**Code complément à 2** (Ch 2 p.12 à 14)

Pour passer du décimal au binaire Cà2 :

1. Si le nombre est positif, on applique la méthode habituelle. Si le nombre est négatif :
2. On commence par traduire la valeur absolue du chiffre en binaire.
3. On inverse tous les bits et on additionne le résultat à 1.

Pour le contraire, il suffit d’écrire le mot en binaire et on applique les opérations habituelles.

S’il y a plusieurs additions, il se peut qu’il y ait plus d’une retenue. On continue donc d’additionner tant qu’il n’y a plus de retenue.

Ex : 0001 + 0001 + 0011 + 0001 = 01**1**0 (3 retenues)

* Multiplication

L’idée est de faire additions au lieu de multiplier. Multiplié **multiplieur**

On additionner le multliplié quand le bit est à , si le bit est à on additionne

1. : Opération normale
2. : On fait le Cà2 sur les 2 valeurs, ensuite on fait l’opération normale
3. : On étend les chiffres en rajoutant bits.   
   Le résultat est sur

Exemple :

WRONG ! Sign extended:

0011 (3) 0000 0011 (3)

x 1011 (-5) x 1111 1011 (-5)

------ -----------

0011 00000011

0011 00000011

0000 00000000

+ 0011 00000011

--------- 00000011

0100001 00000011

not -15 in any 00000011

representation! + 00000011

------------------

1011110001

--------

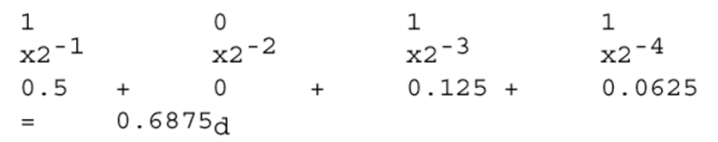
take the least significant 8 bits 11110001 = -15

* Division

On additionne le Ca2 du diviseur (soustrait le diviseur) tant que le résultat est au mieux

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 ÷ 3  =  2 remainder 1 |  | 0000 0111 | = | +7 |  | 0000 0100 | = | +4 |
| + 1111 1101 | = | -3 | + 1111 1101 | = | -3 |
| 0000 0100 | = | +4 | 0000 0001 | = | +1 (remainder) |

**Nombre binaire fractionnaire** (Ch 2 p.16 à 19)



Le point de fraction est placé comme on le souhaite. S’il n’y a pas de doc sur le programme, ça peut être compliqué pour une personne qui prend la relève !  
Pour une multiplication, il faut toujours placer la virgule à la bonne position pour avoir la bonne précision. On additionne les puissances pour avoir le bon nombre à gauche ou à droite du point.

Exemple :

\*.\*\*\*  
x \*\*.\*\*   
…  
\*\*\*.\*\*\*\*\*

Pour représenter où se trouve la virgule, on utilise la notation **Qnm**

\*\*\*\*.\*\*

**Erreurs** (Ch 2 p.24 à 30)

On retrouve différentes sortes d’erreurs :

* Dépassement (ou overflow)
* Saturation lors d’un dépassement
* Arrondis
* Troncatures
* Annulation (underflow)

Il n’y a des problèmes de dépassement que lorsque l’on additionne des nombres que positifs ou que négatifs. Il n’y a pas de problèmes lorsque l’on additionne + et –

1111

+0111

101101

En restant sur 4 bits on tombe bien sur (-1) + (7) = 6

Arrondis et troncature.

Lorsque l’on fait une troncature, le nombre fractionnaire est tronqué :

* au nombre entier positif juste inférieur si le nombre est **positif**
* au nombre entier négatif juste inférieur si le nombre est **négatif**

L’opération d’arrondi consiste juste à tronquer le nombre au-dessus du précédent en lui rajoutant la plus petite variation permise par notre format.

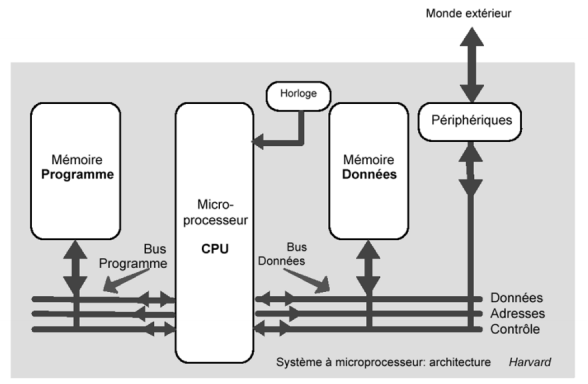
15.8 -1.25  
+ 0.5 + 0.5   
 16.3  **arrondi à 16** -0.75 **arrondi à -1**

Attention, en faisant l’arrondi du plus grand nombre on peut avoir des problèmes de dépassement !

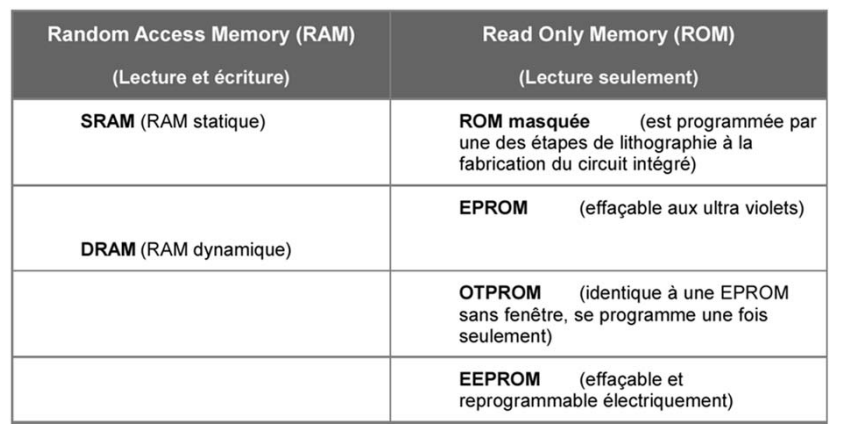
La troncature est plus facile à effectuer qu’un arrondi, mais en général on a moins d’erreur lorsque l’on fait un arrondi !

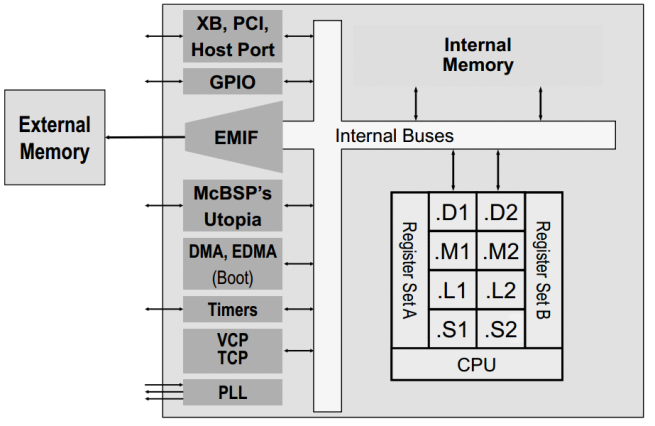
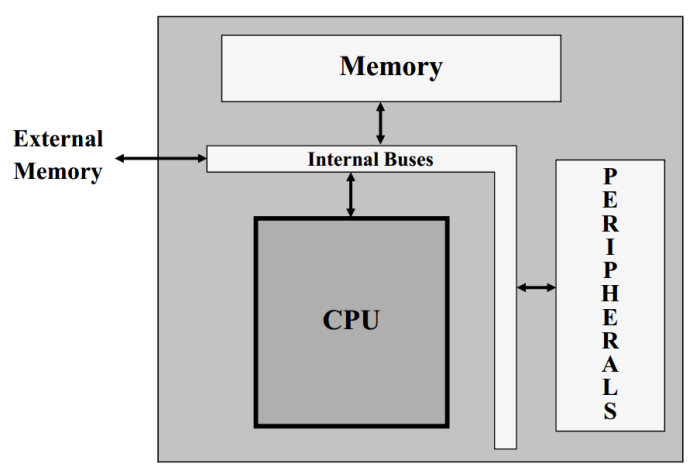
Annulation qui est le fait e perdre l’information après la virgule.

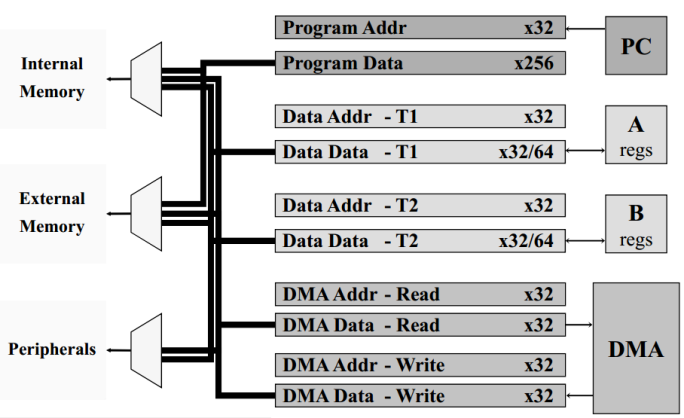
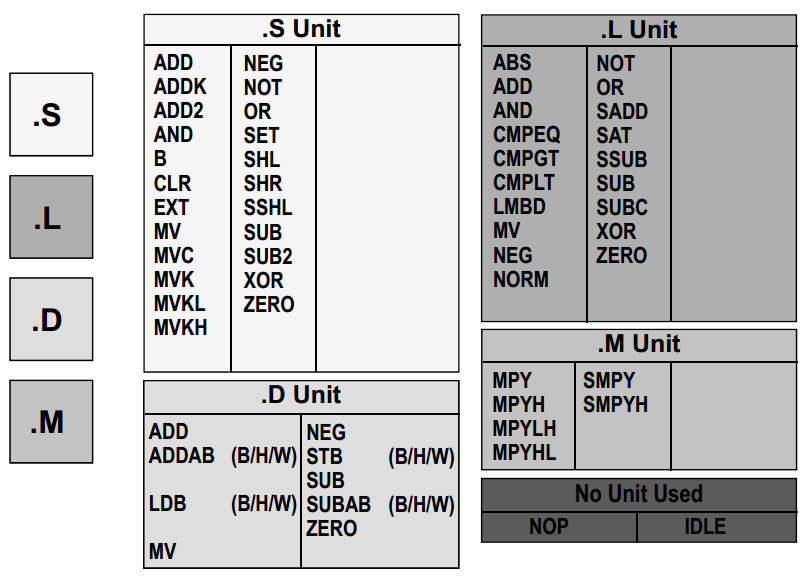
**DSP C6x** (Ch3 et 4)



Les DSP utilisent une architecture Harvard (2/4 accès par cycle d’instruction), alors que les processeurs d’usage général utilisent plutôt une architecture Von Neumann (1 accès par cycle d’instruction).

(Ch 3 p.22 à 29)

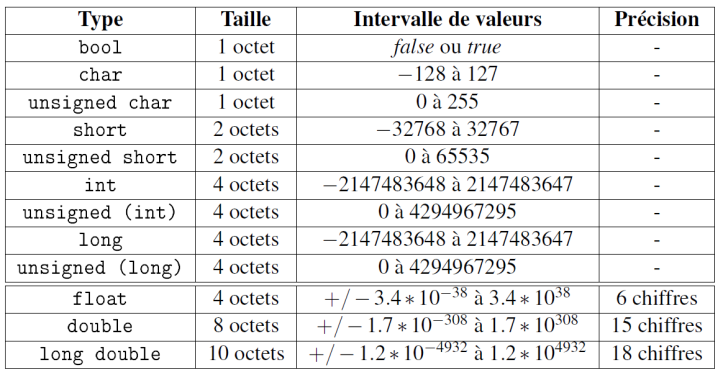




Registres sur 32 bits

Adresse sur 32 bits

Les tailles des bus permettent la programmation parallèle. Par exemple pour effectuer des opérations adresses, le code instruction est sur 32 bits on peut donc faire 8 instructions en même temps (8\*32 = 256 bits). 1 instruction par unité.



**Adressage** (Ch3 p. 37 à 43)

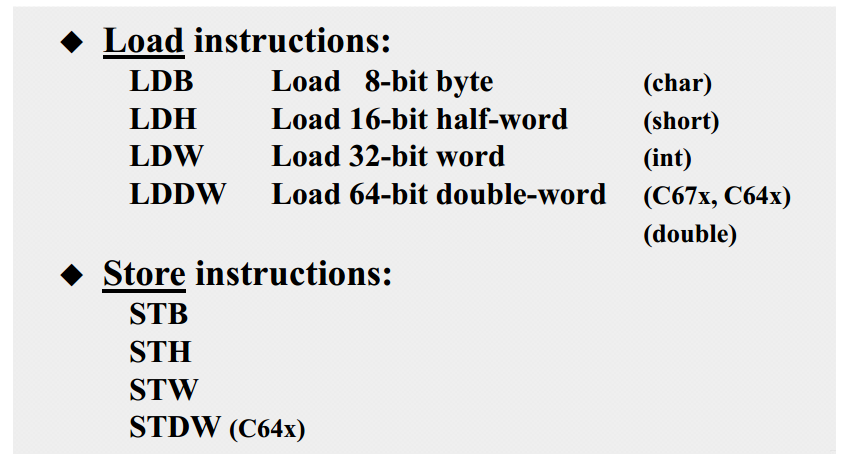
**Adressage Indirect** : Les registres auxiliaires permettent de stocker les registres d’accumulation. Ainsi si on veut utiliser la même variable il sera plus facile d’accès (une espèce de mémoire cache).

**Adressage bit-reverse** : Il a été créé pour la récupération des coefficients de la transformée de Fourier. Ainsi, au lieu d’accéder au 0,1,…7 ème coefficient on accède au 0,4,2… ème coefficient. Si on est sur 4 bits, l’index est à 8. Pour trouver les numéros liés au bit-reverse, il suffit de faire les oppérations usuelles dans ee sens inverse (de gauche à droite au lieu de droite à gauche).

**L’adressage circulaire** : Il est surtout utilisé pour la convolution. On utilise un bloc d’adresse, et on va « circuler » dedans sans en sortir, attention aux pièges !

**Opérations de stockage** (Ch4 p. 25 à 50)

Il y a différentes façon d’aligner les données, que ce soit par octet (byte), short, int etc… La façon d’aligner les données s’appellent **le** (little endian). Par exemple pour 32 bits on écrit x32 (le).



**Opérations croisées** : Les opérandes de registre passent d’un côté à un autre. Il y a 3 étapes pour les opérations croisées :

* Data : Registres en tant que opérande de données
* Adresse : LDW Destination    
   STW source
* Condition : usage illimité pour []

La destination c’est toujours du même côté, par contre la source peut provenir de n’importe quel fichier de registre.

A la fin de l’unité utilisée, on rajoute « x » pour indiquer le cheminement croisé.

**Ecriture des pointeurs** : Il y a plusieurs façons d’écrire des pointeurs :

* Index
* Offset
* Index echelonné

Il ne faut pas confondre

LDH \*A5++(3),A3

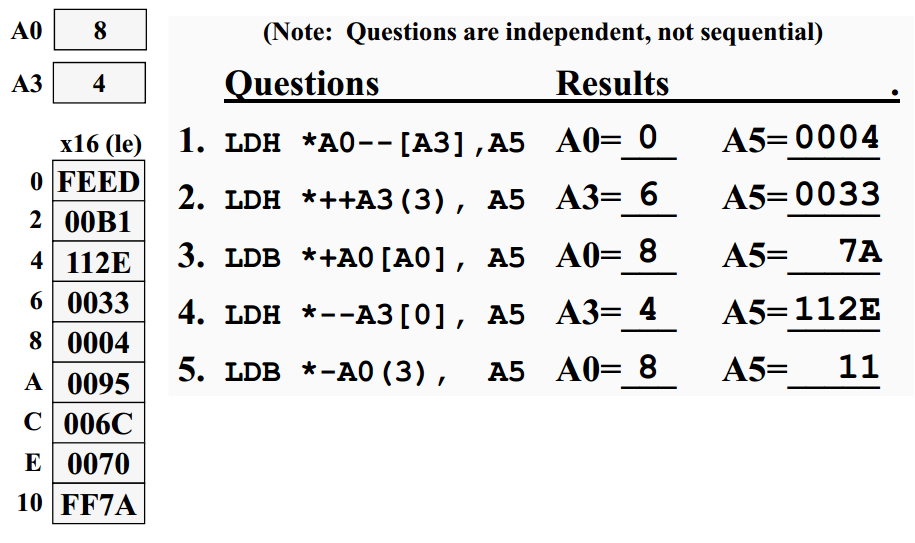
LDH \*A5++[1],A3

Quand il y a des parenthèses, on se déplace dans la mémoire par incrément de 1 octet quoiqu’il arrive. Alors que pour des crochets, on se déplace par incrément spécifié par l’instruction. Ici LDH déplace 2 octets donc on multiplie le nombre entre crochets par 2 pour avoir la vraie adresse.

Entre crochets on ne peut avoir que des entiers, on devrait donc écrire LDH \*A5++[1.5],A3. Le problème c’est que 1.5 n’est pas entier donc on prend 1 pour etre juste en dessous.

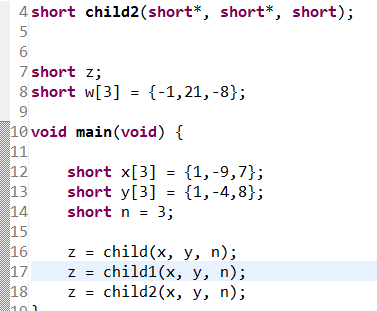
Voici les différents cas (exemple après) :

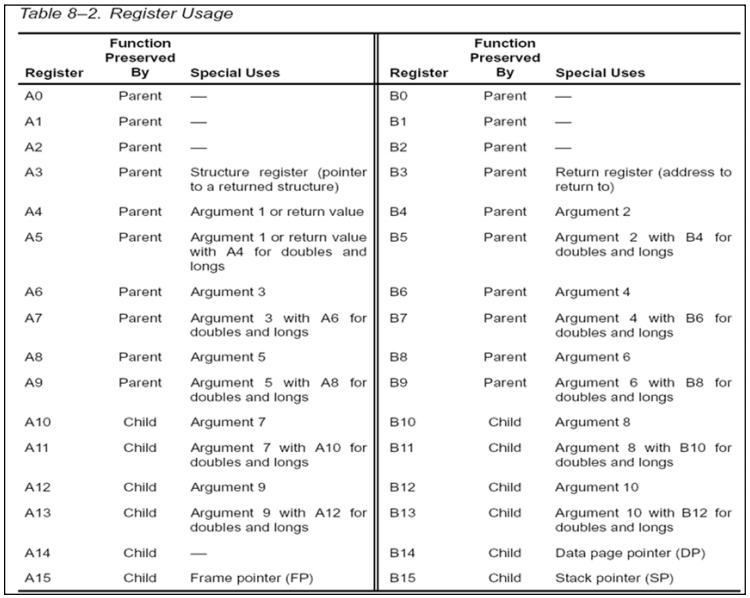
* Pré incrémentation le registre est modifié AVANT et on recherche à cette adresse.
* Post incrémentation le registre est modifié après. On recherche d’abors l’adresse ensuite on modifie le registre.
* Le pré offset c’est juste que le registre (l’adresse) ne change pas. Par contre on recherche bien l’adresse en faisant les calculs habituels.



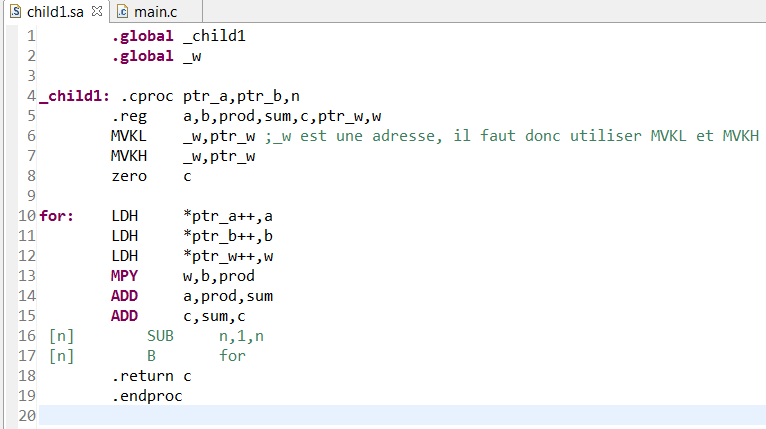
**Assembleur linéaire** (ch5 p.1 à 19)

Comme l’adresse est une constante, on utilise l’instruction MVK. Il peut bouger 16 bits, comme les adresses sont sur 32 bits, on divise l’instruction en 2 :





On utilise les registres A0 à A9. Pour les 6 autres il faut prendre des précautions (sauvegarder dans la pile).



Il faut mettre « \_ » devant les variables globales

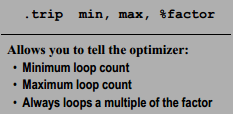
« .cproc » indique les variables d’entrée. Ne pas oublier « .endproc » à la fin

« .reg » indique les variables locales (que l’on définit dans notre fonction)

« .return » qui indique ce que l’on ressort

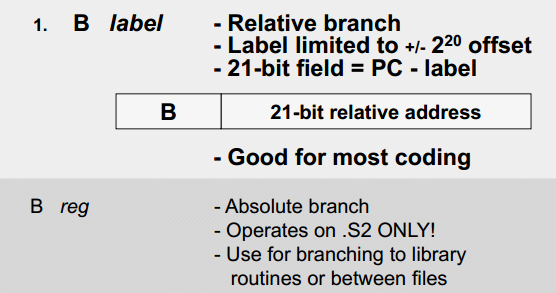
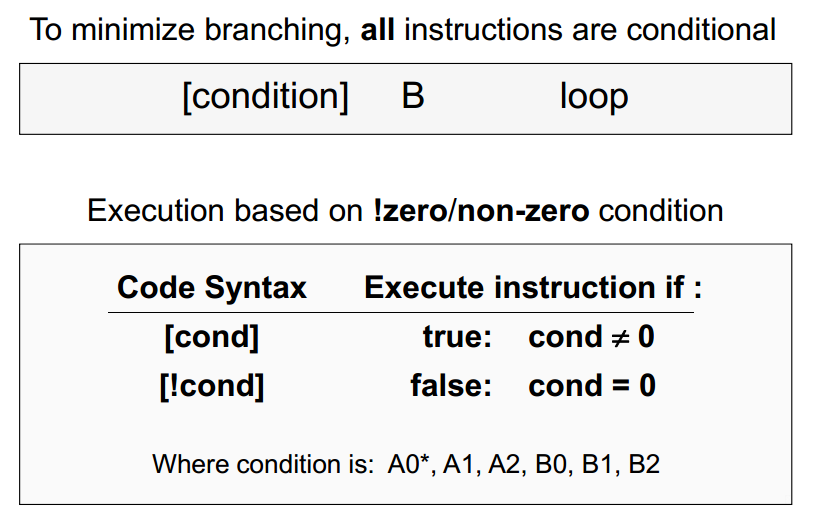
On peut aussi réserver des registres, pour qu’il s’en occupe en priorité. En écrivant 

On peut aussi spécifier le minimum, maximum de la boucle



**Instructions utiles**

Branch : En général, on met aussi la condition entre crochet à l’opération SUB, pour éviter que le programme continue si le résultat est négatif. (Sans la condition il ne s’arrête que quand c’est pile poil égal à 0).



Temps de cycle des instructions :

* Les opérations de multiplication (unité .M) MPY durent 2 cycles.
* Les opérations de stockage ST durent 3 cycles.
* Les opérations de chargement LDH/LDW durent 5 cycles.
* Les opérations de branchement BRANCH durent 6 cycles.

**Méthodes d’optimisation** (voir Lab 3 et Ch 5 p. 20 à 33)

**Pipeline** (voir Lab 3 et Ch 4 p. 54 à 78 et Ch5 p.34 à fin)