Contrôle 2 – Corrigé
Architecture des ordinateurs

Durée : 1 h 30

Exercice 1 (4 points)

Codez les instructions suivantes en langage machine 68000, **vous détaillerez les différents champs** puis vous exprimerez le résultat final sous forme **hexadécimale** en précisant **la taille des mots supplémentaires** lorsque le cas se présente.

1. `MOVE.B  -(A5),(A4)`

MOVE (*cf.* documentation ci-annexée)

0	0	SIZE		DESTINATION						SOURCE					
				REGISTER			MODE			MODE			REGISTER		
0	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**1**
MOVE		**.B**		**(A4)**						**-(A5)**					

Code machine complet en représentation hexadécimale : **`18A5`**

2. `ADDA.L  -1(A3),A2`

ADDA (*cf.* documentation ci-annexée)

1	1	0	1	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		
1	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**1**
ADDA				**A2**			**.L**			**d16(A3)**					

Information à ajouter pour la source : **`d16`** `= -1 =` **`$FFFF`**
`d16` représente un déplacement sur 16 bits signés. Il faut donc convertir `-1` sur 16 bits signés.

Code machine complet en représentation hexadécimale : **`D5EB FFFF`**

3. `MOVE.W  #34,34`

MOVE (*cf.* documentation ci-annexée)

0	0	SIZE		DESTINATION						SOURCE					
				REGISTER			MODE			MODE			REGISTER		
0	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**
MOVE		**.W**		**(xxx).L**						**#<data>**					

- Information à ajouter pour la source : **`#<data>`** `= #34 =` **`#$0022`**
  La taille de la donnée du mode d'adressage immédiat correspond à la taille de l'instruction. L'instruction possède ici l'extension `.W`. La taille de la donnée est donc de 16 bits.

- Information à ajouter pour la destination : **`(xxx).L`** `= 34 =` **`$00000022`**
  Un adressage absolu long représente une adresse sur 32 bits non signés.

Code machine complet en représentation hexadécimale : **`33FC 0022 00000022`**

4. `MOVE.L   #$51,26(A3,A1.W)`

MOVE (*cf.* documentation ci-annexée)

0	0	SIZE		DESTINATION						SOURCE					
				REGISTER			MODE			MODE			REGISTER		
0	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**0**
MOVE		**.L**		**d8(A3,Xn)**						**#<data>**					

- Information à ajouter pour la source : **`#<data>`** `= #$51 =` **`#$00000051`**
  La taille de la donnée du mode d'adressage immédiat correspond à la taille de l'instruction. L'instruction possède ici l'extension `.L`. La taille de la donnée est donc de 32 bits.

- Il y a deux informations à ajouter pour la destination : la valeur de **`d8`** et la valeur de **`Xn`**. Ces deux valeurs doivent être placées dans ce qui s'appelle le **mot d'extension**. Les 5 bits de poids fort du mot d'extension servent à identifier le registre `Xn` et les 8 bits de poids faible à contenir la valeur de `d8`. `d8` est un déplacement codé sur 8 bits signés.

  Mot d'extension du 68000 (*cf.* documentation ci-annexée)

D/A	REGISTER			W/L	0	0	0	DISPLACEMENT INTEGER							
1	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**
A1				**.W**				**d8 = 26 = $1A**							

  La représentation hexadécimale du mot d'extension est : **`901A`**

Code machine complet en représentation hexadécimale : **`27BC 00000051 901A`**

Exercice 2 (4 points)

Vous indiquerez après chaque instruction, le nouveau contenu des registres (sauf le **PC**) et/ou de la mémoire qui viennent d'être modifiés. **Vous utiliserez la représentation hexadécimale**.

Attention : La mémoire et les registres sont réinitialisés à chaque nouvelle instruction.

Valeurs initiales :

```
D0 = $FFFFFFFE  A0 = $00005000  PC = $00006000
D1 = $FFFF0005  A1 = $00005008
D2 = $FFFFF000  A2 = $00005010

$005000   54 AF 18 B9 E7 21 48 C0
$005008   C9 10 11 C8 D4 36 1F 88
$005010   13 79 01 80 42 1A 2D 48
```

1. `MOVE.W  #$27,-(A1)`

Source	Destination
#$27 **#$0027**	(A1) ($5008 - 2) **($5006)**

```
$005000  54 AF 18 B9 E7 21 00 27    A1 = $00005006
```

2. `MOVE.L  D2,4(A2,D0.L)`

Source	Destination
D2 **#$FFFFF000**	4(A2,D0.L) (A2 + D0 + 4) ($5010 + $FFFFFFFE + 4) ($5010 – 2 + 4) **($5012)**

```
$005010  13 79 FF FF F0 00 2D 48
```

3. `MOVE.B  $6006(PC,D2.L),$5010`

Source	Destination
`$6006(PC,D2.L)`	**`($5010)`**
`($6006 + D2)`	
`($6006 + $FFFFF000)`	
`($6006 - $1000)`	
`($5006)`	
`#$48`	

```
$005010  48 79 01 80 42 1A 2D 48
```

4. `MOVE.W  -1(A2,D1.W),2(A0)`

Source	Destination
`-1(A2,D1.W)`	`2(A0)`
`(A2 + D1.W - 1)`	`(A0 + 2)`
`($5010 + 5 - 1)`	`($5000 + 2)`
`($5014)`	**`($5002)`**
`#$421A`	

```
$005000  54 AF 42 1A E7 21 48 C0
```

Exercice 3 (3 points)

Donnez le résultat des additions hexadécimales suivantes, ainsi que le contenu des bits **N**, **Z**, **V** et **C** du registre d'état.

1. `$3D + $E9` **opération en .B**
   = $1**26** (le résultat sur 8 bits est **$26**)
   → N = 0, Z = 0, V = 0, C = 1

2. `$6AB4 + $3FC6` **opération en .W**
   = $**AA7A**
   → N = 1, Z = 0, V = 1, C = 0

Exercice 4 (2 points)

Réalisez le sous-programme **`Add128`** qui réalise une addition sur 128 bits en quelques lignes seulement (pas plus de cinq lignes).

Entrées : **D3:D2:D1:D0** = Entier sur 128 bits (**D0** étant les 32 bits de poids faible).
D7:D6:D5:D4 = Entier sur 128 bits (**D4** étant les 32 bits de poids faible).

Sorties : **D3:D2:D1:D0** = **D3:D2:D1:D0** + **D7:D6:D5:D4**

```
Add128  add.l   d4,d0   ; d4 + d0     → d0, retenue → X
        addx.l  d5,d1   ; d5 + d1 + X → d1, retenue → X
        addx.l  d6,d2   ; d6 + d2 + X → d2, retenue → X
        addx.l  d7,d3   ; d7 + d3 + X → d3, retenue → X
        rts
```

Exercice 5 (3 points)

Réalisez le sous-programme **`GetValue`** en fonction des entrées-sorties ci-dessous (hormis le registre de sortie, aucun registre ne sera modifié en sortie du sous-programme) :

Entrée : **D1.W** = Entier signé sur 16 bits.

Sorties : **D0.L** = 1 si **D1.W** est négatif.
D0.L = 2 si **D1.W** est nul.
D0.L = 3 dans tous les autres cas.

```
GetValue    tst.w   d1          ; Positionne les flags Z et N en fonction de D1.W.
            beq     zero        ; Si D1.W = 0 (Z = 1), saut au label zero.
            bmi     negative    ; Si D1.W < 0 (N = 1), saut au label negative.

positive    moveq.l #3,d0       ; Sortie avec D0.L = 3 (cas où D1.W > 0).
            rts

zero        moveq.l #2,d0       ; Sortie avec D0.L = 2 (cas où D1.W = 0).
            rts

negative    moveq.l #1,d0       ; Sortie avec D0.L = 1 (cas où D1.W < 0).
            rts
```

Exercice 6 (4 points)

Soit les deux instructions suivantes :

- `MOVEM.L  D2/D1/A1/A5,-(A7)`
- `MOVEM.L  (A7)+,A5/A1/D1/D2`

1. Laquelle des deux permet d'empiler les registres ?

   C'est l'instruction **`MOVEM.L D2/D1/A1/A5,-(A7)`** qui permet d'empiler les registres.

2. Dans quel ordre seront-ils empilés ?

   Les registres seront empilés dans l'ordre suivant : **A5**, **A1**, **D2**, **D1**.

3. Dans quel ordre seront-ils dépilés ?

   Les registres seront dépilés dans l'ordre suivant : **D1**, **D2**, **A1**, **A5**.

4. Choisissez la proposition exacte :
   Après l'exécution d'une instruction `RTS`, le pointeur de pile est :
   - ~~incrémenté de deux~~ ;
   - ~~décrémenté de deux~~ ;
   - **incrémenté de quatre** ;
   - ~~décrémenté de quatre~~ **;**
   - ~~inchangé~~.

5. Si un programmeur commet l'erreur d'utiliser une instruction `JMP` à la place d'une instruction `JSR`, quel problème cela peut-il poser ?

   L'instruction `JSR` empile une adresse de retour puis saute à un sous-programme. Le sous-programme doit alors se terminer par une instruction `RTS` qui dépile l'adresse de retour et la place dans le registre **PC** (*Program Counter*).

   L'instruction `JMP` n'empile aucune adresse de retour. Si on l'utilise pour appeler un sous-programme, elle sautera directement à ce dernier. Le sous-programme n'aura alors aucun moyen de connaître son adresse de retour. À la sortie du sous-programme, c'est-à-dire au moment de l'exécution du `RTS`, la valeur qui sera dépilée et placée dans le registre **PC** ne sera pas l'adresse de retour. Par conséquent, le programme continuera son exécution à une adresse incorrecte.

MOVE Move Data from Source to Destination (M68000 Family) MOVE

Operation: Source → Destination

Assembler Syntax: MOVE < ea > , < ea >

Attributes: Size = (Byte, Word, Long)

Description: Moves the data at the source to the destination location and sets the condition codes according to the data. The size of the operation may be specified as byte, word, or long. Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X — Not affected.
N — Set if the result is negative; cleared otherwise.
Z — Set if the result is zero; cleared otherwise.
V — Always cleared.
C — Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	SIZE		DESTINATION						SOURCE					
				REGISTER			MODE			MODE			REGISTER		

Instruction Fields:

Size field—Specifies the size of the operand to be moved.
01 — Byte operation
11 — Word operation
10 — Long operation

MOVE Move Data from Source to Destination (M68000 Family) MOVE

Destination Effective Address field—Specifies the destination location. Only data alterable addressing modes can be used as listed in the following tables:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	—	—
(An) +	011	reg. number:An			
– (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	—	—
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	—	—

MC68020, MC68030, and MC68040 only

(bd,An,Xn)*	110	reg. number:An	(bd,PC,Xn)*	—	—
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	—	—
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	—	—

*Can be used with CPU32.

Source Effective Address field—Specifies the source operand. All addressing modes can be used as listed in the following tables:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An	001	reg. number:An	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	111	100
(An) +	011	reg. number:An			
– (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011

MC68020, MC68030, and MC68040 only

(bd,An,Xn)**	110	reg. number:An	(bd,PC,Xn)**	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

*For byte size operation, address register direct is not allowed.
**Can be used with CPU32.

NOTE

Most assemblers use MOVEA when the destination is an address register.

MOVEQ can be used to move an immediate 8-bit value to a data register.

ADDA — Add Address — ADDA

(M68000 Family)

Operation: Source + Destination $\rightarrow$ Destination

Assembler Syntax: ADDA < ea > , An

Attributes: Size = (Word, Long)

Description: Adds the source operand to the destination address register and stores the result in the address register. The size of the operation may be specified as word or long. The entire destination address register is used regardless of the operation size.

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										EFFECTIVE ADDRESS					
1	1	0	1	REGISTER			OPMODE			MODE			REGISTER		

Instruction Fields:

Register field—Specifies any of the eight address registers. This is always the destination.

Opmode field—Specifies the size of the operation.
011— Word operation; the source operand is sign-extended to a long operand and the operation is performed on the address register using all 32 bits.
111— Long operation.

Effective Address field—Specifies the source operand. All addressing modes can be used as listed in the following tables:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn	000	reg. number:Dn	(xxx).W	111	000
An	001	reg. number:An	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	111	100
(An) +	011	reg. number:An			
– (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011

MC68020, MC68030, and MC68040 only

(bd,An,Xn)*	110	reg. number:An	(bd,PC,Xn)*	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

*Can be used with CPU32

M68000 FAMILY PROGRAMMER'S REFERENCE MANUAL MOTOROLA

BRIEF EXTENSION WORD FORMAT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	REGISTER			W/L	0	0	0	DISPLACEMENT INTEGER							

(a) MC68000, MC68008, and MC68010

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	REGISTER			W/L	SCALE		0	DISPLACEMENT INTEGER							

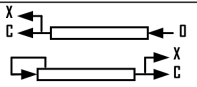
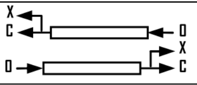
(b) CPU32, MC68020, MC68030, and MC68040

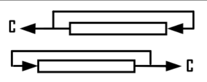
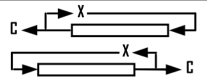
Table 2-1. Instruction Word Format Field Definitions

Field	Definition
Instruction	
Mode	Addressing Mode
Register	General Register Number
Extensions	
D/A	Index Register Type 0 = Dn 1 = An
W/L	Word/Long-Word Index Size 0 = Sign-Extended Word 1 = Long Word
Scale	Scale Factor 00 = 1 01 = 2 10 = 4 11 = 8
BS	Base Register Suppress 0 = Base Register Added 1 = Base Register Suppressed
IS	Index Suppress 0 = Evaluate and Add Index Operand 1 = Suppress Index Operand
BD SIZE	Base Displacement Size 00 = Reserved 01 = Null Displacement 10 = Word Displacement 11 = Long Displacement
I/IS	Index/Indirect Selection Indirect and Indexing Operand Determined in Conjunction with Bit 6, Index Suppress

For effective addresses that use a full extension word format, the index suppress (IS) bit and the index/indirect selection (I/IS) field determine the type of indexing and indirect action. Table 2-2 lists the index and indirect operations corresponding to all combinations of IS and I/IS values.

EASy68K Quick Reference v2.1 www.easy68k.com Copyright © 2004-2009 By: Chuck Kelly

Opcode	Size	Operand	CCR	Effective Address s=source, d=destination, e=either, i=displacement												Operation	Description
	BWL	s,d	XNZVC	Dn	An	(An)	(An)+	-(An)	(i,An)	(i,An,Rn)	abs.W	abs.L	(i,PC)	(i,PC,Rn)	#n		
ABCD	B	Dy,Dx -(Ay),-(Ax)	*U*U*	e -	- -	- -	- -	- e	- -	- -	- -	- -	- -	- -	- -	$Dy_{10} + Dx_{10} + X \rightarrow Dx_{10}$ $-(Ay)_{10} + -(Ax)_{10} + X \rightarrow -(Ax)_{10}$	BCD destination + BCD source + eXtend Z cleared if result not 0 unchanged otherwise
ADD [4]	BWL	s,Dn Dn,d	*****	e e	s d[4]	s d	s d	s d	s d	s d	s d	s d	s -	s -	s[4] -	s + Dn → Dn Dn + d → d	Add binary (ADDI or ADDQ is used when source is #n. Prevent ADDQ with #n.L)
ADDA [4]	WL	s,An	-----	s	e	s	s	s	s	s	s	s	s	s	s	s + An → An	Add address (.W sign-extended to .L)
ADDI [4]	BWL	#n,d	*****	d	-	d	d	d	d	d	d	d	-	-	s	#n + d → d	Add immediate to destination
ADDQ [4]	BWL	#n,d	*****	d	d	d	d	d	d	d	d	d	-	-	s	#n + d → d	Add quick immediate (#n range: 1 to 8)
ADDX	BWL	Dy,Dx -(Ay),-(Ax)	*****	e -	- -	- -	- -	- e	- -	- -	- -	- -	- -	- -	- -	Dy + Dx + X → Dx -(Ay) + -(Ax) + X → -(Ax)	Add source and eXtend bit to destination
AND [4]	BWL	s,Dn Dn,d	-**00	e e	- -	s d	s d	s d	s d	s d	s d	s d	s -	s -	s[4] -	s AND Dn → Dn Dn AND d → d	Logical AND source to destination (ANDI is used when source is #n)
ANDI [4]	BWL	#n,d	-**00	d	-	d	d	d	d	d	d	d	-	-	s	#n AND d → d	Logical AND immediate to destination
ANDI [4]	B	#n,CCR	≡≡≡≡≡	-	-	-	-	-	-	-	-	-	-	-	s	#n AND CCR → CCR	Logical AND immediate to CCR
ANDI [4]	W	#n,SR	≡≡≡≡≡	-	-	-	-	-	-	-	-	-	-	-	s	#n AND SR → SR	Logical AND immediate to SR (Privileged)
ASL ASR	BWL W	Dx,Dy #n,Dy d	*****	e d -	- - -	- - d	- - d	- - d	- - d	- - d	- - d	- - d	- - -	- - -	- s -		Arithmetic shift Dy by Dx bits left/right Arithmetic shift Dy #n bits L/R (#n: 1 to 8) Arithmetic shift ds 1 bit left/right (.W only)
Bcc	BW[3]	address[2]	-----	-	-	-	-	-	-	-	-	-	-	-	-	if cc true then address → PC	Branch conditionally (cc table on back) (8 or 16-bit ± offset to address)
BCHG	B L	Dn,d #n,d	--*--	e[1] d[1]	- -	d d	d d	d d	d d	d d	d d	d d	- -	- -	- s	NOT(bit number of d) → Z NOT(bit n of d)→ bit n of d	Set Z with state of specified bit in d then invert the bit in d
BCLR	B L	Dn,d #n,d	--*--	e[1] d[1]	- -	d d	d d	d d	d d	d d	d d	d d	- -	- -	- s	NOT(bit number of d) → Z 0 → bit number of d	Set Z with state of specified bit in d then clear the bit in d
BFCHG	[5]	d{o:w}	-**00	d	-	d	-	-	d	d	d	d	-	-	-	NOT bit field of d	Complement the bit field at destination
BFCLR	[5]	d{o:w}	-**00	d	-	d	-	-	d	d	d	d	-	-	-	0 → bit field of d	Clear the bit field at destination
BFEXTS	[5]	s{o:w},Dn	-**00	d	-	s	-	-	s	s	s	s	s	s	-	bit field of s extend 32 → Dn	Dn = bit field of s sign extended to 32 bits
BFEXTU	[5]	s{o:w},Dn	-**00	d	-	s	-	-	s	s	s	s	s	s	-	bit field of s unsigned → Dn	Dn = bit field of s zero extended to 32 bits
BFFFO	[5]	s{o:w},Dn	-**00	d	-	s	-	-	s	s	s	s	s	s	-	bit number of 1st 1 → Dn	Dn = bit position of 1st 1 or offset + width
BFINS	[5]	Dn,s{o:w}	-**00	s	-	d	-	-	d	d	d	d	-	-	-	low bits Dn → bit field at d	Insert low bits of Dn to bit field at d
BFSET	[5]	d{o:w}	-**00	d	-	d	-	-	d	d	d	d	-	-	-	1 → bit field of d	Set all bits in bit field of destination
BFTST	[5]	d{o:w}	-**00	d	-	d	-	-	d	d	d	d	d	d	-	set CCR with bit field of d	N = high bit of bit field, Z set if all bits 0
BRA	BW[3]	address[2]	-----	-	-	-	-	-	-	-	-	-	-	-	-	address → PC	Branch always (8 or 16-bit ± offset to addr)
BSET	B L	Dn,d #n,d	--*--	e[1] d[1]	- -	d d	d d	d d	d d	d d	d d	d d	- -	- -	- s	NOT(bit n of d) → Z 1 → bit n of d	Set Z with state of specified bit in d then set the bit in d
BSR	BW[3]	address[2]	-----	-	-	-	-	-	-	-	-	-	-	-	-	PC → -(SP); address → PC	Branch to subroutine (8 or 16-bit ± offset)
BTST	B L	Dn,d #n,d	--*--	e[1] d[1]	- -	d d	d d	d d	d d	d d	d d	d d	d d	d d	- s	NOT(bit Dn of d) → Z NOT(bit #n of d) → Z	Set Z with state of specified bit in d Leave the bit in d unchanged
CHK	W	s,Dn	-*UUU	e	-	s	s	s	s	s	s	s	s	s	s	if Dn<0 or Dn>s then TRAP	Compare Dn with 0 and upper bound [s]
CLR	BWL	d	-0100	d	-	d	d	d	d	d	d	d	-	-	-	0 → d	Clear destination to zero
CMP [4]	BWL	s,Dn	-****	e	s[4]	s	s	s	s	s	s	s	s	s	s[4]	set CCR with Dn – s	Compare Dn to source
CMPA [4]	WL	s,An	-****	s	e	s	s	s	s	s	s	s	s	s	s	set CCR with An – s	Compare An to source
CMPI [4]	BWL	#n,d	-****	d	-	d	d	d	d	d	d	d	-	-	s	set CCR with d - #n	Compare destination to #n
CMPM [4]	BWL	(Ay)+,(Ax)+	-****	-	-	-	e	-	-	-	-	-	-	-	-	set CCR with (Ax) - (Ay)	Compare (Ax) to (Ay); Increment Ax and Ay
DBcc	W	Dn,addres[2]	-----	-	-	-	-	-	-	-	-	-	-	-	-	if cc false then { Dn-1 → Dn if Dn <> -1 then addr →PC }	Test condition, decrement and branch (16-bit ± offset to address)
DIVS	W	s,Dn	-***0	e	-	s	s	s	s	s	s	s	s	s	s	±32bit Dn / ±16bit s → ±Dn	Dn= [16-bit remainder, 16-bit quotient]
DIVU	W	s,Dn	-***0	e	-	s	s	s	s	s	s	s	s	s	s	32bit Dn / 16bit s → Dn	Dn= [16-bit remainder, 16-bit quotient]
EOR [4]	BWL	Dn,d	-**00	e	-	d	d	d	d	d	d	d	-	-	s[4]	Dn XOR d → d	Logical exclusive OR Dn to destination
EORI [4]	BWL	#n,d	-**00	d	-	d	d	d	d	d	d	d	-	-	s	#n XOR d → d	Logical exclusive OR #n to destination
EORI [4]	B	#n,CCR	≡≡≡≡≡	-	-	-	-	-	-	-	-	-	-	-	s	#n XOR CCR → CCR	Logical exclusive OR #n to CCR
EORI [4]	W	#n,SR	≡≡≡≡≡	-	-	-	-	-	-	-	-	-	-	-	s	#n XOR SR → SR	Logical exclusive OR #n to SR (Privileged)
EXG	L	Rx,Ry	-----	e	e	-	-	-	-	-	-	-	-	-	-	register ←→ register	Exchange registers (32-bit only)
EXT	WL	Dn	-**00	d	-	-	-	-	-	-	-	-	-	-	-	Dn.B → Dn.W | Dn.W → Dn.L	Sign extend (change .B to .W or .W to .L)
ILLEGAL			-----	-	-	-	-	-	-	-	-	-	-	-	-	PC→-(SSP); SR→-(SSP)	Generate Illegal Instruction exception
JMP		d	-----	-	-	d	-	-	d	d	d	d	d	d	-	↑d → PC	Jump to effective address of destination
JSR		d	-----	-	-	d	-	-	d	d	d	d	d	d	-	PC → -(SP); ↑d → PC	push PC, jump to subroutine at address d
LEA	L	s,An	-----	-	e	s	-	-	s	s	s	s	s	s	-	↑s → An	Load effective address of s to An
LINK		An,#n	-----	-	-	-	-	-	-	-	-	-	-	-	-	An → -(SP); SP → An; SP + #n → SP	Create local workspace on stack (negative n to allocate space)
LSL LSR	BWL W	Dx,Dy #n,Dy d	***0*	e d -	- - -	- - d	- - d	- - d	- - d	- - d	- - d	- - d	- - -	- - -	- s -		Logical shift Dy, Dx bits left/right Logical shift Dy, #n bits L/R (#n: 1 to 8) Logical shift d 1 bit left/right (.W only)
MOVE [4]	BWL	s,d	-**00	e	s[4]	e	e	e	e	e	e	e	s	s	s[4]	s → d	Move data from source to destination
MOVE	W	s,CCR	≡≡≡≡≡	s	-	s	s	s	s	s	s	s	s	s	s	s → CCR	Move source to Condition Code Register
MOVE	W	s,SR	≡≡≡≡≡	s	-	s	s	s	s	s	s	s	s	s	s	s → SR	Move source to Status Register (Privileged)
MOVE	W	SR,d	-----	d	-	d	d	d	d	d	d	d	-	-	-	SR → d	Move Status Register to destination
	BWL	s,d	XNZVC	Dn	An	(An)	(An)+	-(An)	(i,An)	(i,An,Rn)	abs.W	abs.L	(i,PC)	(i,PC,Rn)	#n		

Opcode	Size	Operand	CCR	Effective Address s=source, d=destination, e=either, i=displacement												Operation	Description
	BWL	s,d	XNZVC	Dn	An	(An)	(An)+	-(An)	(i,An)	(i,An,Rn)	abs.W	abs.L	(i,PC)	(i,PC,Rn)	#n		
MOVE	L	USP,An An,USP	–––––	- -	d s	- -	- -	- -	- -	- -	- -	- -	- -	- -	- -	USP → An An → USP	Move User Stack Pointer to An (Privileged) Move An to User Stack Pointer (Privileged)
MOVEA[4]	WL	s,An	–––––	s	e	s	s	s	s	s	s	s	s	s	s	s → An	Move source to An (MOVE s,An use MOVEA)
MOVEM[4]	WL	Rn-Rn,d s,Rn-Rn	–––––	- -	- -	d s	- s	d -	d s	d s	d s	d s	- s	- s	- -	Registers → d s → Registers	Move specified registers to/from memory (.W source is sign-extended to .L for Rn)
MOVEP	WL	Dn,(i,An) (i,An),Dn	–––––	s d	- -	- -	- -	- -	d s	- -	- -	- -	- -	- -	- -	Dn → (i,An)...(i+2,An)...(i+4,A. (i,An) → Dn...(i+2,An)...(i+4,A.	Move Dn to/from alternate memory bytes (Access only even or odd addresses)
MOVEQ[4]	L	#n,Dn	–**00	d	-	-	-	-	-	-	-	-	-	-	s	#n → Dn	Move sign extended 8-bit #n to Dn
MULS	W	s,Dn	–**00	e	-	s	s	s	s	s	s	s	s	s	s	±16bit s * ±16bit Dn → ±Dn	Multiply signed 16-bit; result: signed 32-bit
MULU	W	s,Dn	–**00	e	-	s	s	s	s	s	s	s	s	s	s	16bit s * 16bit Dn → Dn	Multiply unsig'd 16-bit; result: unsig'd 32-bit
NBCD	B	d	*U*U*	d	-	d	d	d	d	d	d	d	-	-	-	0 - d_{10} - X → d	Negate BCD with eXtend, BCD result
NEG	BWL	d	*****	d	-	d	d	d	d	d	d	d	-	-	-	0 - d → d	Negate destination (2's complement)
NEGX	BWL	d	*****	d	-	d	d	d	d	d	d	d	-	-	-	0 - d - X → d	Negate destination with eXtend
NOP			–––––	-	-	-	-	-	-	-	-	-	-	-	-	None	No operation occurs
NOT	BWL	d	–**00	d	-	d	d	d	d	d	d	d	-	-	-	NOT(d) → d	Logical NOT destination (1's complement)
OR [4]	BWL	s,Dn Dn,d	–**00	e e	- -	s d	s d	s d	s d	s d	s d	s d	s -	s -	s[4] -	s OR Dn → Dn Dn OR d → d	Logical OR (ORI is used when source is #n)
ORI [4]	BWL	#n,d	–**00	d	-	d	d	d	d	d	d	d	-	-	s	#n OR d → d	Logical OR #n to destination
ORI [4]	B	#n,CCR	≡≡≡≡≡	-	-	-	-	-	-	-	-	-	-	-	s	#n OR CCR → CCR	Logical OR #n to CCR
ORI [4]	W	#n,SR	≡≡≡≡≡	-	-	-	-	-	-	-	-	-	-	-	s	#n OR SR → SR	Logical OR #n to SR (Privileged)
PEA	L	s	–––––	-	-	s	-	-	s	s	s	s	s	s	-	↑s → -(SP)	Push effective address of s onto stack
RESET			–––––	-	-	-	-	-	-	-	-	-	-	-	-	Assert RESET Line	Issue a hardware RESET (Privileged)
ROL ROR	BWL W	Dx,Dy #n,Dy d	–**0*	e d -	- - -	- - d	- - d	- - d	- - d	- - d	- - d	- - d	- - -	- - -	- s -		Rotate Dy, Dx bits left/right (without X) Rotate Dy, #n bits left/right (#n: 1 to 8) Rotate d 1-bit left/right (.W only)
ROXL ROXR	BWL W	Dx,Dy #n,Dy d	***0*	e d -	- - -	- - d	- - d	- - d	- - d	- - d	- - d	- - d	- - -	- - -	- s -		Rotate Dy, Dx bits L/R, X used then updated Rotate Dy, #n bits left/right (#n: 1 to 8) Rotate destination 1-bit left/right (.W only)
RTE			≡≡≡≡≡	-	-	-	-	-	-	-	-	-	-	-	-	(SP)+ → SR; (SP)+ → PC	Return from exception (Privileged)
RTR			≡≡≡≡≡	-	-	-	-	-	-	-	-	-	-	-	-	(SP)+ → CCR, (SP)+ → PC	Return from subroutine and restore CCR
RTS			–––––	-	-	-	-	-	-	-	-	-	-	-	-	(SP)+ → PC	Return from subroutine
SBCD	B	Dy,Dx -(Ay),-(Ax)	*U*U*	e -	- -	- -	- -	- e	- -	- -	- -	- -	- -	- -	- -	Dx_{10} - Dy_{10} - X → Dx_{10} $-(Ax)_{10}$ - $-(Ay)_{10}$ – X → $-(Ax)_{10}$	BCD destination – BCD source – eXtend Z cleared if result not 0 unchanged otherwise
Scc	B	d	–––––	d	-	d	d	d	d	d	d	d	-	-	-	If cc is true then 1's → d else 0's → d	If cc true then d.B = 11111111 else d.B = 00000000
STOP		#n	≡≡≡≡≡	-	-	-	-	-	-	-	-	-	-	-	s	#n → SR; STOP	Move #n to SR, stop processor (Privileged)
SUB [4]	BWL	s,Dn Dn,d	*****	e e	s d[4]	s d	s d	s d	s d	s d	s d	s d	s -	s -	s[4] -	Dn - s → Dn d - Dn → d	Subtract binary (SUBI or SUBQ used when source is #n. Prevent SUBQ with #n.L)
SUBA [4]	WL	s,An	–––––	s	e	s	s	s	s	s	s	s	s	s	s	An - s → An	Subtract address (.W sign-extended to .L)
SUBI [4]	BWL	#n,d	*****	d	-	d	d	d	d	d	d	d	-	-	s	d - #n → d	Subtract immediate from destination
SUBQ [4]	BWL	#n,d	*****	d	d	d	d	d	d	d	d	d	-	-	s	d - #n → d	Subtract quick immediate (#n range: 1 to 8)
SUBX	BWL	Dy,Dx -(Ay),-(Ax)	*****	e -	- -	- -	- -	- e	- -	- -	- -	- -	- -	- -	- -	Dx - Dy - X → Dx -(Ax) - -(Ay) – X → -(Ax)	Subtract source and eXtend bit from destination
SWAP	W	Dn	–**00	d	-	-	-	-	-	-	-	-	-	-	-	bits[31:16]←→bits[15:0]	Exchange the 16-bit halves of Dn
TAS	B	d	–**00	d	-	d	d	d	d	d	d	d	-	-	-	test d→CCR; 1 →bit7 of d	N and Z set to reflect d, bit7 of d set to 1
TRAP		#n	–––––	-	-	-	-	-	-	-	-	-	-	-	s	PC→-(SSP);SR→-(SSP); (vector table entry) → PC	Push PC and SR, PC set by vector table #n (#n range: 0 to 15)
TRAPV			–––––	-	-	-	-	-	-	-	-	-	-	-	-	If V then TRAP #7	If overflow, execute an Overflow TRAP
TST	BWL	d	–**00	d	-	d	d	d	d	d	d	d	-	-	-	test d → CCR	N and Z set to reflect destination
UNLK		An	–––––	-	d	-	-	-	-	-	-	-	-	-	-	An → SP; (SP)+ → An	Remove local workspace from stack
	BWL	s,d	XNZVC	Dn	An	(An)	(An)+	-(An)	(i,An)	(i,An,Rn)	abs.W	abs.L	(i,PC)	(i,PC,Rn)	#n		

Condition Tests (**+** OR, **!** NOT, ⊕ XOR; [u] Unsigned, [a] Alternate cc)

cc	Condition	Test	cc	Condition	Test
T	true	1	VC	overflow clear	!V
F	false	0	VS	overflow set	V
HI[u]	higher than	!(C + Z)	PL	plus	!N
LS[u]	lower or same	C + Z	MI	minus	N
HS[u], CC[a]	higher or same	!C	GE	greater or equal	!(N ⊕ V)
LO[u], CS[a]	lower than	C	LT	less than	(N ⊕ V)
NE	not equal	!Z	GT	greater than	![(N ⊕ V) + Z]
EQ	equal	Z	LE	less or equal	(N ⊕ V) + Z

An	Address register (16/32-bit, n=0-7)	**s**	Source,
Dn	Data register (8/16/32-bit, n=0-7)	**d**	Destination
Rn	any data or address register	**e**	Either source or destination
BCD	Binary Coded Decimal	**i**	Displacement
PC	Program Counter (24-bit)	**↑**	Effective address
#n	Immediate data	**{o:w}**	offset:width of bit field
SP	Active Stack Pointer (same as A7)	**SSP**	Supervisor Stack Pointer (32-bit)

USP User Stack Pointer (32-bit)

SR Status Register (16-bit)
CCR Condition Code Register (lower 8-bits of SR)
N negative, **Z** zero, **V** overflow, **C** carry, **X** extend
***** set by operation's result, ≡ set directly
- not affected, **0** cleared, **1** set, **U** undefined

[1] Long only; all others are byte only
[2] Assembler calculates offset
[3] Branch sizes: **.B** or **.S** -128 to +127 bytes, **.W** or **.L** -32768 to +32767 bytes
[4] Assembler automatically uses A, I, Q or M form if possible. Use #n.L to prevent Quick optimization
[5] Bit field determines size. Not supported by 68000. EASy68K hybrid form of 68020 instruction

Commonly Used Simulator Input/Output Tasks TRAP #15 is used to run simulator tasks. Place the task number in register D0. See Help for a complete description of available tasks. (cstring is null terminated)

0	Display n characters of string at (A1), n=D1.W (stops on NULL or max 255) with CR,LF	**1**	Display n characters of string at (A1), n=D1.W (stops on NULL or max 255) without CR,LF	**2**	Read characters from keyboard. Store at (A1). Null terminated. D1.W = length (max 80)	**3**	Display D1.L as signed decimal number
4	Read number from keyboard into D1.L	**5**	Read single character from keyboard in D1.B	**6**	Display D1.B as ASCII character	**7**	Set D1.B to 1 if keyboard input pending else set to 0
8	time in 1/100 second since midnight →D1.L	**9**	Terminate the program. (Halts the simulator)	**10**	Print cstring at (A1) on default printer.	**11**	Position cursor at row,col D1.W=ccrr, $FF00 clears
13	Display cstring at (A1) with CR,LF	**14**	Display cstring at (A1) without CR,LF	**15**	Display unsigned number in D1.L in D2.B base	**17**	Display cstring at (A1) , then display number in D1.L
18	Display cstring at (A1), read number into D1.L	**19**	Return state of keys or scan code. See help	**20**	Display ± number in D1.L, field D2.B columns wide	**21**	Set font properties. See help for details