5.8 Jeu d'instructions.

Définition:
Le jeu d'instructions d'un microprocesseur est l'ensemble des instructions (des ordres) que ce microprocesseur est capable d'exécuter.

Remarque : Un jeu d'instructions se présente sous la forme d'un document « papier ».
Toutefois, pour que sa mise en oeuvre soit possible lors de l'exécution d'un programme qui fait appel à tout ou partie des instructions de cet ensemble, il faut que sur le plan matériel, c'est à dire dans le microprocesseur qui exécute le programme, ce jeu d'instructions soit « implanté ».
Cela signifie qu'il existe effectivement, « sur » le silicium qui constitue le microprocesseur des circuits logiques (fonctions décodage, Unité Arithmétique et Logique, modèle de programmation…), des bus et des multiplexeurs qui matérialisent ce jeu d'instructions.
Sur le papier « on peut laisser aller son imagination » et envisager des opérations très sophistiquées… encore faut-il que le microprocesseur associé intègre les fonctionnalités capables de réaliser ces instructions.

La famille de microprocesseurs Coldfire 5xxx est héritière du modèle de programmation et d'une grande partie du jeu d'instructions de la célèbre famille 680x0. Le premier élément de cette famille, le 68000, a été commercialisé à la fin des années 70 (1978) Il correspondait à ce que l'on nomme aujourd'hui une architecture CISC (Complex Instruction Set Computer), c'est à dire qu'il propose un jeu d'instructions « complexes ».
La famille Coldfire, proposée par Motorola à la fin des années 90 (1998) est postérieure (arrivée après) aux architectures RISC (Reducts Instruction Set Computer), et devrait de ce fait être doté des caractéristiques essentielles de ce type d'architecture (instruction de format fixe, une instruction par cycle d'horloge, beaucoup de registres internes, instructions à trois opérandes…).
C'est par exemple le cas de la famille Power PC du même constructeur.
Toutefois pour fournir une « solution de continuité » aux utilisateurs de la famille 680x0, Motorola a conçu une famille qui « puise » dans les deux types d'architectures. A la fois RICS et CICS.
Ainsi le MC5307 (version 3 de la famille Coldfire) ne dispose pas d'un jeu d'instructions strictement codé sur un format fixe comme l'exige une architecture RISC « pure et dure », mais il impose un format variable, mais qui ne peut excéder 3 mots !
(alors que la famille 680x0 proposait par exemple des instructions de type move.l @absolue_source,@absolue_destination qui exige 1+2+2 mots de codage).
Il n'intègre pas d'unité de calcul sur les nombres réel (la version 4 avec le MCF 5407, supporte la gestion des réels de type IEEE P754) , et un certain nombre d'instructions « sophistiquées » des 680x0 ont été supprimé. (par exemple les opérations portant sur les nombres décimaux telles que ABCD ou NBCD).

En pratique, il reprend prés de 90% des instructions du 68000, mais la contrainte « RISC » des instructions à format « limité » à 3 mots au maximum, interdit toutes les combinaisons des modes d'adressage qui violent cette contrainte.
Le tableau suivant montre pour les instructions à deux opérandes (source et destination), les combinaisons de mode d'adressage possibles.

adressage source	adressage destination
di, ai, (ai), (ai)+, -(ai)	tous les modes sont possibles. (sauf # , toujours impossible!)
dep(ai), dep(pc)	tous sauf : dep(ai,xi) , absolu.
dep(ai,xi) , dep(pc,xi) absolu , # (immédiat)	tous sauf : dep(ai,xi) , dep(ai) , absolu.

De plus beaucoup d'opérations logiques ou arithmétiques ne sont possibles que sur des opérandes longs.
Le tableau suivant liste les instructions les plus « courantes » et leur « degré » de finesse d'opérations, octet, mot ou long.

instructions	taille des opérandes accessibles
move , clr, tst	.b (octet, 8 bits), .w (mot, 16 bits), .l (long, 32 bits)
add , cmp , neg , sub , and , or , eor , not , asl/r , lsl/r , movem , lea , pea	.l (long) SEULEMENT !
moveq	source # octet ➔ etendu en destination .l (32 bits)
ext	source octet ➔ étendu en destination .w (16 bits) source mot ➔ étendu en destination .l (32 bits)
extb	source octet ➔ étendu en destination .l (32 bits)
mulu , muls , div	.w (mot, 16 bits), .l (long, 32 bits)
bchg , bclr , bset , btst	.b (octet, 8 bits) si destination mémoire .l (long, 32 bits) si destination registre

Notez que très peu d'instructions on des portées multiples (.b/.w/.l), cela conduit en langage assembleur, pour beaucoup de situation, à « compliquer » la programmation dés que l'objet manipulé n'est pas de taille 32 bits.

Par exemple vérifier qu'un octet d'adresse mémoire 0x30000 est égal ou non à la valeur 0x33.
Sans contrainte (cas du 68000) le programme est aussi simple que :

```
                cmp.b   #0x33,0x30000   ;compte 1+1+2 mots de codage ! > 3
                                        ;donc impossible avec
                                        ;les microprocesseurs Coldfire !
                beq.s   egal            ;saut à l'étiquette egal, si (0x30000) == 0x33
non_equal:      …                       ;codes executes si (0x30000) != 0x33
egal:           …
```

Ce même programme, pour un microprocesseur Colfire, devient :

```
                clr.l   d0              ;d0 == 0000 0000
                move.b  0x30000,d0      ;d0 == 0000 00??
                cmp.l   #0x33,d0        ;d0 == ? 0000 0033
                beq.s   egal            ;saut à l'étiquette egal, si (0x30000) == 0x33
non_equal:      …                       ;codes executes si (0x30000) != 0x33
egal:           …
```

Soient 2 instructions supplémentaires… c'est la « rançon » à payer pour accéder à l'architecture « pseudo » RICS.
Le gain, une puissance de traitement de l'ordre de 50 MIPS (Million d'Instructions Par Seconde) a comparer au 1 MIPS du 68000 !

5.8.1 Jeu d'instruction (présentation globale)

Nous proposons ci-après de présenter les instructions sous diverses formes

Conventions :

di ou dj ➔ registre de donnée de d0 à d7
ai ➔ registre d'adresse de a0 à a7
<@> ➔ mode d'adressage (applicable, voir détail de l'instruction)
➔ donnée constante (mode d'adressage immédiat)
.l ➔ long de 32 bits, .w ➔ mot de 16 bits , .b ➔ byte (octet) 8 bits.
étiquette ➔ étiquette, « label » de la forme *etiquette :*
liste ➔ énumération de reg. de la forme d0-d5/d7/a2/a4-a7 , qui désigne les registres : d0, d1, d2, d3, d4, d5 , d7, a2, a4, a5, a6, a7
rc ➔ registre de contrôle
sr ➔ registre d'état
ccr ➔ registre de condition (partie basse du registre d'état)
vbr ➔ registre de base des vecteurs d'exception (de la table d'exception)
acc ➔ accumulateur de l'unité MAC
mask ➔ registre de masquage de l'unité MAC
macsr ➔ registre d'état de l'unité MAC

Les lignes grisées correspondent aux instructions privilégiées, seulement exécutables si le microprocesseur est en mode superviseur. Dans le cas contraire, (mode user) déroutement vers l'exécution du programme attaché à l'exception viol de privilége vecteur : 8

5.8.1.1 présentation par ordre alphabétique.

5.8.1.1.1 général (sauf unité MAC)

mnemonique	syntaxe des opérandes	portée	opération réalisée	classe
ADD	di,<@> ou bien <@>,di	.l	addition sur reg. de donnée	arithmétique
ADD*A*	<@>,ai	.l	addition sur reg.d'adresse	arith. non signée
ADDQ	#,di	.l	addition rapide sur reg. de donnée	arithmétique
ADDX	di,dj	.l	addition sur reg. multiprécision (>32 bits)	arithmétique
AND	di,<@> ou bien <@>,di	.l	ET logique	logique
ASL	di,dj ou bien #,di	.l	décalage arithmétique à gauche	arithmétique
ASR	di,dj ou bien #,di	.l	décalage arithmétique à droite	arithmétique
BCHG	di,<@> ou bien #,<@>	.b .l	teste un bit puis l'inverse	logique
BCRL	di,<@> ou bien #,<@>	.b .l	teste un bit puis le met à zéro	logique
BRA	étiquette	.b .l	saut inconditionnel à l'étiquette	saut inconditionnel
BSET	di,<@> ou bien #,<@>	.b .l	teste un bit puis le met à un	logique
BSR	étiquette	.b .w	saut à un sous-programme	gestion de sous-prog.
BTST	di,<@> ou bien #,<@>	.b .l	teste un bit	logique
Bxx	étiquette	.b .w	teste la contition xx, saut à l'étiquette si vraie; suite si faux	saut conditionnel
CLR	<@>,di	.b .w .l	mise à zéro mém. ou reg. donnée	logique
CMP	<@>,di	.l	compare (soustrait) 2 opérandes	arithmétique
CMP*A*	<@> ou bien di	.l	compare reg. d'adresse	arithmétique
CPUSHL	(ai)		"vide" et invalide un cache	gestion système
DIVS	<@>,di	.w .l	division	arith. signée
DIVU	<@>,di	.w .l	division	arith. non signée
EOR	di,<@>	.l	OU exclusif	logique
EXT	di	.w .l	extension du bit de signe 8--> 16--> 32 bits	arithmétique
EXTB	di	.l	extension du bit de signe 8--> 32 bits	arithmétique
HALT			arrêt du micro-processeur	gestion système
JMP	<@>	.l	saut inconditionnel à l'étiquette	saut inconditionnel
JSR	<@>	.l	saut à un sous-programme	appel à sous-prog.
LEA	<@>,ai	.l	chargement d'une adresse ds reg. ai	transfert
LINK	ai,#	.l	réserve zone d'ocets sur la pile	transfert
LSL	di,dj ou bien #,di	.l	décalage logique à gauche	logique
LSR	di,dj ou bien #,di	.l	décalage logique à droite	logique
MOVE	<@>,<@>	.b .w .l	copie un opérande source dans une desti.	transfert
MOVE depuis ccr	di	.w	consultation du reg. de condition	transfert
MOVE depuis SR	sr,di	.w	consultation du registre d'état	gestion système
MOVE vers ccr	#,di	.w	initialisation du reg. de condition	transfert
MOVE vers SR	di,sr	.w	initialisation du registre d'état	gestion système
MOVEC	di,rc	.l	gestion des registres de contrôle	gestion système
MOVEM	liste,<@> ou bien <@>,liste	.l	dépose/extrait n reg. sur/depuis la pile	transfert
MOVEQ	#,di	.l	charge rapidement un reg di	transfert
MULS	<@>,di	.w .l	multiplication	arith. signée
MULU	<@>,di	.w .l	multiplication	arith. non signée
NEG	<@>	.l	complément à 2 d'un opérande	arithmétique
NEQX	<@>	.l	compl. à 2 multiprécision (>32 bits)	arithmétique
NOP			ne fait rien !	synchro/tempo.

NOT	<@>	.l	complément bit à bit	logique
OR	di,<@> ou bien <@>,di	.l	OU logique	logique
PEA	<@>	.l	chargement d'une adresse sur la pile	transfert
PULSE			fabrique un signal ipulsion via PST	gestion système
REMS	<@>,di,dj	.l	reste de la division signée di/<@> dans dj	arithmétique
REMU	<@>,di,dj	.l	reste de la division non signée di/<@> dans dj	arithmétique
RTE			retour d'exception	gestion d'exception
RTS			retour de sous programme	gestion de sous-prog.
STOP	#		arrêt du micro-processeur	gestion système
SUB	<@>,di	.l	soustraction sur reg. de donnée	arithmétique
SUB*A*	<@>,ai	.l	soustraction sur reg. d'adresses	arith. non signée
SUBQ	<@>,di	.l	soustrac. rapide sur reg. de donnée	arithmétique
SUBX	di,dj	.l	soust. sur reg. multiprécision (>32 bits)	arithmétique
SWAP	di	.w	échange mot haut et mot bas du reg. di	transfert
Sxx	di	.b	teste la contition xx, di.b==1 si vraie; ==0 si faux	test
TPF	#	.b .w .l	= = nop mais de durée constante	synchro/tempo.
TRAP	# [1 à 15]	.b .w .l	saut à un sous-prog. en mode superviseur	gestion d'exception
TRAPF	#		nop à longueur variable sur 1 2 ou 3 mots	gestion d'exception
TST	<@>	.b .w .l	compare à zéro	arithmétique
UNLK	<@>	.l	libére zone d'octets sur la pile	transfert
WDDATA	<@>	.b .w .l	gestion du module de mise au point intégré	gestion système
WDEBUG	<@>	64 bits	gestion du module de mise au point intégré	gestion système

5.8.1.1.2 unité MAC (Multiplication et ACcumulation)

mnemonique	**syntaxe des opérandes**	**portée**	**opération réalisée**	**classe**
MAC	di,dj , >> <<	.w .l	((di x dj) + acc) [<< || >>] ==> acc	arithmétique
MACL	di,dj , >> <<,<@>,di	.w .l	((di x dj) + acc) [<< || >>] ==> acc ; <@> ==> di	arithmétique
MOVE	acc , xi	.l	copie le reg. accumulateur dans un reg. Di ou ai	transfert
MOVE	macsr , xi	.l	copie le reg. d'état mac dans un reg. Di ou ai	transfert
MOVE	mask , xi	.l	copie le reg. de masque dans un reg. Di ou ai	transfert
MOVE	# || xi , acc	.l	copie un reg. Di ou ai ou une const. ds le reg. acc.	transfert
MOVE	macsr , ccr	.l	copie le reg. d'état mac ds le reg. d'état général sr	transfert
MOVE	# || xi , macsr	.l	copie un reg. Di ou ai ou une const. ds le reg. d'état mac	transfert
MOVE	# || xi , mask	.l	copie un reg. Di ou ai ou une const. ds le reg. de masque	transfert
MSAC	di,dj , >> <<	.w .l	((di x dj) - acc) [<< || >>] ==> acc	arithmétique
MSACL	di,dj , >> <<,<@>,di	.w .l	((di x dj) - acc) [<< || >>] ==> acc ; <@> ==> di	arithmétique

5.8.1.2 présentation par groupes fonctionnels.

5.8.1.2.1groupe des instructions de transfert

mnemonique	syntaxe des opérandes	portée	opération réalisée
MOVE	<@>,<@>	.b .w .l	copie un opérande source dans une desti.
MOVE depuis ccr	di	.w	consultation du reg. de condition
MOVE vers ccr	#,di	.w	initialisation du reg. de condition
MOVEM	liste,<@> ou bien <@>,liste	.l	dépose/extrait n reg. sur/depuis la pile
MOVEQ	#,di	.l	charge rapidement un reg di
LEA	<@>,ai	.l	chargement d'une adresse ds reg. ai
PEA	<@>	.l	chargement d'une adresse sur la pile
SWAP	di	.w	échange mot haut et mot bas du reg. di
LINK	ai,#	.l	réserve zone de #n ocets sur la pile et sauve ai
UNLK	<@>	.l	libére zone d'octets sur la pile et restitue ai
MOVE	acc , xi	.l	copie le reg. accumulateur dans un reg. Di ou ai
MOVE	macsr , xi	.l	copie le reg. d'état mac dans un reg. Di ou ai
MOVE	mask , xi	.l	copie le reg. de masque dans un reg. Di ou ai
MOVE	# || xi , acc	.l	copie un reg. Di ou ai ou une const. ds le reg. acc.
MOVE	macsr , ccr	.l	copie le reg. d'état mac ds le reg. d'état général sr
MOVE	# || xi , macsr	.l	copie un reg. Di ou ai ou une const. ds le reg. d'état mac
MOVE	# || xi , mask	.l	copie un reg. Di ou ai ou une const. ds le reg. de masque
MOVE depuis SR	sr,di	.w	consultation du registre d'état
MOVE vers SR	di,sr	.w	initialisation du registre d'état
MOVEC	ri,rc	.l	gestion des registres de contrôle

5.8.1.2.2groupe des instructions de traitement logique

mnemonique	syntaxe des opérandes	portée	opération réalisée
AND	di,<@> ou bien <@>,di	.l	ET logique
BCHG	di,<@> ou bien #,<@>	.b .l	teste un bit puis l'inverse
BCRL	di,<@> ou bien #,<@>	.b .l	teste un bit puis le met à zéro
BSET	di,<@> ou bien #,<@>	.b .l	teste un bit puis le met à un
BTST	di,<@> ou bien #,<@>	.b .l	teste un bit
CMP	<@>,di	.l	compare (soustrait) 2 opérandes
CMP*A*	<@> ou bien di	.l	compare reg. d'adresse
CLR	<@>,di	.b .w .l	mise à zéro mém. ou reg. donnée
EOR	di,<@>	.l	OU exclusif
NOT	<@>	.l	complément bit à bit
OR	di,<@> ou bien <@>,di	.l	OU logique
TST	<@>	.b .w .l	compare à zéro

5.8.1.2.3groupe des instructions de décalage

mnemonique	syntaxe des opérandes	portée	opération réalisée
ASL	di,dj ou bien #,di	.l	décalage arithmétique à gauche
ASR	di,dj ou bien #,di	.l	décalage arithmétique à droite
LSL	di,dj ou bien #,di	.l	décalage logique à gauche
LSR	di,dj ou bien #,di	.l	décalage logique à droite

5.8.1.2.4groupe des instructions de traitement arithmétique

mnemonique	syntaxe des opérandes	portée	opération réalisée
ADD*A*	<@>,ai	.l	addition sur reg.d'adresse
ADD	di,<@> ou bien <@>,di	.l	addition sur reg. de donnée
ADDQ	#,di	.l	addition rapide sur reg. de donnée
ADDX	di,dj	.l	addition sur reg. multiprécision (>32 bits)
CLR	<@>,di	.b .w .l	mise à zéro mém. ou reg. donnée
CMP	<@>,di	.l	compare (soustrait) 2 opérandes
CMP*A*	<@> ou bien di	.l	compare reg. d'adresse
DIVS	<@>,di	.w .l	division
DIVU	<@>,di	.w	division
EXT	di	.w .l	extension du bit de signe 8--> 16--> 32 bits
EXTB	di	.l	extension du bit de signe 8--> 32 bits
MULS	<@>,di	.w .l	multiplication
MULU	<@>,di	.w .l	multiplication
NEG	<@>	.l	complément à 2 d'un opérande
NEQX	<@>	.l	compl. à 2 multiprécision (>32 bits)
REMS	<@>,di,dj	.l	reste de la division signée di/<@> dans dj
REMU	<@>,di,dj	.l	reste de la division non signée di/<@> dans dj
SUB	<@>,di	.l	soustraction sur reg. de donnée
SUB*A*	<@>,ai	.l	soustraction sur reg. d'adresses
SUBQ	<@>,di	.l	soustrac. rapide sur reg. de donnée
SUBX	di,dj	.l	soust. sur reg. multiprécision (>32 bits)
TST	<@>	.b .w .l	compare à zéro
MAC	di,dj , >> <<	.w .l	((di x dj) + acc) [<< || >>] ==> acc
MACL	di,dj , >> <<,<@>,di	.w .l	((di x dj) + acc) [<< || >>] ==> acc ; <@> ==> di
MSAC	di,dj , >> <<	.w .l	((di x dj) - acc) [<< || >>] ==> acc
MSACL	di,dj , >> <<,<@>,di	.w .l	((di x dj) - acc) [<< || >>] ==> acc ; <@> ==> di

5.8.1.2.5groupe des instructions de sauts conditionnels et inconditionnels

mnemonique	syntaxe des opérandes	portée	opération réalisée
Bxx	étiquette	.b .w	teste la contition xx, saut à l'étiquette si vraie; suite si faux
BRA	étiquette	.b .l	saut inconditionnel à l'étiquette
JMP	<@>	.l	saut inconditionnel à l'étiquette

5.8.1.2.6groupe des instructions de gestion des sous-programmes et des exceptions

mnemonique	syntaxe des opérandes	portée	opération réalisée
JSR	<@>		appel à un sous-programme
BSR	étiquette	.b .w	appel à un sous-programme
RTS			retour de sous programme
RTE			retour d'exception
TRAP	# [1 à 15]	.b .w .l	saut à un sous-prog. en mode superviseur

5.8.1.2.7groupe des instructions diverses

mnemonique	syntaxe des opérandes	portée	opération réalisée
CPUSHL	(ai)		"vide" et invalide un cache
HALT			arrêt du micro-processeur
NOP			ne fait rien !
PULSE			fabrique un signal ipulsion via PST
STOP	#		arrêt du micro-processeur
Sxx	di	.b	teste la contition xx, di.b==1 si vraie; ==0 si faux
TPF	#	.b .w .l	= = nop mais de durée constante
TRAPF	#		nop à longueur variable sur 1 2 ou 3 mots
WDDATA	<@>	.b .w .l	gestion du module de mise au point intégré
WDEBUG	<@>	64 bits	gestion du module de mise au point intégré

5.8.2 Jeu d'instruction (présentation détaillée) Instructions non privilégiées.

format général d'une page d'instruction détaillée (exemple de l'instruction MOVE)

MOVE.b/w/l ← mnémonique de l'instruction et présence d'une option de taille

copie de l'opérande source dans l'opérande destination ← description de l'opération effectuée

mode d'adressage source possibles	mode d'adressage destination possibles
di, ai, (ai), -(ai), (ai)+	tous les modes sont possibles (sauf # !)
dep(ai), dep(pc)	tous les modes sont possibles saufs : *absolu, dep(ai,xi)* et # !
dep(ai,xi), dep(pc,xi), absolu, #	tous les modes sont possibles saufs : *dep(ai,xi), dep(ai), absolu* et # !

↖combinaisons modes d'adressages possibles

syntaxe générale: **move.taille** <@>,<@> avec taille == b (8 bits) , w (16 bits , l (32 bits)

↖syntaxe assembleur

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code		*taille*		*mode adressage destination*						*mode d'adressage source*					
0	0	-	-	num. reg. ou code			code mode			code mode			num. reg. ou code		

voir tableau des tailles et des modes/codes page : 61 ↖code machine de l'instruction

influence sur bits d'états : ←implication sur les bits d'état

X	N	Z	V	C
-	↑↓	↑↓	0	0

\- non modifié 0 forcé à 0

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)

exemple de mise en oeuvre: …… ← un exemple d'utilisation

TABLEAU DES TAILLES | *long :* **10** | *mot :* **11** | *octet :* **01**

TABLEAU DES MODES/CODES

mode d'adressage	code mode	r e g i s t r e
di	000	de 000 à 111
ai	001	’’
(ai)	010	’’
(ai)+	011	’’
-(ai)	100	’’
dep(ai)	101	’’
dep(ai,xi)	110	’’

mode d'adressage	code mode	code
absolu court	*111*	*000*
absolu	111	001
immédiat #	111	100
dep(pc)	111	010
dep(pc,xi)	111	011

table code mode adressage

5.8.2.1 ADD

ADD.l

Addition arithmétique de l'opérande source avec l'opérande destination.
Le résultat est déposé dans l'opérande destination. Tous les bits d'état sont influencés par l'opération.

modes d'adressage source possibles	modes d'adressage destination possibles
tous les modes sont possibles	*di*
di	tous les modes sont possibles saufs : dep(PC), dep(PC,Xi) et # !

syntaxe générale: **add.l <@>,di**
ou bien **add.l di,<@>**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. di			<@>,di ou di,<@>			*mode d'adressage*					
1	1	0	1	-	-	-	code forme			code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 :
code forme : **010** pour forme <@>,*di* ; **110** pour forme *di*,<@>
Remarques : 1°l'instruction add.l di,dj doit être codée avec la forme 010
2° l'instruction add.l #,*di* (alors notée addi.l #,*di*) peut se coder également : 0x068*r*, *r* étant le numéro du reg. *di* .
Dans ce cas le code opération est suivi sur 2 mots de la valeur de la constante

influence sur les bits d'états :

X	N	Z	V	C
↑↓	↑↓	↑↓	↑↓	↑↓

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
report (retenue) issu du bit 15 du résultat (C et X)
dépassement (over-flow) mis à 1 si signe du résultat abérant

exemple de mise en oeuvre:
additionner les 30 octets rangés à partir de l'adresse 0x600.
La somme sous forme d'un mot sera rangée à l'adresse 0x300000000

```
        …
        lea         0x600,a0        ;pointeur sur zone à gérée
        move.l      #30,d0          ;compteur de boucle
        clr.l       d1              ;reg. support des sommes
        clr.l       d2              ;reg. support des sommes
encore :
        move.b  (a0)+,d1            ;acqui. d'un octet et incrémentation du pointeur
        add.l       d1,d2           ;addition de la somme et nouvel octet (sur long)
        subq.l      #1,d0           ;compteur de boucle –1 → compteur de boucle
        bne.s       encore          ;si < 30 passages alors saut à l'étiquette encore
        move.w d2,0x30000000        ;rangement du résultat
```

5.8.2.2 **ADDA**

ADDA.l

Addition arithmétique de l'opérande source avec un registre d'adresse.
Le résultat est déposé dans un registre d'adresse.
Les opérandes sont traités en tant que nombres non signés. (Il n'existe pas d'adresse négative)
Aucun bits d'état n'est influencé par l'opération.

modes d'adressage source possibles	modes d'adressage destination possibles
tous les modes sont possibles	seuls *ai* possibles

syntaxe générale: **adda.l <@>,ai**
ou bien plus simplement **add.l <@>,ai**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. ai			*code*			*mode d'adressage*					
1	1	0	1				1	1	1	code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est influencé par l'exécution de cette instruction

exemple de mise en oeuvre:
additionner la constante 0x200 au registre a0 et -1 au registre a1.

```
…
add.l     #0x200,a0      ;a0 + 0x00000200 → a0
add.l     #-1,a1         ;a1 + 0xffffffff → a1
…
```

5.8.2.3 **ADDQ**

ADDQ.l

Addition arithmétique « rapide » d'une constante ∈ [1,8] à un opérande destination.
Le résultat est déposé dans un registre ou une adresse de destination.
Tous les bits d'état sont influencés par l'opération.

mode d'adressage source possible	modes d'adressage destination possibles
immédiat # (valeur maxi : 8 ; 0 non admis)	tous les modes sont possibles saufs : *dep(pc) , dep(pc,xi)* et # !

syntaxe générale: **addq.l #,<@>**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				*donnée immédiate*			*code*			*mode d'adressage*					
0	1	0	1	-	-	-	0	1	0	code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 :

influence sur bits d'états :

X	N	Z	V	C
↑↓	↑↓	↑↓	↑↓	↑↓

↑↓ mis à 1 ou à 0 suivant le signe de N ou la valeur nulle ou non pour Z report (retenue) issu du bit 15du résultat (C et X) dépassement (over flow V) mis à 1 si le signe du résultat est aberant

exemple de mise en oeuvre:
additionner la constante 0x8 aux l'adresses 0x100000 , 0x20000000 , 0x20000008 , 0x2000000c et 0x1ffffffc.

```
…
lea         0x20000000,a0   ;pointeur a0 sur la zone 0x20000000
aqqq.l      #8,0x100000             ;la cst. 8 est additionnée au contenu de l'@ 0x100000
addq.l      #8,(a0)         ;la cst. 8 est additionnée au contenu de l'@ 0x20000000
addq.l      #8,8(a0)                ;la cst. 8 est additionnée au contenu de l'@ 0x20000008
addq.l      #8,0xc(a0)              ;la cst. 8 est additionnée au contenu de l'@ 0x2000000c
addq.l      #8,-4(a0)               ;la cst. 8 est additionnée au contenu de l'@ 0x1ffffffc
…
```

5.8.2.4 **ADDX**

ADDX.l

Addition arithmétique du contenu d'un registre source au contenu d'un registre destination en intégrant à l'adition (lsb) la valeur du bit de retenu (X) telle qu'elle est avant l'exécution de l'inst.
Le résultat est déposé dans un registre de destination.
Cette instruction permet de réaliser des additions par « morceaux » sur des nombres de taille >32 bits.
Tous les bits d'état sont influencés par l'opération.

mode d'adressage source possible	mode d'adressage destination possible
di	*di*

syntaxe générale: **addx.l di,di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				*mode d'adr. dest.*			*code*						*mode d'adr. source*		
1	1	0	1	num. registre			1	1	0	0	0	0	num. registre		

influence sur bits d'états :

X	N	Z	V	C
↑↓	↑↓	↑↓	↑↓	↑↓

↑↓ mis à 1 ou à 0 suivant le signe (N) ou la valeur nulle ou non pour Z report (retenue) issu du bit 15du résultat (C et X) dépassement (over flow V) mis à 1 si le signe du résultat est aberant

exemple de mise en oeuvre:
additionner (2 nbr. de 64 bits) les 2 long d'adresse 0x10000 aux 2 longs d'adresse 0x20000.

```
…
lea         0x10000,a0      ;pointeur a0 sur la zone 0x10000+4+4
lea         0x20000,a1      ;pointeur a1 sur la zone 0x20000+4+4
move.l      -(a0),d0        ;d0 chargé avec partie basse (32 bits) du premier nombre
move.l      -(a1),d1        ;d1 chargé avec partie basse (32 bits) du premier nombre
move.w #0,ccr               ;mise à 0 du bit X du regsitre de condition (ccr de SR)
addx.l      d0,d1           ;addition des 32 bits de poids faibles, bit X actualisé.
move.l      d1,0x20004      ;stockage des 32 bits de poids faibles.
move.l      -(a0),d0        ;d0 chargé avec partie haute (32 bits) du premier nombre
move.l      -(a1),d1        ;d1 chargé avec partie haute (32 bits) du premier nombre
addx.l      d0,d1           ;addition des 32 bits de poids haut, bit X actualisé.
move.l      d1,0x20000      ;stockage des 32 bits de poids haut.
…
```

5.8.2.5 **AND**

AND.l

ET logique (bit à bit) entre l'opérande source et l'opérande destination.
Le résultat est déposé dans l'opérande destination. Les bits d'état N et Z sont influencés par l'opération

	modes d'adressage source possibles	modes d'adressage destination possibles
forme <@> , di	tous les modes sont possibles	seuls *di* possibles
forme di , <@>	*di*	tous les modes sont possibles saufs : *dep(ai)* , *dep(ai,xi)* et # !

syntaxe générale: **and.l <@>,di**
ou bien and.l di,<@>

format du code opération : (Attention : pour la forme and.l #,di voir second format)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				Choix de <@>,di ou di, <@>						mode d'adressage					
1	1	0	0	num. reg. di			code forme			code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 :
code forme : 010 pour forme <@>,*di* ;110 pour forme *di*,<@>

format and.l #,di .Ce format occupe 3 mots, le code ci-dessous + 2 mots « constante »

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													num. reg. di		
0	0	0	0	0	0	1	0	1	0	0	0	0	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

↑↓ mis à 1 ou à 0 suivant le signe (N) ou la valeur nulle ou non pour Z
- Non modifié
0 les bits V et C sont forcés à 0

exemple de mise en oeuvre:
Faire un masque en ET de valeur 0x33 sur l'octet d'adresse 0x2000 ? Si le résultat du ET est non nul, effectuer un saut à l'adresse Non_nul, sinon poursuivre.

```
        …
        clr.l       d0          ;reg. préparé pour acceuil de l'octet et opération sur long
        move .b 0x2000,d0       ;acquisition de l'octet à tester, d0 == 0x000000??
        and.l       #0x33,d0    ;ET avec la constante 0x00000033 --> 0x000000??
        bne.s       Non_nul     ;test si l'octet valait  xxxx xx00 ?  (x == 0 ou 1)
                                ;saut vers étiq. Non_nul si l'oct. ne valait pas xxxx xx00
        …..                     ;codes exécutés si l'octet valait xxxx xx00
Non_nul :
                                ,codes exécutés si l'octet ne valait pas xxxx xx00
```

5.8.2.6 **ASL**

ASL.l

Décalage arithmétique de n bits vers la gauche. Cette instruction est tigoureusement identique à l'instruction LSL.l
Le nombre de décalage est limité de 1 à 8 (zéro impossible) s'il est proposé en tant que constante (mode # immédiat).
Le nombre de décalage est limité de 0 à 64 s'il est proposé en tant que contenu de registre.
Cette opération est équivalente à une multiplication par 2^n, *n* étant le nombre de décalages demandés.

mode d'adressage source possible	modes d'adressage destination possibles
immédiat # (valeur maxi : 8 ; 0 non admis) ou bien *di*	seulement *di*

syntaxe générale: **asl.l #,di**
ou bien **asl.l di,di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				*donnée # ,ou num. reg*			*d r*	*c o d e*		*i / r*	*c o d e*		*num. reg. destination*		
1	1	1	0	-	-	-	1	1	0	-	0	0	-	-	-

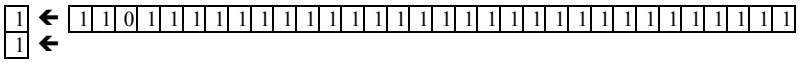
dr = 1 pour décalage à gauche.
i/r = 0 ➔ nbr. décalage mode immédiat, = 1 ➔ nbr. décalage mode registre di

influence sur bits d'états :

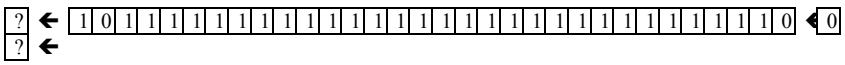
X	N	Z	V	C
↑↓	↑↓	↑↓	0	↑↓

↑↓ mis à 1 ou à 0 suivant le signe (N) ou la valeur nulle ou non du dernier bit décalé (Z) report (retenue) issu du bit 15 du résultat (C et X)
V bit de dépassement (over-flow) est forcé à 0

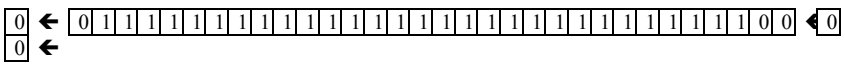
exemple de mise en oeuvre:
décaler de 2 bits à gauche le registre d7 en mode immédiat. **asl.l #2,d7**
d7 initial (0xdfffffff) et bits d'état C et X



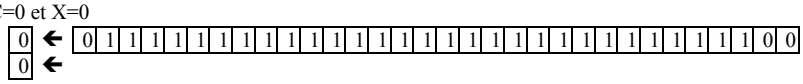
étape 1 et bits d'état C et X



étape 2 et bits d'état C et X



d7 final (0x7ffffffc) et bits d'état C=0 et X=0



5.8.2.7 **ASR**

ASR.l

Décalage arithmétique de n bits vers la droite.
Le nombre de décalage est limité de 1 à 8 (zéro impossible) s'il est proposé en tant que constante (mode # immédiat).
Le nombre de décalage est limité de 0 à 64 s'il est proposé en tant que contenu de registre.
Cette opération est équi. à une division SIGNEE par 2^n , *n* étant le nombre de décalages demandés.

mode d'adressage source possible	modes d'adressage destination possibles
immédiat # (valeur maxi : 8 ; 0 non admis) ou bien *di*	seulement *di*

syntaxe générale: **asr.l #,di**
ou bien **asr.l di,di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				*donnée# ou num. reg*			*d r*	*c o d e*		*i / r*	*c o d e*		*num. reg. destination*		
1	1	1	0	-	-	-	0	1	0	-	0	0	-	-	-

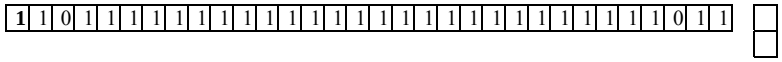
dr = 0 pour décalage à droite.
i/r = 0 ➔ nbr. décalage mode immédiat, = 1 ➔ nbr. décalage mode regsitre di

influence sur bits d'états :

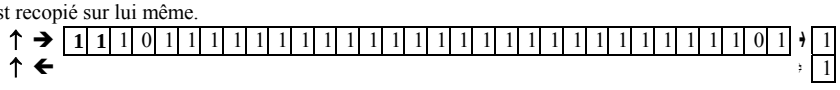
X	N	Z	V	C
↑↓	↑↓	↑↓	0	↑↓

↑↓ mis à 1 ou à 0 suivant le signe (N) ou la valeur nulle ou non du dernier bit décalé (Z) report (retenue) issu du bit 15 du résultat (C et X)
V bit de dépassement (over-flow) est forcé à 0

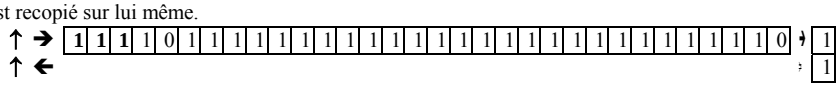
exemple de mise en oeuvre:
décaler de 2 bits à droite le registre d7 en mode immédiat. asr.l #2,d7
d7 initial (0xdffffffb) et bits d'état C et X



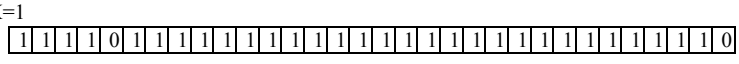
étape 1 et bits d'état C et X. Le bit MSB est recopié sur lui même.



étape 2 et bits d'état C et X. Le bit MSB est recopié sur lui même.



d7 final (0xf7fffffe) et bits d'état C=1 et X=1



5.8.2.8 BCHG et BCLR et BSET et BTST

BCHG.b/l BCLR.b/l BSET.b/l BTST.b/l

Ces 4 instructions testent, avant toute chose, le bit qu'elles « manipulent ». C'est le bit d'état Z qui est le support de ce test.

BCHG inversion de l'état d'un bit dans un octet en mémoire ou dans un registre de donnée.
BCLR mise à 0 d'un bit dans un octet en mémoire ou dans un registre de donnée.
BSET mise à 1 d'un bit dans un octet en mémoire ou dans un registre de donnée.
BTST teste de l'état d'un bit dans un octet en mémoire ou dans un registre de donnée.

modes d'adressage source possibles	modes d'adressage destination possibles
di	tous les modes sont possibles saufs : *ai, dep(pc) , dep(pc,xi)* et # !
immédiat #	tous les modes sont possibles saufs : *ai, dep(ai,xi), absolu, dep(pc), dep(pc,xi)*

syntaxes: **bchg.b/l di,<@>** **bclr.b/l di,<@>** **bset.b/l di,<@>** **btst.b/l di,<@>**
ou bien **bchg.b/l #,<@>** **bclr.b/l #,<@>** **bset.b/l #,<@>** **btst.b/l #,<@>**
si l'opérande de destination est un reg. de donnée di, alors la portée de l'opération est de 32 bits (.l)
si l'opérande de destination est en mémoire, alors la portée de l'opération est de 8 bits (.b)

format du code opération pour les cas **di,<@>**: voir tableau modes d'adressage page 61

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. di			*code**			*mode d'adressage*					
0	0	0	0	-	-	-				code mode			num. reg. ou code		

* code : 101 ➔ BCHG ; 110 ➔ BCLR ; 111 ➔ BSET ; 100 ➔ BTST.

format du code opération pour les cas **#,<@>**: la constante # est dans un mot d'extention

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code							*code**			*mode d'adressage*					
0	0	0	0	1	0	0				code mode			num. reg. ou code		
0	0	0	0	0	0	0	constante # (exploitée en modulo 8 si dest. mémoire ou 64 dest. *di*)								

* code : 001 ➔ BCHG ; 010 ➔ BCLR ; 011 ➔ BSET ; 000 ➔ BTST.

influence sur bits d'états :

X	N	Z	V	C
-	-	↑↓	-	-

↑↓ (Z) mis à 1 ou à 0 suivant la valeur du bit testé avant (éventuel) changement
- X, N, V, C non modifiés

exemple de mise en oeuvre:
Mettre à 0, le bits de poids faible (bit 0) de l'octet d'adresse 0x12000.

```
        lea     0x12000,a2
        bclr.b  #0,(a2)
        …
```

Tester le bit de poids fort (bit 31) du regsitre d7. Si ce bit vaut zéro « sauter » à l'étiquette ICI :

```
        btst.l  #31,d7
        beq.s   ICI
        …
```

5.8.2.9 BRA

BRA

Saut inconditionnel relatif à PC.
L'instruction est associée au déplacement qui sépare l'instruction BRA de l'étiquette que cette dernière spécifie. Ce déplacement ajouté à la valeur courante du compteur ordinal PC, modifie ce dernier et provoque le saut désiré.
Si l'amplitude d'un saut négatif (vers des adresse plus petites) est inférieure à –128 octets ou celle d'un saut positif (vers des adresse plus grandes) est inférieure à +127 il est possible d'utiliser le mnémonique *bra.s* (s comme short, court) qui conduit à un codage de l'instruction sur une mot (16 bits).
Pour des amplitudes plus grandes [-32768 , +32767] il ne faut pas ajouter l'argument *.s*
Des sauts plus importants ne sont pas possibles avec cette instruction. Voir l'instruction jump.

mode d'adressage

dep(pc) seulement

syntaxe générale: **bra** **etiquette** (étiquette « lointaine »)
ou bien **bra.s** **etiquette** (étiquette « proche »)

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code								*dép. (si .s) sinon 00000000 et dép. dans le mot d'extension*							
0	1	1	0	0	0	0	0	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

- Les bits d'état ne sont pas influencés par l'instruction BRA

exemple de mise en oeuvre: réaliser une boucle de programme sans fin .
1er cas d'une boucle de moins de 128 octets

```
                …
Boucle : …
                …                       ;corps de la boucle
                bra .s    Boucle        ;retour au point "Boucle"
                …
```

2ième cas boucle de plus de 128 octets et de moins de 32768 octets.

```
                …
Boucle : …
                …                       ;corps de la boucle
                bra       Boucle        ;retour au point "Boucle"
```

5.8.2.10 **Bxx**

Bxx ==> xx est le mnémonique d'une condition de saut : de CC à VS

Saut conditionnel relatif à PC. (Pour les particularités des amplitudes des sauts et du mode d'adressage relatif à PC voir instruction BRA)

Ces instructions permettent de réaliser des sauts sous certaines conditions. Ces conditions portent sur le résultat arithmétique ou logique de la dernière opération (active en terme de bits d'états) précédent l'instruction Bxx. Si la condition testée est vraie, alors le compteur ordinal PC est modifié (valeur de l'étiquette), il y a rupture de séquence.

Si la condtion testée est fausse, PC n'est pas modifié et le programme se poursuit par l'exécution de l'instruction qui suit l'instruction Bxx. Ce sont des combinaisons binaires des bits d'états qui sont testées.

Le tableau ci-dessous met en relation des conditions « testables » et les mnémoniques Bxx associés.

mnémo.	code	ce qui est testé…	équation testée
BEQ	0111	dernier résultat nul	Z=1
BNE	0110	der. résultat non nul	Z=0
BGE	1100	der. résultat ≥ 0 en comp. à 2	N⊕V=0
BGT	1110	der. résultat > 0 en comp. à 2	N⊕V ou Z=0
BLT	1101	der. résultat < 0 en comp. à 2	N⊕V=1
BLE	1111	der. résultat ≤ 0 en comp. à 2	N⊕V ou Z=1
BVC	1000	der. résultat avec signe correct	V=0

mnémo.	code	ce qui est testé…	équation testée
BMI	1011	nombre négatif	N=1
BPL	1010	nombre positif	N=0
BCC	0100	der. résultat ≥ 0 en bin. naturel	C=0
BHI	0010	der. résultat > 0 en bin. naturel	C ou Z=0
BCS	0101	der. résultat < 0 en bin. naturel	C=1
BLS	0011	der. résultat ≤ 0 en bin. naturel	C ou Z=1
BVC	1001	der. résultat avec sig.incorrect	V=1

syntaxe générale: **bxx etiquette** (étiquette « lointaine »)
ou bien **bxx.s etiquette** (étiquette « proche »)

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				*code condition*				*déplacement (si .s) sinon 00000000 et dép. dans le mot d'extension*							
0	1	1	0	-	-	-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

- Les bits d'état ne sont pas influencés par l'instruction BRA

exemple de mise en oeuvre: réaliser une boucle de programme 50 fois (boucle de moins de 128 octets).
1er cas d'une boucle de moins de 128 octets

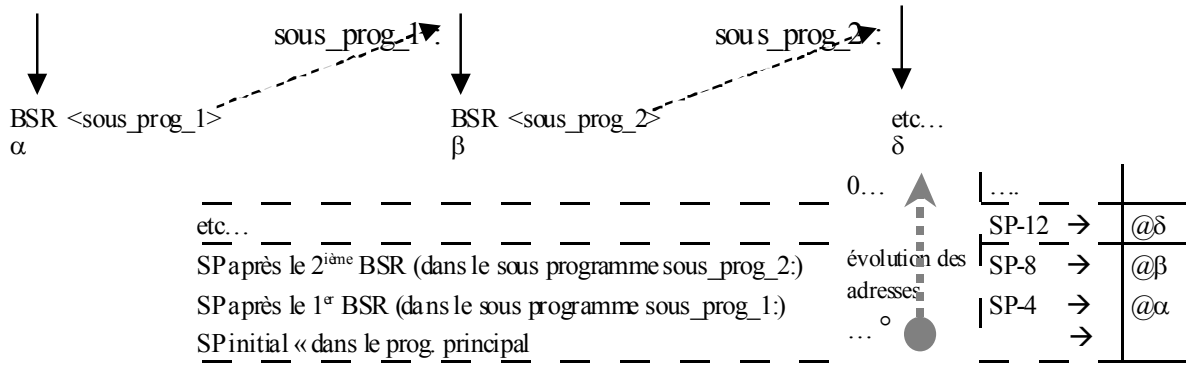
```
                …
                move.l      #50,d0      ;d0 sert de compteur de boucles
Boucle : …
                …                       ;corps de la boucle
                subq.l      #1,d0       ;compteur --
                bne .s      Boucle      ;si moins de 50 passages aller au point "Boucle"
                …                       ;suite du prog. après 50 « bouclages »
```

5.8.2.11 **BSR**

BSR

Appel à un sous programme en mode relatif à PC. (Pour autres modes voir instruction JSR)
Pour les particularités des amplitudes des sauts et du mode d'adressage relatif à PC voir instruction BRA.
Cette instruction assure la sauvegarde de la valeur du compteur ordinal (adresse de l'instruction qui suit BSR) « sur » la pile . PC → (SP-4)
Le pointeur de pile SP est décrémenté de 4. SP = SP – 4.
Cela permet, au sein du sous programme appelé, d'appeler un nouveau sous programme avec le même mécanisme de sauvegarde de PC sans « écrasser » le dernier PC sauvegardé. Cela permet d'effectuer des appels à sous programmes « imbriqués » (voir illustration ci dessous)

Enfin PC est chargé avec la valeur de l'adresse de l'étiquette (point d'entrée du sous programme)
PC = PC + déplacement = point entrée sous programme



syntaxe générale: **bsr** **etiquette** (étiquette « lointaine »)
ou bien **bsr.s** **etiquette** (étiquette « proche »)

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code								*déplacement (si .s) sinon 00000000 et dép. dans le mot d'extension*							
0	1	1	0	0	0	0	1	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit du registre d'état n'est influencé par cette instruction

exemple de mise en oeuvre: Appel du sous programme calcul.
SP initial 0x20000 ; @ de l'instruction BSR 0x1000 ; @ de début du sous prog. calcul 0x4000

```
                …
@0x1000         bsr     calcul      ;appel du sous programme (codage sur 2 mots)
@0x1004         …
@0x4000 calcul : …                  ;début du sous programme calcul. Ici  SP= 0x1fffc = 0x2000
- 4
                …                   ;à partir de l'adresse pile 0x1fffc on trouve : « PC » 0x1004
```

5.8.2.12 **CLR**

CLR.b/w/l

Mise à zéro d'un octet, d'un mot ou d'un long situé en mémoire ou dans un registre.
Les bits d'état, sauf X sont influencés par l'opération.

modes d'adressage possibles

tous les modes sont possibles saufs : *ai* et # !

syntaxe générale: **clr.taille** <@>

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code								*taille*		*mode d'adressage*					
0	1	0	0	0	0	1	0	-	-	code mode			num. reg. ou code		

voir tableau modes d'adressage et taille page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	0	1	0	0

Le bit Z passe à 1. (marque de la mise à zéro faite par l'exécution de l'instruction CLR)
Les bits N,V et C sont mis à 0 (zéro est un nombre « positif ») , X n'est pas modifié

exemple de mise en oeuvre:
mettre à zéro les 1000 longs (4096 octets) à partir de l'adresse 0x600.

```
        …
        lea     0x600,a0        ;pointeur sur zone à gérée
        move.l  #1000,d0        ;compteur de boucle
encore :
        clr.l   (a0)+           ; »clear » d'un long, et pointeur ++
        subq.l  #1,d0           ;compteur de boucle –1 → compteur de boucle
        bne.s   encore          ;si < 1000 passages alors saut à l'étiquette encore
        …
```

même exemple pour 1000 octets.

```
        …
        lea     0x600,a0        ;pointeur sur zone à gérer
        move.l  #1000,d0        ;compteur de boucle
encore :
        clr.b   (a0)+           ; »clear » d'un octet, et pointeur ++
        subq.l  #1,d0           ;compteur de boucle –1 → compteur de boucle
        bne.s   encore          ;si < 1000 passages alors saut à l'étiquette encore
        …
```

5.8.2.13 **CMP**

CMP.l

Opération de comparaison entre le contenu d'un registre de donnée et l'opérande source.
Seuls les bits d'états reflètent le résultat de cette comparaison.
Généralement un saut conditionnel exploite les valeurs de ces bits d'état « derrière » de compare.
Plus précisément, l'opération 'compare' réalise la soustraction entre l'opérande de destination *di* et l'opérande source. [destination – source].
Le résultat arithmétique n'est pas fournit., mais les bits d'état sont influencés par l'opération.

modes d'adressage source possibles	mode d'adressage destination possible
tous les modes sont possibles	seuls *di* possibles

syntaxe générale: **cmp.l <@>,di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. di			*code*			*mode d'adressage*					
1	0	1	1	-	-	-	0	1	0	code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 :

L'opération **destination – source** entraîne :

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	↑↓	↑↓

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
report (retenue) issu du bit 15 du résultat (C)
dépassement (over-flow) mis à 1 si signe du résultat aberrant
- Le bit X n'est pas modifié

exemple de mise en oeuvre:

Vérifier si l'octet d'adresse 0x20000000 est égal à la valeur 0x33. Si oui, « aller » à l'étiquette OUI …

```
        clr.l     d0                ;nettoyage de registre d'accueil de l'octet
        move.b    0x20000000,d0     ;acquisition de l'octet à tester
        cmp.l     #0x00000033,d0    ;comparaison sur un « long  préparé » par le clr de d0
        beq.s     OUI               ;saut si l'octet acquis valait 0x33
NON :   …                           ;suite du prog. si octet diff. de 0x33
        …
OUI :   …                           ;suite du programme si octet == 0x33
        …
```

5.8.2.14 CMPA

CMPA.l

Opération de comparaison entre le contenu d'un registre d'adresse et l'opérande source.
Seuls les bits d'états reflètent le résultat de cette comparaison.
Généralement un saut conditionnel exploite les valeurs de ces bits d'état « derrière » de compare.
Plus précisément, l'opération 'compare' réalise la soustraction entre l'opérande de destination *ai* et l'opérande source. [destination – source].
Le résultat arithmétique n'est pas fournit., mais les bits d'état sont influencés par l'opération.

modes d'adressage source possibles	mode d'adressage destination possible
tous les modes sont possibles	seuls *ai* possibles

syntaxe générale: **cmpa.l <@>,ai** ou bien **cmp.l <@>,ai**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. ai			*code*			*mode d'adressage*					
1	0	1	1	-	-	-	1	1	1	code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	↑↓	↑↓

L'opération **destination – source** entraîne :
↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
report (retenue) issu du bit 15 du résultat (C)
dépassement (over-flow) mis à 1 si signe du résultat aberrant
- Le bit X n'est pas modifié

exemple de mise en oeuvre:
Vérifier si le contenu du registre a5 est ou non égal à la valeur 0x20000000.
Si non, « aller » à l'étiquette NON …

```
        …
        cmp.l     #0x20000000,a5    ;comparaison de a5 avec la constante 0x20000000
        bne.s     NON               ;saut si a5 n'est pas égal à 0x20000000
OUI :   …                           ;suite du prog. si a5 == 0x20000000
        …
NON :   …                           ;suite du programme si a5 != 0x20000000
        …
```

5.8.2.15 **DIVS**

DIVS — Les 2 opérandes divisés sont des nombres signés.

Forme DIVS.W 32/16
Opération de division entre le contenu (long) d'un registre de donnée *di* et l'opérande source (mot).
Le quotient (résultat) est déposé dans le mot bas du registre de destination.
Le reste de la division est déposé dans le mot haut du bregistre de destination.
Forme DIVS.L 32/32
Opération de division entre le contenu (long) d'un registre de donnée *di* et l'opérande source (long).
Le quotient (résultat) est déposé dans le registre de destination (long).
Le reste de la division n'est pas disponible. (si nécessaire, voir instruction REMS)

	modes d'adressage source possibles	mode d'adr. dest. possible
DIVS.W	tous les modes sont possibles sauf *ai*	seuls *di* possibles
DIVS.L	tous les modes sont possibles : sauf *ai, dep(ai,xi), absolu, #, dep(pc), dep(pc,xi)*	seuls *di* possibles

syntaxe générale: **divs.w <@>,di**
divs.l <@>,di

format du code opération : Forme DIVS.W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. di			*code*			*mode d'adressage*					
1	0	0	0	-	-	-	1	1	1	code mode			num. reg. ou code		

format du code opération : Forme DIVS.L (occupe 2 mots de codage)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code										*mode d'adressage source*					
0	1	0	0	1	1	0	0	0	1	code mode			num. reg. ou code		
1	num. reg. di			1	0	0	0	0	0	0	0	0	num. reg. di		

voir tableau modes d'adressage page 61 : Note : Dans la forme .L, le numéro du registre destination apparaît 2 fois

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	↑↓	0

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
dépassement (over-flow) mis à 1 si signe du résultat aberrant
- Le bit X n'est pas modifié C est mis à 0

exemple de mise en oeuvre:
Diviser le contenu du registre (mot) d4 par 9. Ranger le résultat (mot) en 0x10000, et le reste en 0x10002. Le contenu de d4 est un nombre signé.

```
…
divs.w      #9,d4          ;divise d4.w par la constante 9
move.w d4,0x10000          ;range le résulat à l'adresse 0x10000
sawp.w  d4                 ;place le reste dans la partie basse du reg. d4 (résultat ➔ partie haute)
move.w d4,0x10002          ;range le reste de la division à l'adresse 0x10002
…
```

5.8.2.16 DIVU

DIVU

Les 2 opérandes divisés sont des nombres non signés.

Forme DIVS.W 32/16

Opération de division entre le contenu (long) d'un registre de donnée *di* et l'opérande source (mot).
Le quotient (résultat) est déposé dans le mot bas du registre de destination.
Le reste de la division est déposé dans le mot haut du bregistre de destination.

Forme DIVS.L 32/32

Opération de division entre le contenu (long) d'un registre de donnée *di* et l'opérande source (long).
Le quotient (résultat) est déposé dans le registre de destination (long).
Le reste de la division n'est pas disponible. (si nécessaire, voir instruction REMS)

	modes d'adressage source possibles	mode d'adr. desti. possible
DIVU.W	tous les modes sont possibles sauf *ai*	seuls *di* possibles
DIVU.L	tous les modes sont possibles : sauf *ai*, *dep(ai,xi), absolu, #, dep(pc), dep(pc,xi)*	seuls *di* possibles

syntaxe générale: **divu.w <@>,di**
divu.l <@>,di

format du code opération : Forme DIVS.W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. di			*code*			*mode d'adressage*					
1	0	0	0	-	-	-	0	1	1	code mode			num. reg. ou code		

format du code opération : Forme DIVS.L (occupe 2 mots de codage)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code										*mode d'adressage source*					
0	1	0	0	1	1	0	0	0	1	code mode			num. reg. ou code		
0	num. reg. di			0	0	0	0	0	0	0	0	0	num. reg. di		

voir tableau modes d'adressage page 61 : Note : Dans la forme .L, le numéro du registre destination apparaît 2 fois

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	↑↓	0

. ↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
dépassement (over-flow) mis à 1 si signe du résultat aberrant
- Le bit X n'est pas modifié C est mis à 0

exemple de mise en oeuvre:

Diviser le contenu du registre (long) a2 par 9, le contenu de a2 (adresse) est un nombre non signé.
Le résultat de la division doit être rangé dans le registre a2.

```
        …
        move.l      a2,d0       ;déposer a2 dans un registre de donnée (restriction mode d'adressagev
sur ai)
        divs.w      #9,d0       ;divise d4.w par la constante 9
        move.l      d0,a2       ;range le résulat dans le reg. « destination » a2
        …
```

5.8.2.17 EOR

EOR.l

OU exclusif logique (bit à bit) entre l'opérande source et l'opérande destination.
Le résultat est déposé dans l'opérande destination. Les bits d'état N et Z sont influencés par l'opération

modes d'adressage source possibles	modes d'adressage destination possibles
seuls : *di* possibles et immédiat #	Si source *di* tous les modes sont possibles saufs : *dep(pc) , dep(pc,xi)* et # ! Si source # seul le mode *di* est possible pour la destination.

syntaxe générale: **eor.l di,<@>**
ou bien **eor.l #,di** (== eori.l #,di)

format du code opération : Forme eor.l Di,<@>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. di			code			*mode d'adressage*					
1	0	1	1	-	-	-	1	1	0	code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 :

format du code opération : Forme eor.l #,Di (1 mot code opération + 2 mots pour la valeur de la constante)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	1	0	0	0	0	num. reg. di		
partie haute (mot haut) de la constante															
partie basse (mot bat) de la constante															

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
\- non modifié (le bit X conserve sa valeur initiale)
0 les bits V et C sont forçé à zéro.

exemple de mise en oeuvre:
Faire un OU exclusif entre la valeur 0x33 et la valeur de l'octet d'adresse 0x2000 ? Si le résultat du ⊕ est nul, effectuer un saut à l'adresse Egal, sinon poursuivre.

```
…
clr.l       d0          ;reg. préparé pour acceuil de l'octet et opération sur long
move .b 0x2000,d0       ;acquisition de l'octet à tester, d0 == 0x000000??
oer.l       #0x33,d0    ; d0 ⊕  0x00000033 --> d0 == 0x000000??
beq.s       Egal        ;test si l'octet valait  0x33 ?
                        ;saut vers étiq. Egal si l'octet valait 0x33
…..                     ;codes exécutés si l'octet ne valait pas 0x33
Egal :
```

5.8.2.18 **EXT EXT B**

EXT.w/.l EXTB.l

Cette instruction assure lors de son exécution, l'extension du signe (msb de la source) :
d'un octet vers un mot . ext.w dx ……xxxxxxxx **yxxxxxxx** ➔ ……yyyyyyyy yxxxxxxx
d'un mot vers un long . ext.l dx xxxxxxxxxxxxxxxx **yxxxxxxxxxxxxxxx** ➔ yyyyyyyyyyyyyyyy yxxxxxxxxxxxxxxx
d'un octet vers un long. extb.l dx xxxxxxxxxxxxxxxxxxxxxxxx **yxxxxxxx** ➔ yyyyyyyyyyyyyyyyyyyyyyyy yxxxxxxx

seul mode d'adressage destination possible

di

syntaxe générale: **ext.w di** ou bien **ext.l di**
ou bien **extb.l di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code							code forme			*code*			num. reg. di		
0	1	0	0	1	0	0	-	-	-	0	0	0	-	-	-

voir tableau modes d'adressage page 61 :
code forme : 010 pour forme *ext.w* ;011 pour forme *ext.l* *;111* pour forme extb.l

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
\- non modifié (le bit X conserve sa valeur initiale)
0 les bits V et C sont forçé à zéro.

exemple de mise en oeuvre:
Etendre le signe de l'octet bas du registre de donnée d4. d4 initial == 0x12345688
1) en négligeant l'existance de l'instruction extb.l
2) en exploitant l'existance de l'instruction extb.l

1)

```
…
ext.w       d4          ; d4 devient 0x1234ff88   ff est l'extension du msb (1) de 88
ext.l       d4          ; d4 devient FFFFff88   FFFF est l'extension du msb (1) de ff
…
```

2)

```
…

extb.l      d4          ; d4 devient (en 1 seule instruction) ffff ff88
                        ;ffff ff est l'extention du msb (1) de l'octet bas 88
….
```

5.8.2.19 **JMP**

JMP

Saut inconditionnel. PC adopte la valeur de l'adresse élaborée via le mode d'adressage choisi. L'instruction JMP est fonctionnellement compable à l'instruction BRA, mais autorise un saut dans la totalité de l'espace mémoire (2^{32} octets).

mode d'adressage

tous possibles sauf : *di, ai, (ai)+, -(ai), #*

syntaxe générale: **jmp** <@>

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code										*mode d'adressage*					
0	1	0	0	1	1	1	0	1	1	code mode			num. reg. ou code		

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est modifié par cette instruction.

exemple de mise en oeuvre:

via l'instruction jmp réaliser une boucle de programme sans fin .
1) en utilisant le mode d'adressage absolu.
2) en utilisant l'adressage indexé sans déplacement.

1)

```
BOUCLE :
                …
                …                       ;corps de la boucle
                jmp     BOUCLE          ;l'adresse absolue est celle de l'etiquette BOUCLE
2)
                lea     BOUCLE,a4       ;le registre d'adresse a4 est initialise avec la valeur de l'adresse
                                        ;de boucle. Sa valeur est supposée inchangée au sein de la
                                        ; boucle
BOUCLE :
                …
                …                       ;corps de la boucle
                jmp     (a4)            ;la valeur de a4 (BOUCLE) sert d'adresse de destination
                …
```

5.8.2.20 **JSR**

JSR

Appel à un sous programme.
L'adresse du point d'entrée dans le sous programme est élaborée via le mode d'adressage retenu.
Cette instruction assure la sauvegarde de la valeur du compteur ordinal (adresse de l'instruction qui suit JSR) « sur » la pile . PC → (SP-4)
Le pointeur de pile SP est décrémenté de 4. SP = SP – 4.
Cela permet, au sein du sous programme appelé, d'appeler un nouveau sous programme avec le même mécanisme de sauvegarde de PC sans « écrasser » le dernier PC sauvegardé. Cela permet d'effectuer des appels à sous programmes « imbriqués » (voir illustration ci dessous)
Enfin PC est chargé avec la valeur de l'adresse calculée (point d'entrée du sous programme)

PC = valeur de <@> = point entrée sous programm

sous_prog_1 : sous_prog_2 :

JSR <sous_prog_1> JSR <sous_prog_2> etc…
α β δ

	0…	….	
etc…		SP-12 →	@δ
P après le 2ième JSR (dans le sous programme sous_prog_2:)	évolution des adresses	SP-8 →	@β
SP après le 1er JSR (dans le sous programme sous_prog_1:)		SP-4 →	@α
SP initial « dans le prog. principal	… ∞	→	

syntaxe générale: **jsr <@>** (<@> == étiquette point d'entèe du sous programme)

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code										*mode d'adressage*					
0	1	0	0	1	1	1	0	1	0	code mode			num. reg. ou code		

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est modifié par cette instruction.

exemple de mise en oeuvre: (usage du mode absolu)
Appel du sous programme calcul.
SP initial 0x20000 ; @ de l'instruction JSR 0x1000 ; @ de début du sous prog. calcul 0x4000

```
                …
@0x1000         jsr     calcul      ;appel du sous programme (codage sur 1 mot de code opération
                                    ;+ 2 mots qui contiennnent la valeur de l'adresse calcul==
                                    ;0x00004000)
@0x1006                 …
@0x4000         calcul :…           ;début du sous programme calcul. Ici  SP= 0x1fffc = 0x2000 - 4
                …                   ;à partir de l'adresse pile 0x1fffc on trouve : « PC » 0x1006
```

5.8.2.21 **LEA**

LEA

Lors de l'exécution de cette instruction, le mode d'adressage proposé en tant que source est, évalué (calculé). La valeur évaluée est déposée dans la destination qui ne peut être qu'un registre d'adresse (scritement long).
Cette instruction permet d'initialiser un registre d'adresse avec une valeur « complexe » (voir exemple).

modes d'adressage source possibles	mode d'adressage destination possible
tous les modes sont possibles sauf : *di*, *ai*, *(ai)+* , *-(ai)* et # (immédiat) .	seul *ai* possible

syntaxe générale: **lea <@>,ai**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. ai			*code*			*mode d'adressage*					
0	1	0	0	-	-	-	1	1	1	code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est modifié par cette instruction.

exemple de mise en oeuvre:

Mettre dans le registre a3, la valeur de létiquette ICI :

1) en utilisant l'instruction move

```
        …
        move.l      #ICI,a3
        …
ICI :   …
```

2) en utilisant l'instruction lea

```
        …
        lea     ICI,a3
        …
ICI :   …
```

Mettre dans le reg. a3, la valeur résultant du calcul du mode indéxé avec déplacement sur a6 : 12(a6)

1) en utilisant l'instruction move

```
        …
        move.l      a6,d0
        add.l       #12,d0
        move.l      d0,a6
        …
```

2) en utilisant l'instruction lea

```
        …
        lea     12(a6),a3
        …
```

On note dans ce dernier exemple l'efficacité de l'instruction lea. 1 instruction par rapport à 3 instructions, et sans destruction de registre (ici d0 est détruit dans la solution sans lea).

5.8.2.22 LINK

LINK

(voir aussi instruction unlink (unlk))

Cette instruction permet de réserver sur la pile, *n* octets. Le nombre n est fixé par la valeur de la constante #. De plus cette instruction sauve sur la pile la valeur « courante » du registre *ai* désigné, et place dans ce dernier la valeur courante du stack pointeur a7 – 4.
A la fin de l'exécution a7 == a7 initial – (4 + *n*). n est un nombre en général négatif.

Opération résumée liée à link ai,#-n

ai → (sp – 4) ;ai courant sauvé sur la pile à l'adressse sp - 4
sp ← sp – 4 ;sp autodécrémenté de 4 unités
ai ← sp ;ai prend la valeur de sp
sp ← sp – n ;sp est modifié, la valeur signée de *n* est ajoutée à sa valeur courante

L'espace ai – sp de n octets est disponible pour divers stockage (par exemple, pour les variables locales d'une fonction C). Ces variables sont manipulables via l'adressage indexé via *ai*. *ai* est souvent nommé pointeur de cadre (frame pointer)

mode d'adressage source possible	mode d'adressage destination possible
seuls *ai* possibles	seuls # *(immédiat)* possibles

syntaxe générale: **link ai,# # == mot signé**

format du code opération : (occupe 2 mots de codage, 1 code opération + 1 constante # sur mot signé)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													num. reg. *ai*		
0	1	0	0	1	1	1	0	0	1	0	1	0	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

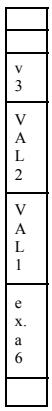
Aucun bit d'état n'est modifié par cette instruction.

exemple de mise en oeuvre:

« compiler le début du programme C suivant : (le registre a6 sera retenu comme « frame pointer »
Si sp == a7 initial = 0x10000000

```
main()
{
int     VAL_1= 0, VAL_2 ;   →   link    a6,#-10  ;-10 == place nécessaire
short   val_3= -34;                              ;pour ranger 2 "int" (4+4)
VAL_2 = VAL_1;                                   ; + 1 "short" (2)
….                              clr.l   -4(a6)   ; == VAL_1=0
                                moveq   #-34,d0  ;prépare val_3 = -34
                                move.w d0,-10(a6) ; == val_3 = -34
                                move.l  -4(a6),-8(a6) ; == VAL_2 = VAL_1
                                …
```

à ce stade a7 = 0x10000000 – 4 – 4 – 4 – 2 = 0xfffffffe



état de la pile

5.8.2.23 **LSL**

LSL.l

Décalage logique de n bits vers la gauche. Cette instruction est rigoureusement identique à l'instruction ASL.l
Le nombre de décalage est limité de 1 à 8 (zéro impossible) s'il est proposé en tant que constante (mode # immédiat).
Le nombre de décalage est limité de 0 à 64 s'il est proposé en tant que contenu de registre.
Cette opération est équivalente à une multiplication par 2^n, *n* étant le nombre de décalages demandés.

mode d'adressage source possible	modes d'adressage destination possibles
immédiat # (valeur maxi : 8 ; 0 non admis) ou bien *di*	seulement *di*

syntaxe générale: **lsl.l #,di**
ou bien **lsl.l di,di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				*donnée # ou num. reg*			*d r*	*c o d e*		*i / r*	*c o d e*		*num. reg. dest.*		
1	1	1	0	-	-	-	1	1	0	-	0	1	-	-	-

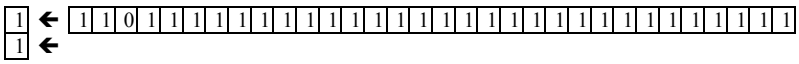
dr = 1 pour décalage à gauche.
i/r = 0 ➔ nbr. décalage mode immédiat, = 1 ➔ nbr. décalage mode registre di

influence sur bits d'états :

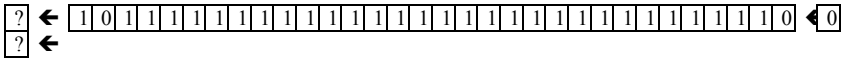
X	N	Z	V	C
↑↓	↑↓	↑↓	0	↑↓

↑↓ mis à 1 ou à 0 suivant le signe (N) ou la valeur nulle ou non du dernier bit décalé (Z) report (retenue) issu du bit 15 du résultat (C et X)
V bit de dépassement (over-flow) est forcé à 0

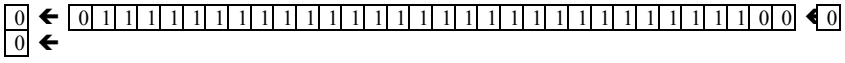
exemple de mise en oeuvre:
décaler de 2 bits à gauche le registre d7 en mode immédiat lsl.l #2,d7
d7 initial (0xdfffffff) et bits d'état C et X



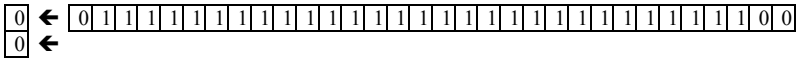
étape 1 et bits d'état C et X



étape 2 et bits d'état C et X



d7 final (0x7ffffffc) et bits d'état C=0 et X=0



5.8.2.24 **LSR**

LSR.l

Décalage logique de n bits vers la droite.
Le nombre de décalage est limité de 1 à 8 (zéro impossible) s'il est proposé en tant que constante (mode # immédiat).

Le nombre de décalage est limité de 0 à 64 s'il est proposé en tant que contenu de registre.

mode d'adressage source possible	modes d'adressage destination possibles
immédiat # (valeur maxi : 8 ; 0 non admis) ou bien *di*	seulement *di*

syntaxe générale: **lsr.l #,di**
ou bien **lsr.l di,di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				*donnée # ou num. reg*			*d r*	*c o d e*		*i / r*	*c o d e*		*num. reg. dest.*		
1	1	1	0	-	-	-	0	1	0	-	0	1	-	-	-

dr = 0 pour décalage à droite.
i/r = 0 ➔ nbr. décalage mode immédiat, = 1 ➔ nbr. décalage mode registre di

influence sur bits d'états :

X	N	Z	V	C
↑↓	↑↓	↑↓	0	↑↓

↑↓ mis à 1 ou à 0 suivant le signe (N) ou la valeur nulle ou non du dernier bit décalé (Z) report (retenue) issu du bit 15 du résultat (C et X)
V bit de dépassement (over-flow) est forcé à 0

exemple de mise en oeuvre:
décaler de 2 bits à droite le registre d7 en mode immédiat. asr.l #2,d7
d7 initial (0xdffffffb) et bits d'état C et X

1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	**1**	*1*		

étape 1 et bits d'état C et X. Le bit MSB ← 0.

0 ➔

0	**1**	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	**1**	*1*		

étape 2 et bits d'état C et X. Le bit MSB ← 0.

0 ➔

0	0	**1**	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	**1**		

d7 final (0x37fffffe) et bits d'état C=1 et X=1

0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

5.8.2.25 **MOVE**

MOVE.b/w/l

Copie de l'opérande source dans l'opérande destination

mode d'adressage source possibles	mode d'adressage destination possibles
di, ai, (ai), -(ai), (ai)+	tous les modes sont possibles (sauf # !)
dep(ai), dep(pc)	tous les modes sont possibles saufs : *absolu, dep(ai,xi)* et # !
dep(ai,xi), dep(pc,xi), absolu, #	tous les modes sont possibles saufs : *dep(ai,xi), dep(ai), absolu* et # !

syntaxe générale: **move.taille <@>,<@>** avec taille == b (8 bits) , w (16 bits , l (32 bits)

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code		*taille*		*mode adressage destination*						*mode d'adressage source*					
0	0	-	-	num. reg. ou code			code mode			code mode			num. reg. ou code		

voir tableau modes d'adressage, et des tailles page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

\- non modifié 0 forcé à 0

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)

Attention, si la forme MOVE.w/l <@>,ai est exploitée, aucun bit d'état n'est modifié.

exemple de mise en oeuvre:

déposer la constante 0x12345678 aux l'adresses 0x100000 , 0x20000000 , 0x20000008 , 0x2000000c et 0x1ffffffc.

```
…
lea       0x20000000,a0     ;pointeur a0 sur la zone 0x20000000
move.l    0x12345678,d0     ;initialisation du reg. d0 avec la constante

move.l    d0,0x100000       ;constante stockée à l'@ 0x100000
move.l    d0,(a0)           ;constante stockée à l'@ 0x20000000
move.l    d0,8(a0)          ;constante stockée à l'@ 0x20000008
move.l    d0,0xc(a0)        ;constante stockée à l'@ 0x2000000c*
move.l    d0,-4(a0)         ;constante stockée à l'@ 0x1ffffffc**
…
```

*l'écriture move.l d0,12(a0) est équivalente

**l'écriture move.l d0,0xfffc(a0) est équivalente

5.8.2.26 **MOVEQ**

MOVEQ.l

Copie de l'opérande source qui ne peut être qu'un octet dans le mode immédiat dans l'opérande destination qui ne peut être qu'un registre de donnée *di* . Le bit de signe (7) de l'octet source est étendu (dupliqué) , aux bits de 8 à 31 de la destination.
Il n'y a pas de différence fonctionnnelle entre, par exemple :
move.l #0xffffff80,d0 et moveq.l #0x80,d0
Mais premier move « classique » nécessite 3 mots de codage, donc sur un bus des données de 16 bits 3 cycles de lecture, alors que le moveq (quick) ne nécessite qu'un seul mot, donc un seul cycle de lecture, où sont concaténés code opération et octet source (voir code de l'instruction).

Le choix de cette instruction (quand il est possible) conduit donc à un gain de place en mémoire de programme et à un gain de temps lors de la phase d'exécution.

mode d'adressage source possible	mode d'adressage destination possibles
seulement *# (immédiat)* sur 1 octet.	seulement *di*

syntaxe générale: **moveq.l #,di** Attention, la constante ∈ [0x80, 0x7f]

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				n u m . r e g			*code*	*source constante*							
0	1	1	1	-	-	-	0	-	-	-	-	-	-	-	-

voir tableau modes d'adressage page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

\- non modifié ↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z) 0 forcé à 0

exemple de mise en oeuvre:
Déposer les constantes 0x33 et 0x333 respectivement dans les registres de données d0 et d1

```
…
moveq.l   #0x33,d0      ;occupe 1 seul mot de codage
move.l    #0x333,d1     ;occupe 1 + 2 mots, mais l'instruction
                        ;moveq.l#0x333,d1 est impossible du fait
                        ;de la taille > l'octet de la source 0x333 !
```

5.8.2.27 **MOV3Q**

MOV3Q (seulement pour le MCF5407)

Copie de l'opérande source qui ne peut être qu'un mot écrit sur 3 bits dans le mode immédiat dans l'opérande destination.
Les nombres « possibles » sont –1 (en fait codé 000) et 1 à 7 (codé en binaire naturel 001 …111), le bit de signe de l'octet source est étendu (dupliqué) aux bits de 3 à 31 de la destination.
Ainsi l'instruction move.l #0x6,3(a0) impossible avec le coldfire devient mov3q.l
Le premier move, « classique » nécessiterait 4 mots de codage !
L'instruction mov3q « transporte » la constante (ici 6) dans son code opération, et du coup le codage total compte ici seulement 2 mots, le code opération et l'offset de déplacement 3.
Le choix de cette instruction (quand il est possible) conduit donc à un gain de place en mémoire de programme et à un grain de temps lors de la phase d'exécution.

mode d'adressage source possible	mode d'adressage destination possibles
# (immédiat)-1, 1, 2, 3, 4 '5, 6, 7 seulement.	Tous possibles sauf , *dep(pc), dep(pc,ai)*

Syntaxe général mov3q.l #,<@> Attention la constante ∈ [-1] 0[1,7] 0exclu

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				d o n n é e			*code*			*mode d'adressage destination*					
1	0	1	0	-	-	-	1	0	1	code mode			num. reg. ou code		

voir tableau modes d'adressage page 63 :

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

\- non modifié 0 forcé à 0
↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)

Exemple de mise en œuvre :
Déposer les constantes 0x3 et –1 respectivement dans les longs d'adresse 0x300000 et 0x400000

```
…
mov3q.l     #0x3,0x300000     ;occupe 3 mots de codage (adressage absolu)
mov3.l      #-1,0x400000      ;idem
```

5.8.2.28 **MVS**

MVS.b ou .w (seulement pour le MCF5407)

Copie de l'opérande source dans le registre de donnée de destination.
Si l'origine est un octet (.b) alors son bit de signe est étendu des bits 8 à 31 du registre destination.
Si l'origine est un mot (.w) alors son bit de signe est étendu des bits 16 à 31 du registre destination.

Ainsi l'instruction (hypothèse a0 = 0x400000, d5 = 0x44444444 et (0x400003)° = 0x80)
mvs.b 3(a0),d5 prend l'octet d'adresse 0x400003, le dépose dans le registre d5et étend le bit de signe à 32 bits. Résultat d5 = 0xffffff80

mode d'adressage source possible	mode d'adressage destination possibles
Tous possibles	*Seulement di*

Syntaxe général mvs.taille <@>,di

format du code opération :

15	14	13	12	11	10	9	8	7	6	5 4 3	2 1 0
Code				Registre			Code		Taille	Mode adressage destination	
0	1	1	1	-	-	-	1	0	-	Code mode	Num. reg. ou code

Taille .b => 0 .w => 1
voir tableau modes d'adressage page 63 :

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

\- non modifié 0 forcé à 0
↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)

Exemple de mise en œuvre :
Déposer l'octet d'adresse 0x400000 dans le registre d5 sous forme d'un long avec prise en compte du signe initial

```
…
mvs.s 0x400000,d5
```

Si l'octet initial est un nombre positif , alors d5 final vaut : 0x000000xx xx ∈ [0x00, 0x7f]
Si l'octet initial est un nombre négatif , alors d5 final vaut : 0xffffffxx xx ∈ [0x80, 0xff]

5.8.2.29 **MVZ**

MVZ.b ou .w (seulement pour le MCF5407)

Copie de l'opérande source dans le registre de donnée de destination.
Si l'origine est un octet (.b) alors les bits de 8 à 31 du registre destination sont mis à 0
Si l'origine est un mot (.w) alors les bits de 16 à 31 du registre destination sont mis à 0.

Ainsi l'instruction (hypothèse a0 = 0x400000, d5 = 0x44444444 et (0x400003)° = 0x80)
mvz.b 3(a0),d5 prend l'octet d'adresse 0x400003, le dépose dans le registre d5et met à 0 les bits de 8 à 32 . Résultat d5 = 0x00000080

mode d'adressage source possible	mode d'adressage destination possibles
Tous possibles	*Seulement di*

Syntaxe général mvs.taille <@>,di

format du code opération :

15	14	13	12	11	10	9	8	7	6	5 4 3	2 1 0
Code				Registre			Code		Taille	Mode adressage destination	
0	1	1	1	-	-	-	1	1	-	Code mode	Num. reg. ou code

Taille .b => 0 .w => 1
voir tableau modes d'adressage page 63 :

influence sur bits d'états :

X	N	Z	V	C
-	0	↑↓	0	0

\- non modifié 0 forcé à 0
↑↓ mis à 1 ou à 0 suivant la valeur nulle ou non (Z)

Exemple de mise en œuvre :
Déposer l'octet d'adresse 0x400000 dans le registre d5 sous forme d'un long

```
…
mvs.s                0x400000,d5
```

Que l'octet initial soit un nombre positif ou négatif d5 final vaut : 0x000000xx xx ∈ [0x00, 0xff]

5.8.2.30 **MOVE from CCR**

MOVE.w ➔ de la partie code condition du registre d'état vers un registre de donnée.

Copie de la partie code de condition (ccr) du registre d'état dans le registre de donnée destiantion (partie basse). L'octet ccr est placé dans l'octet bas de *di*. L' octet suivant est rempli de zéro.

Attention, bien que seul un octet (ccr) soit manipulé, c'est un mot qui doit définir l'argument taille de l'instruction. Cette instruction permet d'effectuer un traitement logique ou de test sur les bits du registre ccr. Voir l'instruction MOVE to CCR pour la copie inverse.

mode d'adressage source possible	mode d'adressage destination possibles
seulement *# (immédiat)* sur 1 octet.	seulement *di*

syntaxe générale: **move.w** **ccr,di** Attention, argument de taille ➔ mot ➔ .w

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													n u m .		r e g
0	1	0	0	0	0	1	0	1	1	0	0	0	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

\- non modifié

Aucun bit d'état n'est modifié par cette instruction.

exemple de mise en oeuvre:

Vérifier que les bits C et X sont tous les à 1 suite à l'exécution de l'instruction « III »
Si c'est le cas aller à OUI, sinon poursuivre en NON.

```
                …
                « III »                 ;instruction dont les concéquences sur les bits C et V sont à
tester
                move.w ccr,d0           ;copie de ccr dans d0 ➔ « 0x?? ?? 00 ccr »
                and.l   #0xff,d0        ;forme d0 ➔ « 00 00 00 ccr » via le masque 0xff
                cmp.l   #0x11,d0        ;teste X et C à 1 ?
                beq.s   OUI             ;si X == C == 1, saut à l'étiquette OUI
NON :           …                       ;si X == C == 0 alors poursuite !
                …
OUI :           …
                …
```

5.8.2.31 MOVE to CCR

MOVE.b ➔ d'un reg. de donnée ou constante vers la partie code condition du d'état d'état.

Copie de l'octet bas du registre de donnée *di* , ou un argument constant (#) dans la partie code de condition (ccr) du registre d'état.
Les divers bit d'état prennent les valeurs des bits équivalents (position) des bits du registre source *di* impliqué ou de la contante # proposée.
Cette instruction permet de donner toutes valeurs particulières aux bits de la partie ccr du registre d'état SR, sans modifier la partie haute de ce dernier.
Voir l'instruction MOVE from CCR pour la copie inverse.

modes d'adressage source possibles	mode d'adressage destination possible
seulement *di* et immpédiat *#.*	seulement *ccr*

syntaxe générale: **move.b di,ccr**
ou bien **move.b #,ccr**
Attention, argument de taille ➔ octet ➔ .b, différent de l'instruction move.w ccr,di

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													n u m .		r e g
0	1	0	0	0	0	1	0	1	1	0	0	0	-	-	-

influence sur bits d'états :

X	N	Z	V	C
↑↓	↑↓	↑↓	↑↓	↑↓

↑↓ les bits d'état de ccr adoptent les valeurs de leur alter-ego de l'argument source

exemple de mise en oeuvre:
Inverser l'état du bit V, sans altérer les bits voisins du regsitre ccr.

```
…
move.w ccr,d0          ;acquisition et "gel" de l'état des bits du reg. ccr
bchg.l   #1,d0         ;inverse le bit de poids 1 de d0, en fait l'image
                       ;initiale du bit V de la partie ccr du bit d'état SR
                       ;suite à l'exécution de cette instruction le reg. ccr
                       ;est modifié (bit Z), mais l'image de ce dernier est
                       ;toujours préservée dans le bit de poids 2 du reg. d0
move.b  d0,ccr         ;ccr == ccr initial, avec la valeur du bit V inversée.
…
```

5.8.2.32 **MOVEM**

MOVEM.l

Transfert vers ou depuis (en fonction de la situation source ou destination des regsitres) la pile, des registres désignés sous forme d'une énumération. Cette instruction optimise les mécanismes de sauvegarde et de restitution des registres dans les modules logiciels tels que les sous-programmes ou les traitements d'interruptions ou d'exceptions logicielles.

	mode d'adressage source possibles	mode d'adressage destination possibles
sauvegarde forme registre ➔ pile	*di, ai* (voir forme « énumérée » ci-dessous)	*(ai), dep(ai)*
restitution forme pile ➔ registre	*(ai), dep(ai)*	*di, ai* (voir forme « énumérée » ci-dessous)

syntaxe générale: sauvegarde **movem.l** **liste_reg_à_sauver,<@>**
restitution **movem.l** **<@>,liste_reg_à_restituer**

exemples d'écriture de liste : désigner tous les registres : d0-a7
forme « énumérée » désigner d0, d4,d5,d6,d7,a0,a1 ,a4 et a6 d0/d4-a1/a4/a6

format du code opération :
Cette instruction occupe 2 mots de codage. Un mot « code » + un mot « liste »

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code					direct.	*code*				*mode d'adressage source*					
0	1	0	0	1	-	0	0	1	1	code mode			num. reg. ou code		
a7	a6	a5	a4	a3	a2	a1	a0	d7	d6	d5	d4	d3	d2	d1	d0

voir tableau modes d'adressage page 61 : si le bit direction = 0 ➔ sauvegarde, registres vers mémoire.
si le bit direction = 1 ➔ restitutions, mémoire vers registres.

Dans le mot « liste » un bit à 1 désigne la sauvegarde (ou la restitution) du registre désigné. Un bit à 0 signifie que le registre associé n'est pas conserné par l'instruction. (exemple l'écriture d0/d4-a1/a4/a6 ➔ mot liste == 0x53f1

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

exemple de mise en oeuvre:
Au sein d'un sous programme sauvegarder les registres d0 d5 et a3 à a5 puis les restituer à la fin du sous programme. Exploiter l'adressage (a6), ménager un espace de sauvegarde de 4x4longs.

```
debut :   link     a6,#-16          ;reservation de 16 octets. (voir inst. link)
          movem.l  d0/d5/a3/a5,-16(a6)  ;sauvegarde des registres source
                                   ;écriture équivalente : movem.l  d0/d5/a3/a5,(sp)
          …                        ;corps du sous-programme
          movem.l  -16(a6),d0/d5/a3/a5  ;restitution des rgistres destination
          unlk     a6              ;récupération de l'espace de sauvegarde
          rts                      ;fin du sous programme
```

5.8.2.33 MULS

MULS Les 2 opérandes multipliés sont des nombres signés.

Forme MULS.W 16x16 → 32

Opération de multiplication entre les contenus (mots) d'un registre de donnée *di* et l'opérande source.
Le produit (résultat) long est déposé dans le registre de destination.

Forme MULS.L 32x32 → 32

Opération de multiplication entre les contenus (long) d'un registre de donnée *di* et l'opérande source.
Du produit (résultat) d'un format théorique de 64 bits, seuls les 32 bits de poids faibles sont conservés et déposés dans le registre de destination (long).
Les 32 bits de poids forts sont perdus.

	modes d'adressage source possibles	mode d'adr. dest. possible
MULS.W	tous les modes sont possibles sauf *ai*	seuls *di* possibles
MULS.L	tous les modes sont possibles : sauf *ai, dep(ai,xi), absolu, #, dep(pc), dep(pc,xi)*	seuls *di* possibles

syntaxe générale: **muls.w <@>,di**
muls.l <@>,di

format du code opération : Forme MULS.W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. di			*code*			*mode d'adressage*					
1	1	0	0	-	-	-	1	1	1	code mode			num. reg. ou code		

format du code opération : Forme MULS.L (occupe 2 mots de codage)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code										*mode d'adressage source*					
0	1	0	0	1	1	0	0	0	0	code mode			num. reg. ou code		
0	num. reg. di			1	0	0	0	0	0	0	0	0	0	0	0

voir tableau modes d'adressage page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
dépassement (over-flow) mis à 1 si signe du résultat abérant
\- le bit X n'est pas modifié, le bit de retenue C et V sont forcés à 0

exemple de mise en oeuvre:

Multiplier (mot) d4 par -999. Ranger le résultat (long) en 0x10000. Le contenu de d4 est un nombre, lui même, signé.

```
…
muls.w      #-999,d4        ;produit de d4.w par la constante –999 (== 0xfc19)
move.l      d4,0x10000      ;range le résulat à l'adresse 0x10000
…
```

5.8.2.34 **MULU**

MULU — Les 2 opérandes multipliés sont des nombres non signés.

Forme MULU.W 16x16 → 32

Opération de multiplication entre les contenus (mots) d'un registre de donnée *di* et l'opérande source.
Le produit (résultat) long est déposé dans le registre de destination.

Forme MULU.L 32x32 → 32

Opération de multiplication entre les contenus (long) d'un registre de donnée *di* et l'opérande source.
Du produit (résultat) d'un format théorique de 64 bits, seuls les 32 bits de poids faibles sont conservés et déposés dans le registre de destination (long).
Les 32 bits de poids forts sont perdus.

	modes d'adressage source possibles	mode d'adr. dest. possible
MULU.W	tous les modes sont possibles sauf *ai*	seuls *di* possibles
MULU.L	tous les modes sont possibles : sauf *ai, dep(ai,xi), absolu, #, dep(pc), dep(pc,xi)*	seuls *di* possibles

syntaxe générale: **mulu.w <@>,di**
mulu.l <@>,di

format du code opération : Forme MULU.W

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. di			*code*			*mode d'adressage*					
1	1	0	0	-	-	-	0	1	1	code mode			num. reg. ou code		

format du code opération : Forme MULS.L (occupe 2 mots de codage)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code										*mode d'adressage source*					
0	1	0	0	1	1	0	0	0	0	code mode			num. reg. ou code		
0	num. reg. di			0	0	0	0	0	0	0	0	0	0	0	0

voir tableau modes d'adressage page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
dépassement (over-flow) mis à 1 si signe du résultat abérant
\- le bit X n'est pas modifié, le bit de retenue C et V sont forcés à 0

exemple de mise en oeuvre:

Multiplier (mot) d4 par 999. Ranger le résultat (long) en 0x10000. Le contenu de d4 est un nombre, lui même, non signé.

```
…
mulu.w      #999,d4      ;produit de d4.w par la constante 999 (== 0x03e7)
move.l      d4,0x10000   ;range le résulat à l'adresse 0x10000
…
```

5.8.2.35 **NEG**

NEG.l

Cette instruction réalise le complément à 2 du contenu long d'un registre de donnée *di*.

mode d'adressage possible
seulement *di*

syntaxe générale: **neg.l di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													n u m .		r e g
0	1	0	0	0	0	1	0	1	0	0	0	0	-	-	-

influence sur bits d'états :

X	N	Z	V	C
↑↓	↑↓	↑↓	↑↓	↑↓

↑↓ Tous les bits d'état sont modifiés par cette instruction

exemple de mise en oeuvre:
Réaliser la soustraction, contenu du registre d1 moins contenu du registre d2, résultat dans le registre d3. Utiliser l'algorithme A – B == A + complément à 2 de B .

```
…
move.l      d2,d3       ;à ce point, d1 == A et d2 == B
neg.l       d3          ;d3 == B
add.l       d1,d3       ;d3 == complement à 2 de B
                        ;d3 == A + complement à 2 de B
…
```

5.8.2.36 **NEGX**

NEGX.l

Cette instruction réalise, comme l'instruction addx pour l'addition, l'opération complément à 2 du contenu long d'un registre de donnée *di* en tenant compte de la valeur du bit d'état X (retenue de la précédente opération. Elle permet en fait de réaliser par morceau le complément à 2 d'un nombre de plus de 32 bits. Il faut toutefois, pour le traitement du « premier » morceau (partie basse), que le bit d'état X ai été mis à 0 par programme

mode d'adressage possible

seulement *di*

syntaxe générale: **negx.l di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													n u m .		r e g
0	1	0	0	0	0	0	0	1	0	0	0	0	-	-	-

influence sur bits d'états :

X	N	Z	V	C
↑↓	↑↓	↑↓	↑↓	↑↓

↑↓ Tous les bits d'état sont influencés par cette instruction

exemple de mise en oeuvre:

Réaliser le complément à 2 du nombre de 64 bits rangé à partir de l'adresse 0x200000.
C'est à la même adresse que doit être le « nouveau » nombre.

```
…
lea       0x200008,a0   ;a0 "pointe" juste au dessous du nombre de 8 octets
move.l    -(a0),d0      ;d0 contient les 4 octets de poids faible du nombre
                        ;après exécution, a0 == 0x200004
move.b    #0x00,ccr     ;mise à 0 de tous les bits d'état ➔ X = 0
neg.l     d0            ;complément à 2 des 4 octets de poids faible du nbr.
                        ; X et C adoptent la valeur 0 ou 1 associée
move.l    d0,(a0)       ;range ces 4 octets à partir de l'adresse 0x200004
                        ;lors de ce move X n'est pas altéré.
move.l    -(a0),d0      ;d0 contient les 4 octets de poids fort du nombre
                        ;après exécution, a0 == 0x200000
neg.l     d0            ;complément à 2 des 4 octets de poids fort du nbr.
                        ;cette opé. intégre la valeur de X issue du 1er neg
move.l    d0,(a0)       ;range ces 4 octets à partir de l'adresse 0x200000
…
```

5.8.2.37 **NOP**

NOP

Cette instruction ne réalise rien !
Son exécution entraîne l'incrémentation du compteur ordinal (PC), de 2 unités.
Son exécution prend du temps.
Ce temps n'est pas constant, il est fonction de l'état du « pipeline ».
Son exécution n'est commencée que lorsque tous les cycles engagés pour l'écoulement du « pipeline » sont achevés. Elle permet ainsi si nécessaire de prévenir des problèmes de recouvrement. d'exécutions des instructions en cours d'exécution dans le pipeline.
Pour obtenir juste, la fonction « perte de temps » utiliser l'instruction tpf

Mode d'adressage possible : sans objet !

format du code opération :

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

syntaxe générale: **nop**

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est modifié par cette instruction

exemple de mise en oeuvre:

```
…
iii                 ;l'exécution de l'instruction iii laisse le CPU dans un état E
nop                 ;si l'instruction iii est à l'adresse α
jjj                 ;l'intruction jjj est à l'adresse α + 2, elle s'exécute à partir
                        ;de l'état CPU, E inchangé
…
```

5.8.2.38 **NOT**

NOT.l

Cette instruction réalise le complément à 1 (inversion bit à bit) du contenu long d'un registre de donnée *di*.

mode d'adressage possible

seulement *di*

syntaxe générale: **not.l di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													n u m .		r e g
0	1	0	0	0	1	1	0	1	0	0	0	0	-	-	-

influence sur bits d'états :

X	N	Z	V	C
↑↓	↑↓	↑↓	↑↓	↑↓

↑↓ Tous les bits d'état sont influencés par cette instruction

exemple de mise en oeuvre:

Réaliser la soustraction, contenu du registre d1 moins contenu du registre d2, résultat dans le registre d3. Utiliser l'algorithme A – B == A + complément à 1 de B + 1.

```
…
move.l      d2,d3       ;à ce point, d1 == A et d2 == B
                        ;d3 == B
not.l       d3          ;d3 == complement à 1 de B
add.l       d1,d3       ;d3 == A + complement à 2 de B
addq.l      #1,d3       ;d3 == A - B
…
```

5.8.2.39 **OR**

OR.l

OU logique (bit à bit) entre l'opérande source et l'opérande destination.
Le résultat est déposé dans l'opérande destination. Les bits d'état N et Z sont influencés par l'opér.

	modes d'adressage source possibles	modes d'adressage destination possibles
forme <@> , di	tous les modes sont possibles	seuls *di* possibles
forme di , <@>	*di*	tous les modes sont possibles saufs : *dep(ai)* , *dep(ai,xi)* et # !

syntaxe générale: **or.l <@>,di**
ou bien **or.l di,<@>**

format du code opération : (Attention : pour la forme or.l #,di voir second format)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				*choix <@>,di ou di,<@>*						*mode d'adressage*					
1	0	0	0	num. reg. di			code forme			code mode			num. reg. ou code		

voir tableau modes d'adressage page 61:
code forme : 010 pour forme <@>,*di* ;110 pour forme *di*,<@>

format or.l #,di .Ce format occupe 3 mots, le code ci-dessous + 2 mots « constante »

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													num. reg. di		
0	0	0	0	0	0	0	0	1	0	0	0	0	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
\- non modifié (le bit X conserve sa valeur initiale)
0 les bits V et C sont forçé à zéro.

exemple de mise en oeuvre:
Lire l'octet d'adresse 0x20000000, et y appliquer un masque en OU de valeur 0x33.
ex : si (0x20000000).b == 0x81, après l'opération ==> (0x20000000).b == 0xb3 == 10**11** 00**1**1

```
…
move .b     0x20000000,d0     ;acquisition de l'octet à modifier, d0 == 0x000000??
or.l        #0x33,d0          ;OU avec la constante 0x00000033
move.b      d0,0x20000000     ;rangement de l'octet « masqué » (modifié)
…
```

5.8.2.40 **PEA**

PEA

Lors de l'exécution de cette instruction, le mode d'adressage proposé en tant qu'argument est, évalué (calculé). La valeur évaluée est ensuite déposée sur la pile (scritement long).
Le pointeur de pile est décrémenté de 4 unités. <@> ➔ (sp –4) ; sp ← sp - 4

Cette instruction permet de réaliser des passage d'arguments de type pointeur via la pile, conformement aux procédure des langages évolués tel que le C (voir exemple).

modes d'adressage de l'argument <@> possibles

tous les modes sont possibles sauf : *di*, *ai*, *(ai)+* , *-(ai)* et # (immédiat) .

syntaxe générale: **pea** **<@>**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code										*mode d'adressage*					
0	1	0	0	1	0	0	0	0	1	code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est modifié par cette instruction

exemple de mise en oeuvre:
Déposer sur la pile (avec décrémentation du pointeur de pile) la valeur associée à l'adresse évaluée par l'expression 12(a4). Il ne faut pas détruire la valeur courante de a4.

1) sans exploiter l'instruction pea.

```
…
move.l      a4,d0       ;copie de a4 dans un reg. de donnée
addq.l      #12,d0      ;calcul (évaluation) de 12(a4)
move.l      d0,-(sp)    ;la valeur calculée est déposée sur la pile et sp ← sp - 4
…
```

2) en utilisant l'instruction pea. (même hypothèse)

```
…
pea         12(a4)      ;remplace les 3 lignes precedentes, et de détruit aucun
                        ;registre
…
```

5.8.2.41 **PULSE**

PULSE

L'exécution de cette instruction, entraîne l'apparition physique du code 0100 sur les broches PST(3-0) du microprocesseur (broches d'état du processeur). Ce code particulier peut être décodé par une logique câblée extérieure et fournir ainsi un signal (impulsion) à chaque exécution de cette instruction.
Placée dans une boucle, par exemple, l'impulsion résultant de l'exécution de l'instruction pulse, sera un signal périodique, dont la mesure temporelle renseignera sur la durée d'exécution de la boucle.
L'exécution de cette instruction ne modifie l'état qu'un des registres du microprocesseur (sauf PC qui est incrémenté de 2 unités)

Pas d'argument.

syntaxe générale: **pulse**

format du code opération :

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

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est modifié par cette instruction

exemple de mise en oeuvre:
Mesure de la période d'exécution d'une boucle. On suppose qu'une logique câblée réalise la matérialisation S = /PST3.PST2./PST1./PST0 . Alors à l'aide d'un période mètre, (ou d'un oscilloscope) la durée qui sépare 2 impulsions S mesure la durée d'exécution des codes de la boucle (y compris la durée d'exécution de l'instruction pulse)

```
boucle :    ….                  ;début de la boucle
            ….                  ;corps de la boucle
            pulse               ;instruction de déclenchement du signal S
            bra     boucle      ;fin de boucle, retour au début de la boucle
            …
```

5.8.2.42 **REMS**

REMS Reste de la division de 2 opérandes 32 bits. Les nombres divisés sont signés.

L'instruction DIVS.L qui réalise la division de 2 nombres signés de 32 bits, ne fournit que le quotient (sur les 32 bits du registre de destination *di*. Si le reste de la division est nécessaire, alors l'instruction rems.l répond à ce besoin en fournissant ce reste, mais ce reste seulement. Les opérandes diviseur et dividante sont respectivement la source <@> et le registre *dj*, le reste est déposé dans le registre *di*. (si nécessaire, voir instruction DIVS.L)

modes d'adressage source possibles (<@>)	mode d'adressage desti. possible (di et dj)
tous les modes sont possibles : sauf *ai, dep(ai,xi), absolu, #, dep(pc), dep(pc,xi)*	seuls *di* possibles

syntaxe générale: **rems.l <@>,di ;dj**

remarques : Si *di* et *dj* sont choisis identiques, c'est l'instruction divs.l qui est exécutée et non rems. l
Si une division par zéro est entreprise, aucun registre n'est modifié, mais l'exception division par zéro est exécutée.
Si le quotient dépasse le plus grand nombre signé possible de 32 bits (>0 ==> 0x7fffffff, <0 ==> 0x80000000), le bit V du registre
ccr est mis à 1), mais le registre *di* n'est pas modifié.

format du code opération : (occupe 2 mots de codage)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code										*mode d'adressage source*					
0	1	0	0	1	1	0	0	0	1	code mode			num. reg. ou code		
0	num. reg. dj			1	0	0	0	0	0	0	0	0	num. reg. di		

voir tableau modes d'adressage page 61 : (rappel, c'est le registre *di* qui contient le reste de la division)

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	↑↓	0

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
dépassement (over-flow) mis à 1 si signe du résultat abérant
\- le bit X n'est pas modifié, le bit de retenue C est forcé à 0

exemple de mise en oeuvre:
Diviser le contenu du registre (long) d4 par 9. Ranger le résultat (long) (quotient) en 0x10000, et le reste en 0x10004. Le contenu de d4 est un nombre signé.

```
…
lea       0x10000,a0   ;prépare un pointeur pour le rangement des résultats
moveq.l   #9,d0        ;prépare le diviseur (qui ne peut pas être une constante sur un .l)
rems.l    d0,d5 ;d4    ;reste de d4/d0 déposé dans d5
divs.l    d0,d4        ;divise d4.l par la constante 9 contenue dans d0, résultat dans d4
move.l    d4,(a0)+     ;range le quotient à l'adresse 0x10000, a0 pointe en 0x10004
move.l    d5,(a0)+     ;range le reste à l'adresse 0x10004, a0 pointe en 0x10008
…
```

5.8.2.43 **REMU**

REMU Reste de la division de 2 opérandes 32 bits. Les nombres divisés sont non signés.

L'instruction DIVU.L qui réalise la division de 2 nombres signés de 32 bits, ne fournit que le quotient (sur les 32 bits du registre de destination *di*. Si le reste de la division est nécessaire, alors l'instruction remu.l répond à ce besoin en fournissant ce reste, mais ce reste seulement.

Les opérandes diviseur et dividante sont respectivement la source <@> et le registre *dj*, le reste est déposé dans le registre *di*. (si nécessaire, voir instruction DIVU.L)

modes d'adressage source possibles (<@>)	mode d'adressage desti. possible (di et dj)
tous les modes sont possibles : sauf *ai, dep(ai,xi), absolu, #, dep(pc), dep(pc,xi)*	seuls *di* possibles

syntaxe générale: **remu.l <@>,di ;dj**

remarques : Si *di* et *dj* sont choisis identiques, c'est l'instruction divu.l qui est exécutée et non remu. l
Si une division par zéro est entreprise, aucun registre n'est modifié, mais l'exception division par zéro est exécutée.
Si le quotient dépasse le plus grand nombre signé possible de 32 bits (>0 ==> 0x7fffffff, <0 ==> 0x80000000) (comme pour l'instruction rems !), le bit V du registre ccr est mis à 1), mais le registre *di* n'est pas modifié.

format du code opération : (occupe 2 mots de codage)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code										*mode d'adressage source*					
0	1	0	0	1	1	0	0	0	1	code mode			num. reg. ou code		
0	num. reg. dj			0	0	0	0	0	0	0	0	0	num. reg. di		

voir tableau modes d'adressage page 61 : (rappel, c'est le registre di qui contient le reste de la division)

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	↑↓	0

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
dépassement (over-flow) mis à 1 si signe du résultat abérant
\- le bit X n'est pas modifié, le bit de retenue C est forcé à 0

exemple de mise en oeuvre: (idem que pour l'instruction remu.l)
Diviser le contenu du registre (long) d4 par 9. Ranger le résultat (long) (quotient) en 0x10000, et le reste en 0x10004.
Le contenu de d4 est un nombre non signé.

```
…
lea       0x10000,a0    ;prépare un pointeur pour le rangement des résultats
moveq.l   #9,d0         ;prépare le diviseur (qui ne peut pas être une constante sur un .l)
remu.l    d0,d5 ;d4     ;reste de d4/d0 déposé dans d5
divu.l    d0,d4         ;divise d4.l par la constante 9 contenue dans d0, résultat dans d4
move.l    d4,(a0)+      ;range le quotient à l'adresse 0x10000, a0 pointe en 0x10004
move.l    d5,(a0)+      ;range le reste à l'adresse 0x10004, a0 pointe en 0x10008
…
```

5.8.2.44 **RTS**

RTS

Retour d'un sous programme.

Les 4 octets pointés sur la pile par le registre sp (= = a7) sont déposés dans le compteur ordinal du microprocesseur. Ces 4 octets étant l'adresse de retour au programme appelant, l'exécution RTS est équivalente à un saut à l'instruction qui suit l'instruction JSR ou BSR du programme appelant (voir instruction JSR ou BSR).

Suite au chargement de PC, le pointeur de pile est incrémenté de 4 unités.

En résumé, l'instruction RTS réalise : (SP).l ==> PC ; SP +4 ==> SP

Mode d'adressage possible : sans objet !

syntaxe générale: **rts**

format du code opération :

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

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

- les bits d'état ne sont pas modifiés par l'instruction RTS

exemple de mise en oeuvre:

Appel du sous programme calcul (usage du mode absolu).

SP initial 0x20000 ; @ de l'instruction JSR 0x1000 ; @ de début du sous prog. calcul 0x4000

Puis à la fin du sous programme retour au programme appelant.

```
                    …               ;programme appelant
@0x1000             jsr     calcul  ;appel du sous programme (codage sur 1 mot de code opération
                                    ;+ 2 mots qui contiennnent la valeur de l'adresse calcul==
                                    ;0x00004000)
@0x1006             yyy             ;instruction qui suit l'instruction jsr

                    …
@0x4000     calcul : …              ;début du sous programme calcul. Ici  SP= 0x1fffc = 0x2000 - 4
                    …               ;à partir de l'adresse pile 0x1fffc on trouve : « PC » 0x1006
                    …               ;corps du sous programme
                    rts             ;PC <== 0x1006 et SP <== SP + 4 = 0x2000
                                    ;c'est donc l'instruction yyy et non xxx qui sera exécutée après
                                    ;l'exécution de rts
                    xxx
                    …
```

5.8.2.45 **SATS**

SATS.l (seulement pour MCF5407)

Cette instruction placée derrière une instruction arithmétique succeptible d'avoir provoqué une saturation (bit V de ccr = 1) réalise sur le registre désigné une éventuelle saturation du contenu de ce registre (si V valait 1) sinon (V=0) le contenu du registre n'est pas modifié

L'algorithme exécuté par l'instruction *sats.l di* est le suivant :

si V de ccr ==1
alors si msb (bit31) de di ==0 alors di = 0x80000000
sinon si msb (bit 31) de di ==1 di = 0x7fffffff
Sinon (V de ccr ==0) alors di est inchangé

Mode d'adressage possible :

Seulement ***di***

syntaxe générale: **sats.l di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													Num. reg.		
0	1	0	0	1	1	0	0	1	0	0	0	0	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
\- le bit X n'est pas modifié, le bit de retenue C et le bit d'over flow sont forcés à 0

exemple de mise en oeuvre:

Assurer la saturation éventuelle du registre d4

```
add.l   (a0),d4   addition d'un opérande pointé par a0 avec le contenu de d4
sats.l  d4        modification de d4 si l'overflow (bit V) s'est mis à 1
```

5.8.2.46 **Sxx**

Sxx ==> xx est le mnémonique d'une condition de test : de CC à VS

Sxx pour Set conditionnel.
Si la condition testée, (via les bits d'état) est vérifiée (est vraie) alors l'octet bas du registre de destination est forcé avec la valeur 0xff, sinon avec la valeur 0x00 (si conditions testées fausses).
Ce sont des combinaisons binaires des bits d'états qui sont testées.
Le tableau ci-dessous met en relation des conditions « testables » et les mnémonique Sxx associés.

mnémo.	code	ce qui est testé…	équation testée
SEQ	0111	dernier résultat nul	Z=1
SNE	0110	der. résultat non nul	Z=0
SGE	1100	der. résultat ≥ 0 en comp. à 2	N⊕V=0
SGT	1110	der. résultat > 0 en comp. à 2	N⊕V ou Z=0
SLT	1101	der. résultat < 0 en comp. à 2	N⊕V=1
SLE	1111	der. résultat ≤ 0 en comp. à 2	N⊕V ou Z=1
SVC	1000	der. résultat avec signe correct	V=0

mnémo.	code	ce qui est testé…	équation testée
SMI	1011	nombre négatif	N=1
SPL	1010	nombre positif	N=0
SCC	0100	der. résultat ≥ 0 en bin. naturel	C=0
SHI	0010	der. résultat > 0 en bin. naturel	C ou Z=0
SCS	0101	der. résultat < 0 en bin. naturel	C=1
SLS	0011	der. résultat ≤ 0 en bin. naturel	C ou Z=1
SVC	1001	der. résultat avec signe incorrect	V=1

syntaxe générale: **Sxx di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				*code condition (voir tableau)*				*code*					num.	reg.	di
0	1	0	1	-	-	-	-	1	1	0	0	0	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Les bits d'état sont testés, mais ne sont pas modifiés par l'instruction Sxx

exemple de mise en oeuvre:
Suite à un test du long d'adresse 0x40000, si ce dernier est un nombre strictement positif, charger dans l'octet bas du registre d4 la valeur –1 (0xff), sinon y mettre 0x00.

```
        tst.l     0x40000    ;teste le long d'adresse 0x40000 et actyalise les bits d'état, en
                             ; particulier le bit de signe N
        sgt       d4         ;si (0x40000).l est positif alors d4== 0x ??????ff sinon 0x ??????00
        …
```

Même programme sans utiliser l'instruction sgt .

```
        tst.l     0x40000    ;teste le long d'adresse 0x40000 et actyalise les bits d'état, en
                             ; particulier le bit de signe N
        bgt       UN         ;saut si la valeur signée testée est strictement positive
        move.b    #-0,d4     ;d4 == 0x ??????00 car valeur testée nulle ou négative
        bra.s     SUITE
UN :    move.b    #-1,d4     ;d4 == 0x ??????ff car valeur testée positive
SUITE : …                    ;suite du programme
```

5.8.2.47 **SUB**

SUB.l

Soustraction arithmétique de l'opérande source à l'opérande destination.
Le résultat est déposé dans l'opérande destination. Tous les bits d'état sont influencés par l'opération.

modes d'adressage source possibles	modes d'adressage destination possibles
tous les modes sont possibles	seuls *di* possibles
di	tous les modes sont possibles saufs : *dep(ai) , dep(ai,xi)* et # !

syntaxe générale: **sub.l <@>,di**
ou bien **sub.l di,<@>**

format du code opération : .

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. di			choix <@>,di ou di,<@>			*mode d'adressage*					
1	0	0	1	-	-	-	code forme			code mode			num. reg. ou code		

voir tableau modes d'adressage page 61:
code forme : 010 pour forme <@>,*di* ;110 pour forme *di*,<@>

Remarque : l'instruction sub.l #,*di* (alors notée subi.l #,*di*) peut se coder également : 0x048*r*, *r* étant le numéro du registre *di* .
Dans ce cas le code opération est suivi sur 2 mots de la valeur de la constante

influence sur bits d'états :

X	N	Z	V	C
↑↓	↑↓	↑↓	↑↓	↑↓

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
report (retenue) issu du bit 15 du résultat (C et X)
dépassement (over-flow) mis à 1 si signe du résultat abérant

exemple de mise en oeuvre:
Effectuer la soustraction du long d'adresse 0x60000 au contenu du registre d7
Le résultat sous forme d'un long sera rangé à l'adresse 0x300000000

```
…
lea      0x60000,a0        ;pointeur sur l'opérande à soustraire au reg . d7
sub.l    (a0),d7           ;d7 – (0x60000) ➔ d7
move.l   d7,0x30000000     ;rangement du résultat en 0x30000000
…
```

5.8.2.48 **SUBA**

SUBA.l

Soustraction arithmétique de l'opérande source à un registre d'adresse.
Le résultat est déposé dans un registre d'adresse.
Les opérandes sont traités en tant que nombres non signés. (Il n'existe pas d'adresse négative)
Aucun bits d'état n'est influencé par l'opération.

modes d'adressage source possibles	modes d'adressage destination possibles
tous les modes sont possibles	seuls *ai* possibles

syntaxe générale: **suba.l <@>,ai**
ou bien plus simplement **sub.l <@>,ai**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				num. reg. ai			*code*			*mode d'adressage*					
1	0	0	1				1	1	1	code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est modifié par l'exécution de cette instruction

exemple de mise en oeuvre:
Soustraire la constante 0x200 au registre a0 et 1 au registre a1.

```
…
sub.l     #0x200,a0      ;a0 - 0x00000200 → a0
sub.l     #1,a1          ;a1 - 0xffffffff → a1
…
```

5.8.2.49 **SUBQ**

SUBQ.l

Soustraction arithmétique « rapide » d'une constante ∈ [1,8] à un opérande destination.
Le résultat est déposé dans un registre ou une adresse de destination.
Tous les bits d'état sont influencés par l'opération.

mode d'adressage source possible	modes d'adressage destination possibles
immédiat # (valeur maxi : 8 ; 0 non admis)	tous les modes sont possibles saufs : *dep(pc) , dep(pc,xi)* et # !

syntaxe générale: **subq.l #,ai**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				*donnée immédiate*			*code*			*mode d'adressage*					
0	1	0	1	-	-	-	1	1	0	code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 :

influence sur bits d'états :

X	N	Z	V	C
↑↓	↑↓	↑↓	↑↓	↑↓

↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)
report (retenue) issu du bit 15 du résultat (C et X)
dépassement (over-flow) mis à 1 si signe du résultat abérant

exemple de mise en oeuvre:
Soustraire la constante 0x8 aux l'adresses 0x20000000 , et 0x1ffffffc.

```
…
lea     0x20000000,a0     ;prépare un pointeur sur l'espace méoire à gérer
subq.l  #8,(a0)           ;8 est retiré au long d'adresse 0x20000000
subq.l  #8,-4(a0)         ; 8 est retiré au long d'adresse 0x1ffffffc
…
```

5.8.2.50 **SUBX**

SUBX.l

Soustraction arithmétique du contenu d'un registre source au contenu d'un registre destination en intégrant à l'adition (lsb) la valeur du bit de retenu (X) telle qu'elle est avant l'exécution de l'inst.
Le résultat est déposé dans un registre de destination.
Cette instruction permet de réaliser des soustractions par « morceaux » sur des nbr. de taille >32 bits.

Tous les bits d'état sont influencés par l'opération.

mode d'adressage source possible	mode d'adressage destination possible
di	*di*

syntaxe générale: **subx.l di,di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code				*mode d'adr. dest.*			*code*						*mode d'adr. source*		
1	0	0	1	num. registre			1	1	0	0	0	0	num. registre		

influence sur bits d'états :

X	N	Z	V	C
↑↓	↑↓	↑↓	↑↓	↑↓

Tous les bits d'état sont influencés par le résultat de l'opération

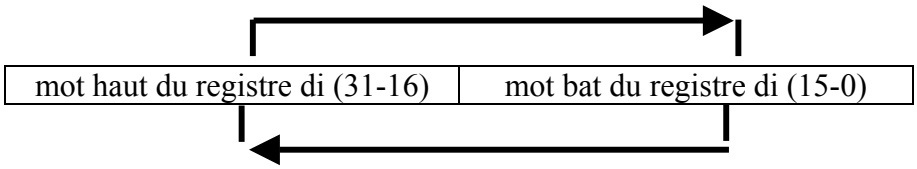
exemple de mise en oeuvre:
Soustraire (2 nbr. de 64 bits) les 2 longs d'adresse 0x10000 aux 2 longs d'adresse 0x20000.

```
…
lea       0x10000,a0      ;pointeur a0 sur la zone 0x10000+4+4
lea       0x20000,a1      ;pointeur a1 sur la zone 0x20000+4+4
move.l    -(a0),d0        ;d0 chargé avec partie basse (32 bits) du premier nombre
move.l    -(a1),d1        ;d1 chargé avec partie basse (32 bits) du premier nombre
move.w    #0,ccr          ;mise à 0 du bit X du regsitre de condition (ccr de SR)
subx.l    d0,d1           ;soustraction des 32 bits de poids faibles, bit X actualisé.
move.l    d1,0x20004      ;stockage des 32 bits de poids faibles.
move.l    -(a0),d0        ;d0 chargé avec partie haute (32 bits) du premier nombre
move.l    -(a1),d1        ;d1 chargé avec partie haute (32 bits) du premier nombre
subx.l    d0,d1           ;addition des 32 bits de poids haut, bit X actualisé.
move.l    d1,0x20000      ;stockage des 32 bits de poids haut.
…
```

5.8.2.51 **SWAP**

SWAP.W

Cette instruction réalise l'inversion des mots haut et bas d'un registre de donnée *di*.



mode d'adressage possible

seulement *di*

syntaxe générale: **swap.w** **di**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													num. reg		
0	1	0	0	1	0	0	0	1	0	0	0	0	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

\- X n'est pas modifié.
↑↓ les bits d'état N et Z modifiés par cette instruction
V et C sont forçés à 0

exemple de mise en oeuvre:
Inverser les mots haut et bas du registre d'adresse a4 !

```
…
move.l      a4,d0       ;passage par un registre de donnée supportant
                        ;l'instruction swap
swap        d0          ;inversion des mots de a4 copiés dans d0
move.l      d0,a4       ;remise des mots inverses dans le registre a4
…
```

5.8.2.52 **TAS**

TAS (seulement pour MCF5407)

Teste le msb (bit7) et l'éventuelle valeur nulle de l'octet mémoire « visé » et actualise les bits d'état N et Z en fonction de la valeur de ce bit.
En suite , SANS RELACHER LES BUS , le msb de cet octet est mis à 1.
Une telle instruction , dite à cycles indivisibles (lecture/modification/ecriture) est essentiellement destinée à la gestion de sémaphores dans le cadre d'une architecture multiprocesseur

modes d'adressage destination possibles

tous sauf di et ai(implicite), #, dep(pc) et dep(pc,xi)

syntaxe générale: **tas.b** <@>

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code										*mode d'adressage destination*					
0	1	0	0	1	0	1	0	1	1	code mode			num. reg. ou code		

voir tableau modes d'adressage et des tailles page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

\- non modifié Vet C forcés à 0
↑↓ les bits N et Z sont modifiés par l'instruction

exemple de mise en oeuvre:
Tester et mettre à 0x80 l'octet « sémaphore » d'adresse 0x4000
Si son msb est déjà à 1 sauter à l'étiquette « occupe » sinon poursuivre.

```
        …
        clr.b       0x4000      ;sémaphore mis à 0, donc son msb == 0
        ….
        tas.b       0x4000      ;teste le sémaphore ET mets son msb à 1
        bne.s       occupe      ;saut si le msb est à 1
        …. Ici poursuite du programme si msb trouvé à 0. msb actuellement à1
        …..
occupe  …. Ici poursuite du programme si msb trouvé à 1
```

5.8.2.53 **TPF**

TPF

Cette instruction, comme l'instruction NOP ne réalise rien, mais suivant l'argument de taille associé sa taille est variable de 2 à 3 mots.
Son exécution entraîne l'incrémentation du compteur ordinal (PC), de 2 unités.
Son exécution prend un temps constant (un cycle d'horloge, quelque soit le format.
Il résulte de son exécution un « saut systématique » de 2 , 4 ou 6 octets suivant qu'elle est invoquée sur un argument de taille .b , .w ou .l
L'exemple montre un cas d'usage pour « éviter » une instruction bra.

mode d'adressage possible
seulement # *(immédiat)*

syntaxe générale: **tpf** ou bien **tpf.w #** **tpf.l #**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													*complément de codage*		
0	1	0	1	0	0	0	1	1	1	1	1	1	-	-	-

avec complément de codage : 100 pour la forme sans mot d'extenstion
010 pour la forme avec un mot d'extenstion
011 pour la forme avec deux mot d'extenstion

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est modifié par l'exécution de cette instruction

exemple de mise en oeuvre:
2 formes équivalentes possibles. 1) avec bra.s 2) avec tpf
qui traduisent toutes les deux le codes C suivant :

```
if      (a == b)
        z = n1;
        else
        z = 2;
        …
```

1)
```
        …
        cmp.l   d0,d1
        beq.s   ETIQ0
        moveq   #2,d2
        bra.s   ETIQ1
ETIQ0:
        moveq.l #1,d2
ETIQ1 : …
```

2)
```
        …
        cmp.l   d0,d1
        beq.s   ETIQ0
        moveq   #2,d2
        tpf.w   #X      ;l'instruction bra disparaît
ETIQ0:                  ;X est ici égal au code OP
                        ;de l'instruction moveq.l #1,d2
ETIQ1 : …               ;l'étiquette ETIQ1 n'est plus
                        ;utile
```

5.8.2.54 **TRAP**

TRAP appel d'un programme d'exception

Cette instruction, réalise deux fonctions distinctes.
a) Passage de l'exécution du programme du mode user (ou superviseur) au mode superviseur (sans qu'il sagisse d'une instruction privilégiée) Pour cela le bit S du registre d'état est mis à 1
b) Mécanisme d'appel à un sous-programme (programme d'exception TRAP n° *n* ($n \in [0,15]$).

Avec : stockage de PC sur la pile en (SP) – 4
stockage du SR initial (avant TRAP) sur la pile en (SP) – 6
stockage d'un mot de « format » sur la pile en (SP) – 8
modification de SP ← SP = SP – (4+2+2)
chargement de PC avec adresse stockée dans la table d'exception.
PC ← (VBR + 0x80 + 4x *n*)

mode d'adressage possible

seulement # *(immédiat)* valeur de *n* ($n \in [0,15]$)

syntaxe générale: **trap #*n***

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code												*valeur de n*			
0	1	0	0	1	1	1	0	0	1	0	0	-	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est modifié par l'exécution de cette instruction

exemple de mise en oeuvre:
Préparer, puis invoquer l'appel de l'exception Calcul, via l'usage de trap #2.
Hypothéses ; VBR = 0 (table d'exception débutant à l'adresse 0) et mode « user »

```
            …
            move.l      #Calcul,d0      ;prepare le stockage de l'adresse "Calcul" ds la table
            move.l      d0,0x88         ;stockage effectif à l'adresse 0x80 + 2x4
            ….                          ;suite du programme
            trap        #2              ;appel « indirect » au programme calcul
            …                           ;retour de calcul en mode « user »

Calcul : ….                             ;corps du programme Calcul
            ….                          ;exécuté en mode superviseur
            rte                         ;instruction de retour d'exception.
```

5.8.2.55 **TST**

TST.b/w/l

Test de l'opérande destination et actualisation des bits d'états.
Le test consiste à réaliser la soustraction opérande destination – 0 (zéro)
Cette instruction (sur une taille .l) derait, si elle exister dans la famille 5307, fonctionnellement équivalente à l'instruction cmp.l #0,<@>

modes d'adressage destination possibles

tous les modes sont possibles*

* Attention, si le mode *ai* est utilisé, seules les tailles .w et .l sont possibles.

syntaxe générale: **tst.taille** **<@>** avec taille == b (8 bits) , w (16 bits , l (32 bits)

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code								*taille*		*mode d'adressage destination*					
0	1	0	0	1	0	1	0	-	-	code mode			num. reg. ou code		

voir tableau modes d'adressage et des tailles page 61 :

influence sur bits d'états :

X	N	Z	V	C
-	↑↓	↑↓	0	0

\- non modifié 0 forcé à 0
↑↓ mis à 1 ou à 0 suivant signe (N) et valeur nulle ou non (Z)

exemple de mise en oeuvre:
Tester si l'octet « binaire naturel », d'adresse 0x50000000 est positif ou nul.
Si oui effectuer un saut à l'étiquette OUI, sinon poursuivre.

```
        …
        tst.b     0x50000000    ;test de l'octet d'adresse 0x50000000
        bge.s     OUI           ;saut conditionnel, effectif si l'octet testé est ≥ 0
NON :   …                       ;codes exécutés si saut non fait ➔ octet testé < 0
        …
OUI :   …                       ;codes exécutés si saut fait ➔ octet testé ≥ 0
```

5.8.2.56 **UNLK**

UNLK (voir aussi instruction link)

Cette instruction permet de libérer de la pile, les *n* octets réservés par l'instruction link.
Elle restitue également le registre désigné en argument. (généralement le registre préservé par l'instruction link)
A la fin de l'exécution a7 == ai initial + 4
Opération résumée liée à unlk ai

```
ai → sp          ;ai courant donne sa valeur au pointeur de pile SP (a7)
(sp) → ai        ;restitution de la valeur ai, préservée sur la pile par l'instruction link
ai,#
sp ← sp + 4      ;récupération des 4 octets de sauvegarde de ai
```

Cette instruction est en général utilisée comme « réplique symétrique » de l'instruction link, est est placée en fin de sous-programme (fonction C) juste avant l'instruction rts (ou rte)

mode d'adr. desti. possible

seuls *ai* possibles

syntaxe générale: **unlk ai**

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code													num. reg. *ai*		
0	1	0	0	1	1	1	0	0	1	0	1	1	-	-	-

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est modifié par l'exécution de cette instruction

exemple de mise en oeuvre: (voir exemple donné pour l'instruction link)
« compiler la fin du programme C donné à l'exemple de l'instruction link :
Si sp == a7 initial = 0x0xfffffffe
Ci dessous état de la pile au sein de l'exécution de la fonction.

```
main()
{
…..
}  ← traduction de cette accolade                                    …

        …               ;corps de la fonction
        unlk    a6      ;sp ← a6 = 0xfffffffc, puis  a6 ← (sp) = ex a6 de la fonc. appelante
                        ;sp ← sp + 4 = 0x10000000
        rts             ;pc ← (sp) = adresse de retour ds la fonction appelante
                        ;sp ← sp + 4
        …
```

5.8.2.57 **WDDATA**

WDDATA.b/w/l écriture de données émises (série) sur la broche dso du module de mise au point « debug »

Les processeurs Coldfire sont dotés de broches dites de mise au point (dsclk,dsi,dso). La broche dso est une sortie série sur laquelle le microprocesseur, via un module interne (debug), présente en série un octet, un mot ou un long (fonc. argument taille) lu à l'adresse définie par l'argument de l'instruction WDDATA
Cette émission série est réalisée sans perturber le fonctionnement « normal » du processeur. On parle de mise au point en temps réel.

L'exécution de l'instruction WDDATA est associé à la génération d'un code (0x04) , comme pour l'exécution de l'instruction PULSE

modes d'adressage possibles de l'argument <@>

tous les modes sont possibles sauf : *di*, *ai*, *dep(pc)*, *dep(pc,xi)*, et # (immédiat) .

syntaxe générale: **wddata.taille** <@>

format du code opération :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
code								*taille*		*mode d'adressage*					
0	1	0	0	1	0	0	0	-	-	code mode			num. reg. ou code		

voir tableau modes d'adressage page 61 : taille 00 ➔ octet, 01 ➔ mot, 10 ➔ long

influence sur bits d'états :

X	N	Z	V	C
-	-	-	-	-

Aucun bit d'état n'est modifié par l'exécution de cette instruction

exemple de mise en oeuvre:
Provoquer la sérialisation du long d'adresse 4(a2) sur la broche dso via l'usage de l'instruction wddata.
Hypothèse, a2 = 0x20000000

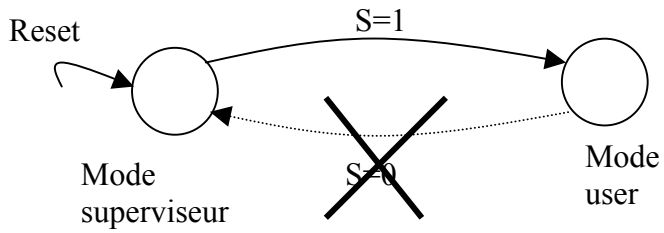
```
…                ;corps du programme
wddata.l4(a2)    ;émission du code 0x04 sur les broches pst et début de
                 ;l'émission série (au rythme du signal dsclk) du long d'adresse
                 ;4(a2) , soit 0x20000004, sur la broche dso.
…                ;suite du programme
```

5.8.3 Jeu d'instructions - Les instructions privilégiées.

Les instructions décrites ci-dessous dites privilégiées ne peuvent être exécutées qu'en mode superviseur. Le processeur est dit en mode superviseur lorsque le bit S de son registre d'état est à 1 si ce n'est pas le cas, si le bit S a été mis à 0, il est en mode utilisateur les instructions « privilègiées » ne sont alors plus exécutables.

Si le processeur rencontre une instruction privilégiée en mode utilisateur, celle-ci n'est pas exécutée et le microprocesseur déroute son activité pour entamer une procédure d'exception nommée, viol de privilège. Ce cas particulier est exposé plus loin.

Suite à un reset (ou une mise sous tension) le microprocesseur est en mode superviseur. Tout le jeu d'instruction est alors possible. Si le bit S est mis à 0 par l'exécution d'une instruction telle par exemple que move.w #0,SR (cette instruction est elle-même une instruction privilégiée, alors le microprocesseur « bascule » en mode dit user (utilisateur). Le retour en mode superviseur est «normalement » impossible. D'où le diagramme d'état suivant :



L'existence de ces 2 modes n'a d'intérêt que dans le cas ou le microprocesseur est retenu pour être l'unité centrale d'une machine type serveur, ou station de travail, ou ordinateur personnel (mais néanmoins multi-utilisateur).
Pour le cas très particulier du Coldfire, c'est son ancêtre, la famille 68000, qui elle équipait effectivement les ordinateurs Apple, qui par « héritage » d'architecture lui confère ce mécanisme de modes. Pour ses applications les plus courantes de système enfoui, ce mécanisme de mode ne présente aucun intérêt. Toutefois l'étude des mécanismes d'exceptions partiellement attachés à ces notions de modes est développé ici, car interressante dans le contexte plus large des processeurs tels que le Pentium ou le Power PC.

Les instructions privilègiées apparaissent en bistre dans le tableau page 56 sont :

CPUSHL	(ai)		"vide" et invalide un cache	gestion système
HALT			arrêt du micro-processeur	gestion système
MOVE depuis SR	sr,di	.w	consultation du registre d'état	gestion système
MOVE vers SR	di,sr	.w	initialisation du registre d'état	gestion système
MOVEC	di,rc	.l	gestion des registres de contrôle	gestion système
RTE			retour d'exception	gestion d'exception
STOP	#		arrêt du micro-processeur	gestion système
WDEBUG	<@>	64 bits	gestion du module de mise au point intégré	gestion système

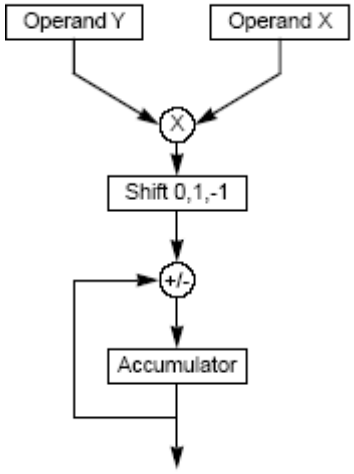
Le processus d'exécution de ces instructions sera développé ultérieurement, veuillez nous en excuser

5.8.4 Unité MAC et jeu d'instruction associé.

Le microprocesseur *Cold* FIRE 5307

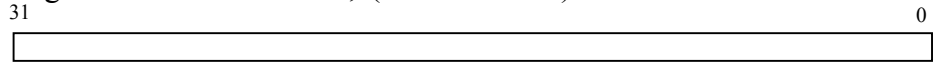
C . GUIRAUDIE

Les processeurs Coldfire 5307 et 5407 intègrent une unité de calcul dite multiplication et accumulation. Associée à 3 registres, elle donne à travers un jeu d'instructions dédié, des fonctionnalités de type traitement du signal. Bien entendu, les performances sont loin d'être celles d'un véritable DSP. L'architecture générale est la suivante.

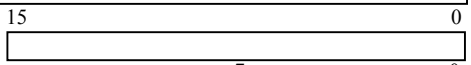


Les registres dédiés sont :

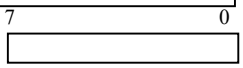
Registre accumulateur, (de résultats) **acc**.



Registre de masquage, **mask**.



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



Format des nombres :

Les nombres manipulés sont soit des entiers signés ou non signés, soit des nombres dits à virgule fixe. (le msb est le bit de signe, les suivant expriment le nombre qui suit la virgule fixe et donne ainsi un nombre qui appartient nécessairement à l'ensemble : $-1 \leq$ nombre codé $\leq 1- 2^{(N-1)}$

Suivant le format retenu, mot (16 bits) ou long (32 bits)
Le nombre le plus négatif est -1 .
Il est représenté respectivement par les valeurs 0x8000 et 0x 80000000
Le nombre le plus positif est $1-2^{-15}$ ou $1-2^{-31}$.
Il est représenté respectivement par les valeurs 0x7FFF et 0x 7FFFFFFF

Jeu d'instructions associé :

Instruction	Mnemonic	Description
Multiply Signed	MULS <ea>y,Dx	Multiplies two signed operands yielding a signed result
Multiply Unsigned	MULU <ea>y,Dx	Multiplies two unsigned operands yielding an unsigned result
Multiply Accumulate	MAC Ry,RxSF MSAC Ry,RxSF	Multiplies two operands, then adds or subtracts the product to/from the accumulator
Multiply Accumulate with Load	MAC Ry,RxSF,Rw MSAC Ry,RxSF,Rw	Multiplies two operands, then adds or subtracts the product to/from the accumulator while loading a register with the memory operand
Load Accumulator	MOV.L {Ry,#imm},ACC	Loads the accumulator with a 32-bit operand
Store Accumulator	MOV.L ACC,Rx	Writes the contents of the accumulator to a register
Load MACSR	MOV.L {Ry,#imm},MACSR	Writes a value to the MACSR
Store MACSR	MOV.L MACSR,Rx	Write the contents of MACSR to a register
Store MACSR to CCR	MOV.L MACSR,CCR	Write the contents of MACSR to the processor's CCR register
Load MASK	MOV.L {Ry,#imm},MASK	Writes a value to MASK
Store MASK	MOV.L MASK,Rx	Writes the contents of MASK to a register