Assembleur et Code Digital

1) INTRODUCTION ...

Avant de commencer à écrire des programmes en langage-machine, il va être nécessaire d'introduire un certain nombre de notions et de définitions :
Tout d'abord, nous allons définir nos objectifs, le rôle et l'utilité de ce cours et du langage-machine en général ...

Les Objectifs :

A l'issu de ce cours, tout étudiant saura :
- Choisir le langage le plus adapté à la résolution d'un problème ou à la traduction d'un algorithme complexe...
- Ecrire des programmes en Assembleur 68000, adaptés aux contraintes des Cahiers Des Charges, et aux divers matériaux proposés lors des Travaux Pratiques ...
- Ecrire des programmes en Assembleur 80286, tournant de manière autonome sous DOS, sur un micro-ordinateur compatible IBM-PC ...
- Ecrire des modules assembleurs, interfacés avec des programmes principaux écrits dans un langage évolué.

De manière générale, l'étudiant aura ajouté à l'éventail de ses outils, l'assembleur sur des machines à base de 68000 ou de 80286 ...

Qu'est ce que le langage-machine ?

C'est le seul langage qui soit directement compréhensible par un microprocesseur. Il est composé d'une suite d'octets localisés en mémoire et dont le contenu, binaire, constitue une suite de codes significatifs pour le microprocesseur.
Il est nécessaire de bien comprendre que quel que soit le langage utilisé en programmation évoluée (C , ADA, Programmation Structurée, Programmation Objets) , les outils de programmation finiront toujours par fournir à notre microprocesseur des codes langage-machine...

Mais pourquoi consacrer un cours à l'assembleur ?

Que d'inconvénients au langage d'assemblage !!! La liste est longue et parfois rédhibitoire . Rares sont les programmeurs qui envisagent avec sérénité la programmation d'une application en langage-machine ...
Citons, sans quantifier ces défauts:

- Lenteur du développement .
- Complexité de la programmation .
- Portabilité nulle .

Il y a ne serait-ce que 10 ans (presque la préhistoire pour nous), une application digne de ce nom ne pouvait cependant qu'être écrite en langage-machine; seuls des ordinateurs de taille respectable (nécessitant une salle blanche réfrigérée de 40 m^2) permettaient l'utilisation de langages comme le FORTRAN . En effet, les caractéristiques des machines rendaientt alors les professionnels dans l'obligation d'accepter les inconvénients du langage-machine ...
Ceux-ci étaient d'ailleurs minimisés par le fait que les performances des machines limitaient l'imagination débordante des programmeurs géniaux de cette époque ... En effet, il n'est guère possible de se perdre dans les méandres d'un programme quand on ne dispose que de 8 Koctets de mémoire, par exemple ...
Les utilisateurs refusaient (comme maintenant d'ailleurs) les temps de réaction quasi-infinis des applications écrites en BASIC par exemple ...

Ainsi, les programmeurs qui maîtrisaient parfaitement le langage-machine étaient-ils alors très prisés sur le marché du travail.
Cependant, au fil des ans, les machines sont devenues de plus en plus performantes, et les outils de programmation sont devenus si performants que l'écart entre les langages évolués et le langage-machine a fortement régressé...
La notion d'application-client quitte peu à peu le domaine du langage-machine pour réapparaître dans des applications ultra-évoluées tournant par exemple dans des environnements comme Windows 3.1...
Parallèlement, les développements en assembleur sont rendus plus simples, grâce à des outils eux-aussi plus évolués... La concurrence entre les produits les plus performants imposent aux créateurs d'applications d'introduire des modules assembleurs rendant leurs applications plus rapides donc plus compétitives vis à vis des autres produits ...
Que faire ??? Que choisir ???
La réponse est éminemment variable, dans le temps, et en fonction de critères qui peuvent devenir aussi évasifs que l'âge du capitaine ...

Critères de choix :

Avantage	Inconvénient	Ratio	Note
	Lenteur du développement	20	Augmente
Vitesse d'exécution		5	Diminue
Compacité du code		4	Constant
Complexité de programmation		2	
Proche du Hardware			
	Aucune Portabilité		
Optimisation des Algorithmes			
	Fiabilité et Mise au point		

Langage Communication Information ...

L'information est définie comme un élément de connaissance qui peut être transmis, archivé, partagé.
La communication peut être définie comme l'échange ou la propagation des informations; cette communication peut utiliser des supports très divers comme la parole, utilisée par notre civilisation, mais aussi les gestes, utilisés par les malentendants, la musique ou la peinture utilisés par les artistes, l'écriture, la lumière...
Le langage constitue en l'ensemble des règles et définitions qui régissent la communication.
Ainsi, toute langue comportera des définitions, ensemble de concepts ou mots qui forment le vocabulaire de notre "langue", et des règles de grammaire qui définiront l'architecture, la structure des communications...

La langue que nous désirons étudier est pour nous d'une complexité extraordinaire... Son vocabulaire est réduit à deux mots...

0 et 1

La grammaire et la syntaxe de notre langue permettent un nombre de combinaisons quasi-infini... et d'une complexité extraordinaire...

01001100 01010011 00001111 11011000 11100010 10101010 11010001 00010111 10000111

Une telle complexité nous oblige à changer de langage...

Dans un premier temps, nous parlerons en Hexadécimal, plutôt qu'en Binaire. Ceci ne présente aucune difficulté au niveau du microprocesseur tant est grande la corrélation entre binaire et hexa. L'époque où cette solution était réellement utilisée n'est pas si lointaine ...
Dans un second temps, nous allons utiliser le langage d'assemblage qui utilise une forme beaucoup plus proche de ce que nous connaissons (utiliser le mot ADD pour effectuer une addition ne devrait normalement pas dépasser nos possibilités) mais nécessitant l'usage de traducteurs qui nous éloignent déjà de la machine... Les mots de Assembleur et de Linker viennent de faire leur apparition dans notre environnement.

2) Rappels de Numération

Toute information est donc stockée, dans un ordinateur, sous forme de BIT (abréviation de l'expression BInary digiT).
Ces informations sont regroupées par blocs de 8 bits nommés OCTETS (BYTE en anglais) ou par blocs de 16 bits nommés MOTS (WORD en anglais)
La représentation de ces informations peut se faire, comme à l'habitude en notation décimale, mais l'introduction de bases de numération différentes pourra être nécessaire ...

Rappelons ci-dessous quelques valeurs dans les bases les plus utilisées:

(Décimal) **10**	(binaire) **2**	**4**	(octal) **8**	(hexadécimal)**16**
0	0	0	0	0
1	1	1	1	1
2	10	2	2	2
3	11	3	3	3
4	100	10	4	4
5	101	11	5	5
6	110	12	6	6
7	111	13	7	7
8	1000	20	10	8
9	1001	21	11	9
10	1010	22	12	A
11	1011	23	13	B
12	1100	30	14	C
13	1101	31	15	D
14	1110	32	16	E
15	1111	33	17	F
16	10000	100	20	10
64	1000000	1000	100	40
256	100000000	10000	400	100
1024	10000000000	100000	2000	400
65536	10000000000000000	100000000	200000	10000
1048576	100000000000000000000	10000000000	4000000	100000
16777216	1000000000000000000000000	1000000000000	100000000	1000000
268435456	1 suivi de 28*0	1 suivi de 14*0	2000000000	10000000
4294967296	1 suivi de 32*0	1 suivi de 16*0	40000000000	100000000

Ce tableau montre des équivalences évidentes entre la base 2 et ses bases multiples.
Ainsi, un digit octal (2^3=8) est équivalent au groupement de 3 digits binaires, comme un digit hexadécimal (2^4=16) est équivalent au groupement de 4 digits binaires...

Sans empiéter sur le cours de mathématiques, on peut très simplement résumer les règles de changement de base par les formules suivantes :

$$(b_k b_{k-1} \ldots b_2 b_1 b_0) = \sum_{i=1}^{k} b_i \times n^i = b_0 + n \times (b_1 + n \times (b_2 + \ldots + n \times (b_{k-1} + n \times b_k) \ldots))$$

Ces formules permettent de passer de n'importe quelle base à n'importe quelle autre base, mais une bonne compréhension de la base d'arrivée est nécessaire... Ainsi pour transformer un nombre de base 12 en base 7, notre esprit très limité devra-t-il passer par la base 10 ... Ceci n'est d'ailleurs pas vrai pour des bases comme 2 et 16, ce qui va quand même simplifier les rapports (déjà bien électriques) entre nous et l'ordinateur ...

3) Représentation des Données ...

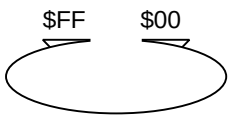
Un microprocesseur n'est pas un élément autonome. Pour fonctionner, il a besoin d'éléments divers parmi lesquels on trouve la mémoire, lieu de stockage de toutes les données que manipule à un instant le microprocesseur...

3.1 Représentation du programme...

Les diverses instructions d'un programme en langage-machine sont stockées en mémoire sous une suite d'octets (en nombre pair sur 68000). Chaque instruction est constituée d'un ensemble de 2 à 10 octets stockés dans la mémoire de notre micro-ordinateur...
Cette partie de la mémoire est la plus difficile à interpréter, puisque le codage fait correspondre à chaque combinaison de 0 et de 1, une instruction particulière. A priori, un désassembleur est nécessaire pour traduire les instructions de l'hexadécimal vers une forme assembleur plus compréhensible.
Notons que bien entendu, chaque microprocesseur a ses propres codes d'instructions rarement voire même jamais portables vers un autre microprocesseur (sauf cas particulier des familles ascendantes 68000-68010-68020-68030... ou 8088-8086-80186-80286-80386-80486) et que la traduction même adopte souvent des conventions différentes voir radicalement opposées... Ceci augmentera bien sûr la difficulté, dès que nous utiliserons 2 microprocesseurs différents ...

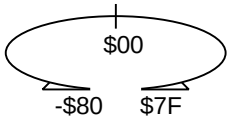
3.1 Données de type BYTE ...

On entend par donnée, tout emplacement recevant une valeur initialisée ou calculée. Du type de variable dépend la taille qui leur est allouée.
Les données de type BYTE occupent 8 bits en mémoire, permettant 2^8 combinaisons, soit 256 valeurs possibles.
Pour le processeur, le problème s'arrête là, mais pour nous, une interprétation différente des valeurs peut être faite. Il est possible pour le programmeur de considérer ses variables comme appartenant à l'ensemble mathématique **N** (Entiers Naturels , non-signés) ou **Z** (Entiers Signés) ...
Comme les nombres de 8 bits ne correspondent qu'à un sous-ensemble de N et Z, il nous faut maintenant définir les limites dans chacun de ces 2 cas :



Entiers Non Signés:
de 0 à 255

Entiers Signés
de -128 à +127

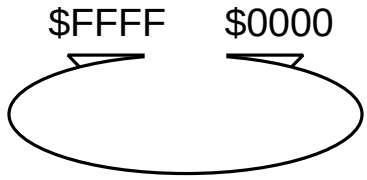


Quelle bizarrerie! on peut quasiment dire que 8 bits permettent de coder 256 valeurs qui s'étendent sur l'intervalle de -128 à +255...
Le bit de poids le plus fort est considéré comme un bit de signe, réduisant à 128 le nombre des possibilités, en positif comme en négatif...
La représentation des nombres signés sera abordé de manière complète dans un prochain paragraphe.
Le 68000 peut manipuler des données de type BYTE, il suffit pour cela de faire suivre chaque mnémonique manipulant un octet du suffixe .B comme par exemple :

```
MOVE.B    #$7C,D0
ADD.B     D0,D0
LSL.B     #2,D0
```

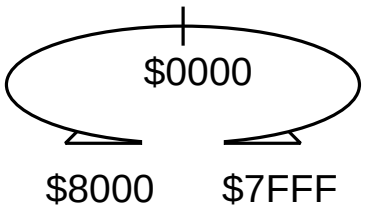
3.2 Les Données de type WORD ...

Les données de type WORD sont stockées sur 16 bits, soit deux octets consécutifs, dont le premier doit impérativement être situé à une adresse paire (cet inconvénient n'existera pas sous la même forme en 80286 ou il se traduira de manière beaucoup plus insidieuse par un ralentissement).
Ce type de données permet 65536 combinaisons que l'on peut encore une fois considérer comme signées ou non signées :



Entiers Non-Signés
de 0 à 65535

Entiers Signés
de -32767 à +32768

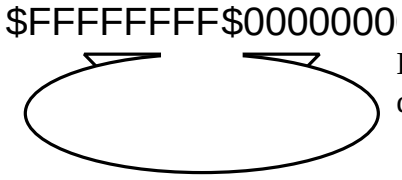


notez que de part sa structure électronique, le 68000 est particulièrement adapté au maniement des données de type WORD. Pour spécifier à l'assembleur qu'une instruction utilise des opérandes de type WORD, un mnémonique doit être suivi de .W comme dans:

```
MOVE.W    #$7F00,D2
ADD.W     D0,D0
ROL.W     #8,D0
```

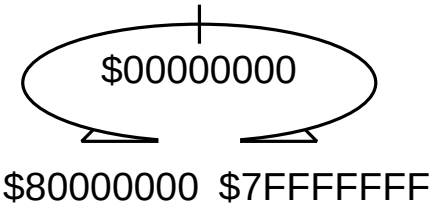
3.3 Les Données de type LONG WORD

Les données de type LONG WORD occupent 32 bits donc 4 octets en mémoire. De part sa structure, le 68000 ne disposant que de 16 fils pour recevoir des données, le processeur manipulera ce type de données en deux vagues, rendant les opérations plus lentes. Ce type de données autorise 2^{32} combinaisons, soit un peu plus de 4 milliards...



Entiers Non-Signés
de 0 à $4*10^9$

Entiers Signés
de $-2*10^9$ à $+2*10^9$



Les instructions travaillant sur des LONG WORD doivent être suivies du suffixe .L, c'est ce type de données qui le plus souvent nous permettra de spécifier au processeur des emplacements mémoires... En effet, sur 68000, les adresses ont une taille physique de 24 bits permettant de désigner 16 millions d'adresses différentes.

3.4 Représentation des Réels...

Les réels ne correspondent pas à des données manipulées directement par les microprocesseurs ; cependant, ce sont des données que l'utilisateur doit parfois manipuler dans ses applications, et il peut être souhaitable d'en connaître le format de stockage...
De plus les processeurs actuels sont souvent couplés, aujourd'hui, à des coprocesseurs arithmétiques, et ce de manière interne. C'est le cas du 68040 de MOTOROLA, ou du 80486 d'INTEL...
Plusieurs formats de stockage des réels ont été proposés dans les dernières années, mais le format IEEE semble s'être imposé sur le marché ...

Pour comprendre le format des réels, il va être indispensable de faire quelques rappels de numération. Vous avez l'habitude de manipuler des valeurs binaires à puissances positives... Mais que signifie un nombre binaire comme 1110,1010 qui introduit des puissances binaires négatives, dont vous trouverez quelques valeurs dans le tableau ci-dessous:

+1	2
0	1
-1	0,5
-2	0,25
-3	0,125
-4	0,0625
-5	0,03125
-6	0,015625
-7	0,0078125
-8	0,00390625
-9	0,001953125
-10	0,0009765625
-11	0,00048828125
-12	0,000244140625
-13	0,0001220703125
-14	0,00006103515625
-15	0,000030517578125
-16	0,0000152587890625
-17	0,00000762939453125

Comme le montre le tableau ci-dessous, la décomposition d'un nombre réel même simple en puissance de 2 pose des problèmes impossibles à résoudre à la main...

Essayez par exemple de coder la valeur décimale 0,1 en binaire ... Peut-être arriverez-vous rapidement à 0,00011001 mais qui a trouvé de tête 0,0001100110011001100110011001100110011001100110011 valeur périodique et donc ne tombant jamais exacte en binaire ...

Quelqu'un peut-il voir du premier coup d'oeil dans la série binaire 0,0101010101010101010101010101010101010101010101010... la valeur 1/3 de notre bonne base 10... Alors que 1/7 se représente par 0,001001001001001001001001001001001001001001001...

Quel casse-tête pour un microprocesseur!!!

Alors imaginez sa légitime angoisse quand vous tentez de faire exécuter l'opération 7,54634E-21 + 9,346E-23

Quel que soit le nombre réel, une chose est sûre, il est converti en une mantisse qui est comprise dans l'intervalle [0,5;1,0[et un exposant puissance de 2...
Ainsi, par exemple, le nombre 14,625 s'exprime en binaire 1110,1010 et est suivant ce qui vient d'être dit transformé en une mantisse valant 0,11101010 et une puissance de 2 qui vaut 4 dans notre exemple...
D'après ce qui vient d'être dit toute mantisse commence par 0,1... puisque supérieure ou égale à 1/2 et inférieure à 1. Ce 0,1 est pour les théoriciens de l'informatique comme une injure à la compacité du codage, aussi tous les systèmes l'éliminent-ils sans scrupule ...
On trouvera à son emplacement théorique un bit de signe. La mantisse étant toujours conservée en valeur absolue...
La puissance de 2 est stockée directement en mémoire après l'ajout d'un offset permettant de coder aussi bien les nombres très petits (à puissance négative) que les nombres très grands (à puissance positive)...
Ceci étant clair, on peut maintenant décrire plus exactement la constitution des formats de stockage IEEE des nombres réels, format FLOAT (occupant 4 octets) et le format DOUBLE (occupant 8 octets).

Format FLOAT: - 1 Bit de signe
- 8 Bits d'Exposant (Offset $7E)
- 23 Bits de Mantisse

Exemple: 14.625_{10} = 1110.1010_2 soit (0,11101010 puissance 4 ($82 avec l'offset))
0 10000010 11010100.... et en Hexa $41 6A 00 00

Exemple: 2.000_{10} = 10_2 soit (0,10000000 puissance 2 ($80 avec l'offset))
0 10000000 00000000.... et en Hexa $40 00 00 00

Exemple: 1000.5_{10} = $1111101000,1_2$ soit (0,1111101000 puissance 10 ($88 avec l'offset))
0 10001000 111101000100.... et en Hexa $44 7A 20 00

Format DOUBLE: - 1 Bit de signe
- 11 Bits d'Exposant (Offset $3FE)
- 52 Bits de Mantisse

Exemple: 14.625_{10} = 1110.1010_2 soit (0,11101010 puissance 4 ($402 avec l'offset))
0 10000000010 11010100.... et en Hexa $40 2D 40 00 00 00 00 00

Exemple: 2.000_{10} = 10_2 soit (0,10000000 puissance 2 ($400 avec l'offset))
0 10000000000 00000000.... et en Hexa $40 00 00 00 00 00 00 00

Exemple: 1000.5_{10} = $1111101000,1_2$ soit (0,1111101000 puissance 10 ($408 avec l'offset))
0 10000001000 111101000100.... et en Hexa $40 8F 44 00 00 00 00 00

Notez que le nombre 0 n' est pas codable par ce système, aussi a-t-il été décidé que la valeur 000 de l'exposant correspondrait au réel 0. La plupart des systèmes mettent la mantisse également à 0 ...
Notez également que l'exposant théorique maximal ($FF en Float et $7FF en Double) n'est pas utilisés dans le format IEEE...

Les principales caractéristiques de ces deux types de codage sont résumées ci-dessous:

	DOUBLE	FLOAT
Valeur Mini	1.39067e-309	2.35099e-38
Valeur Maxi	1.79769e+308	1.70141e+38
Différenciation	1.192e-07	2.222e-16

4) Arithmétique...

Ce paragraphe traite des divers problèmes indispensables à la bonne compréhension du fonctionnement interne d'un microprocesseur en général et du 68000 en particulier. En effet si un microprocesseur compte indéniablement mieux que nous, plus vite et sans erreur, nous autres utilisateurs, l'accusons parfois de se tromper, alors que c'est notre compréhension des résultats qui est erronée...

4.1 Changement de Signe - Complément à 2 -

Nous avons déjà vu que l'on introduisait la notion de bit de signe pour représenter le signe d'un nombre. Ce bit de signe est toujours le bit de poids fort d'un nombre, mais changer la valeur du bit de signe d'un nombre ne suffit pas à en changer le signe...
En effet, si cela suffisait :

+7	en binaire	00000111	
-5	en binaire	10000101	
+2	devtrait être	10001100	qui est -12

ATTENTION !!!
Cet Exemple est volontairement FAUX

Cette convention ne peut qu'être erronée puisqu'une simple addition binaire ne fonctionne pas correctement. En fait la technique à employer est le complément à 2.
Le complément à 2 consiste en une inversion logique (ou complément à 1) suivie d'une incrémentation (d'où le nom sans autre signification de complément à 2)...

En ce cas dans notre exemple précédent:			
+5	en binaire		00000101
	inversion logique		11111010
	incrémentation		+ 1
-5	devient ainsi		11111011
Notre Opération devient:			
+7	en binaire		00000111
-5	en binaire		11111011
+2	fait bien	(1)	00000010

Dans l'exemple, tout se passe maintenant fort bien. Notez cependant une retenue du huitième bit vers l'extérieur.
Cette retenue ne figure bien sûr pas dans notre octet, mais le microprocesseur ne l'oublie pas pour autant. Il mémorise cette retenue dans un indicateur (flag) du nom de CARRY (C) ...
Nous aurons par la suite bien d'autres occasions de reparler des indicateurs.

4.2 Extension de Signe.

Le 68000 manipule quasi-indifféremment des données sur 1, 2 ou 4 octets; on peut donc se poser la question de savoir ce qui se passe lorsque l'on désire passer d'un format de données à un autre...

Passer d'un format à un format plus petit ne pose aucun problème sous réserve que le résultat reste valide dans le nouveau format...

$00006745.L	devient	$6745.W	... Exact Signé	... Exact Non-Signé
$00020000.L	devient	$0000.W	... Faux Signé	... Faux Non-Signé
$00009000.L	devient	$9000.W	... Faux Signé	... Exact Non-Signé
$FFFF8000.L	devient	$8000.W	... Exact Signé	... Faux Non-Signé

Passer d'un format à un format plus grand ne pose aucun problème si la donnée manipulée est considérée comme non signée puisqu'il suffit de rajouter des 0 en tête du nombre (la technique utilisée pour ce faire dépendra énormément du processeur)... Cependant s'il s'agit d'un nombre considéré comme signé, c'est le bit de signe qui doit être reproduit un certain nombre de fois en tête du nombre, ce qui peut être vérifié en utilisant le complément à 2.
Cette opération est en général assumée par une instruction spécifique du processeur (EXT en 68000, CBW et CWD en 8086).

4.3 Contrôle des résultats en Non-Signé.

Tout microprocesseur fournit au programmeur un certain nombre de signaux affectés en réponse à une opération arithmétique. Ces signaux sont des indicateurs à logique positive prenant les valeurs 0 ou 1, et occupant donc un bit dans le microprocesseur où ils sont regroupés dans un registre de flags nommé sur 68000, CCR (Code Condition Register de 8 bits) ou SR (Status Register de 16 bits qui inclut CCR).

Parmi ces bits, l'un a pour rôle de surveiller la validité du résultats des opérations arithmétiques sur des nombres non signés: C'est le Carry noté C ...

Carry est l'équivalent d'une retenue du bit de poids fort d'un résultat vers l'extérieur...
Le Carry est positionné lorsque, à la suite d'une opération, un résultat dépasse la capacité de stockage désirée ou au contraire devient négatif...

Exemples:

Mettez en évidence la retenue de b7 vers l'extérieur, grâce au binaire.

130	soit	$82	65	soit	$41
+ 127	soit	+ $7F	- 80	soit	- $50
257		$ (1) 01	-15		$ (1) F1
Dans les 2 cas, Carry est mis à 1					

4.4 Contrôle des résultats en Signé.

Le second des deux exemples fournis ci-dessus, s'il est faux en non-signé, se révèle juste en Signé...
L'indicateur de Carry n'est donc pas utilisable pour vérifier la validité du résultat d'une opération qui concerne des nombres signés. C'est donc un autre indicateur qu'il va nous falloir utiliser, il s'agit du bit de débordement nommé encore oVerflow, noté V en 68000.
Malheureusement , si la définition du Carry était évidente, celle de l'overflow est plus mystérieuse, et nous allons essayer de la découvrir à travers plusieurs exemples...

64	=	01000000
+ 65	=	01000001
- 127		10000001

Une retenue interne s'est produite du bit 6 vers le bit 7. Cette retenue s'est opérée vers le bit de signe. Le résultat est devenu négatif par "accident".
V est défini comme une retenue de b6 vers b7...

- 1	=	11111111
+ - 1	=	11111111
- 2		11111110

Résultat correct alors qu' une retenue s'est bien produite du bit b6 vers b7. Cependant une deuxième erreur de signe s'est produite, puisque l'on a également eu une retenue de B7 vers l'extérieur (qui a mis C à 1)... Ces deux erreurs se sont annulées fournissant un résultat exact. D'où une nouvelle définition de V = retenue de b6 vers b7 mais pas de Carry ...

- 64	=	11000000
+ - 65	=	10111111
+ 127		01111111

Mais ce troisième exemple invalide notre théorie puisqu' il n'y pas de retenue de b6 vers b7, alors que le résultat est faux...
Carry seul constitue une condition d'overflow...

D'où notre définition : $V = (b_6 \text{ a } b_7) \oplus Carry$

Le débordement se produit:
- lors de l'addition de grands nombres positifs
- lors de l'addition de grands nombres négatifs
- lors de la soustraction d'un grand nombre positif à un grand nombre négatif
- lors de la soustraction d'un grand nombre négatif à un grand nombre positif

4.5 Autres Indicateurs.

C et V sont fondamentaux puisqu'ils permettent de détecter les erreurs de calculs induites par le format des nombres, et il est vital de comprendre que le microprocesseur gère les deux indicateurs simultanément à chaque opération arithmétique. C'est à l'utilisateur de n'utiliser que celui de ces deux indicateurs qui correspond au choix qu'il avait fait pour les opérateurs de départ... Nous testerons C ou V suivant que nous considérons les opérateurs comme non-signés ou signés...

Cependant un microprocesseur rend bien d'autres services grâce à d'autres indicateurs, qui sur le 68000 sont:

Z	- Zero Flag	: Mis à 1 quand le résultat d'une opération vaut 0. Attention à cette interprétation inversée du flag Z.
N	- Negative Flag	: Recopie du bit de signe d'un résultat, ce flag n'a d'intérêt, bien entendu, que si on travaille sur des nombres non-signés.
X	- Extend Flag	: Quasi équivalent au Carry sur 68000, plus particulièrement dédié aux calculs arithmétiques, alors que le Carry possède d'autres caractéristiques supplémentaires...

D'autres microprocesseurs fourniront des indicateurs supplémentaires, ou possédant des définitions légèrement différentes...

5) Exercices

Transcrire en octal et en hexadécimal sur 16 bits (format WORD) les nombres suivants, en indiquant ceux dont la transcription n'est valable qu'en signé ou en non-signé:

1056	donne	@2040	octal	$420	hexadécimal	Signé	Non-Signé
16384	donne		octal		hexadécimal	Signé	Non-Signé
32768	donne		octal		hexadécimal	Signé	Non-Signé
-1264	donne		octal		hexadécimal	Signé	Non-Signé
-35000	donne		octal		hexadécimal	Signé	Non-Signé

Effectuer la transcription hexadécimale des nombres négatifs suivants (en utilisant le complément à 2) au format 16 bits:

-1024	donne	$FC00	-16384	donne
-23130	donne		-1	donne
-1256	donne		-76	donne
-32768	donne		-65536	donne