Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

1) Objectifs et mode d'emploi du polycopié.

Ce polycopié est un document qui veut se situer à la frontière du document technique type « databook » et du manuel pédagogique. Les aspects les plus avancés ne seront pas développés dans ces pages. S'il s'agit de détails techniques sur les produits Coldfire 5307 et 5407 , nous invitons le lecteur à consulter le manuel, MCF5307 ou 5407 ColdFire Integrated Microprocessor User's Manual ® ou bien à visiter le site Freescale (ex Motorola) http://www.freescale.com/

S'il s'agit de concepts particuliers, des documents spécifiques sont disponibles (exemples : gestions des entrées-sorties parallèles, les interruptions…)

Au début de chaque chapitre, sont précisées les connaissances requises, nécessaires, pour une lecture profitable du-dit chapitre.

Les questions émaillent le texte (logo **?**) pour permettre au lecteur de s'assurer de la « solidité » de ses connaissances.Les réponses (logo ⏩) sont en général situées à proximité immédiate des questions posées.

Des rappels peuvent apparaître sur le coté du texte associés à un astérisque*.

1) Rappel
Zone d'explications sommaires données à titre de rappel.
Ou renvoi à telle partie du document ou à tel autre document.

2) Qu'est ce qu'un microprocesseur.

Connaissances requises : Logique binaire combinatoire et séquentielle, fonctions logiques de base, registres, bascule D, compteurs, UAL.

Un microprocesseur est un circuit électrique composé de fonctions logiques (binaire). En conséquence, il ne « manipule » que des 0 et des 1 (bit), en général groupés par paquets de 8, nommés octet (byte).

Il est doté de bus* d'échanges d'informations avec des composants mémoires. (Voir figure 1, le schéma d'un système minimum composé d'un microprocesseur associé à une mémoire contenant les codes d'un programme.)

Via ces échanges, en particulier via des lectures consécutives d'informations stockées dans ces mémoires, le microprocesseur lit des codes (groupe de bits) qu'il interprète en tant qu'ordres. Ces interprétations sont suivies d'exécutions effectives des ordres. Ces exécutions successives correspondent à l'exécution d'un programme (suite d'ordre, de codes d'instructions).

Certaines des instructions sont capables sous certaines conditions de conduire à l'exécution d'un saut (plutôt qu'une exécution « du suivant ») dans le déroulement du programme. Cette capacité de sauts conditionnels, permet d'assimiler un microprocesseur à un ordinateur.

***2) Rappel* bus**.
Il s'agit d'ensembles de fils (de broches) qui « portent » des signaux dont la signification est cohérente et spécifique. exemple les bus de données et d'adresses sont deux groupes de fils distingués par la nature donnée ou adresse des informations qu'ils « transportent »

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

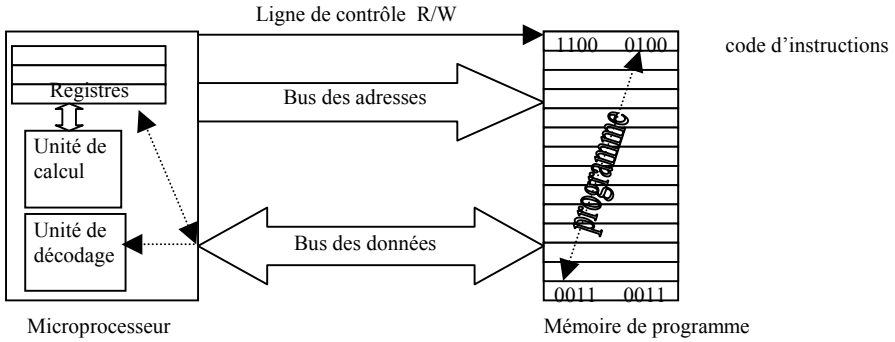


Figure 1 Système minimum : le microprocesseur et sa mémoire de programme

3) Rapide historique des microprocesseurs.

Connaissances requises : Aucune.

A faire

4) Les différents types de microprocesseurs.

Connaissances requises : Aucune.

A faire

1. Les microprocesseurs Motorola : Coldfire 5307 et 5407.

Connaissances requises : Le chapitre 2.

5.1 Caractéristiques générales.

A faire

5.2 Schéma « bloc ».

Le schéma bloc du microprocesseur Coldfire 5307 fait apparaître une structure bien plus large que celle d'un simple microprocesseur type 68000. Ici le bloc CPU (Central Processing Unit) traditionnel, est concentré dans les blocs « Coldfire Core » + « MAC Module ». Les autres blocs sont, par groupe de fonctions :

_un cache (mémoire). (unifié et non séparé en caches instructions et données de 8ko),

_une mémoire RAM (lecture et écriture) de 4 Ko (1 Kilo octets = 1024 octets), à accès rapide (sans insertion de temps de ralentissement « wait state »),

_un contrôleur de mémoire dynamique type SDRAM (Synchronous Dynamic Ramdom Acces Memorie) ou type EDO (Extended Data Out),

_un décodeur de champs d'adresses (8 Chip Select),

_un contrôleur DMA (Accès Mémoire Direct),

_un ensemble de périphériques intégrés (bus parallèle de 16 bits, 2 canaux série synchrones/asynchrones, 2 timers 16 bits, 1 contrôleur de bus I2C).

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

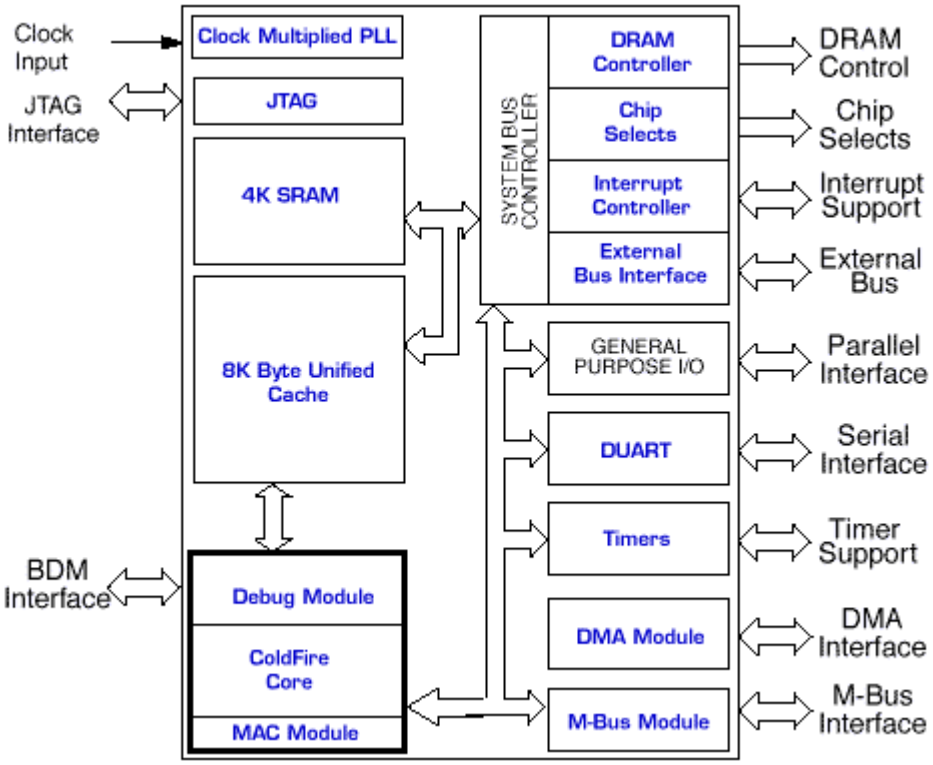


Figure 2 Schéma bloc du microprocesseur Coldfire MCF5307

Certaines de ces fonctions donnent au microprocesseur 5307 des caractéristiques de microcontrôleur (les périphériques intégrés notamment), d'autres le rapprochent de microprocesseurs traditionnels (cache ou contrôleur SDRAM).
En pratique, les pages qui suivent vont traiter presque exclusivement de l'unité centrale du composant. Les autres parties, telles que les divers périphériques, seront étudiées dans d'autre documents.

Le produit 5407 est une évolution du 5307. Ses plus sont :
-Une fréquence de fonctionnement interne plus grande. 220 Mhz au lieu de 90 Mhz (d'ou une puissance de traitement maximale qui passe de 75 MIPS (Million d'Instructions Par Seconde) à plus de 200 MIPS)
-Des caches d'instructions (16ko) et de données (8ko) séparés.
-Ajout d'instructions et « améliorations » du jeu d'instructions

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

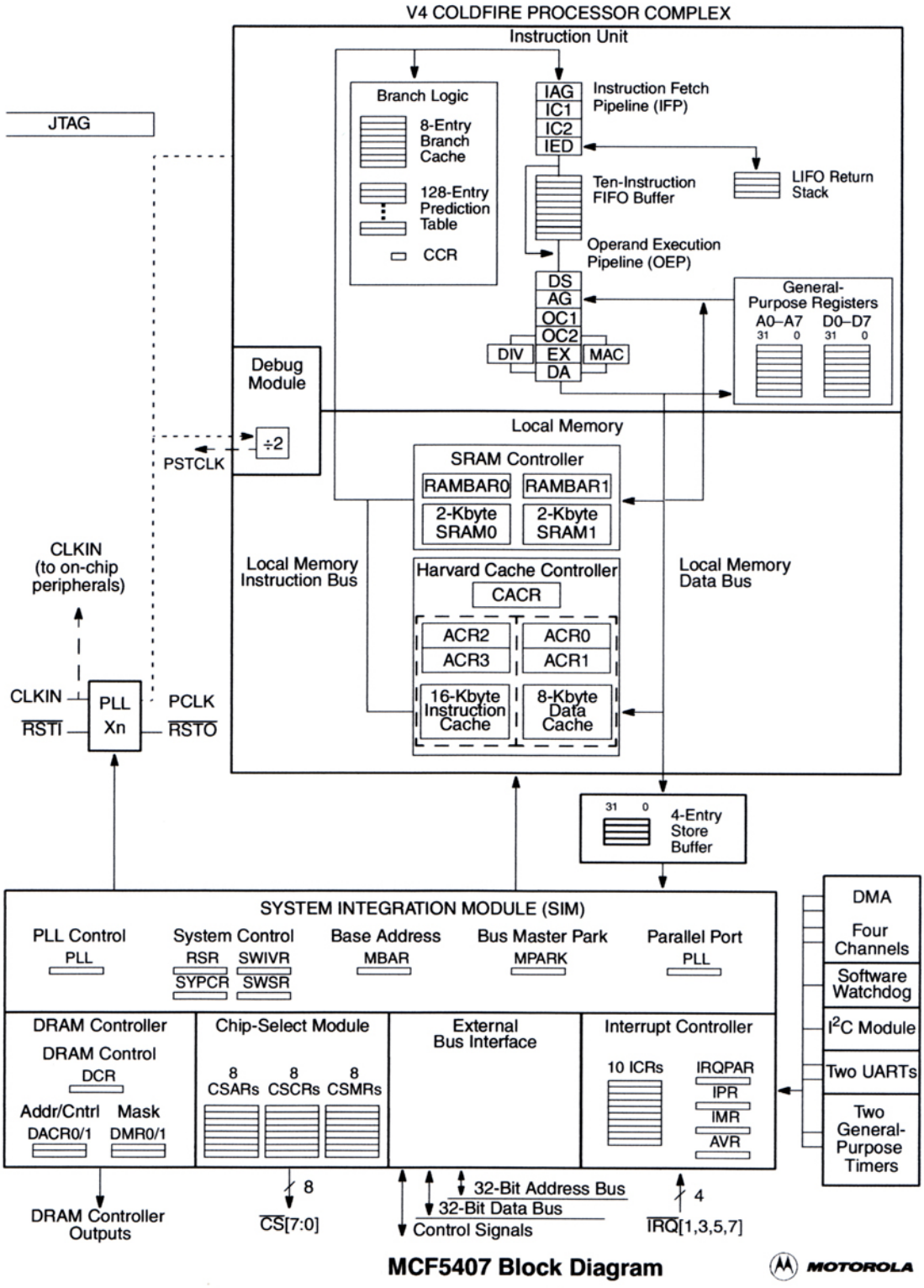


Figure 3 Schéma bloc du microprocesseur Coldfire MCF5407

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

5.3 Modèle de programmation.

Le modèle de programmation d'un microprocesseur est constitué par l'ensemble des registres qui sont accessibles par le biais de l'exécution d'un programme.
Il y a, dans la plupart des microprocesseurs, des registres utiles pour le fonctionnement de composant, mais non exploitables par programmation. Ces registres n'appartiennent pas au modèle de programmation du microprocesseur.

Le microprocesseur Coldfire compte un grand nombre de registres dans son modèle de programmation. Plus de 150 ! Pour des raisons pédagogiques évidentes, les registres qui concernent la programmation des divers périphériques ne seront pas inclus dans le modèle de programmation. Ces registres (env. 120) sont donnés en annexe de ce document.
Le microprocesseur Coldfire (comme les composants de la famille Motorola 68000) est capable de fonctionner soit en mode normal (mode utilisateur), soit en mode superviseur. Quelques registres sont strictement accessibles dans ce dernier mode. (Ils sont représentés grisés dans les pages qui suivent)
Remarque : Les registres d'adresses sont accessibles indifféremment par des accès 16 et 32 bits*. Seuls les accès 32 bits seront retenus dans ces pages.

- Rappels importants :

• Groupe de bits	• nom général	• symbole assembleur	• symbole langage. C ansi
• 8 bits	• octet / byte	• .b	• char
• 16 bits	• mot / word	• .w	• short
• 32 bits	• long / long	• .l	• int

5.3.1 Modes et modèle de programmation.

Quelques registres du modèle de programmation ne sont accessibles qu'en mode superviseur. Il s'agit des registres SR (partie haute du registre d'état) et VBR (registre pointeur de la table d'exceptions)
Ces registres sont « grisés » dans la représentation qui suit.

Les modes de fonctionnement USER ou SUPERVISEUR du point de vue de leur fonctionnalités ou bien des aspects de la programmation qui les concernent sont détaillés plus loin dans ces pages au fur et à mesure de leur implication dans le sujet traité.

Les deux produits MCF 5307 et MCF 5407, ont en commun le même modèle de programmation à quelque détails près qui ne sont pas développés ici.

5.4 Modèle de programmation général.

Cette page donne l'ensemble des registres (principaux) du modèle de programmation.

Registres de données (ou de travail) **d0 à d7**.

31		15	7	0
				d0
				d1
				d2
				d3
				d4
				d5
				d6
				d7

Registres d'adresses, **a0 à a7**.

31		0
		a0
		a1
		a2
		a3
		a4
		a5
		a6
		a7

Registre pointeur d'instruction, dit registre de programme **pc**.

31	0
	pc

Registre d'états, **sr**. (inclus dans sa partie basse le registre **ccr** registre de conditions)

15	7	0
		sr
	← *ccr* →	

Registre pointeur de table d'exception **vbr** (vector base register).

31	19	0
	← *0 à 19 20 bits pour adresses modulo 1 Méga* →	*vbr*

Registres de l'unité « multiplication et accumulation » mac (acc+mask+macsr)
Registre accumulateur, (de résultats) **acc**.

31	0
	acc

Registre de masquage, **mask**.

15	0
	mask

Registre d'état de l'unité, **macsr**.

7	0
	macsr

5.5 Modèle détaillé.

8 registres de données , nommés d0 , d1 ….. à d7.
Accessibles indifféremment, au niveau de l'octet, du mot ou du long.
Seuls les octets et mots « bas » (bits 0 à 7 ou bits 0 à 15) sont manipulables via le sélecteur de taille des mnémoniques des instructions (langage assembleur).
.b pour octet (byte) 8 bits du bit 0 au bit 7. ➔ **char** en langage C .
.w pour mot (word) 16 bits du bit 0 au bit 15. ➔ **short** en langage C .
.l pour long 32 bits du bit 0 au bit 31. ➔ **int** en langage C
.

← Octet - .b – char→
← Mot - .w - short →
← Long - .l - int →

31		15	7	0
				d0
				d1
				d2
				d3
				d4
				d5
				d6
				d7

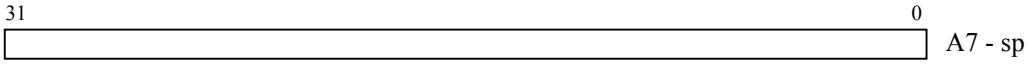
Exemples :
mettre à 0 le registre D7 sur sa totalité (32 bits) *clr.l d7*
Par exemple : Valeur initiale de D7 ➔ 0x12345678 , après exécution de cette instruction d7 = 0x00000000
copier l'octet bas de D1 dans l'octet bas de D3 *move.b d1,d3*
Par exemple : Valeurs initiales de D1 ➔ 0x11111111, de D3 ➔ 0x33333333, après exécution de cette instruction D1 = 0x11111111, de D3 0x33333311
mettre 0x1515 dans le mot bas de d4 *move.w #0x1515,d4*
Par exemple : Valeurs initiales de D4 ➔ 0x44444444, après exécution de cette instruction D4 = 0x44441515

7 registres d'adresse nommés de a0 ,a1 …. à a6.
Accessibles (seulement) au niveau du long (l'accès mot est possible, mais jamais utilisé dans ces pages).

31	0
	a0
	a1
	a2
	a3
	a4
	a5
	a6

Exemples :
copier le contenu du registre a0 dans le registre a5 *move.l a0,a5*
Par exemple : Valeurs initiales de A0 ➔ 0x0000000, de A5 ➔ 0x55555555, après exécution de cette instruction D1 = 0x00000000, de D3 0x00000000
initialiser le registre a4 avec l a valeur 0x30000000
2 possibilités : move.l #0x30000000,a4 ou bien lea 0x30000000,a4

1 registre pointeur de pile (stack pointer) nommé a7 ou sp .
Accessibles (seulement) au niveau du long.

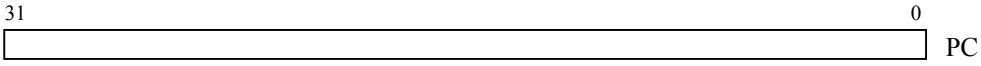


Ce registre est implicitement (automatiquement, systématiquement) utilisé par le microprocesseur pour permettre le rangement de contenu de registre en mémoire à l'occasion d'appel de sous-programme (en langage assembleur ou de fonction en langage C) ou d'exécution d'exceptions logicielles ou d'interruptions matérielles.
Il doit donc contenir l'adresse d'une région de mémoire vive (plus précisément l'adresse +1 du dernier octet de cette région).

Exemple :
initialiser le pointeur de pile avec la valeur 0x10008000.
Plusieurs solutions. (qui montrent en particulier que l'écriture a7==sp)

move.l #0x10008000,sp
ou bien *move.l #0x10008000,a7*
ou bien *lea 0x10008000,sp*
ou bien lea 0x10008000,a7

1 registre pointeur de programme (program pointer) nommé pc .
Accessibles (indirectement *bra,jmp,jsr…)* au niveau du long.

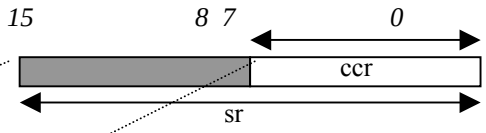


Ce registre doit être initialisé avec l'adresse du premier octet du code de la première instruction d'un programme à exécuter. C'est en effet ce registre qui « pointe » en mémoire de programme (zone .text en assembleur , main en C).
Il fournit à tout instant l'adresse de l'instruction à exécuter.
C'est en général via une commande de type GO que cette valeur est fournit.
A la mise sous tension le Coldfire initialise ce registre avec le contenu du second long de la table d'exception.
Ce registre est modifié ou non suite à l'exécution d'une instruction de saut conditionnel. (exemple beq, bmi, bls…). Il est systématiquement modifié par une instruction de saut inconditionnel. (exemple bra , jmp, jsr…).

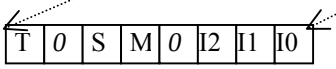
Exemple :
exécuter 50 fois une boucle de prog. (le registre d0 est ici utilisé en tant que compteur de boucles) :

```
        move.l  #50,d0
boucle :  …..          ;corps de la boucle, ici non définit.
        subq.l  #1,d0
        bne.s   boucle      ;ici PC adopte la valeur « boucle » tant que d0 # 0
        …. Suite du programme ,exécutée après 50 « bouclage »
```

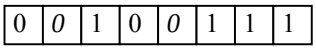
1 registre d'états (status register) nommé sr .
Ce registre de 16 bits n'est que partiellement accessible (dans sa partie basse 0-7) en mode utilisateur. Cette partie est nommée ccr (condition code register).



La partie haute de SR (bits de 8 à 15) contient :



A la mise sous tension, ces bits valent respectivement :



Le bit **T** (Trace) permet, mis à 1, un fonctionnement instruction par instruction. (pas à pas, step by step).

Le bit S (Superviseur) à 1, met le microprocesseur Coldfire en mode SUPERVISEUR. Toutes les instructions sont exécutables, en particulier, celles qui donnent accès au registre SR.
Le bit **S** à 0 met le microprocesseur Coldfire en mode USER.

Le bit **M** (Master) est mis à 0 lors de l'exécution d'une exception ou d'une interruption. Mis à 1 par programme via l'exécution d'une instruction :
move.w #0x3700,SR (si le microprocesseur le permet en étant en mode superviseur à cet instant), ou RTE.

Les bits **I**2/1/0 (Interrupt mask) sont des bits de masquage des niveaux physiques N d'interruption.
Ils forment un code dit masque I. $I \in [7,0]$
Si le niveau N est $\leq I$ ➔ pas d'interruption, sauf si N=7.

Exemple :
placer les masques d'interruption au niveau 110, rester en mode superviseur, T et M mis à 0. Tous les bits d'états de la partie CCR sont mis à 0.

Remarque : pour modifier cette partie de SR il faut être en mode superviseur, et tout le registre d'état (sur 16 bits) est modifié.

move.w #0x2600,sr * *0x2600* → *0010 0**110** 0000 0000*
*T0**S**M0 **III** 000XNZVC*

La partie basse de SR est nommée CCR (Condition Code Register) (bits de 0 à 7) contient :

0	*0*	*0*	X	N	Z	V	C

Les bits X,N,Z,V,C sont des bits d'états. Leurs valeurs évoluent en fonction de certaines caractéristiques des résultats des opérations réalisées par le microprocesseur durant l'exécution des instructions d'un programme.
Ainsi si un résultat est nul, le bit Z en passant à 1 rend compte de ce fait.
Et bien entendu, il passe à 0 si le résultat est non nul.
Alors, si nécessaire, une instruction de saut conditionnel , type beq conduira lors de son exécution à un saut (déroutement du programme par modification du registre pc) si Z=1 ou bien au contraire à poursuivre en séquence le programme (passer à l'exécution de l'instruction qui suit l'instruction beq) si Z=0.
A la mise sous tension, ces bits valent respectivement :

0	*0*	*0*	0	0	0	0	0

0	*0*	*0*	**X**	N	Z	V	C

Le bit **X** est mis à 1 ou à 0 (construit) comme le bit C → voir bit C.
C'est une « doublure » du bit de retenu. Toutefois il n'est pas altéré par certaines instructions, alors que son « jumeau » le bit C est mis à 0. C'est le cas par exemple de l'instruction move , qui met le bit C du registre ccr à 0 , mais n'altère pas la valeur du bit X de ce même registre.

0	*0*	*0*	X	**N**	Z	V	C

Le bit **N** (Negative) est une copie du bit de poids fort du dernier résultat d'une « opération » (y compris un move).
Exemples:
suite à l'exécution de l'instruction : *add.l #1,d1* (addition de la constante 1 au contenu du registre d1 , résultat dans le registre d1)
Si la valeur initiale de d1 est 4, alors d1 après exécution vaudra 5, plus précisément (écrit en binaire sur 32 bits) : ***0***000 0000 0000 0000 0000 0000 000 0101
Le bit de poids fort , à 0, indique un nombre positif. ➔ Le bit N du registre CCR prend la valeur 0.
Si la valeur initiale de d1 est -4, alors d1 après exécution vaudra -3, plus précisément (écrit en binaire sur 32 bits) : ***1***111 1111 1111 1111 1111 1111 1111 1101
Le bit de poids fort , à 1, indique un nombre négatif. ➔ Le bit N du registre CCR prend la valeur 1.

0	*0*	*0*	X	N	**Z**	V	C

Le bit **Z** (Zero) est mis à 1 lorsque le résultat d'une opération est nul (y compris move).
Il est mis à 0 lorsque le résultat est non nul.

Exemple 1:
suite à l'exécution de l'instruction : *add.l #1,d1* (addition de la constante 1 au contenu du registre d1 , résultat dans le registre d1)

Si la valeur initiale de d1 est 4, alors d1 après exécution vaudra 5, et le bit Z=0
Si la valeur initiale de d1 est -1, alors d1 après exécution vaudra 0, et le bit Z=1

Exemple 2:
« boucler » 50 fois.

```
            move.l      #50,d0      /*compteur de boucle*/
boucle :    ….                      /*corps de la boucle*/
            subq.l      #1,d0       /*au 50ieme passage d0 == 0 ➔ Z==1*/
            bne.s   boucle          /*Si Z == 0 le saut vers boucle est fait*/
            ….                      /*suite du programme*/
```

0	*0*	*0*	X	N	Z	**V**	C

Le bit **V** (o**V**erflow, dépassement arithmétique) est mis à 1 lorsque le résultat d'une opération arithmétique donne un signe aberrant vis-à-vis des signes des opérandes impliqués.
Il est mis à 0 lorsque le signe du résultat est correct.
Ainsi la somme de deux nombres signés (complément à 2) positifs devrait toujours donner un résultat positif. S'il en est autrement (débordement du complément à 2 dans le format considéré) le bit V passe à 1.
Le bit V est élaboré pour les opérations d'addition, de soustraction et de division.

Exemples :
suite à l'exécution de l'instruction : *add.l #1,d1* (addition de la constante 1 au contenu du registre d1 , résultat dans le registre d1)
Si la valeur initiale de d1 est 4, alors d1 après exécution vaudra 5.
4 et 1 sont deux nombres positifs le résultat 5 est positif. (tout va bien) V ➔ 0.

Si la valeur initiale de d1 est 0x7FFFFFFF, alors d1 après exécution vaudra 0x80000000.
0x7FFFFFFF et 1 sont deux nombres positifs, le résultat 0x80000000 est négatif. (il y a un problème) V ➔ 1.

Remarque : V == 1 informe qu'il y a un problème de signe, mais ne « corrige » pas le problème. Les sauts conditionnels bvc et bvs ainsi que l'instruction trap V permettent de « constater » la valeur du bit V et de dérouter conditionnellement l'exécution du programme en fonction de cette valeur.

0	*0*	*0*	X	N	Z	V	**C**

Le bit **C** (Carry, retenue, report) est le bit de retenue (0 ou 1) tel qu'il est généré lors d'une opération arithmétique au niveau du bit de poids fort pour le format de travail choisi.

Si l'opération a une portée d'un octet (.b) le bit C est relatif au report arithmétique qui concerne le bit de poids 7 du résultat (msb de l'octet).
Si l'opération a une portée d'un mot (.w) le bit C est relatif au report arithmétique qui concerne le bit de poids 15 du résultat (msb du mot).
Si l'opération a une portée d'un long (.l) le bit C est relatif au report arithmétique qui concerne le bit de poids 31 du résultat (msb du long).

Exemple : suite à l'exécution de l'instruction : *add.l #1,d1*
,addition de la constante 1 au contenu du registre d1 , résultat dans le registre d1.

Si la valeur initiale de d1 est 4, alors d1 après exécution vaudra 5, plus précisément (écrit en binaire sur 32 bits): 0← 0000 0000 0000 0000 0000 0000 000 0101
Il n'y a pas de report (ou le report vaut 0) C ➔ 0.
Si la valeur initiale de d1 est 0xFFFFFFFF, alors d1 après exécution vaudra 0, plus précisément (écrit en binaire sur 32 bits):
1← 0000 0000 0000 0000 0000 0000 000 0000
Il y a un report sur le bit 31 vers un bit 32 qui n'existe pas ! C ➔ 1.
Remarque dans ces 2 exemples le bit V ➔ 0.
0xFFFFFFFF == -1, donc 1 + (-1) ne peut pas conduire à une erreur de signe.

1 registre pointeur de table d'exception. VBR

Ce registre n'est accessible qu'en mode superviseur. Ses bits de 0 à 19 sont toujours à zéro. Cela signifie que la table d'exception dont il est le pointeur est nécessairement placée à des adresses modulo 1 million 2^{20} . Ainsi si ce registre VBR, est laissé à zéro (état de reset) la table d'exception est placée à partir de l'adresse 0 du plan mémoire du microprocesseur Coldfire.

?

-a- Combien compte de registres de données,le modèle de programmation du microprocesseur Coldfire 5307 ? quelle est leur taille ?
-b- Même question vis à vis des registres pointeurs d'adresses ?
-c- Quel est le registre qui pointe l'adresse mémoire où se trouve le code de l'instruction à exécuter ?

⏩

-a- Le modèle de programmation du microprocesseur Coldfire 5307 compte 8 registres de données nommés de d0 à d7.
Ces registres font 32 bits. Ils peuvent être exploités pour contenir un octet (8 bits) ou un mot (16 bits) ou un long (32 bits)
-b- Le modèle de programmation du microprocesseur Coldfire 5307 compte 8 registres d'adresses nommés de a0 à a7.
Le registre a7 est particulier, il s'agit du regsitre pointeur de pile, encore nommé SP (Stack Pointer).
Ces registres font 32 bits. Dans ce polycopié ils ne seront exploités que pour contenir des longs (32 bits) taille d'unz adresse.
-c- Il s'agit du regsitre PC (program pointeur). C'est lui qui est éventuellement modifier à l'occasion des sauts conditionnels.

5.6 Les mnémoniques des instructions (lang. Assembleur)

Comme le précisera le chapitre consacré au jeu d'instructions, tous les ordres que reçoit le microprocesseur sont des groupes de 1 à 3 mots au maximum (16 à 48 bits) appelés codes opération des instructions.
En fait il est plus précis de dire que le premier mot est parfois complété par 1 ou 2 mot supllémentaires.

Programmer (écrire) en mémoire directement ces codes est une tâche complexe et source de beaucoup d'erreurs (0101 1110 1000 1100 ressemble beaucoup à 0001 1110 1000 1100 … et pourtant ne signifie pas la même chose pour le microprocesseur !)
En pratique, au niveau le plus bas de la programmation (au plus près du modèle de programmation), on utilise un éditeur de textes pour composer un fichier (dit source). Ce fichier contient la suite désirée d'instructions fournies sous forme de mnémoniques. C'est à dire quelques lettres qui évoquent l'opération à réaliser.
Ces mnémoniques sont imposés par le constructeur du microprocesseur employé.
Par contre ils ne sont pas standardisés. (autant de jeu de mnémonique que de microprocesseurs différents).
Ce fichier est alors soumis à un logiciel appelé « assembleur » qui va traduire en codes machine (0 et 1) chaque mnémonique.
L'ordinateur qui assume ces tâches, traitement de texte / assemblage, est appelé machine de développement.
Si cette machine de développement « tourne » sur un microprocesseur X et que les travaux de développement concerne un microprocesseur Y , on parle d'assembleur croisé, (cross assembler).
Ces notions sont les mêmes pour l'outil compilateur.
Si cette machine de développement « tourne » sur un microprocesseur X et que les travaux de développement concerne un microprocesseur X , on parle d'assembleur résidant.
Exemple. Un PC équipé d'un pentium, utilisé pour développer du code Coldfire, utilisera un assembleur croisé Coldfire, ou bien un compilateur C++ croisé Coldfire.

Il est donc fondamental de connaître les mnémoniques du microprocesseur sur lequel on souhaite faire des développements logiciels en assembleur.
Si on travaille avec un niveau de langage « supérieur » (C,C++, JAVA…), dits langages évolués, alors on est dispensé de connaître ce niveau de détail. Cela explique l'usage de plus en plus rare de l'assembleur, tout au moins dans l'informatique « de gestion » ou de supervision.

Par contre, dans le domaine de l'informatique industrielle, il est toujours recommandé de maîtriser l'assembleur en particulier lorsqu'il est nécessaire d'optimiser tout ou partie d'un programme vis à vis des critères vitesse d'exécution ou nombre d'octets générés.

La forme générale des mnémoniques du Coldfire (et de la famille 68000) est :

mnémonique.taille opérande_source, opérande_destination

exemple : move.w d4,5(a3)

**** op érande***
Nom générique d'une valeur, d'un nombre impliqué dans une opération

la taille, désigne le nombre de bits utile des opérandes impliqués dans l'opération :

.b pour byte <-> octet <-> 8 bits.
.w pour word <-> mot <-> 2 octets <-> 16 bits.
.l pour long <-> 2 mots <-> 4 octets <->32 bits.

Les opérandes* sont :

pour la source :
Soit une valeur constante, dite immédiate précédée du signe # (diése).
Soit le nom d'un registre, dont le contenu sera traité en tant qu'opérande.
Soit un mode d'adressage (étudié plus loin) qui définit une adresse, dont le contenu sera traité en tant qu'opérande.

pour la destination :
Soit le nom d'un registre, dont le contenu sera traité en tant qu'opérande.
Soit un mode d'adressage (étudié plus loin) qui définit une adresse, dont le contenu sera traité en tant qu'opérande.

L'opérande source est toujours lu. On lit, on extrait , (on « load ») la source.
L'opérande destination est toujours écrit. On écrit, on stocke, (on store) la destination.
Ces 2 dernières lignes sont très importantes à comprendre pour la suite du cours.

Quelques exemples : (voir aussi les quelques exemples du paragraphe précedent)

move.l #0x3,d1 ➔ le nombre 0x00000003 devient le contenu du registre d1.
clr.b d2 ➔ l'octet bas (seul) de d2 est mis à zéro.
clr.l a3 ➔ tout le registre a3 est mis à zéro.
move.w d1,(a3) ➔ le mot d'adresse égale au contenu du registre a3 devient égal au mot bas qui remplit le registre d1.
Dire le contenu (mot) mémoire pointé par a3 vaut la partie basse du registre d1

5.7 Modes d'adressages.

Sauf pour les deux cas, opérande contenu dans un registre (ex. d4)et opérande immédiate (ex. # xxx), pour tous les autres cas, les opérandes sont des valeurs stockées (à lire) ou à stocker (écrire) en mémoire.
En pratique ces opérandes sont « désignés » par leur adresse (ex. (a4) ou bien 0x30000000).
Tous les microprocesseurs offrent un nombre plus ou moins important de procédures, de manières d'élaborer (de calculer, d'évaluer) ces adresses.
Ces procédures sont nommées modes d'adressage.

Lorsqu'un programmeur travaille en langage assembleur, il est de sa responsabilité de choisir pour accéder à tel opérande, tel ou tel mode d'adressage. Un choix pertinent conduit à :

-minimiser le temps d'exécution de l'instruction et donc globalement du programme.
-diminuer la place occupée par l'instruction et donc globalement par le programme.

Ce dernier point est particulièrement critique lorsque le processeur cible (celui qui exécute le programme) est un microcontrôleur dont l'espace mémoire de programme est restreint à quelques kilo-octets.
Le premier point (optimisation de la vitesse d'exécution) est lui critique pour beaucoup d'applications industrielles ou le temps est un facteur important et souvent « compté ».
Un microprocesseur qui offre un nombre important de modes d'adressage est donc un microprocesseur « intéressant » grâce à son potentiel d'optimisation des critères évoqués ci avant.
La famille Coldfire (héritière de la famille 68000) offre un nombre important de mode d'adressages (12 d'après Motorola … mais seulement 9 si l'on reprends la définition donnée plus haut des modes d'adressages).

Tableau des modes d'adressages disponible pour le microprocesseur MCF5307/MCF5407.

nom	**syntaxe**	**usages les plus courants**	nbr. mots d'extension	Détails page :
immédiat	*#valeur*	initialisation des registres par des valeurs données	1 à 2	15
Par registre (implicite)	*di* ou *ai*	copier un registre dans un autre	0	19
étendu (ou absolu)	*adresse*	accès « peu fréquent» à un opérande en mémoire	2	22
indexé sans déplacement	*(ai)*	accès «très fréquent » à un opérande en mémoire	0	26
indexé avec déplacement	*dép(ai)*	accès «fréquent » à un opérande en mémoire	1	30
indexé avec registre (données ou adresse) et déplacement	*dép(ai,xi)*	gestion de tableaux pointés par *ai* et indexés par *xi* (*xi* == *di* ou *ai*)	1	37
auto-indexé : post-incrémenté prè-décrémenté	 *(ai)+* *-(ai)*	 gestion d'opérandes rangés concécutivement gestion d'opérande rangés dans une structure de pile	 0 0	 42 46
indexé via le cpt. ordinal PC avec déplacement	*dép(PC)*	accès à un opérande en mém. indépendamment de sa localisation absolue. prog. dits « relocatables »	1	51
indexé via le compteur ordinal PC avec registre (di ou ai) et déplacement	*dép(PC,xi)*	accès à un opérande (tableaux) en mémoire indépendamment de sa localisation absolue. programmes dits « relocatables »	1	51

5.7.1 Mode d'adressage immédiat.

SYNTAXE → # valeur

Exemple : #0x33 ou bien #nombre (nombre étant préalablement défini en tant que constante (nombre = 0xabcd)

Il ne s'agit pas vraiment d'un mode d'adressage, puisque l'opérande n'est pas désigné par son adresse. En fait l'opérande est une constante, « collée » au code opération de l'instruction qui la manipule. Ce « mode d'adressage » n'est applicable qu'à l'opérande source.
Ecrire move.b d0,#0x3 est tout à fait surréaliste ! # désigne une constante, donc une valeur qui peut être lue, mais en aucun cas modifiée.
Si la taille de l'opérande est l'octet ou le mot (.b ou .w) sa valeur est « collée » juste après le code de l'opération sous la forme d'un mot (00XX taille octet et XXX taille mot).
Si la taille de l'opérande est longue (.l) sa valeur est « collée » juste après le code de l'opération sous la forme d'un long (XXXXXXXX) .

Note :
La « perte » de l'octet 00 dans le mot 00XX pour la cas d'un opérande de taille octet (8 bits) est justifiée par le maintien d'un code de programme aligné sur une adresse paire . S'il en était autrement, pour un programme démarrant à une adresse paire, l'insertion dans ce programme d'une instruction de type move.b #0x33,d0 , elle même à une adresse paire, conduirait toutes les instructions qui la suivraient à être implantées à des adresses impaires. Il en résulterait une dégradation très substantielle des performances temporelles lors de l'exécution du programme. Cette remarque vaut pour des bus de données organisés en port de 16 ou 32 bits. Toutefois, le microprocesseur Coldfire 5307 ne déroge pas à cette règle, même si son bus des données est organisé en bus de 8 fils, en fait, c'est le décodeur des instructions, qui décodant un opérande immédiat de taille 8 bits, donne l'ordre d'effectuer une lecture « mot » de cet opérande, la lecture étant faite le microprocesseur n'exploite que l'octet bas de ce mot en tant que valeur immédiate.

Exemple : move.l #0x12345678,d4
le code de cette instruction est composé de 3 mots (6 octets),
1 mot de codage de l'instruction (voir pour plus de détails l'instruction move).
Ce code est extrait de la mémoire de programme à l'adresse fournie à cet instant par le contenu du compteur ordinal (pc). Il est ensuite analysé.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**Size**		**Destination Adresse Effective**						**Source Adresse Effective**					
				Registre				**Mode**				**Mode**			**Registre**

Lorsque le mot est codé dans sa zone « adresse effective de source » par le groupe binaire 111 100 alors le microprocesseur assume une nouvelle lecture de deux mots en mémoire (via pc + 2 puis pc +4) pour en extraire la valeur 0x1234 5678 qui est rangée dans le registre d4 (voir pour la destination le mode adressage par registre).

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

En résumé, si pc initial = 0x400, si d4 initial = 0x44444444

PC	00	00	04	00	d4	44	44	44	44

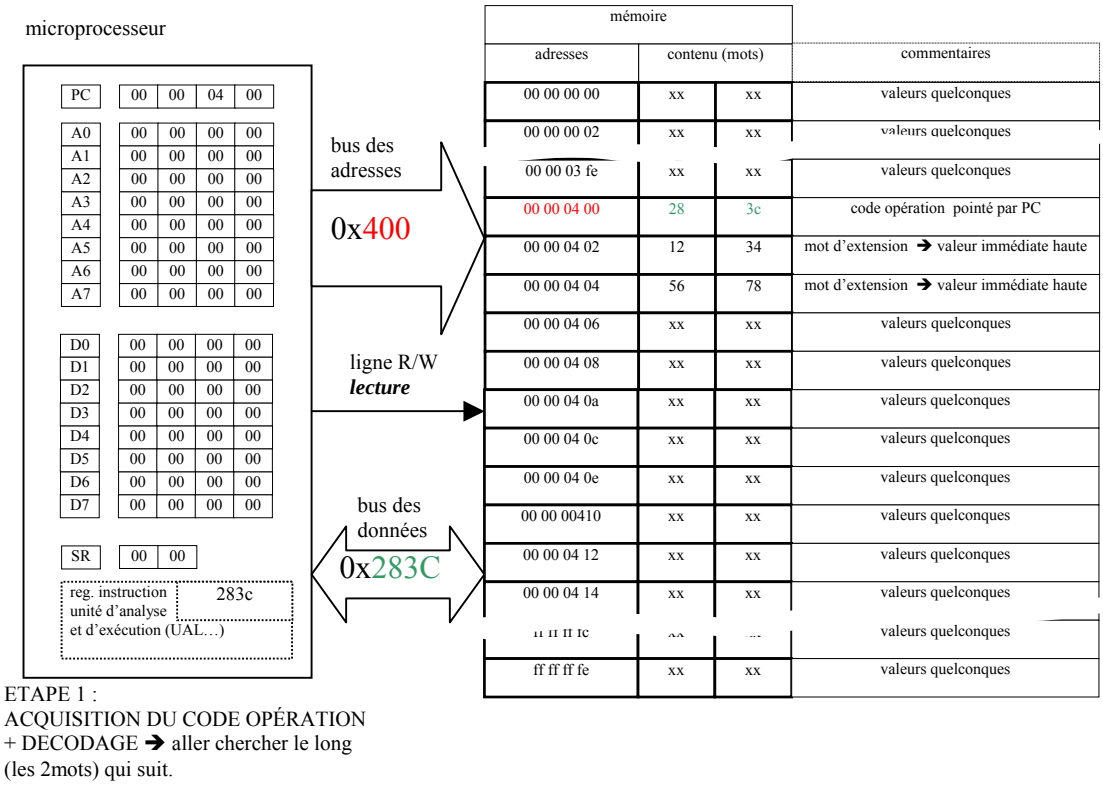
Si l'espace mémoire (organisé ici en mot) dans la région d'adresse 0x 400 est tel que :

adresses	Valeurs	Explications
0x400	0x283c	code de l'instruction move.l #0x12345678,d4
0x402	0x1234	mot d'extension, partie haute de la valeur immédiate
0x404	0x5678	mot d'extension, partie basse de la valeur immédiate

alors après l'exécution de cette instruction :
PC= 0x406 et d4 = 0x12345678

PC	00	00	04	06	d4	12	34	56	78

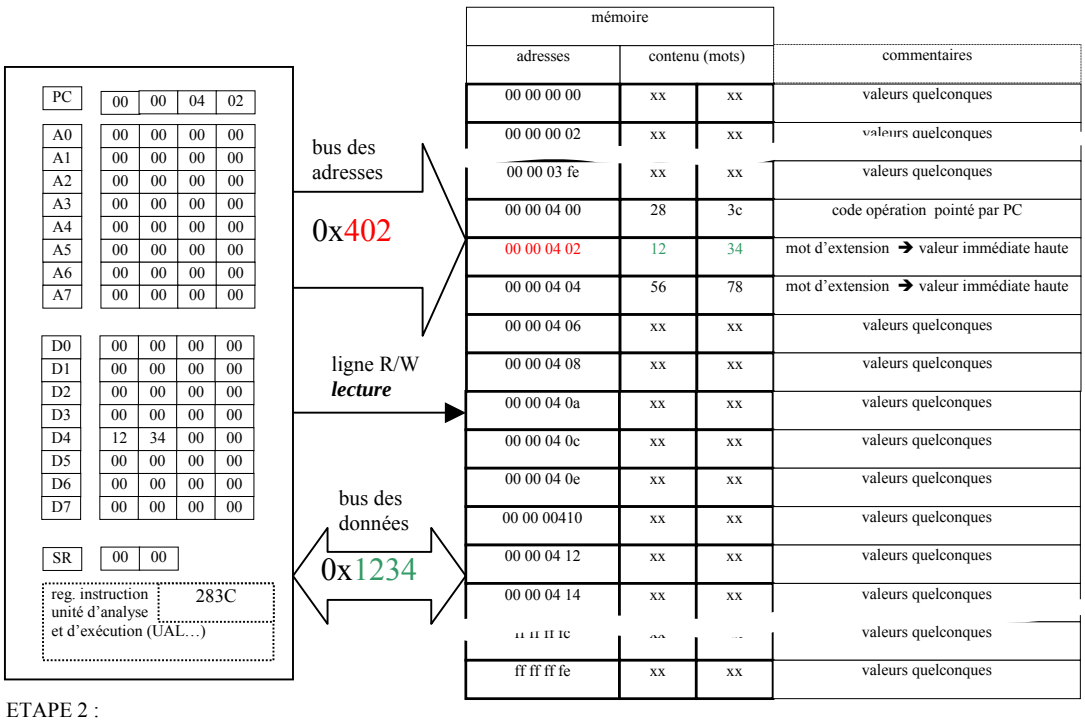
Cheminement des données lié à l'exécution de l'instruction : **move.l #0x12345678,d4**



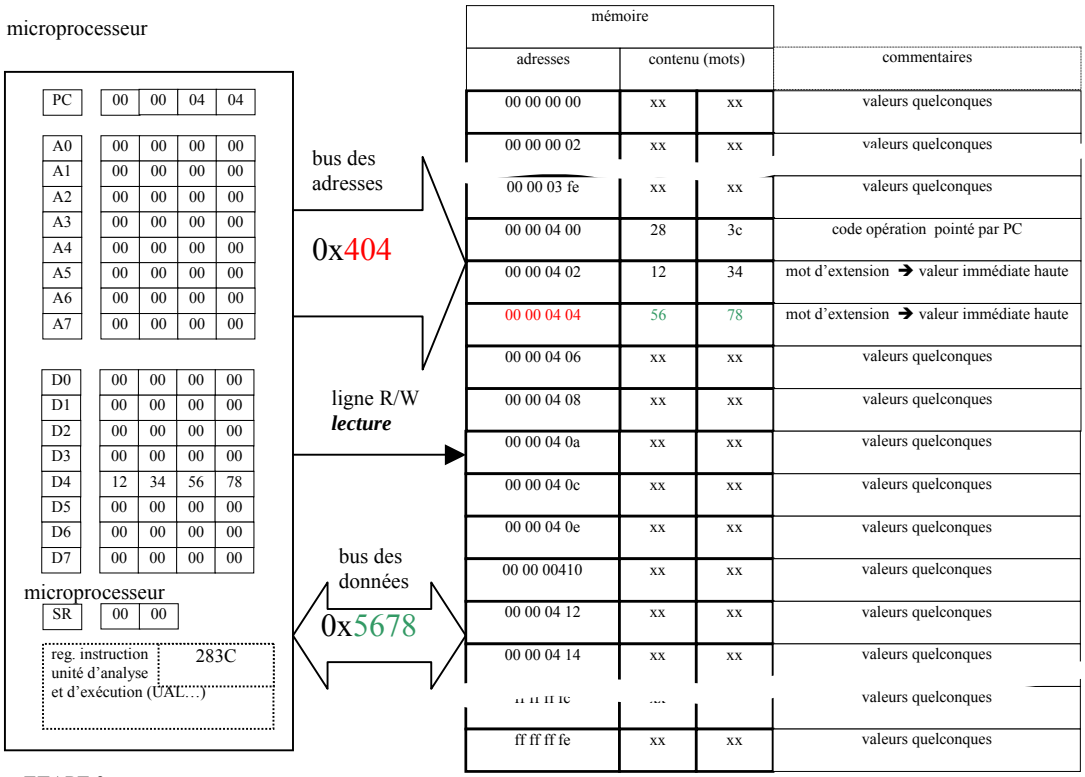
ETAPE 1 :
ACQUISITION DU CODE OPÉRATION
+ DECODAGE ➔ aller chercher le long (les 2mots) qui suit.

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE



ETAPE 2 :
ACQUISITION DU LONG « QUI SUIT3
ici en 2 cycles (bus des données sur 16 bits)
DÉPÔT DANS D4 PARTIE HAUTE.



ETAPE 3:
FIN D'ACQUISITION DU LONG « QUI SUIT»
(bus des données sur 16 bits)
DÉPÔT DANS D4 PARTIE BASSE.
ET PC = PC+2= 0x406 Prêt à accéder au code opération de l'instruction suivante .
Note : Cette valeur n'a pas été « dessinée » dans PC sur cette figure.

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

?

-a- Quel est le mnémonique de l'instruction qui met la constante 33 en tant que long dans le registre de donnée d7 ? *-b- Combien de mots sont nécessaires pour coder l'instruction move.l #0x33,d7 ? , move.w #0x33,d7 ? , move.b #0x33,d7 ?*

-c- Combien de cycles (d'accès mémoire) sont nécessaires (accès sur bus de 16 fils) pour exécuter l'instruction move.l #0x33,d7 ? , move.w #0x33,d7 ? , move.b #0x33,d7 ?

⏩

-a- move.l #33,d7 ou bien avec une présentation hexadécimale, move.l #0x21,d7

-b- le code opération == 1 mot + 2 mots d'extension pour contenir la valeur 0000 0033 ➔ 3 mots
le code opération == 1 mot + 1 mot d'extension pour contenir la valeur 0033 ➔ 2 mots
*le code opération == 1 mot + 1 mot d'extension pour contenir la valeur 00**33** ➔ 2 mots*

-c- 1 cycle pour acquérir le code opération + 1 cycle pour acquérir la valeur 0000 + 1 cycle pour acquérir la valeur 0033
1 cycle pour acquérir le code opération + 1 cycle pour acquérir la valeur 0033
1 cycle pour acquérir le code opération + 1 cycle pour acquérir la valeur 33 prise dans le mot lu 0033

5.7.2 Mode d'adressage par registres (implicite)

SYNTAXE → di ou ai

Il ne s'agit pas vraiment d'un mode d'adressage, puisque l'opérande n'est pas désigné par son adresse. En fait l'opérande est la valeur contenue dans le registre désigné.

Note : Nous avons signalé ci-avant qu'un grand nombre de modes d'adressages est un critère « positif » de choix d'un microprocesseur. Motorola (et tous les autres), au nom de ce principe, tente d'accroître artificiellement ce nombre. Ainsi l'adressage immédiat vu ci_avant ne devrait pas figurer dans le décompte. Pas plus que le mode que nous appellerons adressage par registres. Mais Motorola fait « plus fort » encore puisqu'il invite à distinguer un adressage via di (registres de données) et via ai (registres d'adresses)… Il les nomme respectivement adressage direct par registre de donnée et adressage direct par registre d'adresse… A ce petit jeu Motorola aurait pu proposé l'adressage direct par rapport au registre de donnée d0, puis l'adressage direct par rapport au registre de donnée d1…. l'adressage direct par rapport au registre d'adresse a7. Il aurait ainsi accru de 16 et non de 2 le nombre de modes d'adressages du COLDFIRE.

Exemple : move.l #0x12345678,d4
Il s'agit de l'exemple précédent.
Mais ici c'est l'opérande destination (registre d4) qui est désigné par le mode d'adressage par registre.
Autres exemples : move.l a2,d5
move.b d2,d4

Le code de cette dernière instruction est composé d'un mot (2 octets)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**Size**		**Destination Adresse Effective**						**Source Adresse Effective**					
				Registre				**Mode**				**Mode**			**Registre**

1 mot de codage de l'instruction (voir pour plus détail l'instruction move)
Ce code est extrait de la mémoire de programme à l'adresse fourni à cet instant par le contenu du compteur ordinal (pc). Il est ensuite analysé.
Lorsque mot est codé dans sa zone « adresse effective de source » par le groupe binaire 000 010 alors le microprocesseur interprète ce champ comme désignant le registre d2 en tant que source (mode par registre). De même, si la zone « adresse effective de destination » est codée par le groupe binaire 100 000 alors le microprocesseur interprète ce champ comme désignant le registre d4 en tant que registre de destination (mode par registre).

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

En résumé si pc initial = 0x406, si d2 initial = 0xFFFFFFFF et si d4 initial = 0x12346878

d2	FF	FF	FF	FF		d4	12	34	56	78

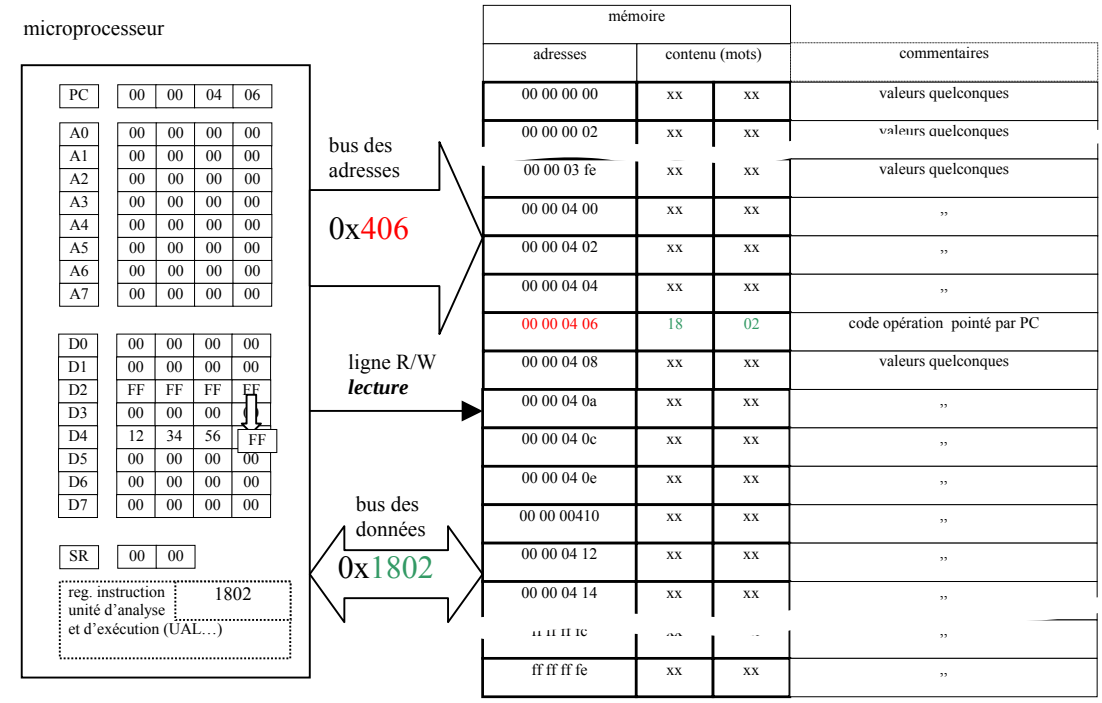
Si l'espace mémoire (organisé ici en mot) dans la région d'adresse 0x 406 est tel que :

Adresses	valeurs	Explications
0x406	0x1802	code de l'instruction move.b d2,d4

alors après l'exécution de cette instruction :
PC= 0x408 , d4 = 0x123456FF et d2 = 0xFFFFFFFF

d2	FF	FF	FF	FF		d4	12	34	56	FF

Cheminement des données lié à l'exécution de l'instruction : **move.b d2,d4**



ETAPE 1 :
ACQUISITION DU CODE OPÉRATION
+ DECODAGE ➔ copier l'octet bas de d2 dans l'octet bas de d4
ETAPE 2 : (sur le même dessin)
L'octet bas de d2 est recopié dans l'octet bas de d4
puis PC = PC +2 . Prêt à accéder au code opération de l'instruction suivante .
Note : Cette valeur n'a pas été « dessinée » dans PC sur cette figure.

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

?

-a- Quel est le mnémonique de l'instruction qui met la moitiée du registre d7 dans la moitiée du registre d4 ? *-b- Combien de mots sont nécessaires pour coder l'instruction move.l d0,d1 ? , move.w d0.d1 ? , move.b d0,d1 ?*

-c- Combien de cycles (d'accès mémoire) sont nécessaires (accès sur bus de 16 fils) pour exécuter l'instruction move.l d0,d1 ? , move.w d0.d1 ? , move.b d0,d1 ?

⏩

-a- move.w d7,d4 , c'est l'argument .w qui précise l'usage des 16 bits de poids faibles des registres de 32 bits d7 et d4.

-b- le code opération == 1 mot pour les trois instructions. Il n'y a aucune extension au code opération.

-c- 1 cycle pour acquérir le code opération et c'est tout ! l'opération transfert de d0 dans d1 est interne au microprocesseur et ne « consomme » pas d'accès mémoire.
1 cycle pour acquérir le code opération (même remarque).
1 cycle pour acquérir le code opération (même remarque).

5.7.3 Mode d'adressage absolu (ou étendu)

SYNTAXE → « adresse en clair »

Pour exploiter ce mode d'adressage il suffit de communiquer au microprocesseur l'adresse à laquelle on souhaite lire (source) ou écrire (destination) l'opérande.
Remarque : Ce mode d'adressage existe en version courte (absolu court), pour laquelle l'adresse est exprimée sur 16 et non 32 bits. Il n'est alors possible que d'accéder à un espace mémoire réduit à 2 zones de 32 koctets situées à chaque extrémités de l'espace mémoire du COLDFIRE.
Ce mode d'adressage « absolu court » ne sera pas développé dans ces pages.

Exemples : clr.l 0x20000000
move.l d4,vitesse (où vitesse est une étiquette du programme).
0x20000000 ou bien vitesse sont deux adresses. L'opérande unique du « clear » ou bien l'opérande destination du « move » sont désignés tous les deux par le mode d'adressage absolu.

Autre exemple : add.l 0x400000,d5
additionner le mot long stocké à l'adresse 0x400000 avec le contenu du registre d5, le résultat sera stocké dans le registre d5.
Le code de cette dernière instruction est composé de 3 mots (6 octets).
1 mot de codage de l'instruction (voir pour plus de détails l'instruction add) plus 2 mots qui valent, pour le premier 0x0040 et pour le second 0x0000
Ce code est extrait de la mémoire de programme à l'adresse fournie à cet instant par le contenu du compteur ordinal (pc). Il est ensuite analysé.

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	**1**	**0**	**1**	**Registre**	**Mode Op**	**Source Adresse Effective**	
						Mode	**Registre**

Lorsque le champ « opmode » vaut 010, et que le champ « register » vaut 101 pour spécifier le registre d5 en tant que registre de destination, et si la zone « adresse effective de source » est codée par le groupe binaire 111 001, alors le microprocesseur interprète ce champ comme : l'opérande source est stocké à l'adresse contenue dans les deux autres mots suivant ce « mot code opération ».

En résumé, si pc initial = 0x408, si d5 initial = 0xFFFFFFFF et si les 4 octets situés aux adresses 0x400000, 0x400001, 0x400002 et 0x400003 * valent respectivement 0x12, 0x34,0x56 et 0x78 .

*on dit plus couramment « le long d'adresse 0x400000 vaut 0x12345678 »

0x400000	12	34	56	78	d5	FF	FF	FF	FF

Remarque important : d5 est un registre du microprocesseur, 0x400000 est une adresse de l'espace mémoire contrôlé par le microprocesseur via ses bus d'adresses de données. Si l'espace mémoire (organisé ici en mot) dans la région d'adresse 0x 408 est tel que :

Le microprocesseur *Cold* FIRE 5307

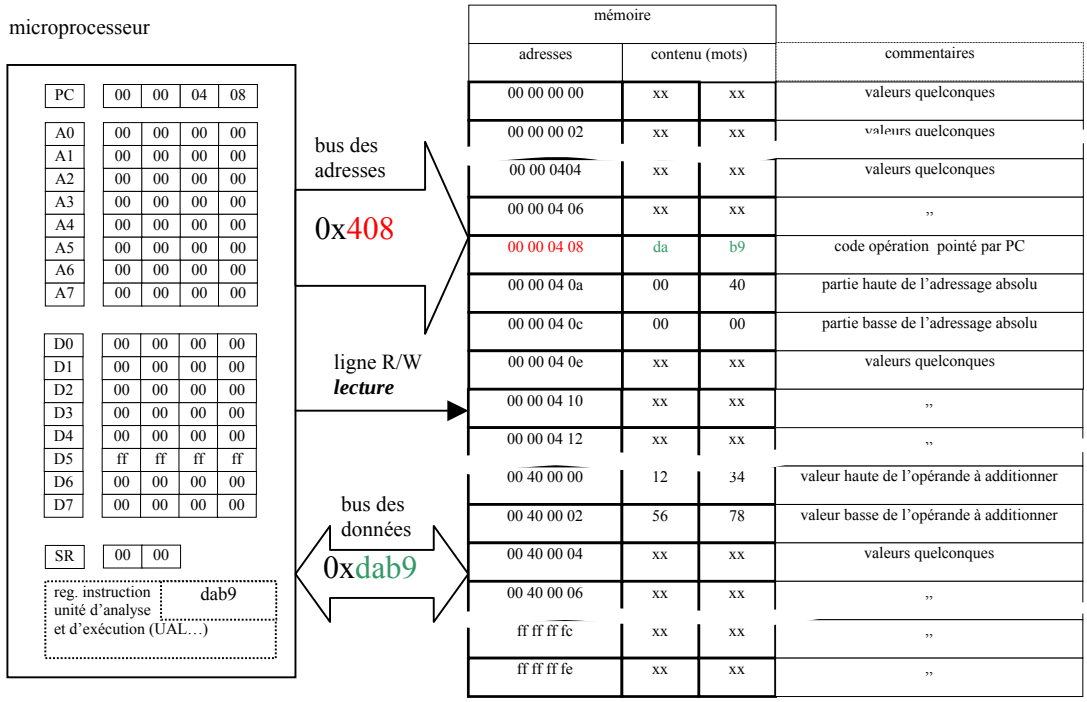
C. GUIRAUDIE

adresses	valeurs	Explications
0x408	0xdab9	code de l'instruction add.l 0x400000,d5
0x40a	0x0040	partie haute de l 'adresse absolue de l'opérande source 0x00400000
0x40c	0x0000	partie basse de l 'adresse absolue de l'opérande source 0x00400000

alors après l'exécution de cette instruction :
PC= 0x40e , d5 = 0x12345677* et le long d'adresse 0x400000 est inchangé.
* le résultat est d'autant plus évident que l'on retient que 0xffffffff == -1 (complément à 2 de 0x00000001)

0x400000	12	34	56	78	d5	12	34	56	77

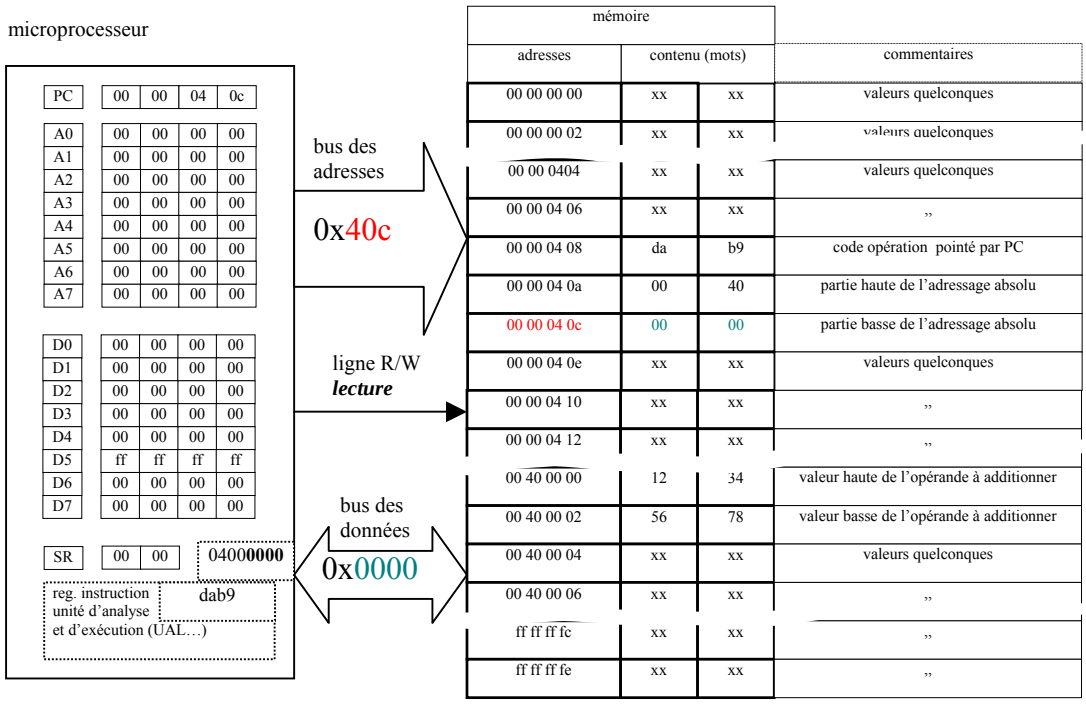
Cheminement des données liées à l'exécution de l'instruction : **add.l 0x400000,d5**



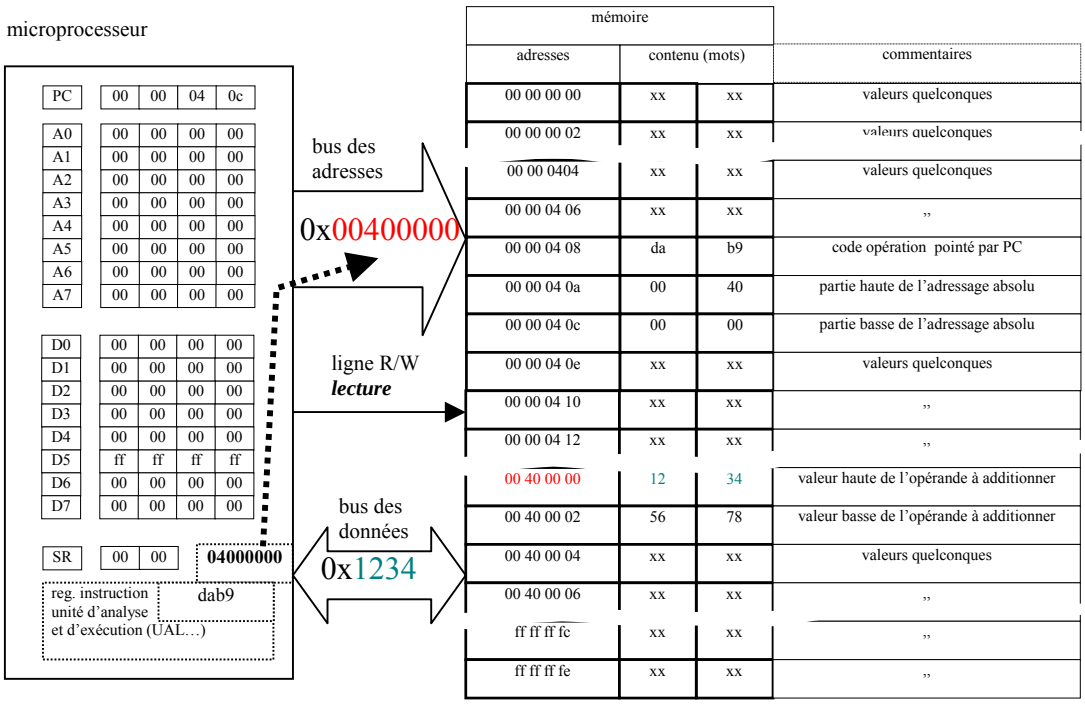
ETAPE 1 :
ACQUISITION DU CODE OPÉRATION
+ DECODAGE ➔ addition 32 bits du mot long d'adresse défini par les 2 mots qui suivent, avec le contenu du registre d5, résultat dans d5.

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE



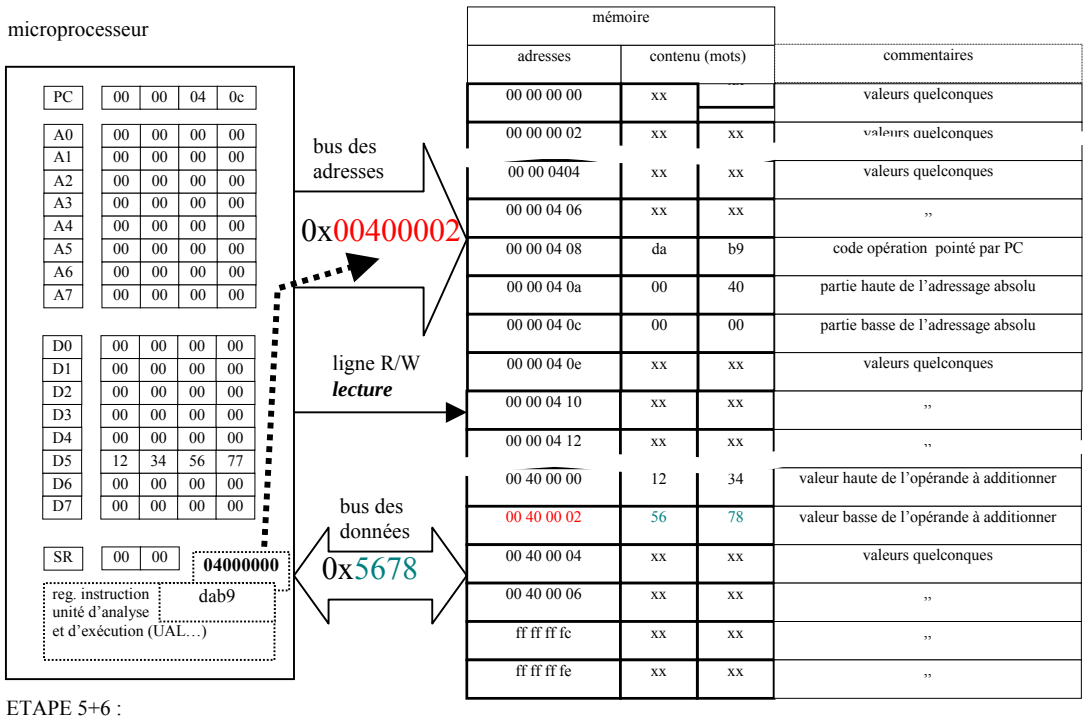
ETAPE 3 :
ACQUISITION EN 2 TEMPS (cause organisation bus des données en 16 bits)
DE PARTIE BASSE DE L'ADRESSE DE LA SOURCE (adressage absolu).
Cette valeur est stockée temporairement dans un registre annexe du microprocesseur.



ETAPE 4 :
ACQUISITION EN 2 TEMPS (cause organisation bus des données en 16 bits)
DE LA PARTIE HAUTE DE L'OPERANDE d'@ 0x400000 (adressage absolu).
Cette valeur est dirigée vers l'UAL (unité arithmétique et logique) du microprocesseur.

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE



ETAPE 5+6 :
ACQUISITION EN 2 TEMPS (cause organisation bus des données en 16 bits)
DE LA PARTIE BASSE DE L'OPERANDE d'@ 0x400000 (adressage absolu).
Cette valeur est dirigée vers l'UAL (unité arithmétique et logique) du microprocesseur.
REALISATION DE L'ADITION. ➔ 0x12345678+0xFFFFFFF => 0x12345677 ➔ registre D5
PC est incrémenté de 2, prêt à accéder au code opération de l'instruction suivante (en 0x40A)

?

-a- *Quel est le mnémonique de l'instruction qui met en mode étendu le mot d'adresse 0x12345678 à l'adresse 0x10000 ? !!!*

-b- *Combien de mots sont nécessaires pour coder l'instruction move.b d0,0x20000000 ?*

-c- *Combien de cycles (d'accès mémoire) sont nécessaires (accès sur bus de 16 fils) pour exécuter l'instruction*
move.b d0,0x20000000 ?, move.l d0,0x20000000 ?

⏩

-a- *QUESTION PIEGE !! Cette operation est impossible en une seule instruction (le modéle théorique serait move.b 0x12345678,0x10000, il occuperait 1+2+2 mots de codage, or le microprocesseur MCF5307 accepte au maximun 3 mots de codage). En conséquence il faut répondre au problème via la succession de 2 instructions :*

```
move.w   0x12345678,d0    ;d0 reçoit provisoirement le contenu 0x10000
move.w   d0,0x10000       ;au total, le mot d'adresse 0x10000 devient la copie de la source d0, donc du
                          ;mot d'adresse 0x12345678
```

-b- *le code opération == 1 mot +1+1 mots pour fournir l'adresse de destination 0x2000 0000. Soit un total de 3 mots.*

-c- *1 cycle pour acquérir le code opération + 2 cycles pour acquérir les 2 mots d'extension qui contiennent l'adresse 0x2000 0000 + 1 cycle d'écriture du la partie haute du bus des données (fils de d16 à d31) pour écrire l'octet bas de d0 à l'adresse (paire) 0x20000000. Soit un total de 3 cycles de lecture + 1 cycle d'écriture.*
1 cycle pour acquérir le code opération + 2 cycles pour acquérir les 2 mots d'extension qui contiennent l'adresse 0x2000 0000 + 2 cycles d'écriture pour écrire respectivement la partie haute de d0 (bits de 16 à 31) à l'adresse 0x20000000, puis la partie basse de d0 (bits de 0 à 15) à l'adresse 0x20000002 . Soit un total de 3 cycles de lecture + 2 cycles d'écriture.

5.7.4 Mode d'adressage indexé

SYNTAXE → (a***i***)

L'adresse est fabriquée par le microprocesseur en utilisant le contenu du registre d'adresse a*i* invoqué.
C'est un mode d'adressage peu « encombrant » , en effet le numéro *i* du registre impliqué est spécifié directement dans le code opération. Par contre via ce mode d'adressage pour accéder à des opérandes d'adresses distinctes il faut utiliser des registres d'adresse différents. Au-delà de 6 adresses distinctes (usage des reg. d'adresse a0 à a6), il faut réinitialiser un ou des registres déjà exploités.
En fait cet adressage est particulière bien adaptée à des algorithmes qui exploitent de très nombreux accès à tel ou tel opérande, en particulier au sein d'une boucle. Dans ces cas il est recommandé de réserver ce mode d'adressage pour les accès à ces opérandes.

Exemple : **move.l #0x12345678,(a4)**
Il s'agit pour l'opérande source du mode d'adressage immédiat.
Ici c'est l'opérande destination qui est désigné par le mode d'adressage indexé via le registre a4.
Si avant l'exécution a4 = 0x80000000, alors après exécution de cette instruction le long 0x12345678 sera rangé à l'adresse 0x80000000.
(Plus précisément l'octet 0x12 sera rangé à l'adresse 0x80000000, l'octet 0x23 sera rangé à l'adresse 0x80000001, l'octet 0x45 sera rangé à l'adresse 0x80000002 et l'octet 0x78 sera rangé à l'adresse 0x80000003)

Autres exemples : **clr.l (a2)**
le long d'adresse égale au contenu du registre a2 sera mis à zéro. ,Ainsi, si avant l'exécution de l'instruction a2 = 0x20000, c'est le long d'adresse 0x20000 qui est mis à zéro. (Plus précisément les octets d'adresses 0x20000, 0x20001 , 0x20002 et 0x20003 sont mis à zéro)

move.w (a2),(a4)
Le mot d'adresse "pointée" par a2 est lu puis rangé en mémoire à l'adresse pointée par a4.
Le code de cette dernière instruction est composé d'un mot (2 octets)
1 mot de codage de l'opération (voir pour plus détail l'instruction move)
Ce code est extrait de la mémoire de programme à l'adresse fournie à cet instant par le contenu du compteur ordinal (pc). Il est ensuite analysé.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**Size**		**Destination Adresse Effective**						**Source Adresse Effective**					
				Registre				**Mode**				**Mode**			**Registre**

Lorsque mot est codé dans sa zone « adresse effective de source » par le groupe binaire 010 010 alors le microprocesseur interprète ce champ comme désignant le registre a2 comme étant le contenant de l'adresse source où se situe l'opérande à lire (adressage indexé via a2). De même, si la zone « adresse effective de destination » est codée par le groupe binaire 100 010 alors le microprocesseur interprète ce champ comme désignant

le registre a4 comme étant le contenant de l'adresse destination où doit être écrit l'opérande destination (adressage indexé via a4).

En résumé si pc initial = 0x40e, si a2 initial = 0x30000000, si a4 initial = 0x60000000 et si cette région mémoire est constituée de mémoire RAM (écriture possible) . Si on suppose ces zones mémoire remplies (par exemple) par les valeurs :

0x30000000 | FF | FF | 0x60000000 | 12 | 34 |

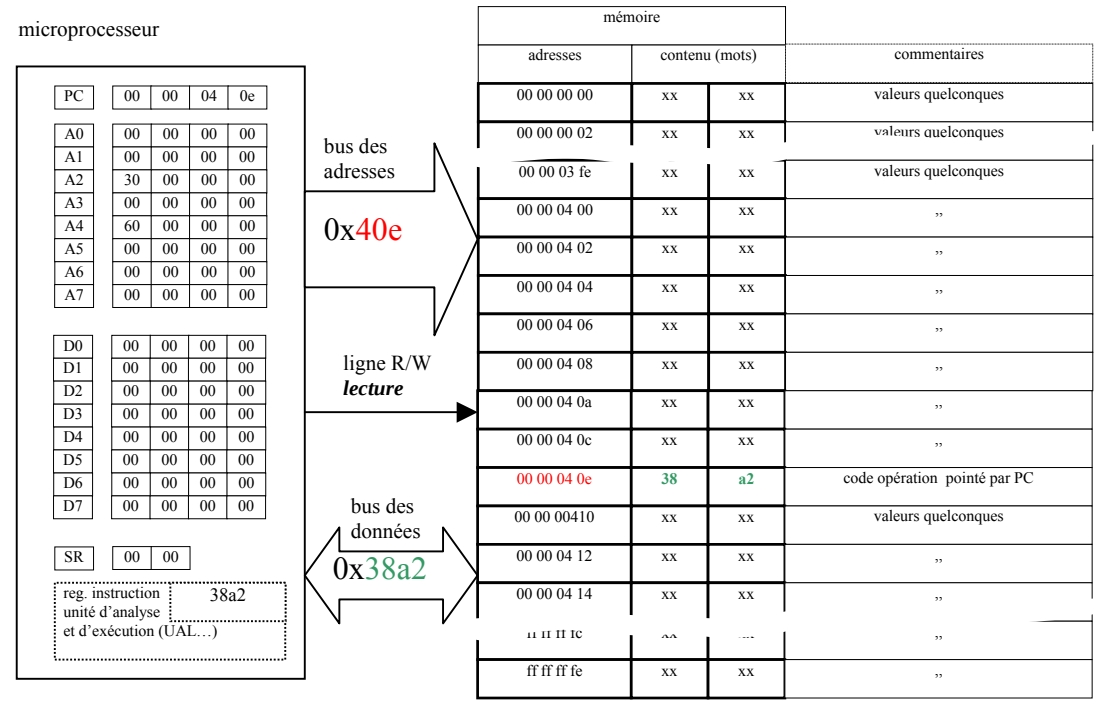
Si l'espace mémoire (organisé ici en mot) dans la région d'adresse 0x 40e est tel que :

adresses	valeurs	Explications
0x40e	0x38a2	code de l'instruction move.w (a2),(a4)

alors après l'exécution de cette instruction :
PC= 0x410 , a2 = 0x30000000 a4 = 0x60000000 et …

0x30000000 | FF | FF | 0x60000000 | FF | FF |

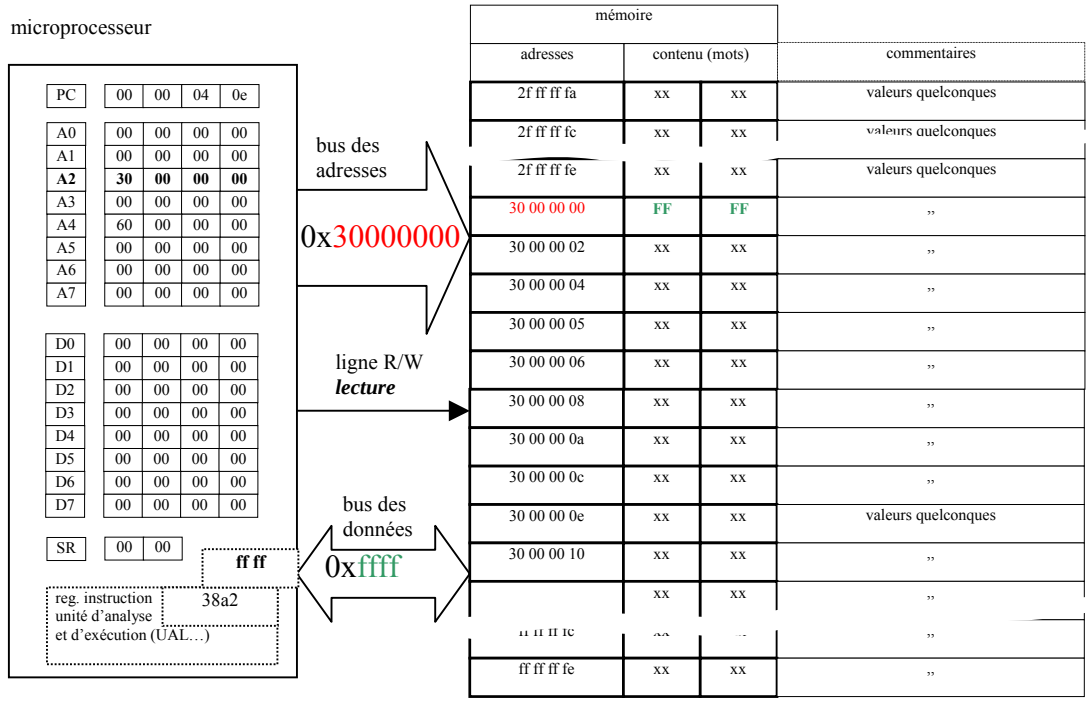
Cheminement des données lié à l'exécution de l'instruction: **move.w (a2),(a4)**



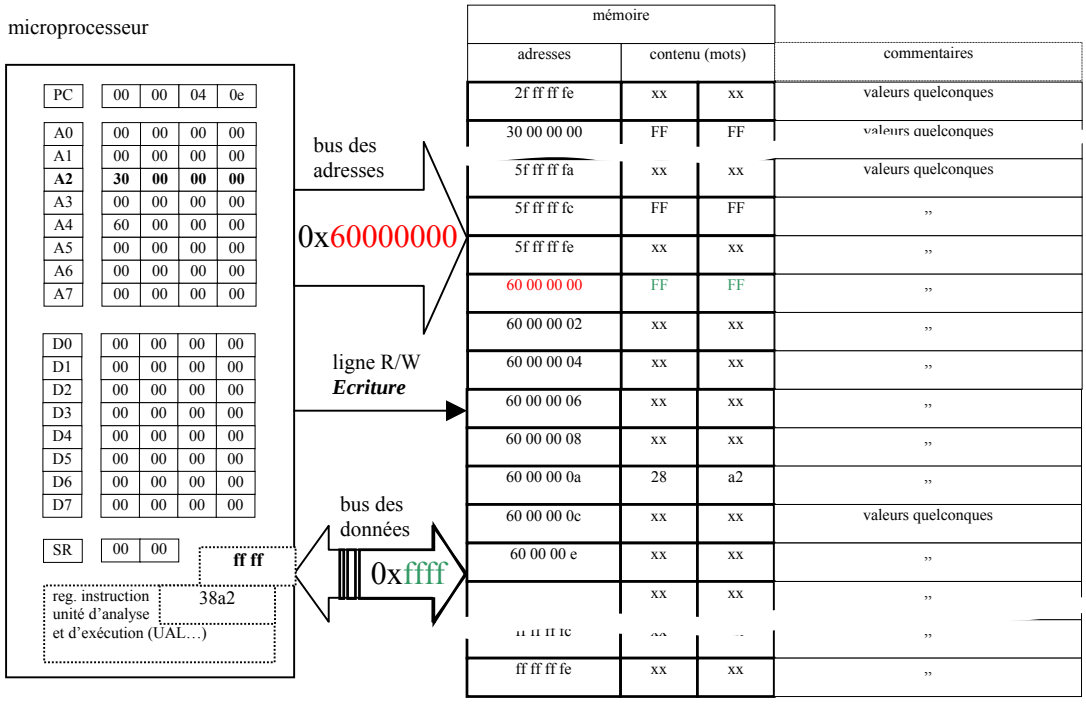
ETAPE 1 :
ACQUISITION DU CODE OPÉRATION
+ DECODAGE ➔ copier le mot d'adresse fournit par le contenu du reg. A2 à l'adresse fournit par le contenu du registre A4.

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE



ETAPE 2 :
ACQUISITION DE L'OPERANDE SOURCE (lue à l'adresse fournie par le registre a2)
(rangement dans un registre « intermédiaire »)



ETAPE 3 :
ECRITURE DE L'OPERANDE DESTINATION (stockée à l'adresse fournie par le registre a4)

Remarque : Compte tenu le la taille16 fils du bus des données, retenue dans cet exemple, si l'instruction analysée avait été move.l (a2),(a4) (et non move.w (a2),(a4)) , les différences auraient été :

_code opér. « légèrement » différent du fait de la taille (size) long (10) et non mot (11)
le code 0x382a devient ➔ 0x282a
_l'acquisition de l'opérande source aurait exigé deux étapes de lecture. Une première à l'adresse 0x30000000 pour y lire la partie haute (2octets) du « long source », suivie d'une seconde à l'adresse 0x30000002 pour y lire la partie basse (2ocets) du « long source ».
_l'émission de l'opérande destination aurait exigé deux étapes d'écriture. Une première à l'adresse 0x60000000 pour y écrire la partie haute (2octets) du « long destination », suivie d'une seconde à l'adresse 0x60000002 pour y écrire la partie basse (2ocets) du « long destination ».
Si le bus retenu avait été de taille 8 fils, alors pour cette même instruction move.l (a2),(a4) quatre cycles de lecture suivis de quatre cycles d'écriture auraient été nécessaires.
Conclusion : Plus le bus des données est large, plus le nombre de cycles pour accéder aux opérandes est faible (dès lors que l'opérande « visée » est de taille plus grande que la taille du bus des données). Un cycle mémoire « coûte » du temps (fonction des performances en temps d'accès des mémoires « visées ») , pour optimiser la vitesse d'exécution d'un programme il est donc nécessaire que l'architecture matérielle offres des caractéristiques, elles mêmes optimisée (ici la taille du bus des données = 32 fils).

?

-a- Quel est le mnémonique de l'instruction qui met, en mode indexé sans déplacement, le mot d'adresse 0x12345678 dans le registre d4 ? (aucun regsitre d'adresse ne contient la valeur 0x12345678) !!!
-b- Combien de mots sont nécessaires pour coder l'instruction move.b d4,(a0) ?
-c- Combien de cycles (d'accès mémoire) sont nécessaires (accès sur bus de 16 fils) pour exécuter l'instruction move.w (a0),d4 ?

⏩

-a- QUESTION PIEGE !! Cette operation est impossible en une seule instruction. Il est nécessaire de commencer par une initialisation d'un registre d'adresse (par exemple a0) avec la valeur d'adresse « visée« . D'ou les 2 instructions :

```
move.l   #0x12345678,a0   ;a0, regsitre d'adresse est initialisé avec la valeur 0x12345678.
                          ;l'instruction lea 0x12345678,a0 est tout aussi convenable !
move.w   (a0),d4          ;le registre d4 est chargé par le mot d'adresse 0x12345678.
```

-b- le code opération == 1 mot seulement !

-c- 1 cycle pour acquérir le code opération + 1 cycle pour acquérir le mot « source » à l'adresse fournie par le registre a0. L'écriture dans le registre d4 est interne au microprocesseur et ne « consomme » pas de cycle mémoire.

5.7.5 Mode d'adressage indexé avec déplacement.

SYNTAXE → dep(a***i***)

Note : dep est une contraction du mot déplacement.

L'adresse est fabriquée par le microprocesseur en ajoutant (addition arithmétique signée) aux 32 bits contenus dans le registre d'adresse désigné, les 16 bits signés du déplacement (étendus sur 32 bits).
C'est un mode d'adressage plus « encombrant » que le mode indexé, puisqu'il nécessite un mot supplémentaire pour contenir le déplacement. Par contre pour accéder à des

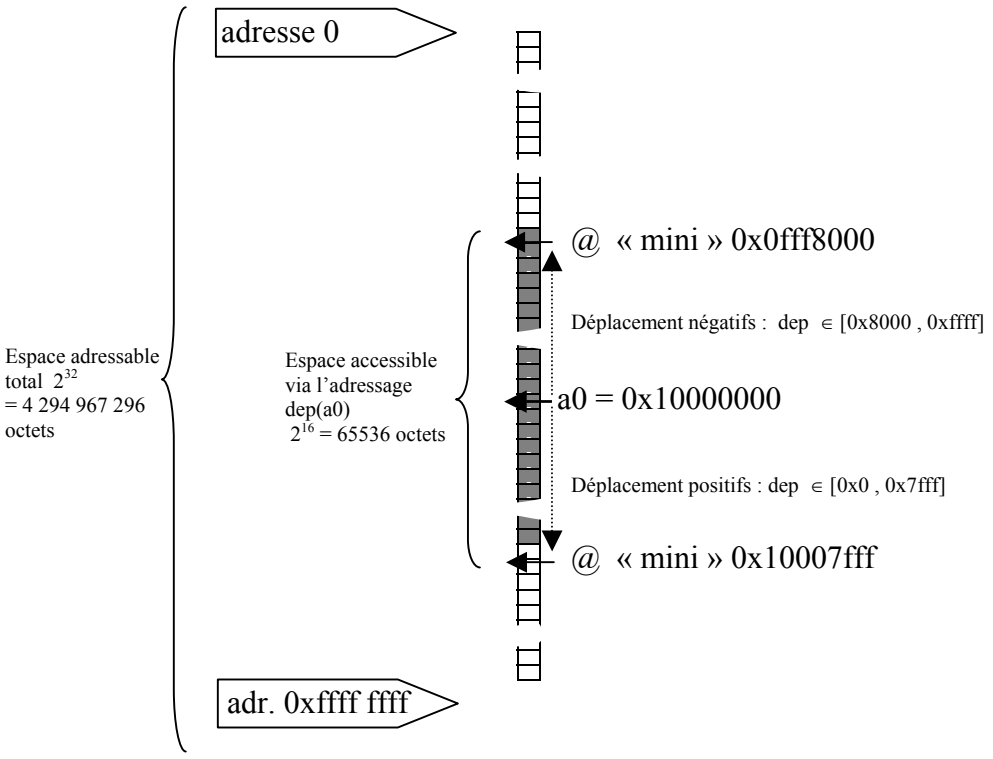
opérandes « proches » les unes des autres, il suffit de disposer d'un seul registre d'adresse initialisé avec une valeur (adresse) elle même proche des adresses des opérandes à atteindre pour pouvoir « pointer » chacun de ces opérandes, sans avoir a changer la valeur du registre d'adresse. L'adressage indexé sans déplacement aurait dans le même contexte nécessité autant d'initialisations distinctes de registres d'adresse qu'il y aurait eut d'opérandes.

Ainsi ce mode d'adressage permet d'accéder à , 2^{15} –1 soit 32767 octets (la moitié de mots et le quart de longs) au delà (après) de l'adresse contenue dans le registre d'adresse impliqué, et à 32767 octets (la moitié de mots et le quart de longs) au dessus de l'adresse contenue dans le registre d'adresse impliqué.

Ex : Si a0 = 0x10000000, alors :

0x8000(a0) permet d'accéder à l'octet (au mot ou au long) d'adresse 0x0fff8000
0x8001(a0) permet d'accéder à l'octet (au mot ou au long) d'adresse 0x0fff8001
…
0xffff(a0) permet d'accéder à l'octet (au mot ou au long) d'adresse 0x0fffffff
0x0(a0) permet d'accéder à l'octet (au mot ou au long) d'adresse 0x10000000
(là l'adressage indexé sans déplacement (a0) serait plus approprié !)
0x1(a0) permet d'accéder à l'octet (au mot ou au long) d'adresse 0x10000001
…
0x7fff(a0) permet d'accéder à l'octet (au mot ou au long) d'adresse 0x10007fff

Le dessin ci-après montre l'espace mémoire du microprocesseur MCF5307 et y précise le champ des opérandes accessibles (65536 « cellules octet » grisées) à partir du registre a0, ce dernier étant initialisé avec la valeur 0x10000000



Exemple : **move.l d0,12(a4)**

Il s'agit pour l'opérande source du mode d'adressage par registre (de donnée).
Ici c'est l'opérande destination qui est désigné par le mode d'adressage indexé via le registre a4.
Si avant l'exécution le registre d0 = 0x11223344 et le registre a4 = 0x80000000, alors après exécution de cette instruction, le long 0x11223344 sera rangé à l'adresse 0x8000000c.
(Plus précisément l'octet 0x11 sera rangé à l'adresse 0x8000000c, l'octet 0x22 sera rangé à l'adresse 0x8000000d, l'octet 0x33 sera rangé à l'adresse 0x8000000e et l'octet 0x44 sera rangé à l'adresse 0x8000000f)
Remarque : Si pour le déplacement une écriture hexadécimale telle que 0x12 avait été retenue (move.l d0,0x12(a4))
alors après exécution de cette instruction le long 0x11223344 sera rangé à l'adresse 0x80000012.
(Plus précisément l'octet 0x11 sera rangé à l'adresse 0x80000012, l'octet 0x22 sera rangé à l'adresse 0x80000013, l'octet 0x33 sera rangé à l'adresse 0x80000014 et l'octet 0x44 sera rangé à l'adresse 0x80000015)

Autres exemples : **clr.l 0xffff(a2)**
le long d'adresse égale au contenu du registre a2 + 0x*ffff**ffff (soit –1) sera mis à zéro.
*Les « *ffff* » résultent de l'extension de 16 à 32 bits réalisée systématiquement par le microprocesseur lors de la mise en œuvre de la somme ai + le déplacement.
Ainsi, si avant l'exécution de l'instruction a2 = 0x20000, c'est le long d'adresse 0x20000-1, soit 0x0001ffff qui est mis à zéro. (Plus précisément les octets d'adresses 0x1ffff, 0x20000 , 0x20001 et 0x20002 sont mis à zéro)
Le registre a2 n'est pas modifié l'usage de ce mode d'adressage. a2 final = 0x20000.
Remarque : l'écriture clr.l 0xffff(a2) aurait pu être remplacée par l'écriture « plus lisible » : clr.l -1(a2)

move.b 4(a2),-6(a4)
L'octet d'adresse "pointée" par a2 +4 est lu, puis rangé en mémoire à l'adresse pointée par a4 -6.
Le code de cette dernière instruction est composé d'un mot code opération + 2 mots d'extension pour spécifier les valeurs des déplacements « source » et « destination » .
Le mot code opération est extrait de la mémoire de programme à l'adresse fournie à cet instant par le contenu du compteur ordinal (pc). Il est ensuite analysé.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**Size**		**Destination Adresse Effective**						**Source Adresse Effective**					
				Registre				**Mode**			**Mode**			**Registre**	

Lorsque mot est codé dans sa zone « adresse effective de source » par le groupe binaire 101 010 alors le microprocesseur interprète ce champ comme désignant le registre a2 comme étant le contenant de l'adresse source associé (additionnée) à un déplacement (Mode 101) où se situe l'opérande à lire (adressage indexé via a2 avec déplacement). Le déplacement est implicitement situé en mémoire de programme juste après le code opération (en PC + 2)
De même, si la zone « adresse effective de destination » est codée par le groupe binaire 100 101 alors le microprocesseur interprète ce champ comme désignant le registre a4 comme étant le contenant de l'adresse source associé (additionnée) à un déplacement (Mode 101) où se situe l'opérande à lire (adressage indexé via a4).).
Le déplacement est implicitement situé en mémoire de programme juste après le code opération (en PC + 4)

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

En résumé si pc initial = 0x410, si a2 initial = 0x30000000, si a4 initial = 0x60000000 et si cette région mémoire est constituée de mémoire RAM (écriture possible) . Si on suppose ces zones mémoire remplies (par exemple) par les valeurs :

0x30000004 | 44 | 0x5ffffffA | 00 | Remarque : 0x5FFFFFFA = 0x60000000-6

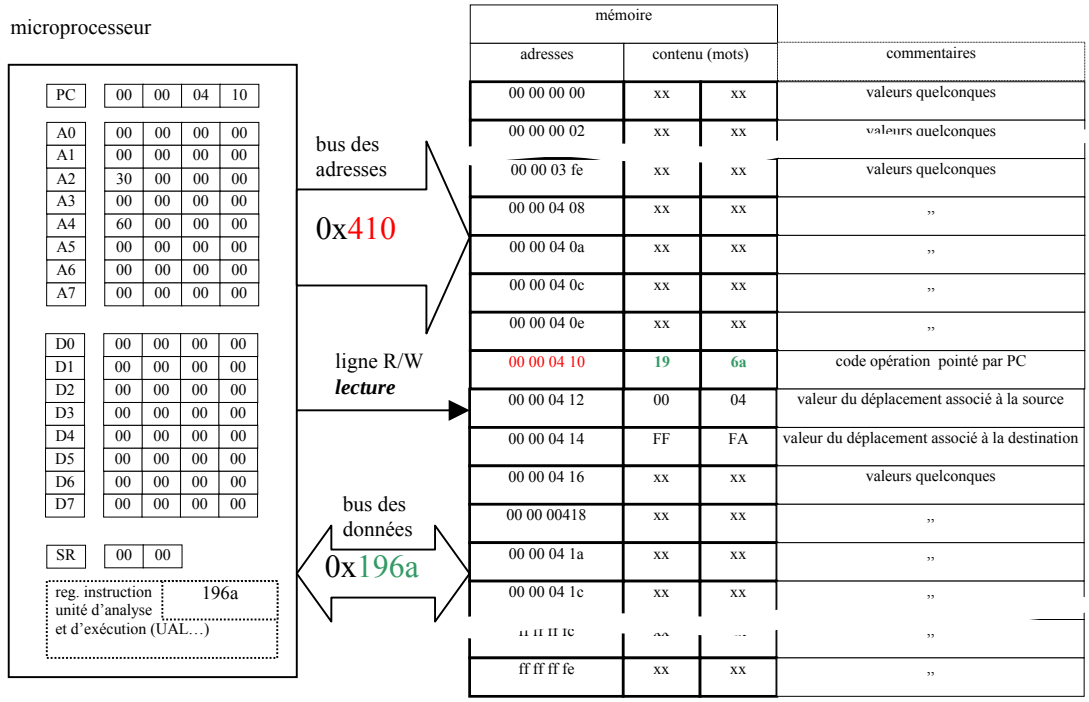
Si l'espace mémoire (organisé ici en mot) dans la région d'adresse 0x 410 est tel que :

adresses	valeurs	Explications
0x410	0x196a	code de l'instruction move.b 4(a2),-6(a4)
0x412	0x0004	valeur du déplacement source +4
0x414	0xfffa	valeur du déplacement destination (-6 ➔ 0xfffa)

alors après l'exécution de cette instruction :
PC= 0x416 , a2 = 0x30000000 a4 = 0x60000000 et …

0x30000004 | 44 | 0x5ffffffA | 44

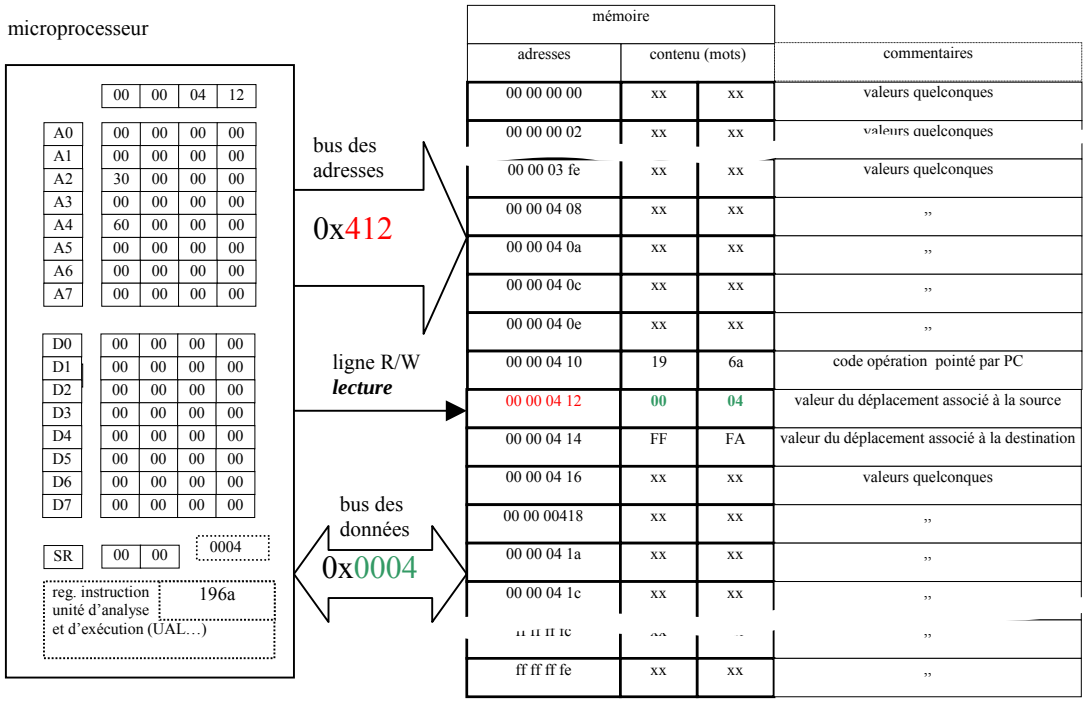
Cheminement des données lié à l'exécution de l'instruction : **move.b 4(a2),-6(a4)**



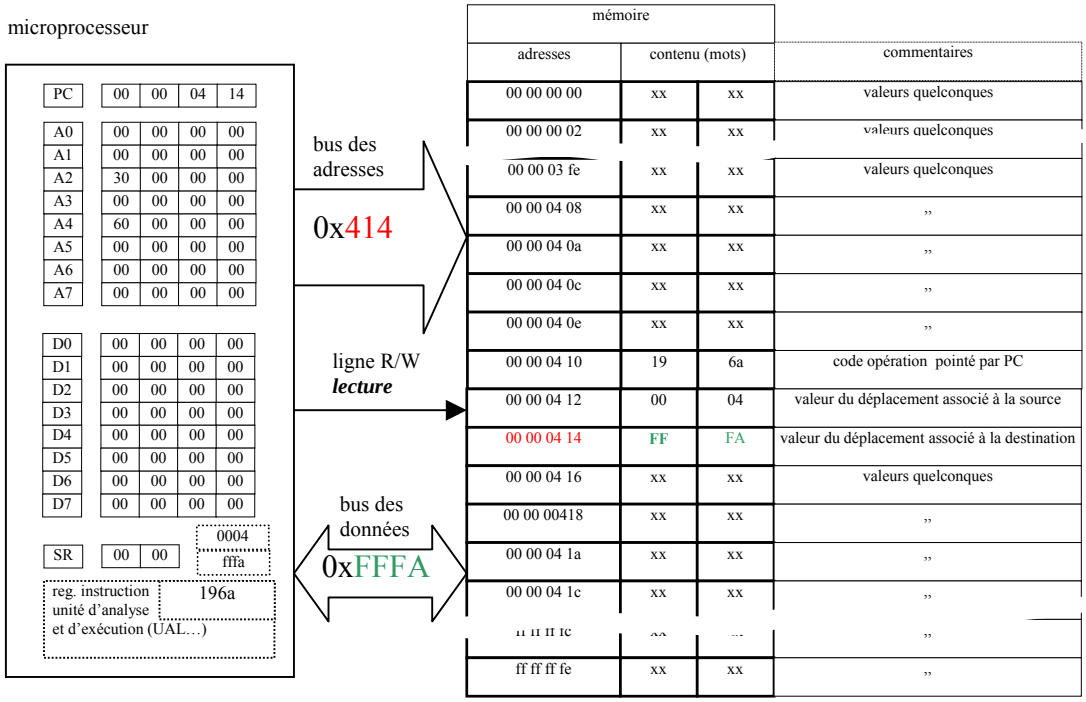
ETAPE 1 :
ACQUISITION DU CODE OPÉRATION + DECODAGE ➔ copier le mot d'adresse fournit par le contenu du reg. A2+ ? (déplacement source) à l'adresse fournit par le contenu du registre A4+ ? (déplacement destination).

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE



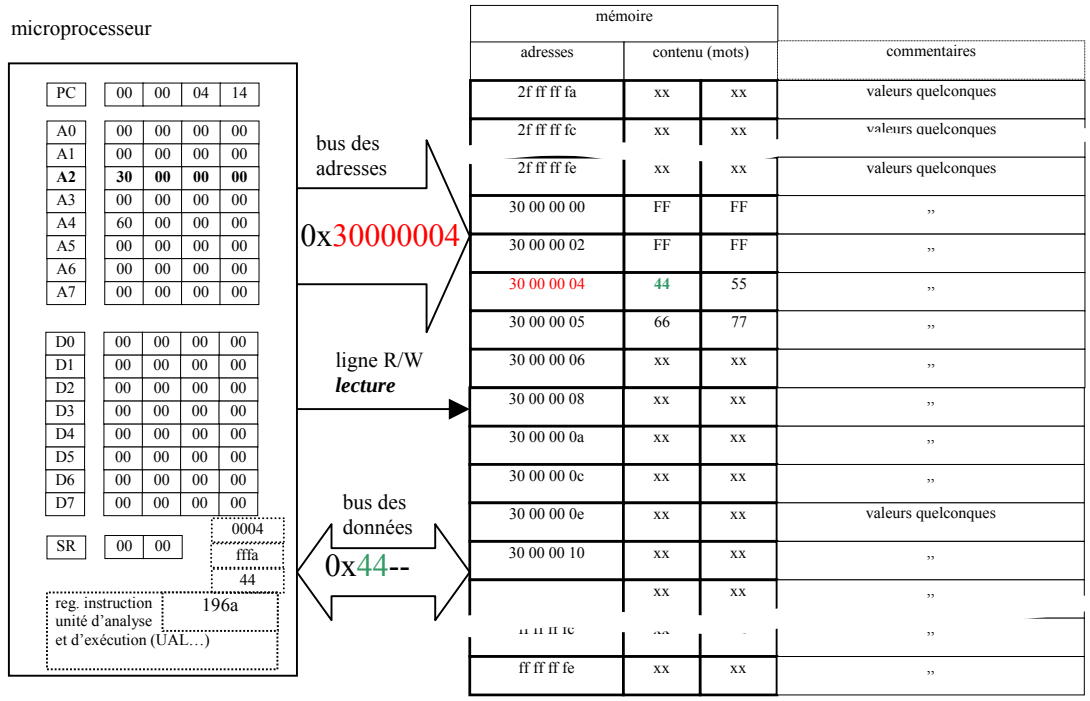
ETAPE 2 :
ACQUISITION DU MOT D'EXTENTION « déplacement source » ici +4 ➔ stockage provisoire.



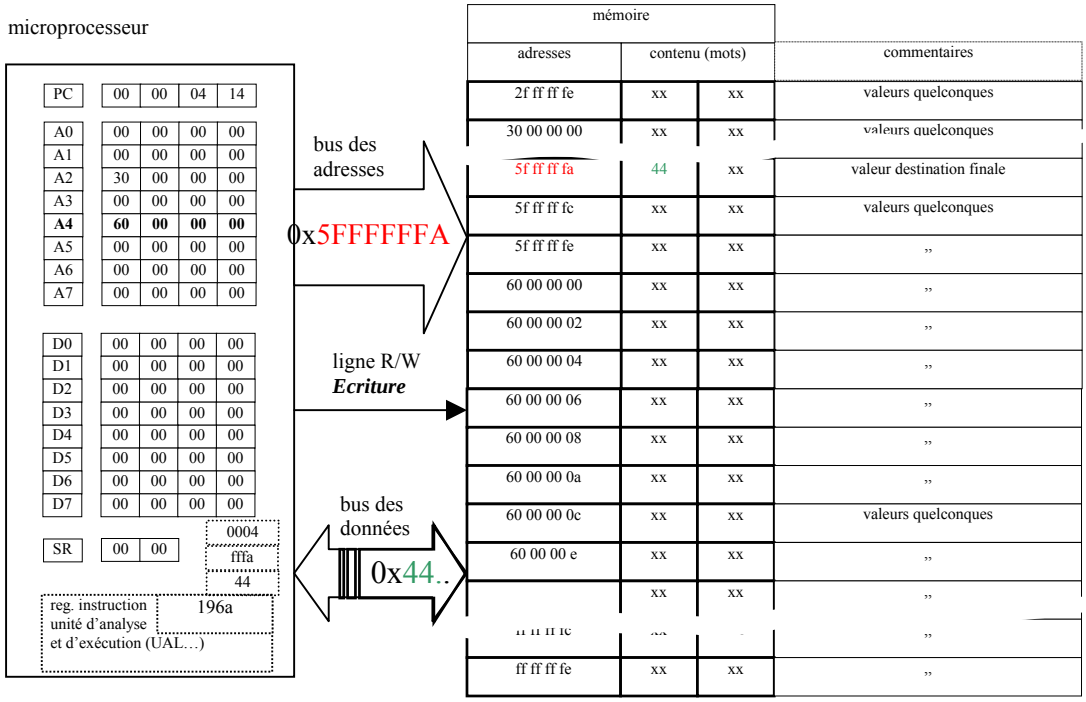
ETAPE 3 :
ACQUISITION DU MOT D'EXTENTION « déplacement destination » ici -6 ➔ stockage provisoire.

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE



ETAPE 4 :
ACQUISITION DE L'OPERANDE SOURCE (lue à l'adresse fournie par le registre a2+déplacement
➔ 0x30000000 + 0x00000004 = 0x30000004)
(rangement dans un registre « intermédiaire »)



ETAPE 5 :
ECRITURE DE L'OPERANDE DESTINATION (stockée à l'adresse fournie par le registre a4+ déplacement)
➔ 0x60000000 + 0xffffffa = 0x5ffffffa)

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

?

-a- *Quel est le mnémonique de l'instruction qui met, en mode indexé avec déplacement, le mot d'adresse 0x8 à l'adresse 0x700 sachant que le registre d'adresse a5 conient la valeur zéro ?*

-b- *Même question si la destination est 0x800 ?!*

-c- *Combien de cycles (d'accès mémoire) sont nécessaires (accès sur bus de 16 fils) pour exécuter l'instruction*
move.l 4(a0,)-12(a5) ?

⏩

-a- *move.w 8(a5), 0x700(a5)*

*_b_* *QUESTION PIEGE !! Cette operation est impossible en une seule instruction, car le déplacement « positif » maximun est 0x7fff. Exécuter 0x800(a5) conduit à ranger l'opérande à l'adresse : contenu de a0+dép. ➔ 0 + 0xffff800 ➔ 0xffff f800 !!!*
Il est nécessaire de commencer par une initialisation d'un registre d'adresse (par exemple a0) avec la valeur d'adresse « visée«, ici 0x800 . D'ou les 2 instructions :

```
move.w   #0x800,a0        ;a0, regsitre d'adresse est initialisé avec la valeur 0x12345678.
                          ;l'instruction lea 0x800,a0 est tout aussi convenable !
move.w   8(a5),0(a0)      ;le mot d'adresse 0x0000 0000 + 0x0000 0005 est rangé à l'adresse
                          ; 0x0000 0800 + 0 ! L'usage de l'instruction move.w 8(a5),(a0) eut été plus pertinante,
                          ;car en exploitant pour la destination l'adressage indexé sans déplacement, elle conduit
                          ;économiser la place du mot d'extension 0x0000 lié au mode indexé avec déplacement.
```

-c- *1 cycle pour acquérir le code opération + 1 cycle pour acquérir le mot d'extension « déplacement source » (0x0008) + 1 cycle pour acquérir le mot d'extension « déplacement destination » (0xfff4 == -12) + 2 cycles de lecture concécutifs pour lire l'opérande source + 2 cycles d'écriture concécutifs pour ranger l'opérande lue à sont adresse de destination.*

5.7.6 Mode d'adressage d'adressage indexé avec registre et déplacement.

SYNTAXE → dep(a**i**,***xi***)

Note : dep est une contraction du mot déplacement. xi signifie ai ou bien di

L'adresse est fabriquée par le microprocesseur en ajoutant (addition arithmétique signée) aux 32 bits contenus dans le registre d'adresse *ai* désigné, les 32 bits contenus dans le registre *di* ou *ai* (noté xi dans la syntaxe) plus les 8 bits signés du déplacement (étendus sur 32 bits).
C'est, comme le mode indexé avec déplacement, un mode d'adressage plus « encombrant » que le mode indexé, puisqu'il nécessite un mot supplémentaire pour contenir le déplacement de 8bits, le codage de l'indice *i* de x*i* , « la nature » registre de donnée ou d'adresse de *x* et d'autres éléments secondaires non vus dans ces pages*.
Notamment la possibilité d'exploiter xi en tant que long ou word via l'usage dans le mnémonique de l'indice de taille .l ou .w . Un facteur de multiplication de xi par 1,2 ou 4 est également possible. ex. move.b dep(a5,d3.w*4),d0 ou bien move.b dep(a5,d3.l*4), d0. Ces pages exploitent par omission de cet indice le cas général du registre xi exploité en tant que long, avec un facteur de multiplication de 1.

Ce mot d'extension a le format suivant :

D/A	N° du reg Xi	W/1	Fac. 1/2/3	Ev	Déplacement signé sur 8 bits

Di➔ D/A=0 ; Ai ➔ D/A=1, taille W ➔ W/l= 0 si ev=1 ; l➔ W/1=1, fact. 1(00) 2(01) 4 (10)
C'est un mode d'adressage parfaitement adapté à la gestion des tables (ou tableaux) . Le registre ai pointe alors sur le premier élément du tableau, tandis que le registre xi (par exemple di) contient l'indice d'accès au tableau. En général le déplacement n'est pas utile et vaut 0.
Ainsi par exemple, si en langage C on écrit :

```
main()
{
char      TAB[10] ;
char      Valeur ;
…
Valeur = TAB[3] ;
….
```

Pour « traduire » la ligne Valeur = TAB[3] ; le compilateur va générer les mnémoniques suivants :
(Prenons deux hypothèses : le tableau TAB est localisé en mémoire à l'adresse 0x20000000, et la variable Valeur est le contenu du registre d7)

```
lea       0x20000000,a2     ;a2 devient pointeur du tableau ➔ a2 == &TAB[0]
move.l    #3,d0             ;d0 devient l'indice d'accès au tableau, ici 3
                            ;sous forme d'une constante, mais le contenu de
                            ;d0 pourrait tout aussi bien être le résultat d'un calcul
                            ;C'est là que réside le « plus » de ce mode d'adressage.
move.b    0(a2,d0),d7       ;le déplacement constant non exploité ici, est mis à 0
                            ;d7 == Valeur, contient le 4eime élément du tableau TAB[3]
```

Même exemple, mais pour un tableau de int (32 bits)….

```
main()
{
int         TAB[10] ;
int         Valeur ;
…
Valeur = TAB[3] ;
….
```

Pour « traduire » la ligne Valeur = TAB[3] ; le compilateur va générer les mnémoniques suivants :
(Prenons deux hypothèses : le tableau TAB est localisé en mémoire à l'adresse 0x20000000, et la variable Valeur est le contenu du registre d7)

```
lea       0x20000000,a2     ;a2 devient pointeur du tableau ➔ a2 == &TAB[0]
move.l    #12,d0            ;d3 devient l'indice d'accès au tableau, ici 3*4
move.l    0(a2,d0),d7       ;le déplacement constant non exploité ici, est mis à 0
                            ;d7 == Valeur, contient le 4eime élément du tableau TAB[3]
```

Nous notons que le compilateur, au niveau de son travail de traduction en langage assembleur intègre le fait que l'objet est un long et assume le fait que dans le tableau il est bien à l'adresse de base augmentée de 12 et non de 3 comme dans le premier exemple (objet char == octet) !

Exemple : **move.l d0,12(a4,d1)**

Il s'agit pour l'opérande source du mode d'adressage par registre (de donnée).
Ici c'est l'opérande destination qui est désigné par le mode d'adressage indexé avec registre et déplacement. indexé via le registre d'adresse a4 avec usage du registre de donnée d1 et du déplacement de +12.
Si avant l'exécution le registre d0 = 0x11223344 , d1 = 0x00000004 et le registre a4 = 0x80000000, alors après exécution de cette instruction le long 0x11223344 sera rangé à l'adresse 0x80000010 == 0x80000000+0x00000004+0x0000000c.
(12 == 0xc)
(Plus précisément l'octet 0x11 sera rangé à l'adresse 0x80000010, l'octet 0x22 sera rangé à l'adresse 0x80000011, l'octet 0x33 sera rangé à l'adresse 0x80000012 et l'octet 0x44 sera rangé à l'adresse 0x80000013)

Autres exemples : **clr.l 0xfe(a2,d0)**
Le long d'adresse égale au contenu du registre a2 + 0x*ffffff**fe (soit –2) + d0 sera mis à zéro.
*Les « *ffffff* » résultent de l'extension de 8 à 32 bits, réalisée systématiquement par le microprocesseur lors de la mise en oeuvre de la somme ai + xi + le déplacement.
Ainsi, si avant l'exécution de l'instruction a2 = 0x20000, si d0 = 0x10, c'est le long d'adresse 0x20000-2+0x10, soit 0x0002000e qui est mis à zéro. (Plus précisément les octets d'adresses 0x2000e , 0x2000f , 0x20010 et 0x20011 sont mis à zéro)
Le registre a2 n'est pas modifié l'usage de ce mode d'adressage. a2 final = 0x20000.
Remarque : l'écriture clr.l 0xfe(a2,d0) aurait pu être remplacée par l'écriture « plus lisible » : clr.l -2(a2,d0)

move.w d7,-6(a4,d1)

Le mot bas contenu dans le registre d7 est rangé en mémoire à l'adresse pointée par a4 + d1 +-6.
Le code de cette dernière instruction est composé d'un mot code opération + 1 mot d'extension pour spécifier en autre la valeur du déplacement « destination » .
Le mot code opération est extrait de la mémoire de programme à l'adresse fournie à cet instant par le contenu du compteur ordinal (pc). Il est ensuite analysé.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**Size**		**Destination Adresse Effective**						**Source Adresse Effective**					
				Registre				**Mode**				**Mode**			**Registre**

Lorsque mot est codé dans sa zone « adresse effective de source » par le groupe binaire 000 111 alors le microprocesseur interprète ce champ comme désignant le registre d7 comme étant le contenant l'opérande source.
De même, si la zone « adresse effective de destination » est codée par le groupe binaire 100 110 alors le microprocesseur interprète ce champ comme désignant le registre a4 comme étant le contenant de l'adresse source associé (additinnée) à un déplacement (Mode 110) exprimé sur 8 bits (ici –6 == 0xfa) ainsi qu'au contenu du regsitre de donnée d1 où ce situe l'opérande à lire (adressage indexé avec registre et déplacement).
Le déplacement est implicitement situé en mémoire de programme dans un mot d'extension juste après le code opération (en PC + 2) (voir ci-avant)
En résumé si pc initial = 0x416, si d7 initial = 0x11223344, si a4 initial = 0x60000000 si d1 initial = 0x0000000c et si cette région mémoire (0x60000000+0xfffffffa+0x0000000c == 0x60000006) est constituée de mémoire RAM (écriture possible) . Si on suppose cette zone mémoire remplie (par exemple) par les valeurs :

0x60000006 | 0000 | Remarque : 0x60000006 == 0x60000000+0xfffffffa+0x0000000c

Si l'espace mémoire (organisé ici en mot) dans la région d'adresse 0x 410 est tel que :

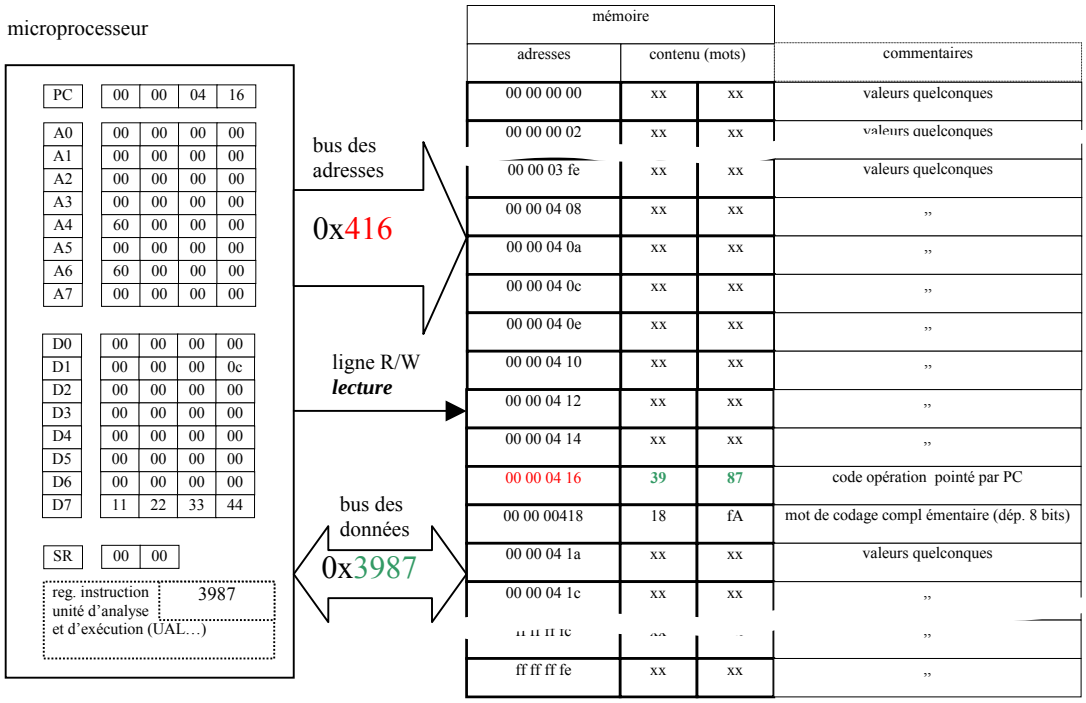
adresses	valeurs	Explications
0x416	0x3987	code de l'instruction move.w d7,-6(a4,d1)
0x418	0x18fa	codage de d1(.l) + valeur du déplacement destination -6

alors après l'exécution de cette instruction :
PC= 0x41a a4 = 0x60000000 et …
0x60000006 | 3344

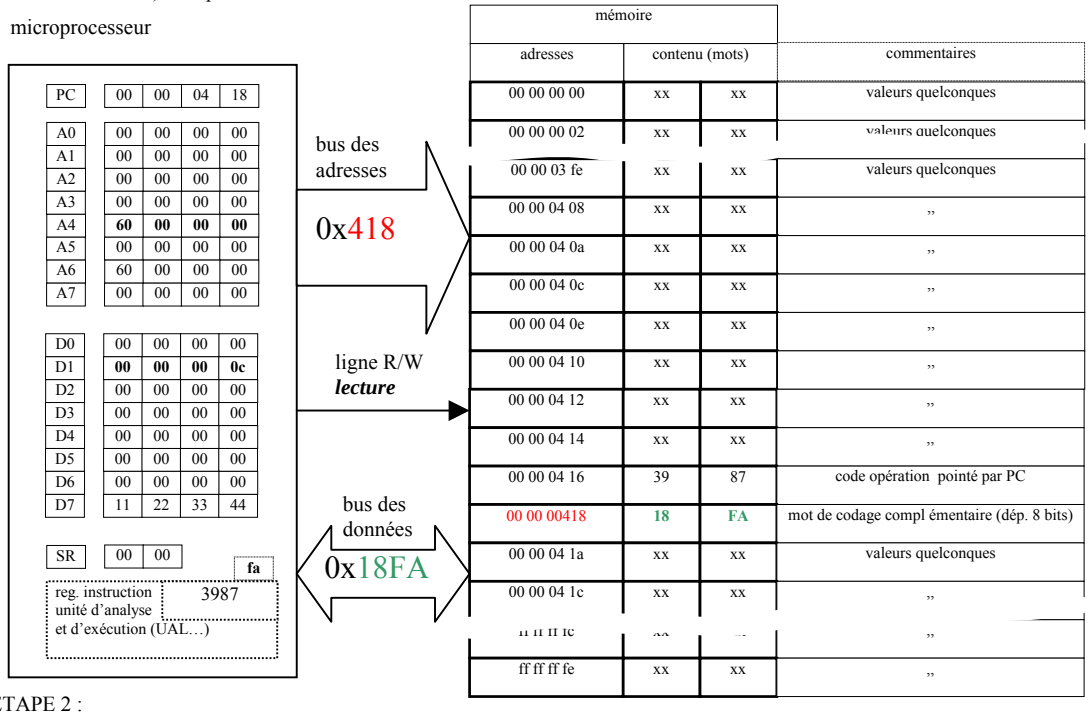
Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

Cheminement des données lié à l'exécution de l'instruction : **move.w d7,-6(a4,d1)**



ETAPE 1 :
ACQUISITION DU CODE OPÉRATION
+ DECODAGE ➔ copier le mot bas de d7 à l'adresse fournit par la somme du contenu du reg. A4+ ? (déplacement source) à lire dans le mot d'extension suivant + le contenu du registre ? dont le numéro et la nature (donnée ou adresse) sont précisés dans le mot d'extension suivant.

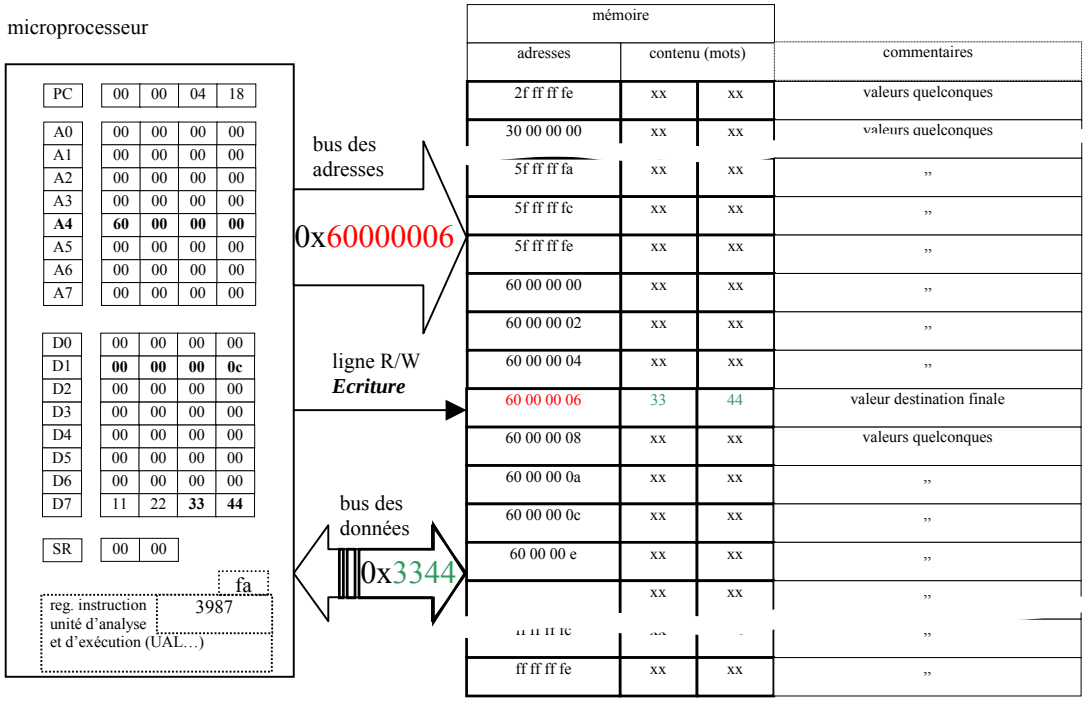


ETAPE 2 :
ACQUISITION MOT D'EXTENSION stockage temporel pour décodage
➔ déplacement 0xfa == -6 ➔0xfffffffa
+ registre d1 == 0x0000000c ➔0x0000000c
+ regsitre a4 == 0x60000000 ➔0x60000000

= adresse de destination ➔0x60000006

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE



ETAPE 3 :
ECRITURE DE L'OPERANDE DESTINATION Ici la partie basse du registre d7.
d7.w == 0x3344 ➔ 0x60000000 + 0xffffffa + 0x0000000c = 0x60000006)

?

-a- *Quel est le mnémonique de l'instruction qui met, en mode indexé avec registre et déplacement, à zéro le mot d'adresse 0x80000000 sachant que le registre d'adresse a5 contient la valeur 0x70000000 et que le registre d6 contient la valeur 0x0fffffff ?*

-b- *Même question si la destination est 0x0 , a5 = 0x7000000 et d6 est à initialiser ?*

-c- *Combien de cycles (d'accès mémoire) sont nécessaires (accès sur bus de 16 fils) pour exécuter l'instruction clr.l 0(a5,d6) ?*

⏩

-a- *clr.l 1(a5,d6) le long d'adresse 0x7000000 + 0x0fffffff + 1 == 0x80000000 sera mis à zero.*

_b_ *Il faut que 0x70000000 + dép sur 8 bits + d6 = 0x800. Si le déplacement est pris nul.*
0x70000000 + 0 + ? = 0x800 ➔ ?= 0x90000800 . Donc si d6 = 0x90000800 (nombre négatif), alors
clr.l 0(a5,d6) mettra à zéro le long d'adresse : 0x70000000+0+0x90000800 == 0x800 ! (sur 32 bits)
Donc il est nécessaire de commencer par une initialisation du registre d6 avec la valeur 0x90000800 . D'ou les 2 instructions :

```
move.l  #0x90000800,d6   ;d6, registre de donnée initialisé avec la valeur 0x90000800.
clr.l   0(a5,d6)         ;le long  d'adresse 0x800 est mis à 0
```

-c- *1 cycle pour acquérir le code opération + 1 cycle pour acquérir le mot d'extension spécifique à ce mode d'adressage. C'est en particulier dans ce mot d'extension que ce trouve le déplacement … ici nul !*
+ 2 cycles d'écriture concécutifs pourmettre à zéro le long (2 motys) à l'adresse élaborée par le microprocesseur, à savoir l'adresse 0x800 dans les conditions proposées à la question –b-.
En pratique un programmeur écrira plus simplement : clr.l (a5,d6). Le déplacement nonexplicité sera en fait nul. Il sera effectivement codé 00 dans le mot d'extension.

5.7.7 Mode d'adressage post-incrémenté

SYNTAXE → (**ai**)+

L'adresse utilisée par le microprocesseur est l'adresse contenue dans le registre d'adresse *ai* désigné.
L'opération spécifiée par l'instruction étant faite, le microprocesseur modifie ensuite le registre *ai* :

ai ← *ai* + 1 si l'instruction porte sur un opérande octet (byte)
ai ← *ai* + 2 si l'instruction porte sur un opérande mot (word)
ai ← *ai* + 4 si l'instruction porte sur un opérande long.

Ce mode d'adressage modifie le registre d'adresse qu'il implique, après exécution de l'instruction !
C'est un mode d'adressage parfaitement adapté à la gestion de suites d'opérandes adjacents (voisin). Cette suite doit avoir une structure « classique » de table, c'est à dire qu'elle doit se développer d'adresses petites vers des adresses grandes.
(exemple une liste d'octets rangés de l'adresse 0x10000 à l'adresse 0x10300, le premier octet étant à l'adresse 0x10000, le dernier étant à l'adresse 0x10300).
Par exemple, copier N octets (mots ou longs) d'un espace mémoire vers un autre espace mémoire. Il suffit d'initialiser deux registres d'adresses respectivement avec l'adresse de début de la zone « source » et avec l'adresse de début de la zone destination. Ensuite dans une boucle il suffit de répéter N fois une instruction de type : move.b/w/l (ai source),(aj destination)
Le mode d'adressage post-incrémenté assurera lors de l'exécution ai++ et aj++
(++ == incrémentation de 1 pour choix .b, de 2 pour choix .w, de 4 pour choix .l)

Mise en œuvre de cet exemple pour recopier 500 mots présents à partir de l'adresse source 0x10000000 vers l'adresse destination 0x40000000

```
            lea         0x10000000,a0   ;a0 pointeur de la zone source
            lea         0x40000000,a1   ;a1 pointeur de la zone destination
            move.l      #500,d0         ;d0 compteur de mots à copier
boucle :    move.w (a0)+,(a1)+          ;copie et auto incrémentation
                                        ;ici à chaque exécution a0 ← a0+2
                                        ;et a1 ← a1+2
            subq.l      #1,d0           ; «compteur de mots» décrementé
            bne.s       boucle          ;test si 500 boucles « effectuées »
            …
```

Remarque : Pour optimiser la vitesse d'exécution de ce programme il eut-été plus rationnel de transférer 250 longs et non 500 mots.
Le programme devient :

```
            lea         0x10000000,a0   ;a0 pointeur de la zone source
            lea         0x40000000,a1   ;a1 pointeur de la zone destination
            move.l      #250,d0         ;d0 compteur de mots à copier
boucle :    move.l      (a0)+,(a1)+     ;copie et auto incrémentation
                                        ;ici à chaque exécution a0 ← a0+4
                                        ;et a1 ← a1+4
            subq.l      #1,d0           ; «compteur de longs» decrémenté
            bne.s       boucle          ;test si 500 boucles « effectuées »
            …
```

« L'avance » des pointeurs est prise en charge implicitement par le mode d'adressage.
Le second exemple (.long), en l'abscence de ce mode deviendrait :

```
        lea     0x10000000,a0   ;a0 pointeur de la zone source
        lea     0x40000000,a1   ;a1 pointeur de la zone destination
        move.l  #250,d0         ;d0 compteur de mots à copier
boucle : move.l (a0),(a1)       ;copie
        add.l   #4,a0           ;ici à chaque exécution il faut faire a0 ← a0+4
        add.l   #4,a1           ;puis ici il faut faire a1 ← a1+4
        subq.l  #1,d0           ; «compteur de longs» decrémenté
        bne.s   boucle          ;test si 500 boucles « effectuées »
```

Il est évidant que ce mode d'adressage est, dans cet algorithme de gestion d'opérandes voisins, particulièrement efficace.

Autre exemple : **move.l d7,(a5)+**

Le long contenu dans le registre d7 est rangé en mémoire à l'adresse pointée par a5 , puis a5 est modifié et devient égal à a5 + 4.
Le code de cette dernière instruction est composé d'un mot code .
Le mot code opération est extrait de la mémoire de programme à l'adresse fournie à cet instant par le contenu du compteur ordinal (pc). Il est ensuite analysé.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**Size**		**Destination Adresse Effective**						**Source Adresse Effective**					
				Registre				**Mode**				**Mode**			**Registre**

Lorsque le mot est codé dans sa zone « adresse effective de source » par le groupe binaire 000 111 alors le microprocesseur interprète ce champ comme désignant le registre d7 comme étant le contenant de l'opérande source.
De même, si la zone « adresse effective de destination » est codée par le groupe binaire 101 011 alors le microprocesseur interprète ce champ comme désignant le registre a5 comme étant le contenant de l'adresse à laquelle se trouve l'opérande destination. Puis après exécution de l'opération (ici move) le microprocesseur réalisera une incrémentation de 4 unités (du fait du codage long 10 du champ size) du registre impliqué a5.
En résumé si pc initial = 0x41a, si d7 initial = 0x11223344, si a5 initial = 0x60000000 et si cette région mémoire (0x60000000/1/2/3) est constituée de mémoire RAM (écriture possible). Si on suppose cette zone mémoire remplie par les valeurs :

0x6000000	00000000

Si l'espace mémoire (organisé ici en mot) dans la région d'adresse 0x 41a est tel que :

adresses	valeurs	Explications
0x41a	0x2ac7	code de l'instruction move.l d7,(a5)+

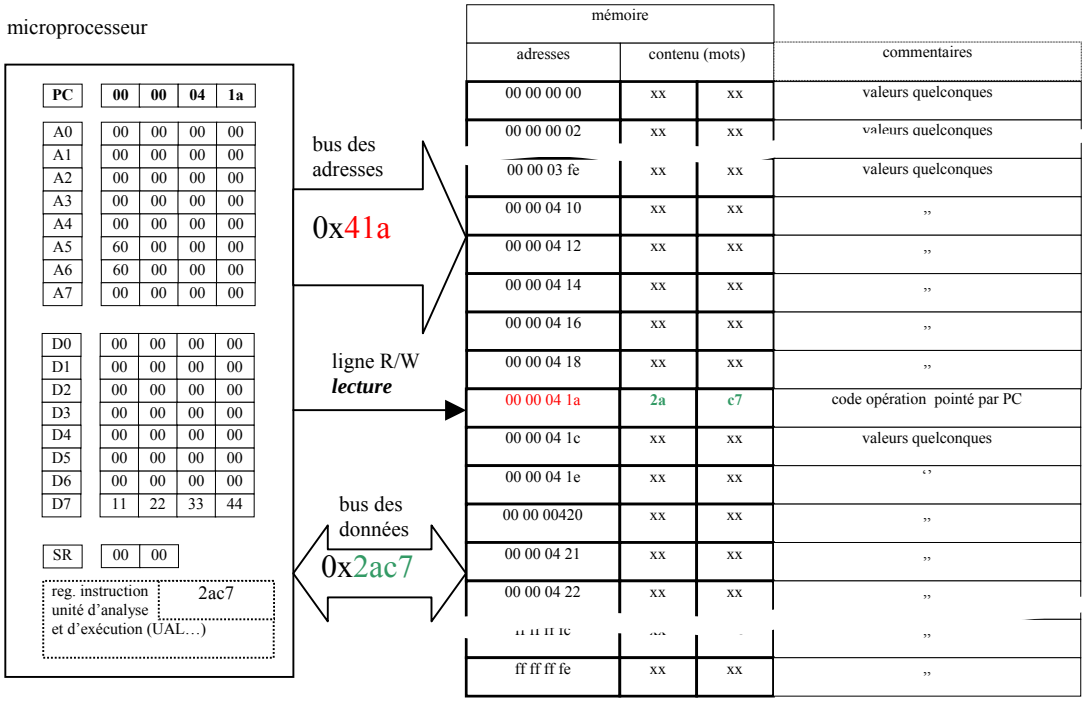
alors après l'exécution de cette instruction :
PC= 0x41c a5 = 0x60000004 et …

0x60000000	11223344

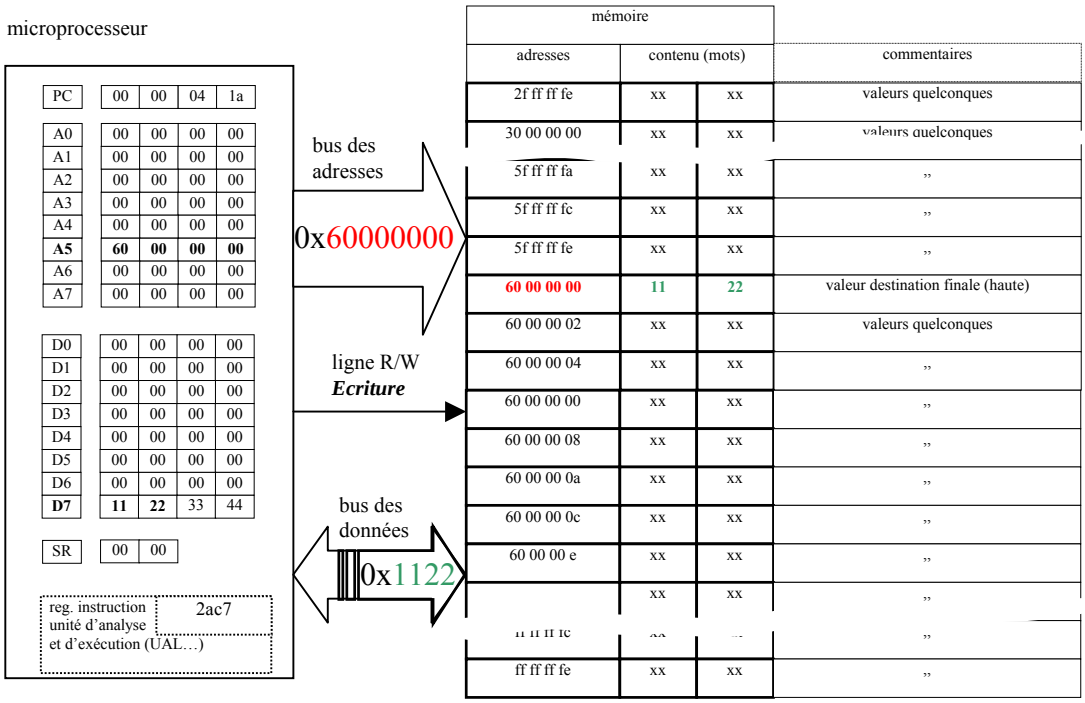
Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

Cheminement des données lié à l'exécution de l'instruction : **move.l d7,(a5)+**



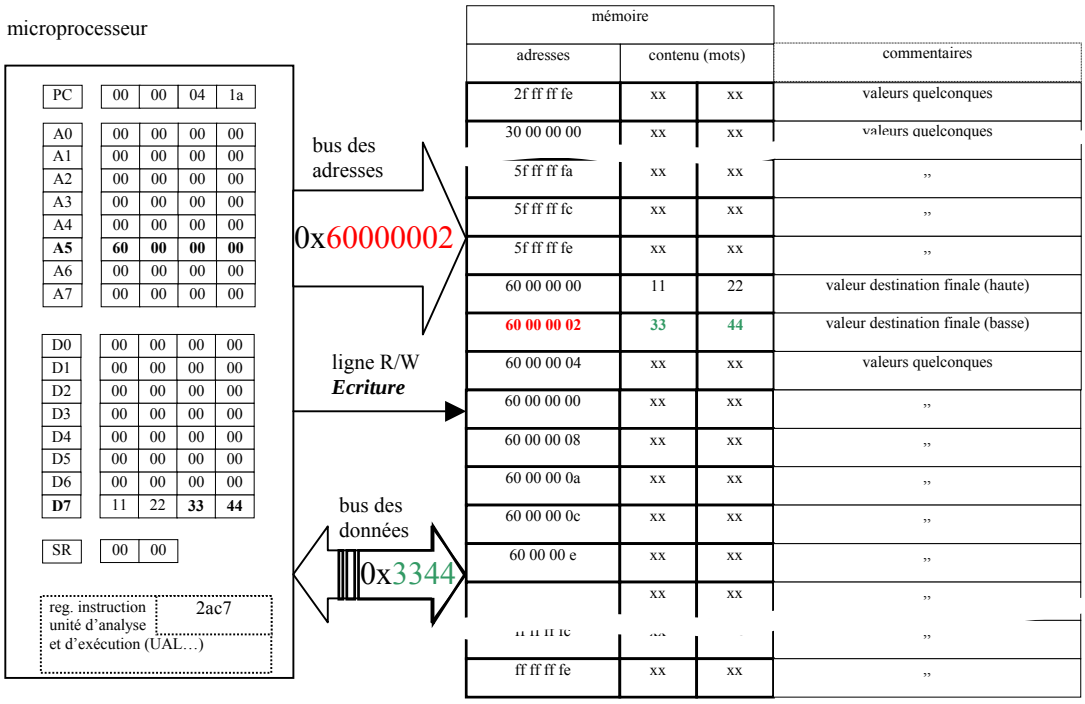
ETAPE 1 :
ACQUISITION DU CODE OPÉRATION
+ DECODAGE ➔ copier le registre (long) d7 à l'adresse fournie par le contenu du registre d'adresse A5.
Après exécution de ce « move », additionner +4 au contenu du registre a5.



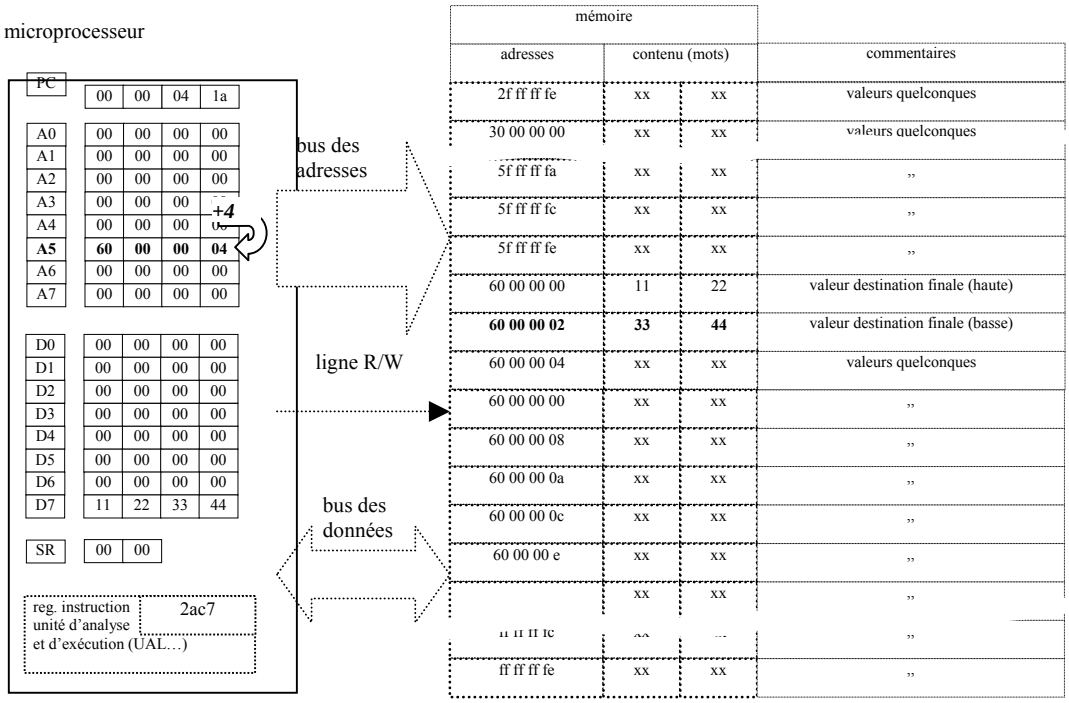
ETAPE 2 :
ECRITURE DE L'OPERANDE DESTINATION Ici la partie haute du registre d7.
d7.l == 0x**1122**3344 ➔ 0x60000000

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE



ETAPE 3 :
ECRITURE DE L'OPERANDE DESTINATION Ici la partie basse du registre d7.
d7.l == 0x1122**3344** ➔ 0x60000002



ETAPE 4 :
Activité interne : A5 <== A5 + 4 ; A5 = 0x6000 0004

?

-a- *Quel est le mnémonique de l'instruction qui réalise sur un long (32 bits), en mode post-incrémenté, un OU logique entre le contenu du regsitre d7 et le long d'adresse 0x80000000 sachant que le registre d'adresse a5 contient la valeur 0x80000000. Le résultat doit être rangé en 0x80000000 ? Quelle est la valeur du registre a5 à l'issus de l'exécution de cette instruction ?*

-b- *Combien de cycles (d'accès mémoire) sont nécessaires (accès sur bus de 16 fils) pour exécuter cette instruction ?*

⏩

-a- *or.l d7,(a5)+ ; a5 final = 0x80000004*

-b- *1 cycle pour acquérir le code opération*
+ 2 cycles pour acquérir l'opérande long d'adresse 0x80000000. (à cause du bus des données de 16 fils cet accès prend 2 cycles [32 = 16+16] de lectures)
l'opération d7 || (0x80000000) est réalisée en interne au sein de l'unité arithmétique et logique du microprocesseur U.A.L)
+ 2 cycles d'écriture concécutifs pour ranger le long (2 mots) résultat en 0x80000000.
Enfin, en interne, le microprocesseur ajoute 4 (action sur un opérande long) au registre impliqué dans le mode d'adressage pos-incrémenté. Ici a5= a5+4 ➔ a5 = 0x80000004

5.7.8 Mode d'adressage pré-décrémentation

v

SYNTAXE → - (**a**i)

5.7.9

Ce mode d'adressage modifie le registre d'adresse qu'il implique, avant exécution de l'instruction !
Le microprocesseur modifie le registre *ai* :

ai ← *ai* - 1 si l'instruction porte sur un opérande octet (byte)
ai ← *ai* - 2 si l'instruction porte sur un opérande mot (word)
ai ← *ai* - 4 si l'instruction porte sur un opérande long.

L'adresse utilisée par le microprocesseur pour réaliser l'opération est l'adresse contenue dans le registre d'adresse *ai* ainsi modifié.

C'est un mode d'adressage parfaitement adapté à la gestion de suites d'opérandes adjacents (voisin). Cette suite doit avoir une structure de pile (stack), c'est à dire qu'elle doit se développer d'adresses grandes vers des adresses petites.
(exemple une suite de longs résultant de la sauvegarde de registres en pile lors d'un mécanisme d'interruption. Si la valeur initiale du pointeur de pile est 0x1000, alors le premier long rangé est à l'adresse 0x10000- 4 (soit 0xfffc) le second à l'adresse 0x10000 –8 (soit 0xfff8) … le n^{ieme} long est rangé à l'adresse 0x10000 – (n x 4).
Le mode d'adressage pré-décrémenté permet en particulier d'émuler (de faire comme) un pointeur de pile avec un registre d'adresse.

Il trouvera également un usage adapté à l'arithmétique des nombres de plus de 4 octets (multiple-précision) lorsque ces derniers, c'est le cas des microprocesseurs Motorola, sont de type « big indian ».

Par exemple, additionner 2 nombres de 12 octets (3 longs), respectivement rangés à partir des adresses 0x10000 [0x10000 (octet msb) → 0x1000b (octet lsb)] et 0x20000 [0x20000 (octet msb) → 0x2000b (octet lsb)]. La somme (supposée de 12 octets) sera rangée à l'adresse 0x30000 [0x30000 (octet msb) → 0x3000b (octet lsb)]. Cette zone mémoire sera supposée de type RAM.

Algorithme :
Copier par morceaux des poids faibles vers les poids forts les 2 nombres dans des registres du microprocesseur (par exemple les registres d0 et d1), en faire la somme en utilisant une instruction d'addition adaptée pour les parties 2 et 3 (addx.l) puis ranger les résultats (dans le bon ordre) à partir de l'adresse 0x30000.

Mise en œuvre de cet exemple :

```
lea       0x10000+12,a0   ;a0 pointeur de la zone source nbr. 1 en 0x1000c
lea       0x20000+12,a1   ;a1 pointeur de la zone source nbr. 2 en 0x2000c
lea       0x30000+12,a2   ;a1 pointeur de la zone resultat en 0x3000c

move.l    -(a0),d0        ;d0 long lsb du nbr. 1, et a0= 0x10008
move.l    -(a1),d1        ;d1 long lsb du nbr. 2, et a0= 0x20008
add.l     d1,d0           ;d0= lsb de la somme des « lsb » des nbr. 1 et 2
move.l    d0,-(a2)        ;resultat lsb rangé en 0x30008/9/c/b et
                          ;a2= 0x30008
move.l    -(a0),d0        ;d0 long milieu du nbr. 1, et a0= 0x10004
move.l    -(a1),d1        ;d1 long milieu du nbr. 2, et a0= 0x20004
addx.l    d1,d0           ;d0= « milieu » de la somme des « milieux» des
                          ;nbr. et 2 en tenant compte du report (bit X) issu de
                          ;la somme  des parties lsb.
move.l    d0,-(a2)        ;resultat milieu rangé en 0x30004/5/6/7 et
                          ;a2= 0x30004
move.l    -(a0),d0        ;d0 long msb du nbr. 1, et a0= 0x10000
move.l    -(a1),d1        ;d1 long msb du nbr. 2, et a0= 0x20000
addx.l    d1,d0           ;d0= msb de la somme des msb des nbr. 1
                          ;et 2 en tenant compte du report (bit X) issu de la
                          ;somme  des parties « milieu».
move.l    d0,-(a2)        ;resultat msb rangé en 0x30000/1/2/3 et
                          ;a2= 0x30000
```

« Le recul » des pointeurs est pris en charge implicitement par de mode d'adressage pré-décrémenté.

Il est évidant que ce mode d'adressage est, dans cet algorithme de gestion d'opérandes voisins en « arrangement » de type pile, particulièrement efficace.

Autre exemple : **move.w d7,-(a5)**

Le registre a5 est d'abord modifié ; a5 <== a5 -2 (-2 car taille .w), puis le mot bas contenu dans le registre d7 est rangé en mémoire à l'adresse pointée par a5.
Le code de cette dernière instruction est composé d'un mot code .
Le mot code opération est extrait de la mémoire de programme à l'adresse fournie à cet instant par le contenu du compteur ordinal (pc). Il est ensuite analysé.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**Size**		**Destination Adresse Effective**						**Source Adresse Effective**					
				Registre				**Mode**				**Mode**			**Registre**

Lorsque mot est codé dans sa zone « adresse effective de source » par le groupe binaire 000 111 alors le microprocesseur interprète ce champ comme désignant le registre d7 comme étant le contenant de l'opérande source.
De même, si la zone « adresse effective de destination » est codée par le groupe binaire 101 101, alors le microprocesseur interprète ce champ comme désignant le registre a5 qui, une fois modifié, sera le contenant de l'adresse à laquelle se trouve l'opérande destination.
Le microprocesseur réalisera donc tout d'abord une décrémentation de 2 unités (du fait du codage long 11 du champ size) du registre impliqué a5, puis il réalisera l'opération (ici move)

En résumé si pc initial = 0x41c, si d7 initial = 0x11223344, si a5 initial = 0x60000000 et si cette région mémoire (0x5fff fffc/d/e/f) est constituée de mémoire RAM (écriture possible) .
Si on suppose cette zone mémoire remplie (par exemple) par les valeurs :

0x5fff fffc	00000000

Si l'espace mémoire (organisé ici en mot) dans la région d'adresse 0x 41a est tel que :

adresses	valeurs	Explications
0x41c	0x3b47	code de l'instruction move.w d7,-(a5)

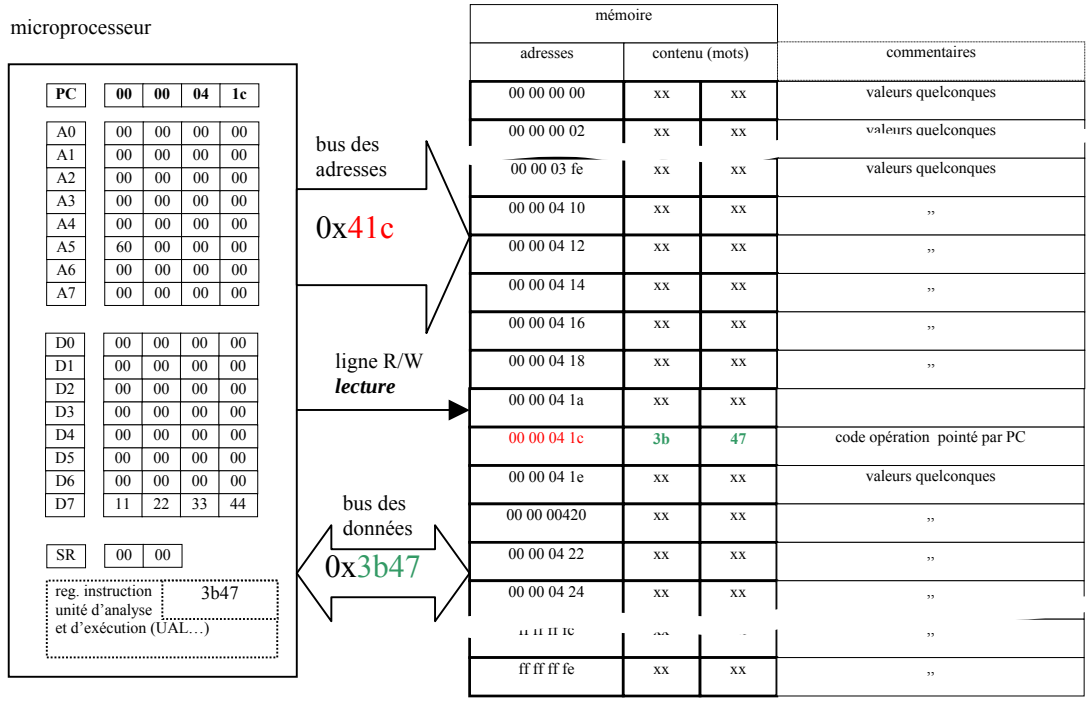
alors après l'exécution de cette instruction :
PC= 0x41e a5 = 0x5fff fffc et …

0x5fff fffc	11223344

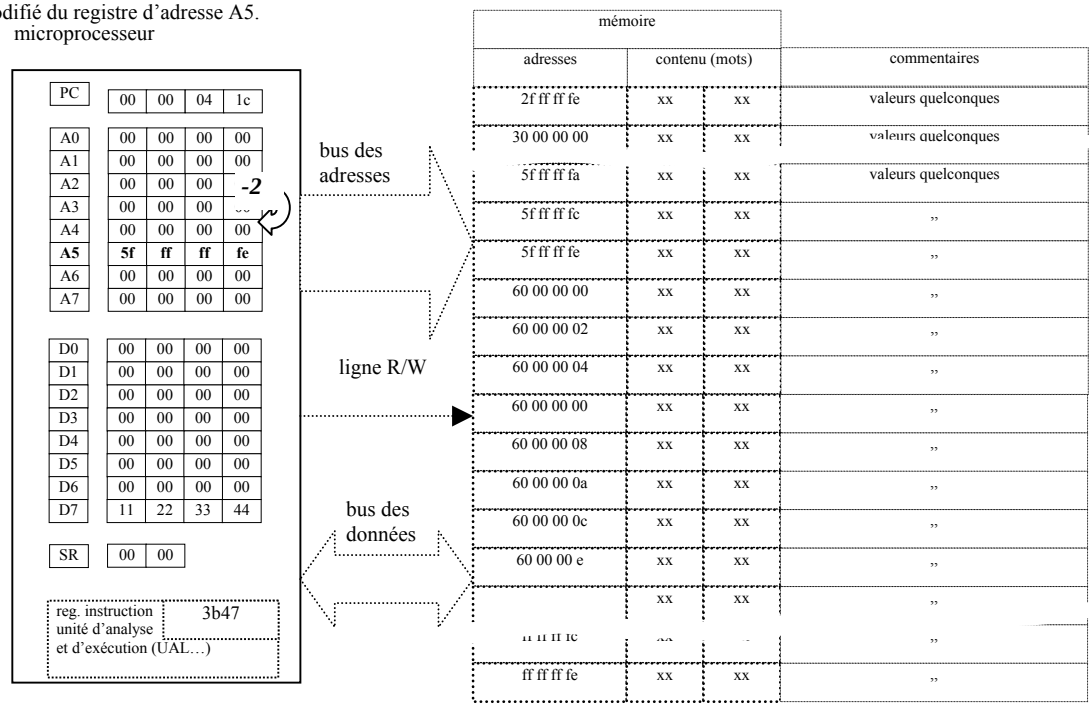
Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE

Cheminement des données lié à l'exécution de l'instruction : **move.w d7,-(a5)**



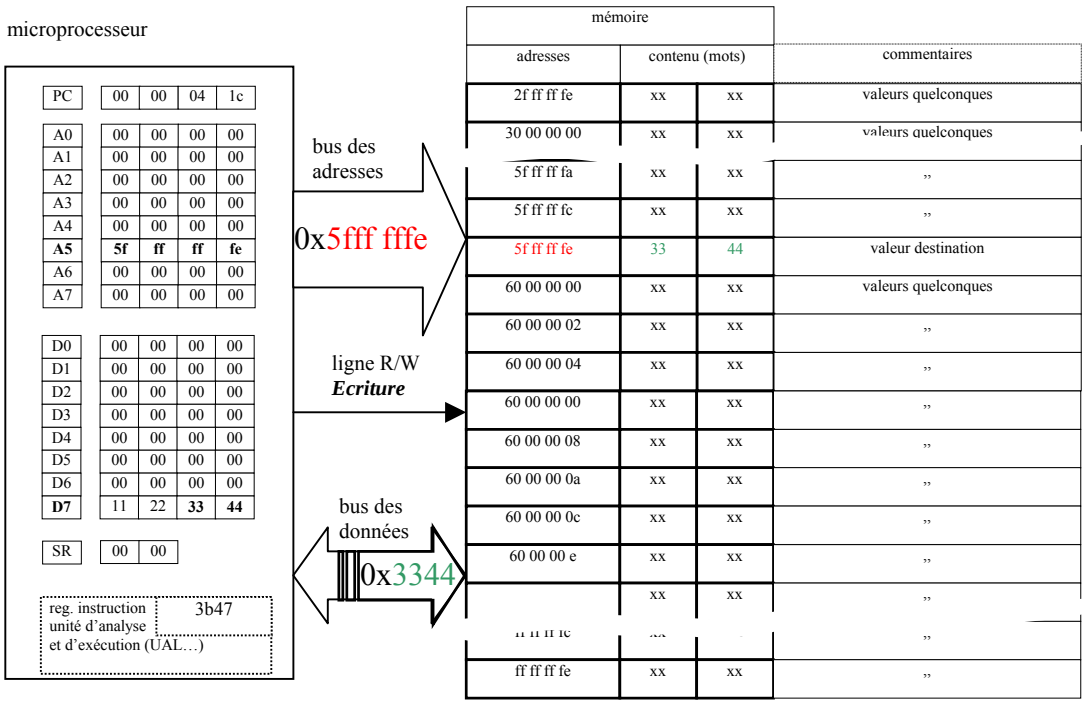
ETAPE 1 :
ACQUISITION DU CODE OPÉRATION
+ DECODAGE ➔ modifier a5 , a5 <== a5 – 2 (ici 0x6000000 –2 = 0x5fff fffe) puis copier le registre (long) d7 à l'adresse fournie par le contenu modifié du registre d'adresse A5.



ETAPE 2:
Activité interne : A5 <== A5 -2 ; A5 = 0x5fff fffe

Le microprocesseur *Cold* FIRE 5307

C. GUIRAUDIE



ETAPE 3 :
ECRITURE DE L'OPERANDE DESTINATION Ici le mot bas du registre d7.
d7.w == 0x1122**3344** ➔ 0x5fff fffe

?

-a- *Quel est le mnémonique de l'instruction qui réalise sur un octet (8 bits), en mode pré-décrémenté, un test (par rapport à zéro) sur le contenu de l'octet d'adresse 0x80000000 sachant que le registre d'adresse a5 contient la valeur 0x80000000 ?!*
Quelle est la valeur du registre a5 à l'issus de l'exécution de cette instruction ?

-b- *Combien de cycles (d'accès mémoire) sont nécessaires (accès sur bus de 16 fils) pour exécuter cette instruction ?*

⏩

-a- *QUESTION PIEGE !!!*
Si a5 a pour valeur initiale 0x80000000, alors l'usage suggéré de l'instruction tst.b d7,-(a5) conduira a tester non l'octet d'adresse 0x80000000 , mais l'octet d'adresse 0x7fff ffff. ceci est lié au mécanisme de pré-décrémentation du registre impliqué dans l'instruction. Il faut donc pour satisfaire le cahier des charges commencerpar une incrémentation du registre a5.

```
lea     1(a5),a5        ;ou bien addq.l     #1,a5
tst.b   -(a5)           ;ici c'est bien l'octet d'adresse 0x80000000+1 –1 donc 0x8000 0000 qui est testé.

;       a5 final = 0x8000 0000
```

-b- *1 cycle pour acquérir le code opération*
+ 1 cycles pour acquérir l'opérande octet d'adresse 0x80000000.

5.7.9 Mode d'adressage indexé via PC avec déplacement.

SYNTAXE → dep(pc)

Note : dep est une contraction du mot déplacement, pc signifie compteur de programme.

Exemple : **move.l d0,12(pc)**

voir chapitre suivant :

5.7.10 Mode d'adressage indexé via PC avec registre et déplacement.

SYNTAXE → dep(a**i,*xi***)

Note : dep est une contraction du mot déplacement, pc signifie compteur de programme, xi désigne un registre de donnée di ou d'adresse ai.

Exemple : **move.l d0,12(pc,d3)**

Les deux modes dep(pc) et dep(pc,xi) sont dans les détails des mécanismes mis en œuvre, tout à fait identiques respectivement aux modes indexé avec déplacement dep(ai) et indexé avec registre et déplacement dep(ai,xi).
La différence réside dans la mise en œuvre du registre compteur ordinal PC en lieu et place d'un registre d'adresse ai.
Ces modes d'adressages sont dédiés aux programmes dont on souhaite que l'implantation en mémoire soit indépendante de l'adresse absolue.
« Il faut que ça marche aussi bien (sans re compilation/assemblage) que le code soit implanté en 0x1000 ou en 0x10000000 ! »
On parle d'un code relogeable ou « relocatable ».
Exemple:

```
        .text
        ….
        lea     cible,a0
        move.l 4(a0),d0
        ….
cible:  .long   1,2,3,4…
```

Si ce programme est assemblé en définissant la zone de code (.text) à l'adresse 0x1000, alors si cible est en 0x1044 (par exemple) ; a0 vaudra lors de l'exécution 0x00001044 et d0 sera égal à 0x00000002, c'est à dire le long d'adresse 0x00001048.
Si ce code « qui fonctionne très bien », est un jour implanté (le code initial, non re-assembler) à partir de l'adresse 0x2000, alors a0 vaudra toujours 0x1044, alors que l'étiquette cible sera à l'adresse 0x2044. Donc lors de l'exécution d0 prendra la valeur du long d'adresse 0x00001048 alors qu'il aurait fallu « ramener » le long d'adresse 0x00002048.

Pour régler cette contrainte, deux solutions :

-Si le programme source est disponible et si l'on possède l'assembleur, il est possible en modifiant les paramètres de l'éditeur de lien de re-assembler le programme et de générer ainsi un nouveau code qui « tient » compte de sa nouvelle localisation. (Dans ce cas lors de l'exécution a0 = 0x00002044… tout rentre dans l'ordre !)

-Utiliser dès la conception du programme source l'adressage indexé via PC.
Le nouveau programme devient :

```
            .text
            ….
ici :       move.l          cible+2+4-ici(pc),d0
            ….
cible:      .long   1,2,3,4…
```

L'assembleur calcule l'expression cible+2+4-ici qui (si $< 2^{15} - 1$) devient en pratique le déplacement du mode indexé. Or cette valeur du déplacement n'est plus fonction de l'implantation absolue du code, seul l'écart entre les étiquettes ici et cible intervient. Le code ainsi généré devient relogeable.

Remarque : Le +2 de l'écriture ici+2-cible+4 intègre le fait que l'étiquette ici pointe sur le code opération de l'instruction move.l , alors qu'en fait, lors le l'exécution PC pointe sur le mot d'extension (le déplacement) qui se trouve 2 octets après. En fait les assembleurs acceptent une écriture simplifiée du type
move.l cible+4(pc). La syntaxe ..(pc) induit de la part de l'assembleur l'usage implicite de la réfence == ici et de l'offset adéquat (+2).
La valeur + 4 est elle, liée à l'accés au second lond de la suite 1,2,3…. l'accés au long 1 se ferait avec un offset de 0, l'accés au long 2 se fait 4 octets plus loin.