

Architecture des ordinateurs

Partiel 2 – Corrigé

Durée : 1h30

Exercice 1 (9 points)

Toutes les questions de cet exercice sont indépendantes. À l'exception des registres utilisés pour renvoyer une valeur de sortie, **aucun registre ne devra être modifié en sortie de vos sous-programmes**. Une chaîne de caractères se termine toujours par un caractère nul (la valeur zéro). On suppose pour tout l'exercice que les chaînes ne sont jamais nulles (elles possèdent au moins un caractère non nul).

1. Réalisez le sous-programme `IsNumber` qui détermine si une chaîne de caractères contient uniquement des chiffres.

Entrée : `A0.L` pointe sur une chaîne non nulle.

Sortie : Si la chaîne contient uniquement des chiffres, `D0.L` renvoie 0.

Si la chaîne contient d'autres caractères que des chiffres, `D0.L` renvoie 1.

```
IsNumber      ; Sauvegarde les registres modifiés.
               move.l  a0,-(a7)

\loop         ; Copie le caractère à tester dans D0
               ; (et fait pointer A0 sur le caractère suivant).
               ; Si fin chaîne, la chaîne ne contient que des chiffres.
               ; (Jamais de chaîne nulle.)
               move.b   (a0)+,d0
               beq      \number

               ; Comparaison au caractère '0'.
               ; Si inférieur, ce n'est pas un chiffre (on sort de la boucle).
               cmp.b    #'0',d0
               blt.s    \notANumber

               ; Comparaison au caractère '9'.
               ; Si inférieur ou égal, c'est un chiffre (on reboucle).
               ; Si supérieur, ce n'est pas un chiffre (on sort de la boucle).
               cmp.b    #'9',d0
               ble.s    \loop

\notANumber   ; Le dernier caractère testé n'est pas un chiffre.
               ; On renvoie la valeur 1 dans D0 (sur 32 bits).
               moveq.l  #1,d0
               bra.s    \quit

\number       ; La chaîne contient uniquement des chiffres.
               ; On renvoie la valeur 0 dans D0 (sur 32 bits).
               moveq.l  #0,d0

\quit         ; Restaure les registres et sortie.
               movea.l  (a7)+,a0
               rts
```

2. Réalisez le sous-programme **GetSum** qui additionne tous les chiffres présents dans une chaîne de caractères.

Entrée : **A0.L** pointe sur une chaîne non nulle contenant uniquement des chiffres.

Sortie : **D0.L** renvoie la somme de tous les chiffres de la chaîne.

Exemple :

A0 →

'7'	'0'	'4'	'8'	'9'	'4'	'2'	'0'	'3'	0
-----	-----	-----	-----	-----	-----	-----	-----	-----	---

D0 doit renvoyer la valeur 37 ($37 = 7 + 0 + 4 + 8 + 9 + 4 + 2 + 0 + 3$).

Indications :

Réalisez une boucle qui pour chaque caractère de la chaîne :

- Copie le caractère en cours dans le registre **D1.B**.
- Convertit le caractère en une valeur numérique.
- Ajoute la valeur numérique du caractère au registre **D0.L**.

```
GetSum          ; Sauvegarde les registres modifiés.
                movem.l a0/d1,-(a7)

                ; D0 et D1 doivent être initialisés à 0 sur 32 bits.
                ; (D1 sera additionné à D0 sur 32 bits.)
                clr.l   d0
                clr.l   d1

\loop           ; Copie le caractère de la chaîne dans D1
                ; (et fait pointer A0 sur le caractère suivant).
                ; Si le caractère est nul, on quitte.
                move.b  (a0)+,d1
                beq.s   \quit

                ; Convertit le caractère en valeur numérique.
                sub.b   #'0',d1

                ; Ajoute cette valeur à la somme (sur 32 bits).
                add.l   d1,d0

                ; Passe au caractère suivant.
                bra.s   \loop

\quit          ; Restaure les registres et sortie.
                movem.l (a7)+,a0/d1
                rts
```

3. À l'aide des sous-programmes **IsNumber** et **GetSum**, réalisez le sous-programme **Checksum** qui renvoie la somme des chiffres d'une chaîne de caractères.

Entrée : **A0.L** pointe sur une chaîne non nulle.

Sortie : Si la chaîne contient uniquement des chiffres :

D0.L renvoie 0 et **D1.L** renvoie la somme.

Si la chaîne contient d'autres caractères que des chiffres :

D0.L renvoie 1 et **D1.L** renvoie 0.

```

Checksum      ; Si la chaîne contient d'autres caractères que des chiffres,
               ; on saute à \notANumber (avec D0 = 1).
               jsr      IsNumber
               tst.l    d0
               bne.s    \notANumber

\number        ; La chaîne contient uniquement des chiffres.
               ; On renvoie la somme dans D1, et 0 dans D0.
               jsr      GetSum
               move.l   d0,d1
               moveq.l  #0,d0
               rts

\notANumber    ; La chaîne contient d'autres caractères que des chiffres.
               ; On renvoie D0 = 1 et D1 = 0.
               moveq.l  #0,d1
               rts

```

Exercice 2 (2 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 \$19,29(A3)

MOVE (cf. [documentation ci-annexée](#))

0	0	SIZE		DESTINATION						SOURCE					
				REGISTER			MODE			MODE			REGISTER		
0	0	0	1	0	1	1	1	0	1	1	1	1	0	0	1
MOVE		.B		d16 (A3)						(xxx) .L					

Information à ajouter pour la source : **(xxx) .L = \$00000019**

Un adressage absolu long représente une adresse sur 32 bits non signés.

Information à ajouter pour la destination : **d16 = 29 = \$001D**

Code machine complet en représentation hexadécimale : **1779 00000019 001D**

2. MOVE.W #\$19,29(A4,D6.L)

MOVE (cf. [documentation ci-annexée](#))

0	0	SIZE		DESTINATION						SOURCE					
				REGISTER			MODE			MODE			REGISTER		
0	0	1	1	1	0	0	1	1	0	1	1	1	1	0	0
MOVE		.W		d8 (A4, Xn)						#<data>					

Information à ajouter pour la source : **#<data> = #\$19 = #\$0019**

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.

Il y a deux informations à ajouter pour la destination : la valeur de **d8** et la valeur de **Xn**. Ces deux informations doivent être contenues dans ce qui s'appelle le **mot d'extension**.

Mot d'extension du 68000 (cf. [documentation ci-annexée](#))

D/A	REGISTER			W/L	0	0	0	DISPLACEMENT INTEGER							
0	1	1	0	1	0	0	0	0	0	0	1	1	1	0	1
D6				.L				d8 = 29 = \$1D							

Les 5 bits de poids fort du mot d'extension servent à identifier le registre **Xn** et les 8 bits de poids faible contiennent la valeur de **d8**. **d8** est un déplacement codé sur 8 bits signés. La représentation hexadécimale du mot d'extension est : **681D**

Code machine complet en représentation hexadécimale : **39BC 0019 681D**

Exercice 3 (2 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 = \$FFFFFFFB A0 = \$00005000 PC = \$00006000
 D1 = \$FFFF0003 A1 = \$00005008
 D2 = \$FFFFE000 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 #50, -(A1)**

Source	Destination
#50	(A1)
#\$0032	(\$5008 - 2) (\$5006)

\$005000 54 AF 18 B9 E7 21 **00 32** **A1 = \$00005006**

2. `MOVE.B $5002(PC), -2(A2, D0.L)`

Source	Destination
(\$5002)	-2(A2, D0.L)
#\$18	(A2 + D0.L - 2)
	(\$5010 - 5 - 2)
	(\$5009)

\$005008 C9 18 11 C8 D4 36 1F 88

Exercice 4 (4 points)

Soit le sous-programme `select` qui utilise comme registre d'entrée `D1.B` et comme registre de sortie `D0.L`.

```
Select      tst.b    d1      ; Mise à jour de N et de Z en fonction de D1.B.
             bne.s    suite1  ; Si Z = 0 (D1.B ≠ 0), saut à suite1.
             move.l   #200,d0 ; Sinon (D1.B = 0), charge 200 dans D0.L.
             rts      ; Sortie.
suite1      bmi.s     suite3   ; Si N = 1 (D1.B < 0), saut à suite3.
             cmp.b    #$61,d1 ; Sinon D1.B est comparé à $61 ($61 = 97).
             blt.s    suite2   ; Si D1.B < $61, saut à suite2.
             move.l   #400,d0 ; Sinon D1.B ≥ $61, charge 400 dans D0.L.
             rts      ; Sortie.
suite2      move.l   #600,d0 ; D1.B < $61, charge 600 dans D0.L.
             rts      ; Sortie.
suite3      move.l   #800,d0 ; D1.B < 0, charge 800 dans D0.L
             rts      ; Sortie.
```

1. Quelle valeur renvoie `select` pour une valeur d'entrée égale à 18 ?

`select` renvoie la valeur **600**.

2. Quelle valeur renvoie `select` pour une valeur d'entrée égale à -5 ?

`select` renvoie la valeur **800**.

3. Quelle valeur renvoie `select` pour une valeur d'entrée nulle ?

`select` renvoie la valeur **200**.

4. Quelle valeur renvoie `select` pour une valeur d'entrée égale à 96 ?

`select` renvoie la valeur **600**.

Exercice 5 (3 points)

En utilisant l'extrait du [DataSheet](#) fourni en annexe, déterminez pour le **PIC 12F509** :

1. La taille d'une donnée.

La taille d'une donnée est de **8 bits**.

2. Le nombre et la taille des mots programme.

Le **PIC 12F509** possède **33 mots programme** (instructions) de **12 bits** chacun.

3. Le nombre et le(s) type(s) de mémoire. Précisez dans chacun des cas la particularité (volatile ou non), la taille de la mémoire et les informations stockées dans cette mémoire.

- **Mémoire flash**

Elle est **non volatile** et permet de stocker des **instructions codées sur 12 bits**. Sa taille est de **1024 mots de 12 bits**.

- **SRAM**

Elle est **volatile** et permet de stocker des **données**. Sa taille est de **41 octets**.

Integer Instructions

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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d ₁₆ ,An)	101	reg. number:An
(d ₈ ,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d ₁₆ ,PC)	—	—
(d ₈ ,PC,Xn)	—	—

MC68020, MC68030, and MC68040 only

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

(bd,PC,Xn)*	—	—
([bd,PC,Xn],od)	—	—
([bd,PC],Xn,od)	—	—

*Can be used with CPU32.

Integer Instructions

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

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.

2.4 BRIEF EXTENSION WORD FORMAT COMPATIBILITY

Programs can be easily transported from one member of the M68000 family to another in an upward-compatible fashion. The user object code of each early member of the family, which is upward compatible with newer members, can be executed on the newer microprocessor without change. Brief extension word formats are encoded with information that allows the CPU32, MC68020, MC68030, and MC68040 to distinguish the basic M68000 family architecture's new address extensions. Figure 2-3 illustrates these brief extension word formats. The encoding for SCALE used by the CPU32, MC68020, MC68030, and MC68040 is a compatible extension of the M68000 family architecture. A value of zero for SCALE is the same encoding for both extension words. Software that uses this encoding is compatible with all processors in the M68000 family. Both brief extension word formats do not contain the other values of SCALE. Software can be easily migrated in an upward-compatible direction, with downward support only for nonscaled addressing. If the MC68000 were to execute an instruction that encoded a scaling factor, the scaling factor would be ignored and would not access the desired memory address. The earlier microprocessors do not recognize the brief extension word formats implemented by newer processors. Although they can detect illegal instructions, they do not decode invalid encodings of the brief extension word formats as exceptions.

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

Figure 2-3. M68000 Family Brief Extension Word Formats

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.



PIC12F508/509/16F505

8/14-Pin, 8-Bit Flash Microcontrollers

Devices Included In This Data Sheet:

- PIC12F508
- PIC12F509
- PIC16F505

High-Performance RISC CPU:

- Only 33 Single-Word Instructions to Learn
- All Single-Cycle Instructions Except for Program Branches, which are Two-Cycle
- 12-Bit Wide Instructions
- 2-Level Deep Hardware Stack
- Direct, Indirect and Relative Addressing modes for Data and Instructions
- 8-Bit Wide Data Path
- 8 Special Function Hardware Registers
- Operating Speed:
 - DC – 20 MHz clock input (PIC16F505 only)
 - DC – 200 ns instruction cycle (PIC16F505 only)
 - DC – 4 MHz clock input
 - DC – 1000 ns instruction cycle

Special Microcontroller Features:

- 4 MHz Precision Internal Oscillator:
 - Factory calibrated to $\pm 1\%$
- In-Circuit Serial Programming™ (ICSP™)
- In-Circuit Debugging (ICD) Support
- Power-On Reset (POR)
- Device Reset Timer (DRT)
- Watchdog Timer (WDT) with Dedicated On-Chip RC Oscillator for Reliable Operation
- Programmable Code Protection
- Multiplexed MCLR Input Pin
- Internal Weak Pull-Ups on I/O Pins
- Power-Saving Sleep mode
- Wake-Wp from Sleep on Pin Change
- Selectable Oscillator Options:
 - INTRC: 4 MHz precision Internal oscillator
 - EXTRC: External low-cost RC oscillator
 - XT: Standard crystal/resonator
 - HS: High-speed crystal/resonator (PIC16F505 only)
 - LP: Power-saving, low-frequency crystal
 - EC: High-speed external clock input (PIC16F505 only)

Low-Power Features/CMOS Technology:

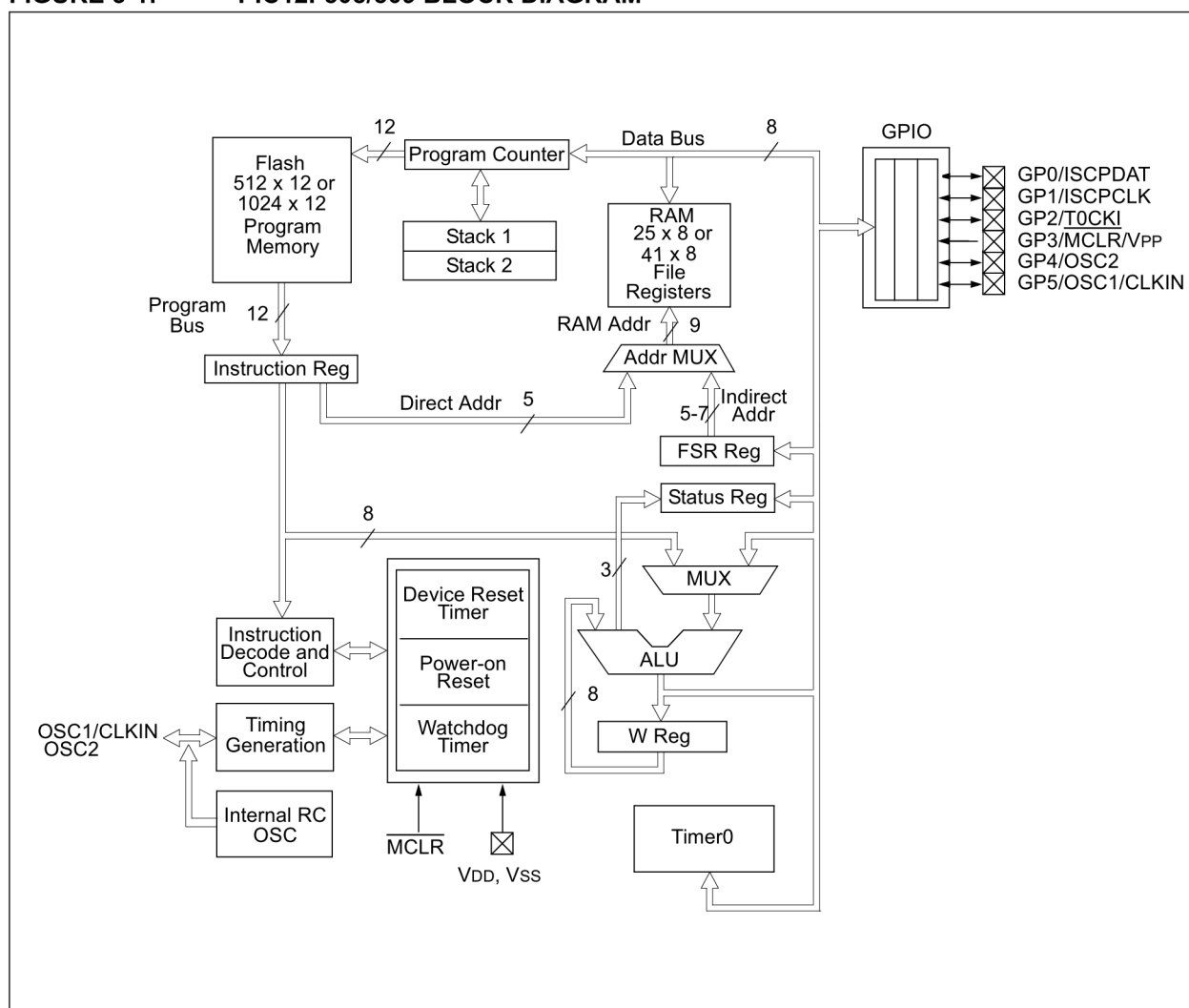
- Operating Current:
 - $< 175 \mu\text{A}$ @ 2V, 4 MHz, typical
- Standby Current:
 - 100 nA @ 2V, typical
- Low-Power, High-Speed Flash Technology:
 - 100,000 Flash endurance
 - > 40 year retention
- Fully Static Design
- Wide Operating Voltage Range: 2.0V to 5.5V
- Wide Temperature Range:
 - Industrial: -40°C to $+85^{\circ}\text{C}$
 - Extended: -40°C to $+125^{\circ}\text{C}$

Peripheral Features (PIC12F508/509):

- 6 I/O Pins:
 - 5 I/O pins with individual direction control
 - 1 input only pin
 - High current sink/source for direct LED drive
 - Wake-on-change
 - Weak pull-ups
- 8-Bit Real-Time Clock/Counter (TMR0) with 8-Bit Programmable Prescaler

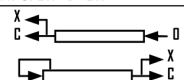
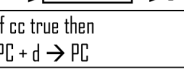
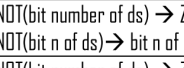
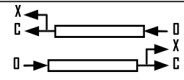
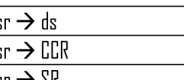
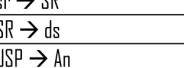
Peripheral Features (PIC16F505):

- 12 I/O Pins:
 - 11 I/O pins with individual direction control
 - 1 input only pin
 - High current sink/source for direct LED drive
 - Wake-on-change
 - Weak pull-ups
- 8-Bit Real-Time Clock/Counter (TMR0) with 8-Bit Programmable Prescaler

FIGURE 3-1: PIC12F508/509 BLOCK DIAGRAM

Device	Program Memory	Data Memory	I/O	Timers 8-bit
	Flash (words)	SRAM (bytes)		
PIC12F508	512	25	6	1
PIC12F509	1024	41	6	1
PIC16F505	1024	72	12	1

68000 Quick Reference

Opcode	Size	Operand	CCR	Effective Address												Operation	Description
	BWL	sr,ds	XNZVC	Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Rn)	abs.W	abs.L	(d,PC)	(d,PC,Rn)	#n		
ABCD	B	Dy,Dx -(Ay),-(Ax)	*U*U*	-	-	-	-	-	-	-	-	-	-	-	-	$Dy_{10} + Dx_{10} + X \rightarrow Dx_{10}$ $-(Ay)_{10} + -(Ax)_{10} + X \rightarrow -(Ax)_{10}$	Add BCD with Extend
ADD	BWL	sr,Dn Dn,ds	*****	sr ds	sr -	sr ds	sr ds	sr ds	sr ds	sr ds	sr ds	sr ds	sr -	sr -	sr -	$sr + Dn \rightarrow Dn$ $Dn + ds \rightarrow ds$	Add binary (ADDI or ADDQ is used when source is #n)
ADDA ²	WL	sr,An	-----	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	$sr + An \rightarrow An$	Add address (W sign-extended to .L)
ADDI ²	BWL	#n,ds	*****	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	$\#n + ds \rightarrow ds$	Add immediate
ADDQ ²	BWL	#n,ds	*****	ds	ds	ds	ds	ds	ds	ds	ds	ds	-	-	-	$\#n + ds \rightarrow ds$	Add quick immediate (#n range: l to 8)
ADDX	BWL	Dy,Dx -(Ay),-(Ax)	*****	-	-	-	-	-	-	-	-	-	-	-	-	$Dy + Dx + X \rightarrow Dx$ $-(Ay) + -(Ax) + X \rightarrow -(Ax)$	Add with Extend
AND	BWL	sr,Dn Dn,ds	---00	sr ds	- -	sr ds	sr ds	sr ds	sr ds	sr ds	sr ds	sr ds	sr -	sr -	sr -	$sr \& Dn \rightarrow Dn$ $Dn \& ds \rightarrow ds$	Logical AND (ANDI is used when source is #n)
ANDI ²	BWL	#n,ds	---00	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	$\#n \& ds \rightarrow ds$	Logical AND immediate
ANDI ²	B	#n,CCR	*****	-	-	-	-	-	-	-	-	-	-	-	-	$\#n \& CCR \rightarrow CCR$	Logical AND immediate to CCR
ANDI ²	W	#n,SR	*****	-	-	-	-	-	-	-	-	-	-	-	-	$\#n \& SR \rightarrow SR$	Logical AND immediate to SR (Privileged)
ASL	BWL	Dx,Dy	*****	-	-	-	-	-	-	-	-	-	-	-	-		Arithmetic shift Dy Dx bits left/right
ASR	BWL	#n,Dy	*****	-	-	-	-	-	-	-	-	-	-	-	-		Arithmetic shift Dy #n bits L/R (#n: l to 8)
	W	ds	*****	-	-	ds	ds	ds	ds	ds	ds	ds	-	-	-		Arithmetic shift ds l bit left/right (W only)
Bcc	BW ³	label	-----	-	-	-	-	-	-	-	-	-	-	-	-	if cc true then $PC + d \rightarrow PC$	Branch conditionally (cc: See Table next pg) (d: 8/16-bit signed integer)
BCHG	B L	Dn,ds #n,ds	---00	ds ds	- -	ds ds	ds ds	ds ds	ds ds	ds ds	ds ds	ds ds	- -	- -	- -	$\text{NOT}(\text{bit number of } ds) \rightarrow Z$ $\text{NOT}(\text{bit } n \text{ of } ds) \rightarrow \text{bit } n \text{ of } ds$	Set Z with state of specified bit in ds then invert the bit in ds
BCLR	B L	Dn,ds #n,ds	---00	ds ds	- -	ds ds	ds ds	ds ds	ds ds	ds ds	ds ds	ds ds	- -	- -	- -	$\text{NOT}(\text{bit number of } ds) \rightarrow Z$ $0 \rightarrow \text{bit number of } ds$	Set Z with state of specified bit in ds then clear the bit in ds
BRA	BW ³	label	-----	-	-	-	-	-	-	-	-	-	-	-	-	$PC + d \rightarrow PC$	Branch always (d: 8/16-bit signed integer)
BSET	B L	Dn,ds #n,ds	---00	ds ds	- -	ds ds	ds ds	ds ds	ds ds	ds ds	ds ds	ds ds	- -	- -	- -	$\text{NOT}(\text{bit } n \text{ of } ds) \rightarrow Z$ $1 \rightarrow \text{bit } n \text{ of } ds$	Set Z with state of specified bit in ds then set the bit in ds
BSR	BW ³	label	-----	-	-	-	-	-	-	-	-	-	-	-	-	$PC \rightarrow -(SP); PC + d \rightarrow PC$	Branch to subroutine (d: 8/16-bit sign-int)
BTST	B L	Dn,ds #n,ds	---00	ds ds	- -	ds ds	ds ds	ds ds	ds ds	ds ds	ds ds	ds ds	ds ds	ds ds	ds ds	$\text{NOT}(\text{bit } Dn \text{ of } ds) \rightarrow Z$ $\text{NOT}(\text{bit } \#n \text{ of } ds) \rightarrow Z$	Set Z with state of specified bit in ds Leave the bit in ds unchanged
CHK	W	sr,Dn	-*UUU	sr	-	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	if $Dn < 0$ or $Dn > sr$ then TRAP	Compare Dn with 0 and upper bound [sr]
CLR	BWL	ds	-0100	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	$0 \rightarrow ds$	Clear destination to zero
CMP	BWL	sr,Dn	*****	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	set CCR with $Dn - sr$	Compare Dn to source
CMPA ²	WL	sr,An	*****	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	set CCR with $An - sr$	Compare An to source
CMPI ²	BWL	#n,ds	*****	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	set CCR with $ds - \#n$	Compare destination to #n
CMPM ²	BWL	(Ay)+,(Ax)+	*****	-	-	-	ea	-	-	-	-	-	-	-	-	set CCR with $(Ax) - (Ay)$	Compare (Ax) to (Ay); Increment Ax & Ay
DBcc	W	Dn,label	-----	-	-	-	-	-	-	-	-	-	-	-	-	if cc false then { $Dn - l \rightarrow Dn$ if $Dn < -l$ then $PC + d \rightarrow PC$ }	Test condition, decrement & branch (d: 16-bit signed integer)
DIVS	W	sr,Dn	---00	sr	-	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	$\pm 32\text{bit } Dn / \pm 16\text{bit } sr \rightarrow \pm Dn$	$Dn = [16\text{-bit remainder}, 16\text{-bit quotient}]$
DIVU	W	sr,Dn	---00	sr	-	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	$32\text{bit } Dn / 16\text{bit } sr \rightarrow Dn$	$Dn = [16\text{-bit remainder}, 16\text{-bit quotient}]$
EOR	BWL	Dn,ds	---00	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	$Dn \text{ XOR } ds \rightarrow ds$	Logical exclusive OR Dn to ds
EORI ²	BWL	#n,ds	---00	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	$\#n \text{ XOR } ds \rightarrow ds$	Logical exclusive OR #n to ds
EORI ²	B	#n,CCR	*****	-	-	-	-	-	-	-	-	-	-	-	-	$\#n \text{ XOR } CCR \rightarrow CCR$	Logical exclusive OR #n to CCR
EORI ²	W	#n,SR	*****	-	-	-	-	-	-	-	-	-	-	-	-	$\#n \text{ XOR } SR \rightarrow SR$	Logical exclusive OR #n to SR (Privileged)
EXG	L	Rx,Ry	-----	ea	ea	-	-	-	-	-	-	-	-	-	-	register \leftrightarrow register	Exchange registers (32-bit only)
EXT	WL	Dn	---00	-	-	-	-	-	-	-	-	-	-	-	-	$Dn.B \rightarrow Dn.W \mid Dn.W \rightarrow Dn.L$	Sign extend (change .B to .W or .W to .L)
ILLEGAL			-----	-	-	-	-	-	-	-	-	-	-	-	-	$PC \rightarrow -(SSP); SR \rightarrow -(SSP)$	Generate Illegal Instruction exception
JMP		ds	-----	-	-	ds	-	-	ds	ds	ds	ds	ds	ds	ds	$ds \rightarrow PC$	Jump to address specified by ds
JSR		ds	-----	-	-	ds	-	-	ds	ds	ds	ds	ds	ds	ds	$PC \rightarrow -(SP); ds \rightarrow PC$	push PC, jump to subroutine at address ds
LEA	L	sr,An	-----	-	-	sr	-	-	sr	sr	sr	sr	sr	sr	sr	$sr \rightarrow An$	Load effective address of sr to An
LINK		An,#n	-----	-	-	-	-	-	-	-	-	-	-	-	-	$An \rightarrow -(SP); SP \rightarrow An;$ $SP + \#n \rightarrow SP$	Create local workspace on stack (n must be negative to allocate!)
LSL	BWL	Dx,Dy	***0*	-	-	-	-	-	-	-	-	-	-	-	-		Logical shift Dy, Dx bits left/right
LSR	BWL	#n,Dy	***0*	-	-	-	-	-	-	-	-	-	-	-	-		Logical shift Dy, #n bits L/R (#n: l to 8)
	W	ds	***0*	-	-	ds	ds	ds	ds	ds	ds	ds	-	-	-		Logical shift ds l bit left/right (W only)
MOVE	BWL	ea,ea	---00	ea	sr	ea	ea	ea	ea	ea	ea	ea	sr	sr	sr	$sr \rightarrow ds$	Move data from source to destination
MOVE	W	sr,CCR	*****	sr	-	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	$sr \rightarrow CCR$	Move source to Condition Code Register
MOVE	W	sr,SR	*****	sr	-	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	$sr \rightarrow SR$	Move source to Status Register (Privileged)
MOVE	W	SR,ds	-----	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	$SR \rightarrow ds$	Move Status Register to destination
MOVE	L	USP,An	-----	-	-	-	-	-	-	-	-	-	-	-	-	$USP \rightarrow An$ $An \rightarrow USP$	Move User Stack Pointer to An (Privileged) Move An to User Stack Pointer (Privileged)

Opcode	Size	Operand	CCR	Effective Address												Operation	Description
				Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Rn)	abs.W	abs.L	(d,PC)	(d,PC,Rn)	#n		
MOVEA ²	WL	sr,An	-----	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr → An	Move source to An (MOVE sr,An use MOVEA)
MOVEM ²	WL	Rn-Rn,ds sr,Rn-Rn	-----	-	-	ds	-	ds	ds	ds	ds	ds	-	-	-	Registers → ds sr → Registers	Move specified registers to/from memory (W source is sign-extended to .L for Rn)
MOVEP	WL	Dn,d(An) d(An),Dn	-----	-	-	-	-	-	-	-	-	-	-	-	-	Dn → d(An)...d+2(An)...d+4(A) d(An) → Dn; d+2(An)...d+4(A)	Move Dn to/from alternate memory bytes (Access only even or odd addresses)
MOVEQ ²	L	#n,Dn	-**00	-	-	-	-	-	-	-	-	-	-	-	-	#n → Dn	Move sign extended 8-bit #n to Dn
MULS	W	sr,Dn	-**00	sr	-	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	±16bit sr * ±16bit Dn → ±Dn	Multiply signed 16-bit; result: signed 32-bit
MULU	W	sr,Dn	-**00	sr	-	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	16bit sr * 16bit Dn → Dn	Multiply unisig'd 16-bit; result: unisig'd 32-bit
NBCD	B	ds	*U*U*	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	0 - ds ₁₀ - X → ds	Negate BCD with Extend
NEG	BWL	ds	*****	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	0 - ds → ds	Negate ds
NEGX	BWL	ds	*****	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	0 - ds - X → ds	Negate ds with Extend
NOP			-----	-	-	-	-	-	-	-	-	-	-	-	-	None	No operation occurs
NOT	BWL	ds	-**00	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	NOT(ds) → ds	Logical NOT (ones complement of ds)
OR	BWL	sr,Dn Dn,ds	-**00	sr	-	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr OR Dn → Dn Dn OR ds → ds	Logical OR (ORI is used when source is #n)
ORI ²	BWL	#n,ds	-**00	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	#n OR ds → ds	Logical OR #n to ds
ORI ²	B	#n,CCR	*****	-	-	-	-	-	-	-	-	-	-	-	-	#n OR CCR → CCR	Logical OR #n to CCR
ORI ²	W	#n,SR	*****	-	-	-	-	-	-	-	-	-	-	-	-	#n OR SR → SR	Logical OR #n to SR (Privileged)
PEA	L	ds	-----	-	-	ds	-	-	ds	ds	ds	ds	ds	ds	-	ds → -(SP)	Push effective address of ds onto stack
RESET			-----	-	-	-	-	-	-	-	-	-	-	-	-	Assert RESET Line	Issue a hardware RESET (Privileged)
ROL	BWL	Dx,Dy	-**0*	-	-	-	-	-	-	-	-	-	-	-	-		Rotate Dx, Dy bits left/right (without X)
ROR	BWL	#n,Dy	-**0*	-	-	-	-	-	-	-	-	-	-	-	-		Rotate Dy, #n bits left/right (#n: 1 to 8)
	W	ds	-**0*	-	-	ds	ds	ds	ds	ds	ds	ds	-	-	-		Rotate ds l-bit left/right (W only)
ROXL	BWL	Dx,Dy	***0*	-	-	-	-	-	-	-	-	-	-	-	-		Rotate Dx, Dy bits L/R (X used then updated)
ROXR	BWL	#n,Dy	***0*	-	-	-	-	-	-	-	-	-	-	-	-		Rotate Dy, #n bits left/right (#n: 1 to 8)
	W	ds	***0*	-	-	ds	ds	ds	ds	ds	ds	ds	-	-	-		Rotate ds l-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*	-	-	-	-	-	-	-	-	-	-	-	-	Dx ₁₀ - Dy ₁₀ - X → Dx ₁₀ -(Ax) ₁₀ - -(Ay) ₁₀ - X → -(Ax) ₁₀	Subtract BCD with Extend
Scc	B	ds	-----	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	If cc is true then l's → ds else 0's → ds	If cc true then ds.B = 11111111 else ds.B = 00000000
STOP		#n	*****	-	-	-	-	-	-	-	-	-	-	-	-	#n → SR; STOP	Move #n to SR, stop processor (Privileged)
SUB	BWL	sr,Dn Dn,ds	*****	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	Dn - sr → Dn ds - Dn → ds	Subtract binary (SUBI or SUBQ is used when source is #n)
SUBA ²	WL	sr,An	-----	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	sr	An - sr → An	Subtract address (W sign-extended to .L)
SUBI ²	BWL	#n,ds	*****	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	ds - #n → ds	Subtract immediate
SUBQ ²	BWL	#n,ds	*****	ds	ds	ds	ds	ds	ds	ds	ds	ds	-	-	-	ds - #n → ds	Subtract quick immediate (#n range: 1 to 8)
SUBX	BWL	Dy,Dx -(Ay),-(Ax)	*****	-	-	-	-	-	-	-	-	-	-	-	-	Dx - Dy - X → Dx -(Ax) - -(Ay) - X → -(Ax)	Subtract with Extend
SWAP	W	Dn	-**00	-	-	-	-	-	-	-	-	-	-	-	-	bits[31:16] ↔ bits[15:0]	Exchange the 16-bit halves of Dn
TAS	B	ds	-**00	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	test ds → CCR; 1 → bit7 of ds	N and Z set to reflect ds, bit7 of ds set to 1
TRAP		#n	-----	-	-	-	-	-	-	-	-	-	-	-	-	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	ds	-**00	ds	-	ds	ds	ds	ds	ds	ds	ds	-	-	-	test ds → CCR	N and Z set to reflect ds
UNLK		An	-----	-	-	-	-	-	-	-	-	-	-	-	-	An → SP; (SP)+ → An	Remove local workspace from stack

Condition Tests (& logical AND, + logical OR, ! logical NOT, " Unsigned)					
cc	Condition	Test	cc	Condition	Test
T	true	I	VC	overflow clear	!V
F	false	O	VS	overflow set	V
HI ^u	high	!C & !Z	PL	plus	!N
LS ^u	low or same	C + Z	MI	minus	N
CC, HS ^u	carry clear	!C	GE	greater or equal	N & V + !N & !V
CS, LO ^u	carry set	C	LT	less than	N & !V + !N & V
NE	not equal	!Z	GT	greater than	N & V & !Z + !N & !V & !Z
EQ	equal	Z	LE	less or equal	Z + N & !V + !N & V

An Address register (16/32-bit, n=0-7)

Dn Data register (8/16/32-bit, n=0-7)

Rn any data or address register

PC Program Counter (24-bit)

sr Source ds Destination

#n Immediate data d Displacement

ea Effective Address (source or destination)

BCD Binary Coded Decimal

SSP Supervisor Stack Pointer (32-bit)

USP User Stack Pointer (32-bit)

SP Active Stack Pointer (same as AT)

label Destination of Branch (Assembler calculates displacement value)

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 according to result of operation

- not affected, O cleared, I set, U undefined

1 Long only; all others are byte only

2 Assembler selects appropriate opcode

3 Branch sizes: .B or .S -128 to +127 bytes, .W or .L -32768 to +32767 bytes