l'on veuille charger un registre par une valeur immédiate, disons $CAFE. Il faut donc prévoir l'instruction suivante : MOVE #$CAFE,D1. Le signe # est utilisé pour dire que $CAFE n'est pas une adresse de la mémoire mais une donnée immédiate. Le code opératoire que l'on peut inventer peut être la suivante :

Copier une donnée immédiate dans le registre R_D.

15 12	11-9	8-6	5-3	2-0
0011	R_D	000	111	100

Bien entendu le mot suivant cette instruction doit être la donnée : $CAFE par exemple. Le code opératoire contiendra donc deux mots : $323C, $CAFE

Il faut donc prévoir un code pour chaque instruction possible. Normalement dans les microprocesseurs commercialisés, le code opératoire se divise en champs. Les champs indiquent la nature de l'opération, la source et la destination. Suivant l'instruction, on peut avoir des mots supplémentaires qui fourniront les adresses nécessaires pour compléter l'instruction comme nous l'avons vu ci-dessus.

2.6.Exécution de programme

Il faut maintenant imaginer un mécanisme pour faire exécuter les commandes une par une. Imaginons que les codes opératoires sont mis au préalable dans la mémoire dès l'adresse zéro. Au moment de lancement du µP (la mise sous tension), celui-ci commence donc exécuter la commande se trouvant à l'adresse zéro. Il faut donc qu'il mette l'adresse zéro sur le bus d'adresse et qu'il récupère la valeur mise sur le bus de donnée par la mémoire. Puis, il exécute cette commande. Il faut maintenant qu'il cherche le code opératoire suivant. L'opération de chercher un code opératoire s'appelle "*Fetch*". C'est-à-dire que le µP met l'adresse du code suivant sur le bus d'adresse pour récupérer aussi tôt le code opératoire suivant. Cette adresse n'est pas forcement 1, voir paragraphe précédent pour comprendre qu'une instruction peut être codée sur plusieurs mots.

Il faut donc traquer les codes opératoires. Cette tâche se fait en utilisant un registre de taille 23 bits s'appelant PC (Compteur de Programme) (nous utilisons un registre de taille 32 dont les 9 bits MSB ne sont pas utilisés : toujours à zéro). Au moment de la mise sous tension du µp, ce registre est initialisé à 0. Dans le cycle de *fetch*, le contenu de ce registre est mis sur le bus d'adresse pour récupérer le code opératoire de l'instruction courante. Suivant la nature de l'instruction, ce registre est incrémenté de 1, 2 ou 3 pour pointer sur l'instruction suivante. Ainsi, le PC pointe toujours sur la prochaine instruction à exécuter. La Figure 5 présente l'architecture modifiée de notre µP.

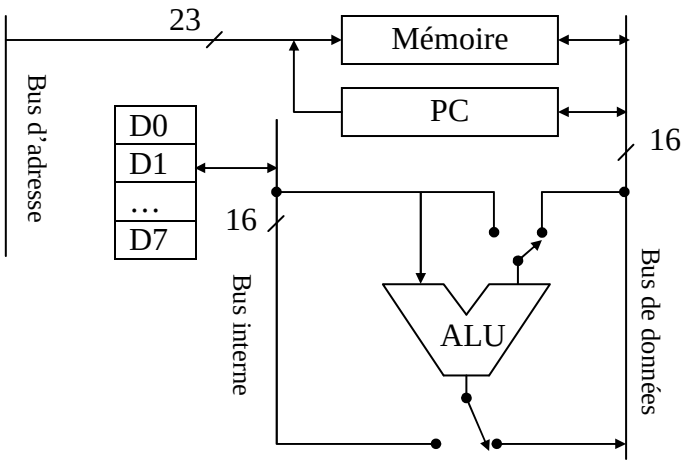


Figure 5- Ajout du compteur de programme à l'architecture du µP

2.7.Branchement

Normalement les instructions sont exécutées une par une. Il est cependant possible que l'on veuille sauter à une autre instruction qui se trouve plus haut ou plus bas dans la mémoire. On peut donc prévoir la commande BRA (pour *branch*) pour sauter à quelques lignes plus hautes ou plus basses. Dans ce cas, il suffit d'indiquer dans le code opératoire, la valeur de décalage (positif ou négatif) par rapport au PC courant. Le montant de ce décalage est à récupérer à l'adresse suivant du code opératoire. Cette valeur s'additionne donc au compteur de programme pour pointer à la prochaine instruction à exécuter. On peut deviner que pour des sauts longs cette instruction a besoin de deux octets : 1 pour le type d'opération (branchement dans ce cas là) et l'autre pour indiquer le montant de décalage. Par contre si le montant de décalage est limité (8 bits) on peut envisager de l'intégrer dans le code. Dans ce cas un seul mot suffit pour coder l'instruction et le branchement est limité à 127 mots plus bas ou 128 mots plus hauts car le décalage est codé sur 8 bits en complément à 2. Remarquez que ce saut et relatif, c-à-d que la référence est la valeur courante du PC.

Pour obtenir des décalages plus importants, on peut envisager un deuxième type d'instruction qui donne un décalage sur 23 bits qui va être donc codée sur 2 mots.

Pour le moment on se contente d'un décalage limité sur 8 bits. La syntaxe de l'instruction est la suivante :

BRA décalage => PC ← PC + offset

On image le code opératoire de cette instruction égale à [0110 0000 (8 bit de décalage)].
Exemple : BRA -2 avec le op-code $60FE, est une boucle infinie car le PC tombe toujours sur la même ligne
Exemple : BRA 0 n'a aucun effet car c'est toujours l'instruction suivante qui sera exécutée

Remarque : Une partie de l'architecture qui s'occupe de l'addition du décalage avec le PC n'est pas présentée sur la Figure 5.

2.8.Indicateurs (drapeaux ou *flags*)

Dans un langage de programmation haut niveau nous avons des instructions de type "*if*". On souhaite réaliser des branchements conditionnels. Nous essayons de modifier notre architecture afin de permettre ce genre d'opération. Imaginer la commande ci-dessous dans un langage évolué :

IF A>B GOTO lable

On remarque une comparaison suivie par un branchement. Pour réaliser cela en langage bas niveau (langage du µp, langage de machine ou encore langage en assembleur), on va prévoir la commande "CMP x,y". Cette commande va exécuter l'opération y-x et suivant le résultat, quelques indicateurs vont être positionnés. Un indicateur est un bit qui indique un état particulier du µp, d'une opération, etc.
Pour un résultat zéro, on prévoit un indicateur (un drapeau ou encor un *flag*) que l'on appèle Z. Si le résultat est zéro, cet indicateur passe à un sinon à zéro. Ce drapeau est positionné à chaque fois qu'une opération arithmétique s'exécute.

On envisage un autre drapeau pour indiquer si le résultat est négatif. Ce drapeau s'appelant N prend la valeur du bit MSB du résultat d'une opération arithmétique.

Un autre drapeau qui s'avère très important est le drapeau de dépassement. Supposons que l'on travaille sur 8 bits, et que l'on voudrait additionner 120 et 20. 120 en binaire est 0111 1000 et 20 en binaire 0001 0100. La somme sera donc 140 ou en binaire 1000 1100. Puisque nous optons la représentation en complément à deux, celui-ci n'est que -(0111 0100) = -116. C'est à dire que la somme de deux nombres positifs devient négative. Dans ce cas, on prévoit un drapeau V pour indiquer le dépassement (*oVerflow*). Ainsi, le programmeur, en testant ce bit, peut se rendre compte que le résultat fourni par ALU est faux.

Remarque : Ces drapeaux sont positionnés à chaque fois qu'une donnée passe à travers l'ALU : toutes les opérations de MOVE, ADD, NEG, AND, OR, XOR ,DEC, NEG, CMP.

On modifie l'architecture de notre processeur pour tenir compte des drapeaux voir Figure 6)

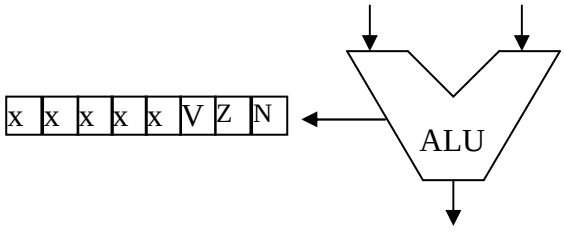


Figure 6- Insertion des drapeaux

Le registre qui contient les drapeaux s'appelle le registre de code condition CCR.

2.9.Branchement conditionnel

La condition est uniquement sur les indicateurs. On présente trois branchements conditionnels suivants :

2.9.1. Branchement suivant Z

- « BEQ décalage » : Le branchement s'effectue si le drapeau Z est à un. Cela signifie que le dernier résultat de calcul dans ALU était zéro.
- « BNE décalage » : Le branchement s'effectue si le drapeau Z est à zéro. Cela signifie que le dernier résultat de calcul dans ALU était différent de zéro

```
Exemple :   MOVE    1001,D1     2 mots
            CMP     #500,D1     2 mots
            BEQ     -5          1 mot
            NEG     B
```

Le résultat est que si le contenu de D1 (qui est le contenu de la case mémoire 1001) est égal à 500, le programme reboucle à l'instruction "MOVE", sinon il continue en exécutant la ligne "NEG".

2.9.2. Branchement suivant V

« BRV décalage » : Le branchement s'effectue si le drapeau V est à un. Cela signifie que le dernier résultat de calcul dans ALU représenté un dépassement et que le résultat était faux.

```
Exemple :   ADD   D7,47
            BRV   FAUX
```

FAUX est une étiquette indiquant une adresse dans la mémoire.

2.9.3. Branchement suivant N

« BMI décalage » : le branchement s'effectue si le drapeau N est à un. Cela signifie que le dernier résultat de calcul dans ALU était négatif. Si on utilise cette instruction toute suite après une comparaison "CMP x,y", si y<x le drapeau N se met à 1 car y-x devient négatif.
Remarque : C'est une comparaison non signée. C'est à dire que le contenu de chaque registre varie de 0 à 2^{16}-1 et non pas de -2^{15} à 2^{15}-1. Ainsi $FFFF > $0000.

2.10. Registre d'index

Il se peut que nous voulions adresser la mémoire de manière indirecte. On peut donc prévoir des registres de taille 23 bits que l'on appelle A0-A7 pour adresser la mémoire (dans la pratique ce sont des registres de taille 32 bits). Il faut donc inventer des instructions adéquates pour charger ou utiliser ce registre. Par exemple pour charger dans D0 la case mémoire adressée par A1, on peut utiliser l'instruction suivante.

```
MOVE   (A1), D0
```

Mettre des parenthèses autour de A1 signifie que l'adresse de donnée à charger est fournie par A1. Autrement dit, ce n'est pas le contenu de A1 qui sera chargé dans A. On dit que **l'adresse effective** (l'adresse de l'opérande) est le contenu de A1. On peut utiliser l'instruction ADD sur les registres d'adresse pour les incrémenter ou décrémenter :

ADD #1, A1 ➔ A1 ← A1+1
ADD #-1, A7 ➔ A7 ← A7-1

2.11. Architecture retenue

On intègre les registres de A et de drapeaux à l'architecture présentée précédemment pour arriver à la Figure 7.

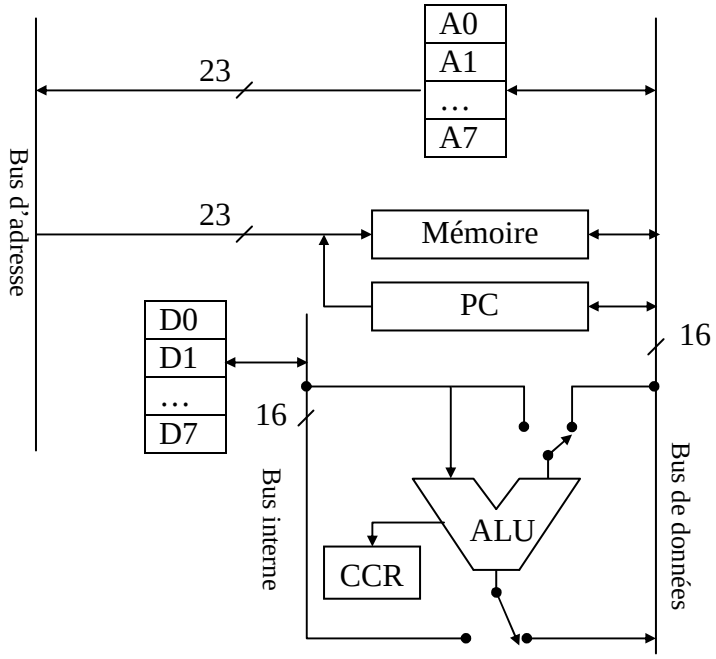


Figure 7- Architecture du processeur retenu

2.12. Modes d'adressage

Le mode d'adressage explique comment on peut obtenir la référence pour aller chercher une donnée. La donnée peut être dans les registres A0-A7, D0-D7, PC, dans la mémoire dont l'adresse est fournie dans le code opératoire ou dans la mémoire mais pointé par A_i, etc.
On peut prévoir les modes suivants :

2.12.1. Inhérent

La donnée se trouve dans un registre explicitement indiqué dans l'instruction. Par exemple :

MOVE D1, D2

La donnée est le contenu du registre D1.

2.12.2. Direct

L'adresse de la donnée se trouve directement dans l'instruction :

MOVE 47, D0

La donnée à l'adresse 47 sera chargée dans D0.

2.12.3. Indexé

L'adresse effective (l'adresse de donnée à chercher) est fournie par le registre d'index A_i.

ADD D0, (A1)

La mémoire pointée par A1 sera incrémentée avec le contenu de D0.

2.12.4. Immédiat

La donnée est explicitement donnée dans l'instruction. Par convention on ajoute le signe # devant la donnée pour préciser ce mode d'adressage.

ADD #4, D1

Le contenu de D1 est incrémenté de 4.

MOVE #$380, A1

Le registre d'adresse A1 est chargé par la valeur en hexadécimal de 380.

Remarque : Une donnée immédiate ne peut jamais être le deuxième opérande. Ainsi la commande "MOVE D1, #$380" n'a pas de sens.

2.13. Cycle d'instruction

Le microprocesseur est une machine à états finis. On peut schématiser le fonctionnement de cette machine comme présentée sur la Figure 8.

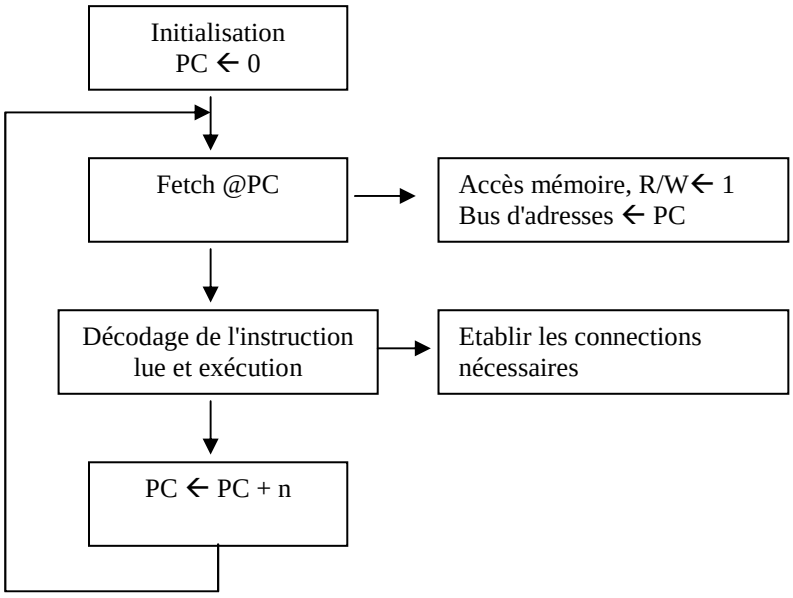


Figure 8- Machine à états finis du µP

2.14. Jeu d'instruction complet

La table ci-dessous présente les instructions déjà vues et leurs codes opératoires.

Mnémonique	Code opératoire
MOVE Dy,Dx	0011 Dx 000 000 Dy
MOVE mem,Dx	0011 Dx 000 111 001
MOVE Dy,mem	0011 001 111 000 Dy
MOVE Dy,(Ax)	0011 Ax 010 000 Dy
MOVE (Ay),Dx	0011 Dx 000 010 Ay
MOVE #data$_{16}$,Dx	0011 Dx 000 111 100 xxxx xxxx xxxx xxxx
MOVE #data$_{32}$,Ax	0010 Ax 001 111 100 xxxx xxxx xxxx xxx xxxx xxxx xxxx xxx
ADD Dy,Dx	1101 Dx 001 000 Dy
ADD mem,Dx	1101 Dx 001 111 001 xxxx xxxx xxxx xxx xxxx xxxx xxxx xxx
ADD Dy,mem	1101 Dy 101 111 001 xxxx xxxx xxxx xxx xxxx xxxx xxxx xxx
ADD #data$_{16}$,Dx	0000 0110 01 000 Dx xxxx xxxx xxxx xxx
ADD #data$_{32}$,Ax	1101 Ax 111 111 100 xxxx xxxx xxxx xxx xxxx xxxx xxxx xxx
ADD Dy,(Ax)	1101 Dy 101 010 Ax
ADD (Ay),Dx	1101 Dx 001 010 Ay
NEG Dx	0100 010 001 000 Dx
CMP #data$_{16}$,Dx	1011 Dx 001 111 100 xxxx xxxx xxxx xxx
AND #data$_{16}$,Dx	1100 Dx 001 111 100 xxxx xxxx xxxx xxx
Or #data$_{16}$,Dx	1000 Dx 001 111 100 xxxx xxxx xxxx xxx
BRA offset	0110 0000 (offset 8 bits)
BNE offset	0110 0110 (offset 8 bits)
BEQ offset	0110 0111 (offset 8 bits)
BMI offset	0110 1011 (offset 8 bits)

2.15. Exemple de programmation

Faire un programme pour calculer la somme des entiers de 1 à 10. Stockez le résultat dans la mémoire $1000.

Corrigé : On écrit d'abord le programme en langage assembleur (avec des mnémoniques). Ensuite suivant le tableau ci-dessus on génère les codes opératoires.

```
            MOVE  #0,D0
            MOVE  #1,D1
Boucle      ADD   D1,D0
            ADD   #1,D1
            CMP   #11,D1
            BNE   Boucle
```

```
            MOVE  D0,$1000
Wait        BRA   WAIT
```

Nous avons utilisé deux étiquettes "Boucle" et « Wait » car au moment d'écriture du programme, nous ne connaissons pas exactement le nombre de mots qui séparent l'instruction de branchement et l'instruction "cible".

Le programme d'assembleur génère les codes opératoires à partir des mnémoniques. Ici, on va le faire à la main !

Sachant que l'on commence à partir de l'adresse zéro, on aura à mettre les codes suivants en mémoire.

étiquette	adresse	Op. Code	mnémonique	Descriptions
	000	$303C $0000	MOVE #0,D0	
	002	$323C $0001	MOVE #1,D1	
	004	$D041	ADD D1,D0	
Boucle	005	$0641 $0001	ADD #1,D1	
	007	$B27C $000B	CMP #11,D1	
	009	$66FB	BNE Boucle	
	00A	$33C0 $0000 $1000	MOVE D0,$1000	
Wait	00D	$60FF	BRA Wait	

2.16. Exercices

1- Quelle est la valeur finale du registre D0 ? Transformez ce programme en codes opératoires et exécutez-le en détail. Les codes opératoires sont mis en mémoire à partir de l'adresse zéro. (La réponse n'est pas 5 !)

```
        MOVE    #10,D0
        MOVE    D0,$000006
        MOVE    #5,D0
Wait    BRA     Wait
```

2- Faire un programme pour multiplier par trois le contenu des mémoires de l'adresse $300 à l'adresse $3FF (donner uniquement les mnémoniques)

3. PERFECTIONNEMENT DE NOTRE PROCESSEUR

Dans le chapitre précédent, nous avons identifié un certain nombre de besoins pour que notre microprocesseur soit opérationnel. En écrivant quelques programmes simples, on se rend compte d'un certain nombre de limitations. Par exemple, on pouvait envisager un jeu d'instructions plus riche.

Cette limitation rend la tache de programmation plus difficile mais cela reste quand même faisable. Par contre, cette architecture ne permet pas de gérer des sous programmes. Sauf si le programmeur prend en charge tous les détails relatifs aux sous programmes. Par contre, les interruptions ne seront pas gérées même avec la meilleure volonté du monde !!

Dans ce chapitre, nous abordons la notion de pile qui nous permettra de gérer le passage à des sous programme beaucoup plus facilement et qui rendra possible la gestion d'interruption.

3.1.Notion de Pile

On entend par une pile, un buffer de type FILO (*First In Last Out*) ou LIFO (*Last In First Out*) pointé par un registre. Ce registre s'appelle couramment SP (*stack pointer*) ou le pointeur de pile. Le dessus de pile est normalement vers les adresses basses. La Figure 3-1 schématise la pile avec le pointeur correspondant.

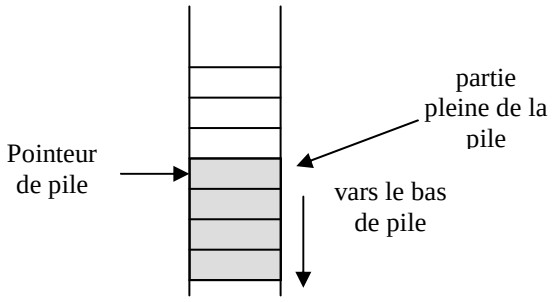


Figure 3-1- Pile

Quand le µp est mis sous tension, le contenu de SP (pointeur de pile) est à zéro. Il pointe donc sur la case mémoire se trouvant en haut de la RAM. Maintenant pour mettre une donnée sur la pile, il suffit décrémenter le registre SP et ensuite mettre la donnée à l'adresse pointée par SP. Ainsi cette donnée est sauvegardée à l'adresse $7FFFFF car "0-1" sur 23 bits, n'est que "$7FFFFF". On remarque que maintenant, le SP pointe sur la dernière donnée mise en pile. Pour mettre encore une donnée, il suffit de faire la même opération : décrémenter le SP et sauvegarder à l'adresse pointée par SP.

Au fur et à mesure que l'on stock des données sur la pile, le pointeur monte. Il faut faire attention que le SP n'arrive pas à une zone contenant des données utiles ou des codes opératoires du programme en cours d'exécution. Dans ce cas, la pile va déborder sur les données utiles en les écrasant. C'est le programmeur qui doit veiller à ce que la pile ne soit pas trop petite en réservant assez de cases mémoires pour la pile.

3.1.1. Architecture

On pourrait prévoir un registre supplémentaire SP sur 23 bits pour garder le pointeur de pile. Cependant, nous préférons d'utiliser un des registres d'adresse en tant que le pointeur de pile. C'est le registre A7 qui jouera le rôle du pointeur de pile.

3.1.2. Jeu d'instruction

On peut donc prévoir quelques instructions supplémentaires pour pouvoir gérer la pile. Pour mettre une donnée sur la pile on peut envisager la commande :

```
PUSH    Dx
```

Le résultat de cette commende est que
1- le registre de A7 est décrémenté
2- le registre source et sauvegardée à l'adresse pointée par la nouvelle valeur de A7
On peut aussi sauvegarder un registre d'adresse Ax sur la pile. Dans ce cas là, il nous faut sauvegarder deux mots sur la pile. Alors :

1- le A7 est décrémenté
2- les 16 LSB de Ax sont sauvegardés à l'adresse pointée par A7
3- le A7 est décrémenté de nouveau
4- les 16 MSB de Ax sont sauvegardés à l'adresse pointée par A7

Pour récupérer une donnée de la pile, on invente la commande suivante :

```
PULL    Dx
```

Le résultat d'exécution de cette commande est le suivant :
1- la valeur pointée par A7 est sauvegardée dans le registre de destination
2- le A7 est incrémenté

Remarque : si la destination est un registre d'adresse Ax, le résultat est le suivant :
1- la valeur pointée par A7 est sauvegardée dans le mot au poids fort de Ax
2- le A7 est incrémenté
3- la valeur pointée par A7 est sauvegardée dans le mot au poids faible du registre de destination Ax
4- le A7 est incrémenté de nouveau

Pour initialiser le pointeur de pile, on aura donc besoin d'une instruction de chargement :

```
MOVE    #data32,A7
```

Exemples d'utilisation :

Changer le contenu des registres D0 et D1

```
PUSH    D0
PUSH    D1
PULL    D0
PULL    D1
```

3.2.Sous programme

Un sous programme est une partie du programme que l'on peut appeler à n'importe quel instant dans la routine principale. La Figure 3-2 schématise cette procédure :

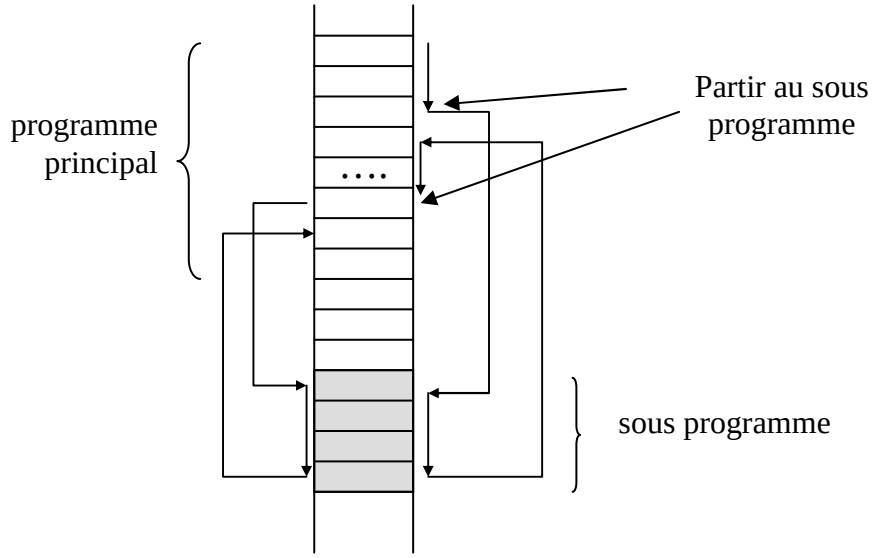


Figure 3-2- Appel d'un sous programme

On remarque que partir vers un sous programme se ressemble à un branchement normal car l'adresse de la routine de sous programme est connue. Par contre le retour est un peu plus compliqué car l'adresse de retour n'est pas connue à priori et qu'il change d'un appel à l'autre. Il faut donc inventer un mécanisme pour gérer correctement le retour d'un sous programme. Le paragraphe suivant aborde l'utilisation de la pile pour conserver l'adresse de retour d'un sous programme.

3.2.1. Mécanisme de sous programme

Pour partir à un sous programme, il suffit de mettre l'adresse de la routine correspondant dans le PC. Alors, automatiquement, l'instruction suivante à exécuter sera la première instruction du sous programmer. Cependant, le PC, avant d'être initialisé par cette nouvelle valeur, contenait l'adresse de l'instruction suivant le branchement vers le sous programme. Autrement dit, il contenait l'adresse de retour du sous programme.

Alors, il ne faut surtout pas l'écraser car il contient une information importante. Mais où est-ce que l'on peut le sauvegarder ? Sachant qu'une fois on est retourné du sous programme, la valeur sauvegardée ne sert plus à rien : il faut donc un emplacement provisoire. La pile est un excellent choix. La procédure d'appel est donc de sauvegarder le PC sur la pile, et ensuite, charger le PC par l'adresse de la routine. Au retour, on récupère dans le PC la valeur de dessus de pile. Il nous faut donc inventer quelques instructions.

3.2.2. Instructions relatives au sous programme

Pour partir à un sous programme, on invente la commande BSR (*Branch to SubRoutin*). On donne dans l'octet suivant du code opératoire le décalage d'adresse, exactement comme c'était le cas dans les instructions de branchement normales. La syntaxe de l'instruction est la suivante :

```
BSR    décalage =>  empiler le PC
                =>  PC  ←  PC + offset
```

Pour retourner d'un sous programme, on invente la commande RTS (*ReTurn from Subroutine*). Le résultat de cette commande est le suivant :

```
RTS   =>   Dépiler dans PC
```

3.3. Interruption

Imaginer qu'un événement extérieur demande un traitement particulier du µp. Cet événement peut intervenir à n'importe quel instant. Le processeur doit être capable d'abandonner le programme en cours et servir la routine correspondant à cette demande d'intervention. Une fois cette demande traitée, le µp reprend le programme qui était en cours d'exécution sans aucune perte de donnée ou d'état. Sachant que la routine appelée va de toute façon, utiliser les mêmes registres et les mêmes drapeaux que le programme principal, il faut sauvegarder l'état complet du système avant de commencer la routine d'interruption et les récupérer juste avant de revenir au programme principal.

Comment faire ? Déjà, au niveau hardware, il faudra prévoir une broche qui signale au µp une requête d'interruption. La tension de cette broche ($\overline{IRQ}$) étant à 5 volts (par exemple) signifie qu'aucune demande d'interruption est signalée. Dès que cette tension passe à 0 volt, une requête d'interruption est détectée. En principe, plusieurs périphériques peuvent être connectés sur le même fils à condition qu'ils soient tous en collecteur ouvert. Un seul périphérique qui demande un service peut faire baisser la tension de cette ligne : le µp est interrompu et il peut maintenant chercher l'origine de l'interruption pour exécuter la routine correspondant à la source de l'interruption.

Paragraphe ci-dessous aborde le mécanisme exact à mettre en place pour répondre aux interruptions.

3.3.1. Mécanisme de l'interruption

La routine d'interruption est placée dans la mémoire à une adresse donnée. Cette adresse peut changer suivant l'application. Alors, une fois le µp ressent un niveau bas sur la broche IRQ, comment peut-on le faire exécuter la routine correspondant ?
On peut réserver deux octets à une adresse fixe contenant l'adresse de la routine à exécuter. Ainsi, le programmeur peut mettre l'adresse de sa routine à cet endroit. En résumé donc, on fige l'adresse $7FFFFE et $7FFFFF pour accueillir une adresse. A la réception d'une requête d'interruption, le contenu de l'adresse $7FFFFE et $7FFFFF est chargé dans PC. Du coup, le branchement s'effectue. Par contre, comme on a vu dans le cadre de sous programme, il faut sauvegarder le contenu de PC pour pouvoir retourner. On fait donc la même opération ici :

1- sauvegarder le PC sur la pile
2- charger le PC par le contenu des mémoires $7FFFFE et $7FFFFF

Dans la routine d'interruption, on sauvegarde sur la pile les registres qui vont être utilisés par la routine. C'est le souci du programmeur et le µp n'y est pour rien. Donc, en principe, les premières lignes de la routine d'interruption ne sont que des PUSH. La routine de l'interruption est ensuite exécutée jusqu'à la dernière instruction. Avant de retourner au

programme principal, il faut récupérer le contenu des registres avant de quitter la routine d'interruption. Alors, une série d'instructions PULL est utilisée pour récupérer ces valeurs dans l'ordre. Le seul problème s'agit des indicateurs. Puisque nous n'avons pas prévu des instructions pour sauvegarder les drapeaux, on laisse le microprocesseur de gérer cette partie. C'est à dire qu'avant partir à la routine d'interruption, le µp met sur la pile le registre de code condition. Les taches effectuées par le µP sont donc les suivantes :

1- sauvegarder le PC sur la pile
2- sauvegarder le registre de code condition sur la pile
3- charger le PC par le contenu des mémoires $7FFFFE et $7FFFFF

La dernière ligne est l'instruction RTE (*Return from Intrupt*). Concrètement parlant, la seul chose qui se passe c'est que la pile est dépilée dans le registre de code condition et ensuite dans le PC. Le retour vers là-où on avait quitté le programme principal se fait donc automatiquement. Par contre, si la parité entre les instructions PUSH et PULL n'est pas respectée au sein de la routine d'interruption, une valeur foireuse sera chargée dans CCR et dans PC qui va faire brancher le µp dieu sait où !!!

3.3.2. Acquittement de l'interruption

Le circuit périphérique doit être informé du fait que sa demande d'interruption a été prise en compte et qu'il peut maintenant "lâcher" la ligne IRQ. On prévoit donc une sortie ACK qui est normalement à 5 volts. Dès qu'une interruption est servie, le processeur va mettre 0 volt sur cette ligne. Le périphérique en voyant un front descendant sur cette ligne, relâche la ligne IRQ. Le processeur de son côté, en voyant le passage de la ligne IRQ à 5 vols, remet en tension haute la ligne ACK. Ce mécanisme est nécessaire afin d'éviter l'interruption sans arrêt dès le retour de la routine d'interruption au programme principal.

3.3.3. Interruption imbriquée

On peut autoriser ou non la possibilité de re-interrompre un processeur qui est en train d'exécuter une routine d'interruption. On prévoit dans le registre de code condition CCR, un bit pour masquer l'interruption. Alors, quand l'interruption arrive, le µp teste ce bit là. Si ce bit est à 1, le processeur ne répond pas à l'interruption et exécute l'instruction suivante comme s'il n'y avait jamais eu une demande d'interruption. Par contre, si ce bit est à zéro, c'est que l'interruption n'est pas masquée : le µp répond. Cette fois-ci, pour éviter les interruptions successives, il masque les nouvelles interruptions en mettant à 1 ce bit et en remettant à zéro au moment d'exécuter la commande RTI. On reprend les opérations à effectuées au moment de répondre à une interruption :

1- mettre le bit I du CCR à 1
2- sauvegarder le PC sur la pile
3- sauvegarder le registre de code condition sur la pile
4- charger le PC par le contenu des mémoires $3FE et $3FF

Pour la commande RTI, on aura donc les opérations suivantes :

1- mettre le bit I du CCR à 0
2- dépiler dans le CCR

3- dépiler dans PC

3.4.Exercice

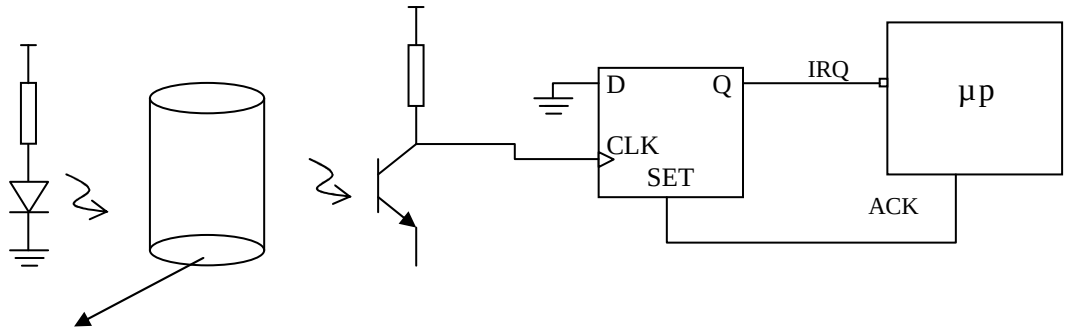
1- Faire un sous programme qui multiplie le registre D0 par D1 et retourne le résultat dans D2. (On considère une multiplication non signée et on ne gère pas le dépassement)

2- Faire un sous programme qui prend dans A0 l'adresse de début d'un buffer et dans D0 sa taille. Le sous programme remplace ce buffer par la valeur absolue de tous ces éléments.

3- Faire un programme qui multiplie deux valeurs déjà sauvegardées sur la pile et qui retourne le résultat sur la pile (multiplication signée et pas de dépassement). Un exemple du programme appelant est le suivant :

```
      MOVE      #$FFFF00,A7
      MOVE      #$05,D0
      PUSH      D0   premier opérande
      MOVE      #$06,D0
      PUSH      D0   deuxième opérande
      BSR       MULT
      PULL      D0   le résultat
Wait  BRA       Wait
```

4- Imaginer que la patte d'IRQ de notre processeur est connectée à un signal signalant le passage d'un objet devant un capteur infrarouge. Le but est de compter le nombre d'objet qui passe. Faite la routine d'interruption correspondant.



4. MICROPROCESSEUR 68000

Le 68000 est un microprocesseur 16/32 bits sorti en 1979. Il devint rapidement le processeur le plus prisé de sa génération. Le formidable succès des machines comme Mac, Amiga ou Atari, est en grande partie due à cet innovant processeur.

Voici les caractéristiques globales du composant :

- Mémoire adressable de 16Mo (24 lignes d'adresse).
- Jeu d'instruction très complet (56 Instructions).
- 14 modes d'adressage différents.
- 8 registres de donnée 32 bits.
- 7 registres d'adresse 32 bits.
- Gestion de privilèges (Utilisateur / Superviseur).
- Deux pointeurs de pile 32 bits différents.

Le terme 16/32 bits résume la façon dont le circuit manipule les données. Le bus de données interne est sur 32 bits mais le processeur communique avec le reste du monde sur un bus externe de 16 bits.

Aujourd'hui la série "microprocesseur 680x0" se compose du 68010, 68020, 68030, 68040 et 68060. Ces processeurs constituent la base des versions microcontrôleurs 683xx. Les caractéristiques de ces composants ainsi que de nombreux manuels utilisateurs peuvent être télécharger sur le site de MOTOROLA.

Nous détaillons l'architecture de ce microprocesseur au fur et à mesure que nous abordons chaque partie du système.

4.1.Les bus

4.1.1. Bus de données

Le bus de données comporte 16 bits. Il y a donc 16 broches nommées de D0 à D15 qui sont connectées à une mémoire externe.

4.1.2. Bus d'adresse

Ce bus comporte 23 fils, c'est-à-dire 2^{23} combinaisons pour adresser donc 8 mega cases mémoires à 16 bits. Ces lignes se nomment A1 à A23 (et non pas de A0 à A22). En effet la ligne A0 n'existe pas, comme si elle est toujours égale à zéro et que nous avons un bus d'adresse de taille 24. Comme effet, les adresses vont de zéro à 2^{24}-2 par un pas de 2 : 0, 2, 4, 6, …, 16 777 214. La Figure 4-1 montre la mémoire et les adresses correspondant :

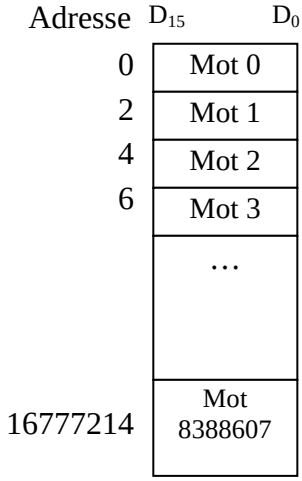


Figure 4-1- organisation de mémoire par des mots sur 16 bits

Cependant on peut aussi adresser les données sur 8 bits (un octet). Pour pouvoir adresser donc des données sur 8 bits aux adresses paires ou impaires (sans avoir A0 à notre disposition), 68000 possède deux broches supplémentaires qui s'appellent USD (*Upper Data Strobe*) et LDS (*Lower Data Strobe*). Ces deux lignes nous permettent d'adresser la partie haute de 16 bits (octet au poids fort) ou la partie basse de 16 bits (octet au poids faible). En effet ces deux lignes jouent le rôle de A0 pour former finalement un bus d'adresse de taille 24. On peut donc adresser 2^{24} octets. L'organisation de la mémoire, vue octet par octet, est schématisée sur la Figure 4-2.

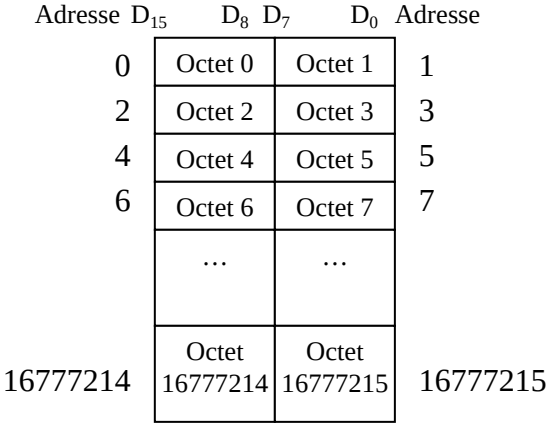


Figure 4-2- organisation de mémoire par des octets (sur 8 bits)

On remarque que les octets sont adressés par des adresses allant de 0 à 2^{24}-1 tandis que les mots sont adressés uniquement par des adresses paires dans le même intervalle.

On peut aussi adresser la mémoire pour traiter des données encore plus longues (sur 32 bits). Dans ce cas cette donnée est écrite sur 4 octets (ou deux mots) comme schématisé sur la Figure 4-3.

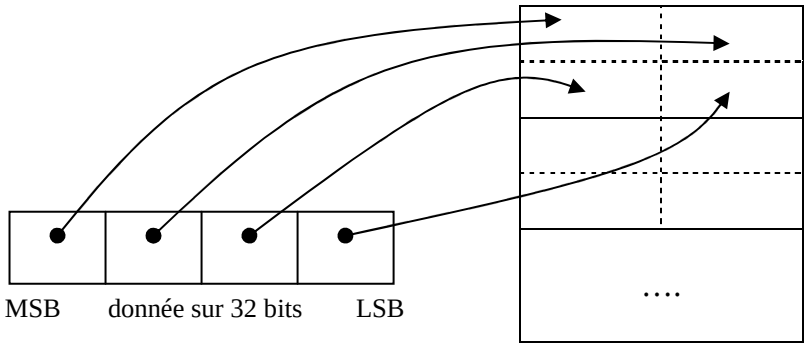


Figure 4-3- Organisation de mémoire pour les mots longs

Les mots longs sont adressés uniquement par des adresses paires.
Exercice : Lesquelles des lignes ci-dessous sont incorrectes ?

[$FFABCD] ← #$1234
[$122101] ← #$12
[$000001] ← #$12345678
[$000004] ← #$4556788910

4.2.Registres internes

4.2.1. Registres de données

68000 est un microprocesseur 32 bits, les registres sont tous de taille 32 bits. Il y a 8 registres de données, nommés D0 à D7. Ces registres sont complètement interchangeables, c'est-à-dire qu'une opération qui fonctionne sur Di, fonctionnera sur Dj. Dans beaucoup de cas, nous n'avons pas besoin de 32 bits pour les opérandes. Par exemple pour traiter les textes, 8 bits suffiront. 68000 permet d'effectuer les opérations soit sur des mots longs (32 bits) soit sur des mots (16 bits) soit sur des octets (8 bits). Pour spécifier la taille des opérandes, à chaque instruction correspond une attribution : « l » pour long mot, « w » pour mot et « b » pour octet. La Figure 4-4 présente la répartition des registres sur différents attributs possibles.

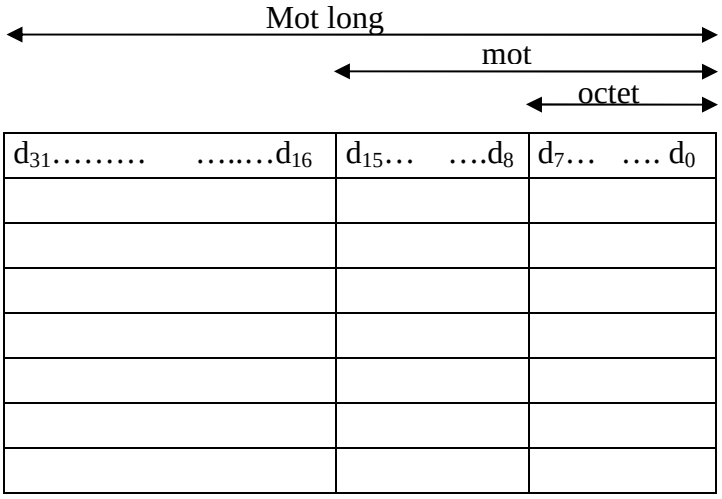


Figure 4-4- Les registres de données du 68000

Par exemple pour ajouter au D1 le contenu de D0, on utilise l'instruction : ADD.l D0,D1. Pour faire la même opération mais uniquement sur les opérandes de taille 16 bits on a : ADD.w D0,D1. Dans ce cas, les 16 MSB de la destination ne sont pas modifiés. Faire la même opération sur des octets : ADD.b D0,D1. Dans ce cas, les 24 MSB de la destination ne sont pas modifiés.

Remarque : On ne peut pas faire des opérations sur l'octet (mot) au poids fort. C'est uniquement l'octet (mot) au poids faible qui peut être utilisé en tant que l'opérande.

4.2.2. Registres d'adresse

68000 a 8 registres d'adresse s'appelant A0 à A7 présenté sur la Figure 4-5.

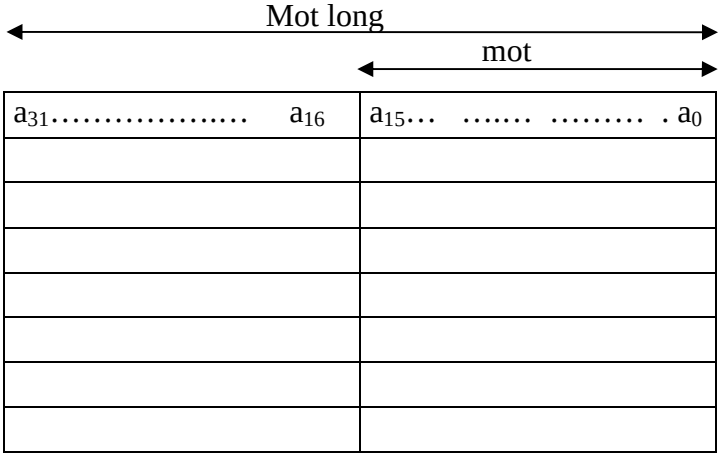


Figure 4-5- Les registres d'adresse du 68000

Ces registres sont des pointeurs et sont utilisés pour adresser la mémoire. Contrairement au cas des registres de donnée, tous les 32 bits de ces registres vont être affectés à la suite d'une opération. Des opérations sur octet ne sont pas autorisées.

Remarque : on peut effectuer l'opération de chargement sur le mot au poids faible d'un registre A mais tous les 32 bits de la destination vont être chargés. C'est-à-dire que les 16 bits MSB sont l'extension de signe du mot au poids faible.

Exemple : Si on voulait charger $AB12 dans le mot au poids faible du registre A0, le contenu de ce registre sera : $FFFF AB12 car le bit de signe de la donnée est égale à 1. Par contre, un chargement de $1234 dans le registre A1 a comme effet de sauvegarder $00001234 dans ce registre.

Remarque : Il vaut mieux ne pas interpréter le contenu d'un registre d'adresse en complément à deux mais en entier non signé.

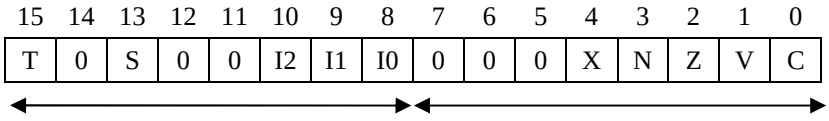
Remarque : Toutes les opérations peuvent être effectuées sur tous les Aj. Cependant, le registre A7 joue le rôle de pointeur de pile.

4.2.3. Compteur de programme

Le compteur de programme est un registre de 32 bits qui contient l'adresse de la prochaine instruction à exécuter. Le bus d'adresse du 68000 étant sur 24 bits, les 8 MSB de ce registre ne sont pas utilisés.

4.2.4. Registre de statut

Le registre de statut est un registre de taille 16 contenant un certain nombre de drapeaux. On peut le diviser en deux octets : statut et CCR. La Figure 4-6 détaille les différents drapeaux de ce registre.



C	Carry bit
V	Overflow bit
Z	Zero bit
N	Negative bit
X	Extend bit
I0-I2	Interrupt mask
S	Status bit
T	Trace bit

Figure 4-6- Registre de statut du 68000

Les drapeaux de l'octet de statut définissent le mode d'opération du processeur. On y reviendra un peu plus trad. L'octet CCR (*Code Condition Register*) contient les drapeaux qui se positionnent dans un certain nombre d'instructions comme les opérations arithmétiques ou logiques. Par exemple le bit carry (retenue) est la retenue du bit 31, 15 ou 7 (suivant l'attribut utilisé l, w ou b).

Exercice: Quel est le contenu du registre D1 et le drapeau C pour chacun des ces ci-dessous : (sachant que D0=$1234 5678, D1=$1357 9BDF
ADD.L D0,D1
ADD.W D0,D1
ADD.B D0,D1

Le drapeau X est une copie de C mais uniquement après un certain nombre d'instructions : addition, soustraction, négation, et décalage. Par contre pour les instructions suivantes, c'est uniquement le drapeau C qui est affecté et le X ne se modifie pas : CMP, MOVE, AND, OR, CLR, TST, MUL et DIV.
Le rôle de X est un vrai bit d'extension qui va étendre les opérations sur les octets à un 9ième bit, sur les mots à un 17ième bit et sur les mots longs à un 33ième bit.
Le bit X a été prévu dans 68000 car le bit C est par fois utilisé pour autre chose qu'un bit d'extension.

Le drapeau V est le dépassement. Il est positionné quand le signe du résultat d'une opération mathématique en complément à 2 n'est pas correct.
Le drapeau Z est à 1 quand le résultat (sur octet, mot ou mot long) est zéro. Le bit N est une copie du bit de signe du résultat.

4.3.Modes d'adressage

Comme nous l'avons vu dans le chapitre précédent, pour faire une opération il faut aller chercher des opérandes. Ces opérandes se trouvent soit dans la mémoire soit dans un registre interne du 68000. Le mode d'adressage aborde la modalité d'adresser ces opérandes.

Avant de détailler ces modes, nous allons opter la notation suivante pour éviter d'éventuelles ambiguïtés.

- Une flèche s'utilise pour designer l'opération d'affectation.
- Des crochets sont utilisés pour désigner le contenu d'un registre ou d'une mémoire.
- $ est utilisé pour indiquer une valeur en hexadécimale.
- % est utilisé pour indiquer une valeur en binaire.
- Pour designer mot long, mot ou octet on utilise .l, .w ou .b.
- Pour designer la mémoire, nous utilisons la lettre M(adr). Entre parenthèses l'adresse de la mémoire est donnée.
- <> une expression est requise.
- [Di(0:7)] désigne uniquement les bits de 0 à 7 du registre Di
- ae est un abréviation d'adresse effective
- d8, d16, d32 sont des données sur 8, 16 ou 32 bits
- Xn pour dire que l'on peut utiliser à la fois Dn ou An

Exemple :
[D0.l] ← $50 signifie que l'on met la valeur décimale de 80 dans la totalité du registre D0.
[D0.b] ← $34 signifie que les 8 LSB du registre D0 sont chargés par $34 et le reste du registre n'est pas modifié.
[M($123456).W] ← $32 signifie que l'adresse $123456 est chargée par 0 et l'adresse suivante ($123457) par $32 (pourquoi ?)
Remarque : [M($123457).w] ← $32 est une erreur (pourquoi ?)

La syntaxe d'instruction est la suivante :

Opération <source>,<destination>

Pour aborder les modes d'adressage, nous utilisons très souvent les commandes MOVE et ADD. Par exemple la commande :

MOVE <source>,<destination>

sera présentée comme :

[destination] ← [source]

et la commande

ADD <source>, <destination>

Est présentée :

[destination] ← [destination]+[source]

4.3.1. Adressage registre direct

4.3.1.1. Direct avec Dn

L'opérande est un registre de donnée du microprocesseur directement spécifié dans l'instruction

Exemple

MOVE.B	**D0**,$1234	[$001234.B] ← [D0.B]
MOVE.W	$9876,**D1**	[D1.W] ← [$FF9876.W] (**attention à l'extension de signe**)
MOVE.L	**D0**,**D1**	[D1.L]← [D0.L]

4.3.1.2. Direct avec An

L'opérande est un registre d'adresse du microprocesseur directement spécifié dans l'instruction

Exemple

MOVE.W	D0,**A0**;	[A0.L] ← [D0.W] (attention à l'extension de signe de 16 bits sur 32 bits)
MOVE.L	D0,**A0**;	[A0.L] ← [D0.L]
MOVE.W	**A0**,D0;	[D0.W] ← [A0.W]
MOVE.L	**A0**,D0;	[D0.L] ← [A0.L]

4.3.2. Adressage immédiat

L'opérande est donné immédiatement dans l'instruction. C'est dire que le µp n'a pas besoin d'aller le chercher en mémoire ni dans un registre. Dans ce cas, nous utilisons le signe « # » pour spécifier ce mode d'adressage.

Exemple

MOVE.W	#18,D0	[D0.W] ← $0012
MOVE.W	#$9876,A0	[A0.L] ← $FFFF9876 (attention à l'extension de signe)
MOVE.B	#%10100111,D1	[D1.B] ← $A7

Remarque : la destination ne peut pas être une donnée immédiate.

4.3.3. Adressage direct

L'opérande se trouve à l'adresse donnée directement dans l'instruction.

Exemple :

MOVE.B	$123,D1	[D1] ← [M($000123)]
MOVE.L	$45678,D0	[D0(31:24)] ← [M($045678)]
		[D0(23:16)] ← [M($045679)]
		[D0(15:8)] ← [M($04567A)]
		[D0(7:0)] ← [M($04567B)]

Remarque : si la donnée est un mot ou un mot long, l'adresse ne peut pas être imapire

4.3.4. Adressage registre indirect

L'opérande se trouve à une adresse donnée par un registre d'adresse. C'est-à-dire que l'adresse de l'opérande n'est pas donnée directement par l'instruction. Dans l'instruction le registre d'adresse est donné entre parenthèses pour indiquer le mode indirect.

Exemple :

MOVE	#$1234,A0	[A0] ← $00001234
MOVE.L	(A0),D3	[D3(31:0)] ← [M([A0]).L] c'est-à-dire :
		[D3(31 :24)] ← [M($001234)]
		[D3(23 :16)] ← [M($001235)]
		[D3(15 :8)] ← [M($001236)]
		[D3(7 :0)] ← [M($001237)]
MOVE.W	D4,(A6)	[M([A6]).W] ← [D4.W]

On parle de l'adresse effective qui est l'adresse de l'opérande. Dans le premier exemple ci-dessus, l'adresse effective est $1234. Dans le deuxième exemple, cette adresse se trouve dans A0.

Remarque : Ce mode d'adressage est intéressant car l'adresse effective se trouve à l'intérieur du µp et il n'a pas à aller la chercher en mémoire. Le code opératoire de l'instruction est donc plus court ce qui signifie une vitesse d'exécution plus importante.

4.3.5. Adressage registre indirect avec post-incrémentation

L'opérande se trouve à l'adresse contenue dans Ai. Une fois l'opérande est récupérer, le registre Ai est incrémenter de 1, de 2 ou de 4 suivant l'attribut de l'opération. Ceci permet d'avancer dans un tableau de valeurs en mémoire de manière automatique. Pour indiquer ce mode d'adressage dans l'instruction, on ajout le signe de « + » après le registre An.

Remarque : Il y a une exception pour le registre A7. Si la taille de l'opérande est un octet, le registre d'adresse A7 qui est le pointeur de la pile sera post-incrémenté de 2 et non de 1, ceci pour avoir en permanence un pointeur de pile dont l'adresse est paire.

Exemple :

MOVE.B	(A0)+, D1	[D1(7:0)] ← [M([A0]).B]

```
                        [A0] ← [A0]+1

MOVE.W    (A0)+, D1     [D1(15:0)] ← [M([A0]).W]
                        [A0] ← [A0]+2

MOVE.L    (A0)+, D1     [D1(31:0)] ← [M([A0]).L]
                        [A0] ← [A0]+4

MOVE.B    D0, (A7)+     [M([A7])] ← [D0(7:0)]
                        [A7] ← [A7]+2
```

4.3.6. Adressage registre indirect avec pré-décrémentation

Le registre Ai est d'abord décrémenté de 1, 2 ou 4. Ensuite, le processeur va chercher l'opérande dont l'adresse se trouve dans le registre An. . Pour indiquer ce mode d'adressage dans l'instruction, on ajout le signe de « - » avent le registre An.

Remarque : Il y a une exception pour le registre A7. Si la taille de l'opérande est un octet, le registre d'adresse A7 qui est le pointeur de la pile sera pré-décrementé de 2 et non de 1, ceci pour avoir en permanence un pointeur de pile dont l'adresse est paire.

Exemple :

```
MOVE.B    -(A0),D1      [A0] ← [A0]-1
                        [D1.B] ← [M([A0]).B]

MOVE.W    -(A0),D1      [A0] ← [A0]-2
                        [D1.W] ← [M([A0]).W]

MOVE.L    -(A0),D1      [A0] ← [A0]-4
                        [D1.L] ← [M([A0]).L]

MOVE.B    D0,-(A7)      [A7] ← [A7]-2
                        [D1.B] ← [M([A7]).B]
```

4.3.7. Adressage registre indirect avec déplacement

L'adresse effective de l'opérande à chercher est la somme du contenu du registre d'adresse indiqué dans l'instruction et un déplacement immédiatement donné dans l'instruction. Le contenu du registre d'adresse n'est évidemment pas modifié.

Remarque : Si l'opérande est un mot ou un mot long, il faut que l'adresse finale soit paire.
Remarque : Le déplacement est donné sur 16 bits. Il est donc dans l'intervalle (-32768, 32767).

Exemple

```
MOVE.W    $19(A0), D1       [D1(0:15)] ← [M([A0]+$19).W]
                            (Il faut que [A0]+$19 soit pair)
MOVE.L    -$04(A1),D0       [D0(0:31)] ← [M([A1]-$04).L]
```

4.3.8. Adressage registre indirect avec index

L'adresse effective de l'opérande est la somme du registre d'adresse donnée dans l'instruction, une valeur immédiate indiquée dans l'instruction, et le contenu d'un registre d'index (Di ou Ai) précisé dans l'instruction. Les registres d'adresse et d'index ne sont évidemment pas modifiés. Le registre d'index peut avoir l'attribut .W ou .L. La donnée immédiate est limitée à 8 bits, elle est donc dans l'intervalle [-128,127].

Remarque : Si l'opérande est un mot ou un mot long, il faut que l'adresse effective finale soit paire.

Exemple
MOVE.W 9(A1, D0.W), D3 [D3] ← [M([A1]+[D0(0:15)]+9)]

4.3.9. Adressage relatif PC

Ce mode d'adressage ressemble au mode d'adressage registre indirect sauf que le registre d'adresse est remplacé par le PC. Deux formes de l'adressage relatif PC est implantés en 68000 : avec déplacement et avec index.
- Adressage relatif PC avec déplacement ae=[PC] + déplacement (16 bits)
- Adressage relatif PC avec index ae=[PC] + [Di ou Ai] + déplacement (8 bits)

Les déplacements sont en complément à 2.

Exemple :

```
        MOVE.B    10(PC),D2     [D2.B] ← [M([PC]+10).B]

        MOVE.B    TBL(PC),D2    [D2.B] ← Valeur 1
        .
        .
        .
TBL     DC.B      Valeur 1      TBL est une étiquette
        DC.B      Valeur 2      DC est une directive d'assembleur (on le verra plus tard)
```

4.4. Introduction au jeu d'instructions

4.4.1. Instruction de chargement

4.4.1.1. MOVE

Recopie le contenu de l'adresse source dans l'adresse destination. Tous les modes d'adressage sont supportés. La destination ne peut pas être une donnée immédiate.

Modes d'Adressage (source)
Dn, An, (An), -(An), (An)+, d16(An), d8(An,Xn.s), d16(PC), d8(PC,Xn.s), adr, #d8, #d16, #d32

- Quand la source est le registre An, la taille de donnée doit être un mot ou un mot long.

Modes d'Adressage (destination)
Dn, (An), -(An), (An)+, d16(An), d8(An,Xn.s), adr

- Quand la destination est le An, vous devez utiliser l'instruction MOVEA.
- quand on utilise le registre A7 et mode d'adressage registre indirecte avec pré-décrémentation ou post-incrémentation, même l'attribut octet fait incrémenter ou décrémenter le A7 par deux. Ceci est pour assurer que le pointeur de pile ne pointe jamais sur des adresses impaires.

Taille des données
Octet / Mot / Mot long

Indicateur

X	Inchangé
N	Standard
Z	Standard
V	0
C	0

Syntaxe
MOVE <adresse>, <adresse>

4.4.1.2. MOVEA

Copie la valeur d'une adresse effective dans un registre d'adresse.

Modes d'Adressage (source)
An, (An), -(An), (An)+, d16(An), d8(An,Xn.s), adr, d16(PC), d8(PC,Xn.s), #d16, #d32

- Pour le mode d'adressage An, la taille des données doit être un mot ou un mot long. Cependant tous les 32 bit de la destination sont affectés moyennant une extension de signe.

Modes d'Adressage (destination) :
An

Taille des données
Mot, Mot long

Indicateurs
Inchangés

Syntaxe
MOVEA <adresse>,An

4.4.1.3. MOVEM

Cette instruction transfert rapidement un groupe de registres dans la mémoire ou bien la mémoire dans un groupe de registres. Cette opération ne marchera qu'avec des mots ou des mots longs. Lors du transfert vers les registres, les mots sont automatiquement étendus au niveau du signe donc les 32 bits seront affectés.
Notez que cette instruction est très utilisée pour déposer plusieurs registres sur la pile avec le mode d'adressage -(An) et les récupérer intactes avec le mode d'adressage (An)+ après plusieurs instructions, retenez que la plupart des sous-programmes (ceux des libraires par exemple) détruisent plusieurs registres lors de leur exécution.
La liste des registres est constituée de registres séparés par des /, vous pouvez aussi faire des intervalles avec des -. Par exemple :

D0-D2/A3-A6 désigne une liste de registres constituée des registres D0, D1, D2, A3, A4, A5 et A6.

Modes d'adressage (liste de registres dans la mémoire)
(An), -(An), d16(An), d8(An, Xn.s), adr

Modes d'adressage (mémoire dans la liste de registres)
(An), (An)+, d16(An), d8(An, Xn.s), adr, d16(PC), d8(PC, Xn.s)

Taille des données **Indicateurs**
Mot / Mot long Inchangés

Syntaxe
MOVEM <liste de registres>, <adresse>
MOVEM <adresse>, <liste de registres>
Exemple : MOVE.W D0-D2/D5/A1-A3,-(A7)

4.4.1.4. MOVEP

MOVEP permet d'envoyer les données aux périphériques. Voir les détaille dans le manuel du 68000.

4.4.1.5. EXG

Les 32 bits des deux registres sont échangés. Les registres peuvent être aussi bien des registres de données que des registres d'adresses, on peut échanger un registre de données avec un registre d'adresses.

Taille des données **Indicateurs**
Mot long Inchangés

Syntaxe
EXG Rx, Ry

4.4.1.6. LEA

Charge dans le registre An l'adresse effective de la source et non pas son contenu. L'adresse peut être fournie dans tous les modes d'adressage sauf immédiat bien sûr. Cette instruction est très couramment utilisée pour écrire des programmes qui doivent être indépendant de leur position dans la mémoire. Dans ce cas, nous utiliserons les modes d'adressages d16(PC).

Modes d'adressage
(An), d16(An), d8(An, Xn.s), adr, d16(PC), d8(PC, Xn.s)

Taille des données **Indicateurs**
Mot long Inchangés

Syntaxe
LEA <adresse>, An

4.4.1.7. PEA

Push Effective Address : Cette instruction calcule une adresse effective et la dépose sur la pile, l'adresse est toujours un mot long.

Modes d'adressage
(An), d16(An), d8(An, Xn.s), adr, d16(PC), d8(PC, Xn.s)

Taille des données
Mot long

Indicateurs
Inchangés

Syntaxe
PEA <adresse>

4.4.2. Opérations arithmétiques

Nous nous contentons ici de lister uniquement les instructions mathématiques. Les détailles de chaque instruction sont à découvrir dans les manuels de 68000.

Mnemonic	Description	Operation
ADD	Add binary	(destination)+(source)→ destination
ADDA	Add address	(destination)+(source)→ destination
ADDI	Add immediate	(destination)+immediate data→ destination
ADDQ	Add quick	(destination)+immediate data→ destination
ADDX	Add extended	(destination)+(source)+X→ destination
ABCD	Add decimal with extend	$(destination)_{10}+(source)_{10}+X$→ destination
SUB	Subtract binary	(destination)-(source)→ destination
SUBA	Subtract address	(destination)-(source)→ destination
SUBI	Subtract Immediate	(destination)-immediate data→ destination
SUBQ	Subtract quick	(destination)-immediate data→ destination
SUBX	Subtract with extend	(destination)-(source)-X→ destination
MULS	Signed multiply	(destination)*(source)→ destination
MULU	Unsigned multiply	(destination)*(source)→ destination
DIVS	Signed divide	(destination)/(source)→ destination
DIVU	Unsigned divide	(destination)/(source)→ destination
EXT	Sign extend	(destination) sign extended → destination
NEG	Negate	0 - (destination) → destination
NEGX	Negate with extend	0 - (destination) – X → destination

4.4.3. Opérations logiques

Ci-dessous une liste des opérations logiques est donnée. Pour obtenir des détails particuliers sur chaque commande, voir les manuels du 68000.

EOR	Exclusive OR logical	(destination)⊕(source)→ (destination)
EORI	Exclusive OR immediate	(destination)⊕immediate data→ (destination)
NOT	Logical complement	~(destination)→ (destination)
OR	Or logical	(destination) OR (source)→ (destination)
ORI	Or immediate	(destination) OR immediate data→ (destination)

AND	And logical	(destination) AND (source)→ (destination)
ANDI	And immediate	(destination) AND immediate data→ (destination)
BCLR	Test a bit and clear	~(<bit number>) OF destination → Z 0 → ~(<bit number>) OF destination
BSET	Test a bit and set	~(<bit number>) OF destination → Z 1 → ~(<bit number>) OF destination

4.4.4. Opérations de branchement

4.4.4.1. Branchement sans condition

Deux types de branchement sont utilisés dans 68000. Le premier est un branchement relatif. C'est-à-dire que le processeur va sauter un nombre précis d'octets plus bas ou plus haut. C'est dire que l'instruction fournit le déplacement (l'offset) par rapport au PC courant. L'instruction correspondant est « BRA déplacement ». Le déplacement est donné soit sur 8 bits (128 octets plus haut ou 127 octet plus bas) soit sur 16 bits (32768 octets plus haut ou 32767 octets plus bas).

Remarque : on parle de l'octet et non pas de l'instruction.

Dans le deuxième type de branchement, on donne carrément l'adresse de l'octet à brancher dessus. Ce n'est donc pas un branchement relatif PC. Dans ce cas l'instruction contient l'adresse de la destination sur 32 bits. L'instruction à utiliser est « JMP destination ».

BRA	Branch always	PC + Displacement → PC
JMP	Jump	Destination → PC

Pour l'instruction BRA voir le paragraphe suivant.

Modes d'adressage pour l'instruction JMP

(An), d16(An), d8(An, Xn.s), adr, d16(PC), d8(PC, Xn.s)

Indicateurs

Inchangés

Syntaxe

JMP <adresse>

4.4.4.2. Branchement conditionnel

Ce type de branchement vérifie d'abord la condition donnée dans l'instruction et ensuit branche à la destination si cette condition est vérifiée. La syntaxe générale de cette instruction est la suivante :

Bcc déplacement

Le cc désigne la condition de branchement qui est en effet une fonction des indicateurs (drapeaux). A chaque fois, cc dans Bcc est remplacé par deux lettres pour avoir au total 15 cas différents. Le tableau ci-dessous en donne la liste :

Mnémonique	Texte	Branchement si	Pour les nombres
BCC	Branch Carry Clear	C=0	
BCS	Branch Carry Set	C=1	

BEQ	Branch Equal	Z=1	
BNE	Branch Not Equal	Z=0	
BGE	Branch Greater or Equal	N=V	signés
BGT	Branch Greater Than	N=V ET Z=0	signés
BHI	Branch HIgher than	C=0 ET Z=0	non signés
BLE	Branch Less or Equal	Z=1 OU NV	signés
BLS	Branch Lower or Same	C=1 OU Z=1	non signés
BLT	Branch Less Than	N=~V	signés
BMI	Branch Minus	N=1	non signés
BPL	Branch Plus	N=0	non signés
BVC	Branch V Clear	V=0	
BVS	Branch V Set	V=1	
BRA	BRanch Always	Sans condition	

Taille de déplacement
Octet / Mot

Indicateurs
Inchangés

Syntaxe
Bcc.B <label>
Bcc.W <label>

4.4.4.3. Branchement avec décrémentation

L'instruction « DBcc » (Decrease then Branch) permet de faire des branchements (sauts) dans un programme suivant l'état des indicateurs, et ce autant de fois que nous le définissons dans un registre de données.

Mnémonique	Texte	Branchement si	Pour les nombres
DBCC	Decrease and Branch Carry Clear	C=0	
DBCS	Decrease and Branch Carry Set	C=1	
DBEQ	Decrease and Branch EQual	Z=1	
DBNE	Decrease and Branch Not Equal	Z=0	
DBGE	Decrease and Branch Greater or Equal	N=V	signés
DBGT	Decrease and Branch Greater Than	N=V ET Z=0	signés
DBHI	Decrease and Branch HIgher than	C=0 ET Z=0	non signés
DBLE	Decrease and Branch Less or Equal	Z=1 OU NV	signés
DBLS	Decrease and Branch Lower or Same	C=1 OU Z=1	non signés
DBLT	Decrease and Branch Less Than	N=~V	signés
DBMI	Decrease and Branch MInus	N=1	non signés
DBPL	Decrease and Branch PLus	N=0	non signés
DBVC	Decrease and Branch V Clear	V=0	
DBVS	Decrease and Branch V Set	V=1	
DBRA	Decrease and BRanch Always	-	
DBF(DBRA)	Decrease and BRanch never terminate	-	
DBT	Decrease and BRanch always terminate	-	

Cette instruction a donc deux opérandes. Le premier est le registre de données qui devra être décrémenté de 1 à chaque fois que l'instruction est exécutée. Le deuxième donne le

déplacement par rapport au PC courant. Dans la pratique, on spécifie uniquement une étiquette (ou un label). L'assembleur changera le nom de votre label par une adresse, celle-ci aura comme taille un octet ou un mot suivant la distance du saut. Vous pouvez ne rien spécifier car le compilateur choisira toujours la bonne taille.

Le branchement s'effectue si la condition n'est pas vérifié et en même temp le contenu du registre spécifié n'est pas -1. Le comportement de cette instruction peut être défini comme suit

```
DBcc  Dn,<label> :   IF cc TRUE THEN EXIT
                     ELSE
                          BEGIN
                               [Dn] := [Dn]-1
                               IF [Dn]=-1 THEN EXIT
                               ELSE  [PC] := label
                               END IF
                          END
                     END IF
                     EXIT
```

Cette instruction est très utilisée dans le cadre des boucles dont le compteur est le registre à décrémenter. Dans ce cas il faut deux conditions pour terminer la boucle.

Remarque : Si la condition est vérifiée, le branchement ne s'effectuera pas.
Remarque : Le cas le plus utilisé est l'instruction DBF qui peut être remplacée de manière équivalent par DBRA.

Taille de registre
Mot

Indicateur
Inchangés

Syntaxe
DBcc Dx, <label>
DBcc Dx, <label>

4.4.5. Opérations diverses

Ci-dessous quelques instructions qui sont très souvent utilisées sont présentées. Pour avoir des détailles sur chacune, je vous invite d'aller consulter les manuels du 68000.

CMP	Compare	(Destination)-(source)
CMPA	Compare address	(Destination)-(source)
CMPI	Compare immediate	(Destination)-immediate data
CMPM	Compare memory	(Destination)-(source)
TST	Test an operand	(Destination) tested → CCR
BTST	Test a bit	~(<bit number>) OF (destination) → Z
ASL, ASR	Arithmetic shift	(Destination) shifted by <count> →(destination)
LSL, LSR	Logical shift	(Destination) shifted by <count> →(destination)
SWAP	Swap register halves	Register (31:16) ↔ register (15:0)

4.5. Pile

La pile est gérée exactement de la même manière que notre processeur vu au chapitre précédent. Ici, c'est le registre A7 qui joue le rôle du pointeur de pile.

Remarque : C'est à la charge de programmeur d'initialiser ce registre avant toute utilisation directe ou indirecte de la pile.

Pour empiler une donnée on utilise l'instruction suivante :
MOVE.x X.s,-(A7)

Et pour dépiler :
MOVE.x (A7)+,X.s

Remarque : On peut aussi mettre sur la pile plusieurs registre en utilisant l'instruction MOVEM.

4.6. Sous programme

Un sous programme est une partie d'un programme qui peut être appelé à partir de n'importe quel point du programme. C'est une façon d'économiser la mémoire car cette partie de code peut être appelée autant de fois que l'on désire sans la répéter. L'autre utilité de sous programme c'est qu'il rend le programme plus lisible et plus modulaire.

Le passage vers un sous programme ressemble à un branchement sauf qu'ici, il faut pouvoir revenir. C'est la raison pour laquelle, le processeur met en mémoire (sur la pile) l'adresse de retour (plus précisément le PC). Il existe deux types d'appel à un sous programme exactement comme c'était le cas pour le branchement : BSR (Branch to Sub Routine) et JSR (jump to Sub Routine). La première effectue un adressage relatif tandis que la seconde est absolue.

On peut avoir des sous programmes imbriqués. A chaque fois qu'il y a un appel à un sous programme, le PC est mis sur la pile et à chaque fois qu'on arrive à la fin de la routine d'interruption, l'adresse de retour est dépilée dans PC.

4.6.1. BSR

Branch to SubRoutine : Dépose l'adresse de la prochaine instruction à exécuter (le contenu du registre PC) sur la pile et effectue un saut au label spécifié pour exécuter un sous-programme.

C'est un saut inconditionnel comme nous le faisions avec les instructions BRA ou JMP à l'exception près que pour retourner du sous-programme il faut utiliser l'instruction RTS qui ressaute à l'adresse déposée sur la pile précédemment.

Le compilateur remplacera le nom de votre label par une adresse, celle-ci aura comme taille un octet ou un mot suivant la distance du saut. Vous pouvez ne rien spécifier car le compilateur choisira toujours la bonne taille.

Taille des données	**Indicateurs**
Octet / Mot	Inchangés

Syntaxe

BSR.B <label>
BSR.W <label>

4.6.2. JSR

Jump to SubRoutine : Fonctionne de la même façon que l'instruction JMP, mais dépose au préalable l'adresse de la prochaine instruction à exécuter (le contenu du registre PC) sur la pile et effectue un saut au label spécifié pour exécuter un sous-programme.

C'est un saut inconditionnel comme nous le faisions avec les instructions BRA ou JMP à l'exception près que pour retourner du sous-programme il faut utiliser l'instruction RTS qui ressaute à l'adresse déposée sur la pile précédemment.

Mode d'adressage
(An), d16(An), d8(An, Xn.s), adr, d16(PC), d8(PC, Xn.s)

Indicateur
Inchangés

syntaxe
JSR <adresse>

4.6.3. RTS

Pour retourner d'un sous programme vers le programme principal, il suffit de mettre l'instruction RTS à la dernière ligne du sous programme. Ce qui se passe c'est que le PC est chargé par l'adresse qui se trouve en haut de pile.

syntaxe
RTS

4.7. Conclusion

Ce chapitre a présenté une partie du modèle du 68000 pour le programmeur. La partie concernant le circuit (hardware) reste à aborder. Sur la partie programmation, nous n'avons donné qu'une introduction. Le chapitre suivant est consacré au langage assembleur qui nous aide à commencer de faire des programmes de manière propre. Nous nous consacrerons ensuite, au développement des instructions plus approfondies telles que les sous programmes et les interruptions. Le circuit du microprocesseur avec un minimum de périphériques sera l'objet du chapitre suivant.

5. ASSEMBLEUR DU 68000

L'assembleur est un programme qui produit les codes opératoires compréhensibles par le microprocesseur à partir d'un fichier contenant les mnémoniques. La sortie de l'assembleur est donc un fichier binaire à charger dans une mémoire. Cette mémoire, souvent de type EPROM, contiendra donc le programme à exécuter par le microprocesseur. Au moment de la mise sous tension du système, le 68000 exécutera ces codes opératoires un par un.

Afin de permettre à l'assembleur d'avoir une bonne compréhension de nos codes en mnémonique, il faut respecter un certain nombre de règles.

5.1. Mise en mémoire

A chaque mnémonique correspond un code opératoire qui peut être codé sur un ou plusieurs mots. Ces mots sont écrits dans la mémoire à partir de l'adresse zéro. Par contre, le programmeur peut choisir l'adresse de début de mémoire de son programme. Il faut donc informer le programme d'assembleur de supposer que ces codes seront écrits à une adresse donnée. Cette information est passée à l'assembleur moyennant la directive "ORG".
Exemple :

```
ORG   $1000
MOVE.B      #10,D0
…
```

5.2. Style d'écriture

Chaque ligne de programme assembleur a 4 champs : label, commande, opérande, commentaire. Ces champs sont séparer par au moins un espace. Le label est écrit obligatoirement dès la première colonne.
Exemple :

Label	Commande	Opérande(s)	commentaire
BOUCLE	LEA	TABLE(PC),A3	Récupérer l'adresse de la table de valeurs

Les commentaires sont à écrire intelligemment ! Des commentaires évidents sont à éviter.
Exemple de mauvais commentaire :

```
DEBUT       ADD.W       #1,D1       incrémenter D1 de 1
```

Par contre, on peut dire par exemple :

```
DEBUT       ADD.W       #1,D1       incrémenter le compteur de boucle
```

On peut avoir une ligne complète de commentaire, dans ce cas il faut obligatoirement une étoile sur la première colonne.
Exemple :

```
* On Calcule maintenant la somme de convolution.
```

5.3.Etiquette

Quand on écrit un programme, on ne sait pas quelle instruction va être sauvegardée à quelle adresse mémoire. Ceci est gênant surtout pour écrire des instructions de branchement, et aussi pour adresser des données sauvegardées en mémoire. Pour se débarrasser de cette difficulté, on est amené à utiliser des étiquettes (labels). L'assembleur qui génère les codes opératoires, remplace ces étiquettes par les adresses correspondantes.

Exemple :

```
          . . .
          LEA  TABLE(PC),A0
          CMP  #0,D0
          BLT  NEGAT
          BRA  POS
NEGAT     MOVE . . .
          . . .
TABLE     DC.b 'il fait beau !'
```

5.4.Définir des constantes

Pour augmenter la lisibilité de votre programme, il est préférable de remplacer les constantes par des chaînes de caractères. Par exemple, si nous avons à traiter un tableau de taille 50, on peut définir une constante "TAILLE" dans le programme et demander à l'assembleur de remplacer cette chaîne de caractère (TAILLE) par 50 à chaque fois qu'il la voit. La directive associée à cette opération est "EQU".

Exemple :

```
TAILLE    EQU       50
ADR_X     EQU       $1000
…
          MOVE.W    #TAILLE,D0
          MOVE.W    ADR_X,D1
```

5.5.Réservation de mémoire

Dans beaucoup de cas, on souhaite réserver ou réserver et initialiser une partie de mémoire. Ceci se fait quand le programme assembleur est en train de mettre les codes opératoires en mémoire. Pour signaler à l'assembleur la taille ou les valeurs à mettre en mémoire, on utilise les directives DS et DC. Ces directives sont accompagnées par un des attributs communs b, w ou l.

Exemple :

```
SINUS     DC.B      0,90,127,90,0,-90,-127,-90
RESUL     DS.W      4   Réserve 4 mots sans les initialiser
```

5.6.Exemple

Ecrire un programme pour additionner dix nombres sur 16 bits donnés dans un buffer en mémoire. Les codes opératoires sont à mettre en mémoire à partir de l'adresse $1000.

```
*************************************
*        la somme de 16 mots          *
*************************************
SIZE  EQU        10              La taille du tableau
      ORG        $1000           Tout ce qui suit sera mis à
                                 partir de l'adr 1000
DEBUT LEA        BUFF(PC),A0     Adresse de buffer est mise
                                 dans A0
      LEA        RES(PC),A1      Adresse du résultat est mise
                                 dans A0
      MOVEQ      #SIZE-1,D0      Init le compteur de boucle
      MOVE.W     (A0)+,D1        La première donnée dans D1
BCL   ADD.W      (A0)+,D1        Accumulation
      DBRA       D0,BCL
      MOVE.L     D1,(A1)         Sauvegarde du résultat
      TRAP       #0              pour terminer le programme
* On réserve et initialise le tableau de taille 10 mots
BUFF  DC.W       1,2,3,4,5,6,7,8,9,10   Les nombres à additionner
RES   DS.W       1               Réservation de mémoire pour le
                                 résultat
```

Ci-dessous on peut apprécier le travail de l'assembleur. La première colonne est le numéro de ligne. La deuxième colonne est l'adresse de la mémoire dans laquelle le code opératoire est mis. La troisième colonne présente le code opératoire correspondant à l'instruction qui se trouve sur la cinquième colonne. On remarque les points suivants :
Ligne 4 : Cette ligne est une directive d'assemblage : il n'y a aucun code opératoire exécutable par le processeur.
Ligne 5 indique que les codes vont être mis en mémoire à partir de l'adresse 1000; il n'y a toujours pas de code opératoire.
Ligne 6 : L'adresse du buffer est mise en registre A0. Examinons le code opératoire. 41FA est le code de l'instruction LEA x(PC),A0, le mot suivant donne la valeur de x qui est égale à 14 (pourquoi 14 ?). Une fois cette instruction est exécutée, quel est le contenu de A0 ?
Ligne 11 : Le deuxième mot de l'instruction DBRA D0,BCL est égal à FFFC. Pourquoi ?
Remarque : Si on change la directive "ORG $1000" en "ORG $2000", le programme continue à fonctionner correctement. C'est dire que ces codes fonctionnent indépendamment de leur emplacement mémoire. Ceci est grâce au fait que nous avons utilisé l'adressage relatif PC. Par contre si on remplace la ligne 6 par MOVEA.L #BUFF,A0, les codes ne fonctionnent que pour l'adresse $1000. Le code ne sera plus indépendant de l'adresse d'implantation du programme (assurez-vous de bien appréhender ce principe).

```
1*************************************
2*        la somme de 16 mots          *
3*************************************
4           0000000A     SIZE:     EQU       10
5  00001000                        ORG       $1000
6  00001000 41FA0014     DEBUT:    LEA       BUFF(PC),A0
7  00001004 43FA0024               LEA       RES(PC),A1
8  00001008 7009                   MOVEQ     #SIZE-1,D0
9  0000100A 3218                   MOVE.W    (A0)+,D1
10 0000100C D258         BCL:      ADD.W     (A0)+,D1
11 0000100E 51C8FFFC               DBRA      D0,BCL
12 00001012 2281                   MOVE.L    D1,(A1)
13 00001014 4E40                   TRAP      #0
14 00001016 000100020003 BUFF:     DC.W      1,2,3,4,5,6,7,8,9,10
           000400050006
           000700080009
           000A
```

```
15 0000102A 00000002     RES:      DS.W      1
16          00001000               END       $1000
```

Exercice :

1- Nous avons un fichier texte en mémoire à partir de l'adresse $2000. La fin de texte est marquée par un zéro.

a- Faire un programme pour compter le nombre total des mots (sachant que les mots sont séparés par un espace).
b- Faire un programme pour compter le nombre total des phrases (sachant qu'il est égal au nombre de points)
c- faire un programme pour transformer le texte en majuscule.

Corrigé 1-a :

```
      org         $1000
      LEA         $2000,A0
      LEA         RES(PC),A1
      MOVE.W      #1,D1
BCL   MOVE.B      (A0)+,D0
      BEQ         FIN
      CMP.B       #' ',D0
      BNE         BCL
      ADDQ.W      #1,D1
      BRA         BCL
FIN   MOVE.W      D1,A1
      TRAP        #0          pour arrêter le programme
RES   DC.W        0

      org         $2000
      DC.B        'Ceci est un test. Il fait beau. ok.',0
```

6. EXERCICES

1- Mettre la valeur $1245 à l'adresse $FFFF00 et la valeur $3467 à l'adresse suivante ($FFFF02).

2- Sauvegarder les valeurs (en octet) de 0 à 99 dans les mémoires allant de $FFFF00 à $FFFF63.

3- La même question mais dans l'ordre inverse (d'abord 99 et à la fin 0)

4- Lisez le contenu (en mot) de la mémoire $FFFF00. S'il est positif ne le touchez pas, sinon inversez-le.

5- Un tableau d'octets signés de taille 10 octets est déjà dans la mémoire à partir de l'adresse $FE0000. Calculez la somme des éléments de ce tableau et stockez le résultat sur un octet à l'adresse $FF0000.

6- Le même tableau que la question précédente mais pour éviter le dépassement, on fera le calcul de la somme sur 16 bits. On stockera le mot résultant à l'adresse $FF0000.

7- Le même tableau que la question précédente, on souhaite trouver l'élément le plus petit et le sauvegarder à l'adresse $FF0000.

8- Le même tableau que la question précédente, multipliez chaque élément du tableau par 4 et sauvegardez le tableau des mots (pour éviter le dépassement) résultant à partir de l'adresse $FFF000.

9- Deux tableaux des mots longs de taille 100 sont déjà dans la mémoire. Le premier commence à partir de l'adresse $FF0000, et le seconde à partir de l'adresse $FF1000. Calculez la somme élément par élément de ces deux tableaux et stockez le tableau résultant dans la mémoire à partie de l'adresse $FF2000.

10- Les mêmes deux tableaux mais avec les mots au lieu des mots longs. Calculer le produit scalaire de ces deux vecteur et mettre le résultat à l'adresse $FF2000.

Dans les exercices suivants le but est d'écrire un sous programme qui récupère les deux mots déjà placés sur la pile, et qui les traite, et qui sauvegarde le résultat sur la pile. Le programme principal est le suivant

```
LEA         $FFFFF0,A7
MOVE.W      #15,-(A7)         Mise sur la pile du premier paramètre
MOVE.W      #-89,-(A7)        Mise sur la pile du seconde paramètre
BSR         CALCUL            Sous programme de calcul
MOVE.(?)    (A7)+,D0          Récupération du résultat
…
```

11- Somme de ces deux paramètres (résultat sur 16 bits)
12- Somme de ces deux paramètres (résultat sur 32 bits)
13- Produit de ces deux paramètres (le résultat est sur 32 bits)
14- Minimum des deux

15- Maximum des deux

16- Paramètre 1 au carré plus paramètre 2 au carré, c'est-à-dire ($a^2 + b^2$) (le résultat est un mot long)

17- Commentez le programme ci-dessous :

```
        LEA   PILE(PC),A7
        …
        TRAP  #0
        DS.W  100
PILE    DS.W  1
```