Assembler 68000 ASM68K Programmer's Reference Manual

Corvus Concept™

CORVUS SYSTEMS

ASMK68
68000 ASSEMBLER REFERENCE MANUAL

Corvus Concept™

PART NO.: 7100-03035

DOCUMENT NO.: CCC/37-33/1.1

RELEASE DATE: November, 1982

TABLE OF CONTENTS

PREFACE

This Assembler Reference Manual describes the ASM68K assembly language used by the Corvus Concept. This version was implemented by Silicon Valley Software Incorporated.

This is a user reference manual for ASM68K, and not a tutorial. Readers should have some grasp of programming concepts, terminology, and an understanding of assembly level programming.

INTRODUCTION

ASM68K is an assembler for 68000 based computer systems. ASM68K reads 68000 assembly language statements, and generates relocatable object code for the linking loader.

This Manual describes the assembly language that is acceptable to ASM68K, and in addition describes how to use ASM68K on 68000 based computer systems.

Overview and Layout of this Manual

Chapter 1 *(this chapter)* is a general introduction to ASM68K, the notation used to describe the assembler instructions and directives, and a short list of applicable documents.

Chapter 2 contains the overview of the assembly language, character set, format of source programs, syntax, and so on.

Chapter 3 describes the rules for formation of operands, operand expressions, addressing modes and categories.

Chapter 4 describes the assembler directives which introduce symbolic constants, reserve data, and control the actions of the assembler.

Chapter 5 describes the instruction syntax in detail for the individual instructions and provides a short description of each instruction's actions and effects on condition codes.

Chapter 6 describes how to use the assembler under control of the CCOS operating system.

Chapter 7 describes the files that the assembler uses, and describes the messages that the assembler generates for errors found during the assembly.

Appendix A contains an alphabetical list of instructions, and affected condition codes.

Appendix B contains an alphabetical list of assembler directives.

Appendix C is a quick reference summary of reserved operand names.

Notation and Conventions Used in this Manual

This section describes the syntactic notation used in this manual to describe assembler language elements.

An element enclosed in "angle brackets" < > is a syntactic entity that is defined in terms of other such entities. Eventually, the definition process gets to a stage where the entities are defined in terms of basic elements such as letters and digits.

The construct : : = is to be read as "is defined to be", and is used to define one syntactic element in terms of others.

A construct enclosed in square brackets [ ] is an optional element.

The "vertical bar" character | is used as an "or" symbol when describing choices among syntactic constructs.

A construct followed by an ellipsis ... is taken to be repeated some number of times.

It is recognized that the syntactic description method used here is not completely rigorous in that it does not consider semantic issues. On such occasions, the descriptions lapse into narrative English, supplemented with examples, to clarify matters.

Applicable Documents

MC68000 16-bit Microprocessor User's Manual. Published by Motorola. Motorola publication number MC68000UM (AD2).

Linker and Library utility manual. Describes the linker and the library management utility. Published by Corvus Systems, Inc.

LAYOUT OF AN ASSEMBLER PROGRAM

This chapter describes the basic layout of a program written in the assembler language. Covered here are the basic elements such as identifiers, constants, comments, and so on. The next chapter describes the rules for operands and operand expressions.

Character Set Recognized by the Assembler

ASM68K recognizes the following character set:

```
the letters  A through Z and a through z,
the digits   0 through 9,
the ASCII    !  @  #  $  %  ^  &  *  (  )  —  +  -
graphic      =  :  ;  "  '  <  >  ´  .  ?  /  |  \
characters   [  ]  {  }     `
```

ASCII non graphics: space, tab, carriage return and line feed.

Identifiers

An identifier consists of up to eight characters, selected from the set of characters:

```
upper case letters  A  through  Z,
lower case letters  a  through  z,
digits              0  through  9,
```

the characters   percent % and underline __ .

```
<identifier> : : = <letter> | <identifier> <letter> ... |
                              <identifier> <digit> ...
```

Upper and lower case letters are treated identically for the purpose of identifiers.

The underline __character and the percent % character are considered to be letters in the context of identifiers—an identifier can start with or contain either of those characters.

An identifier may not begin with a digit.

Examples of Identifiers

widget	in all lower case,
BLIVET	in all upper case,
knothead, KNOTHEAD, and KnOtHeAd	are all equivalent,
try_more	is a valid identifier, and so is:
try_3_on	and also the identifier:
%_pcent	illustrating that identifiers can start with % signs or _ signs, but:
5th_time	is wrong because it starts with a digit, and:
over_long	is wrong because it contains more than eight characters, and:
‡ofttimes	is wrong because of the # sign.

Numeric Constants

ASM68K accepts numeric constants in either decimal (base 10), or hexadecimal (base 16) notation.

Decimal Constants

A Decimal Constant consists of one to 10 decimal digits 0 through 9, with an optional plus (+) or minus (−) sign.

<decimal constant> : : = [<sign>] <decimal digit> ...

<sign> : : = + | −

Although a 32-bit quantity can represent numbers up to ten decimal places, the actual range of (signed) decimal numbers is from −2,147,483,648 to +2,147,483,647.

Numeric constants can not have embedded commas, even if it would make long ones more readable as in the paragraph just above.

Hexadecimal Constants

A Hexadecimal Constant starts with a dollar ($) sign, and contains up to eight hexadecimal digits 0 through 9, A through F, or a through f.

<hexadecimal constant> : : = $ <hexadecimal digit> ...

Negative hexadecimal constants are represented by an 8 character constant with the sign bit set to one. For example:

$ffffff80 represents the value −128

Character String Constants

There are essentially two distinct forms of Character String Constants. A Character String Literal generates a string of bytes the same length as the literal. A String Data Item generates a length delimited string, where the first byte of the string contains the length and the rest of the string follows the length byte.

A character string literal is enclosed in apostrophe signs '.

<character string literal> : : = ' <ASCII character> ... '

The length of a character string literal depends on the context of the specific operand type. A byte operand can have at most one character in the string. A word operand can have at most two characters in the string. A long word operand can have at most four characters in the string.

The apostrophe sign itself is represented by a pair of juxtaposed apostrophe signs in the string.

A STRING DATA ITEM is a string of characters enclosed in quotes instead of apostrophes.

<string data item> : : = " <ASCII character> ... "

A quote is represented by a pair of juxtaposed quotes in the string.

The maximum length of a string data item is 256 characters.

Examples of Character and String Constants

'A'	is a one byte character literal,
'PQ'	is a two byte character literal,
'HELP'	is a four byte character literal,
"Drawn Out"	is a string data item,
'MN' 'Q'	is a four byte character literal with an embedded apostrophe,
"Ten O" "Clock"	is a string data item with an embedded quote.

Format of a Source Statement

An Assembler source statement, in general, consists of an optional (but sometimes required, as for example in an EQU statement) label field, an operation field, an operand field (if applicable to the specific operation) and an optional comment field.

<source statement> : : = [<label>] <operation> <operand> [<comment>]

Examples of Source Statements

```
BozeNite      equ     $100        ;   example of an EQU directive
              ble.s   NotAgain    ;   a short conditional branch
```

Label Field

The Label Field of a source statement consists of an identifier starting in column one on the statement line. If a space (blank), or a comment delimiter (see the section on comments below) character appears in column one, the source statement is considered to be unlabelled.

<label> : : = <identifier>

Examples of Label Field

```
ThisLine     ;This line has a label called "ThisLine".
 ThreeDog    ;"ThreeDog" is not a label because there is a space in
             ;column one. It will generate an error.
             ;This source statement is
             ;unlabelled because of the comment.
```

Operation Field

The Operation Field of a source statement consists of a valid assembly language operation code or a valid assembler directive starting in column two onwards on the source line.

<operation field> : : =	<operation code> | <assembler directive>
<operation code> : : =	discussed in Chapter 5 and in the alphabetical summary in Appendix A
<assembler directive> : : =	discussed in Chapter 4 and in the alphabetical list in Appendix B.

Operation Code Size Attributes

Many of the operation codes have an associated size attribute. This indicates the size of the operand upon which that instruction operates. The sizes are indicated by a qualifier following the operation code field. The size attributes are Byte (eight bits), Word (16 bits) and Long (32 bits). The size attribute field is a period immediately following the operation code, followed by the letter B for byte, W for word and L for long. For branch instructions, the size attribute field is S for short. The long form of the relative branch does not have a qualifier field.

Examples of Operation Field

NOP	;is a no-operation instruction.
RTS	;is a return from subroutine.
MOVE.W	;is a word move instruction.
BEQ.S	;is a short relative branch.

Operand Field

The operand field of an assembler statement line varies widely depending on the particular operation code or assembler directive. The next chapter covers the rules for forming operands, operand expressions and addressing modes in detail.

Comment Field

<operation field> : : =	<operation code> | <assembler directive>
<operation code> : : =	discussed in Chapter 5 and in the alphabetical summary in Appendix A
<assembler directive> : : =	discussed in Chapter 4 and in the alphabetical list in Appendix B.

Operation Code Size Attributes

Many of the operation codes have an associated size attribute. This indicates the size of the operand upon which that instruction operates. The sizes are indicated by a qualifier following the operation code field. The size attributes are Byte (eight bits), Word (16 bits) and Long (32 bits). The size attribute field is a period immediately

following the operation code, followed by the letter B for byte, W for word and L for long. For branch instructions, the size attribute field is S for short. The long form of the relative branch does not have a qualifier field.

Examples of Operation Field

```
NOP      ;is a no-operation instruction.
RTS      ;is a return from subroutine.
MOVE.W;is a word move instruction.
BEQ.S    ;is a short relative branch.
```

Operand Field

The operand field of an assembler statement line varies widely depending on the particular operation code or assembler directive. The next chapter covers the rules for forming operands, operand expressions and addressing modes in detail.

Comment Field

The Comment Field of a source statement starts with a semicolon and is any sequence of ASCII characters. An end of line terminates a comment. There is no provision for "block comments".

A blank line is equivalent to a comment. A statement line which contains only a comment, or only a label followed by a comment, is a valid statement line.

Examples of Comments in Statements

```
; This line is a comment line.
                                    ; This line is also a comment line.
; The next line is all blank and is considered a comment line.
LineTag        ; Label followed by comment is a valid statement.
```

OPERAND FORMATION

This chapter describes the rules for forming operands. The syntax for operands is discussed, followed by the rules for operand expressions. Finally, there is a discussion on addressing modes.

Register Operands

A register operand for an instruction may be either:

- one of the Data Registers D0 through D7,
- one of the Address Registers A0 through A7,
- one of the Special Registers such as CCR (the Condition Code Register), SR (the Status Register) and so on. The Stack Pointer (either User or System) is in fact register A7 in both cases, but it may also be referred to as SP.

<data register> : : = Dn where n is 0 through 7

<address register> : : = An where n is 0 through 7

Throughout the remainder of this manual, the notation "Dn" means a data register, "An" means an address register and "Rx" means any kind of register.

In supervisor mode the User Stack Pointer may be addressed separately by the name USP in certain instructions.

Register Range Specification

The MOVEM (Move Multiple) instruction can have its register operands specified as ranges, or as a list of ranges.

<register range>	: : =	<low register>—<high register>
<low register>	: : =	<register>
<high register>	: : =	<register>
<register range list>	: : =	<register range>
	|	<register range list>/<register range>

Example of Register Ranges

This example moves register D1, D3 through D5, D6, and registers A2 through A6 to a save area called "savearea".

```
MOVEM   d1/d3-d5/d6/A2-A6,savearea
```

Symbolic Operands

A symbolic operand uses an identifier as described in chapter 2 to reference a data item or a program instruction location. Symbolic operands are considered relocatable unless they are specifically stated as EXTERNAL (see the section below on absolute and relocatable operands and expressions).

A symbolic operand can have a size attribute following it. The size attribute can be W for word or L for long. This topic is covered below in the discussion on addressing modes.

Immediate or Literal Operands

Immediate Operands are signalled by placing a # sign in front of the operand.

<immediate operand> : : = #<label> | #<numeric constant

Examples of Immediate Operands

DIVU	#9,D0	;	divides d0 by 9
MOVEQ	#diskwryt,D5	;	place disk write command in D5
MOVE.W	#$48,$a (a1)	;	move 48 hex to (al) + 10

Operand Expressions

Operands may be combined with arithmetic operators to form Operand Expressions.

When forming operand expressions, there are four valid operators:

+	plus or adding operator. This may occur as a unary operator or a binary operator.
–	minus or subtracting operator. This may occur as a unary operator or a binary operator.
*	multiply operator. This can only be a binary operator.
/	divide operator. This can only be a binary operator.

Operator Precedence

Expressions are evaluated left to right. The multiply and divide operators have a higher precedence (bind tighter) than the add and subtract operators. Parentheses may be used to alter the precedence. Operators of equal precedence are evaluated left to right.

Examples of Operand Expressions

```
124*1024
Board*128
MBufFrst+ (BufrSize/PageSize-1)*PMapIncr
UARTRead-UARTTabl
```

Relocatable and Absolute Operands

Terms in an operand expression are absolute, relocatable or external.

A relocatable value bears a fixed relationship to the current value of the program counter, such that moving the entire section of code to a different place in memory does not alter the value of that expression.

An absolute value is independent of the program counter and refers either to absolute memory locations or to immediate operands.

An external reference is filled in by the linker.

Operand expressions containing absolute and relocatable terms evaluate to absolute, relocatable or malformed. If "A" is an absolute term and "R" is a relocatable term, the following relationships hold:

R ± A −> R
A + R −> R
R − R −> A
A + A, A − A, A * A and A / A −> A
R + R, R * R and R / R are all malformed.

Terms may be parenthesized such that a relocatable term can be added to the difference of two relocatable terms to yield a relocatable term:

R + (R − R) −> R,
R + (R − R) + (R − R) and so on.

External references are filled in at link time. The linker either generates a program counter relative address or it generates an indexed JMP relative to A4, where A4 points to the start of the jump table for a particular segment.

Addressing Modes

This section covers the different addressing modes that the MC68000 can process, and describes the assembler syntax for each mode. Here is a summary of the addressing modes:

- Register Direct,
- Address Register Indirect,
- Address Register Indirect with Post-Increment,
- Address Register Indirect with Pre-Decrement,
- Address Register Indirect with Displacement,
- Address Register Indirect with Index,
- Absolute Short Address,
- Absolute Long Address,
- Program Counter with Displacement,
- Program Counter with Index,
- Immediate Data,
- Condition Codes or Status Register.

Register Direct Addressing Modes

Data Register Direct

The Data Register Direct addressing mode uses one of the eight data registers as an operand. The syntax of the operand is:

```
Dn
```

where 'n' is a number between 0 and 7.

Address Register Direct

The Address Register Direct addressing mode uses one of the eight address registers as an operand. The syntax of the operand is:

```
An
```

where 'n' is a number between 0 and 7. In general, if the size field of the instruction specifies a byte operation, the address register direct addressing mode cannot be used.

Memory Addressing Modes

Address Register Indirect

The Address Register Indirect addressing mode specifies that the address of the operand is in one of the 8 address registers. The syntax is:

```
(An)
```

where 'n' is a number between 0 and 7.

Address Register Indirect with Post Increment

The Address Register Indirect with Post-increment specifies that the address of the required operand is in one of the eight address registers. When the operation is complete, the specified address register is incremented by the size field of the operation. The syntax of the operand is:

```
(An)+
```

where 'n' is a number between 0 and 7. If the address register specified is A7 (the Stack-Pointer), it is always incremented by two—the Stack-Pointer always aligns to a word boundary.

Address Register Indirect with Pre Decrement

The Address Register Indirect with Pre-decrement specifies that the address of the required operand is in one of the eight address registers. When the operation is complete, the specified address register is decremented by the size field of the operation. The syntax of the operand is:

```
-(An)
```

where 'n' is a number between 0 and 7. If the address register specified is A7 (the Stack-Pointer), it is always decremented by two—the Stack-Pointer always aligns to a word boundary.

Address Register Indirect with Displacement

The Address Register Indirect with Displacement addressing mode specifies that a displacement is added to the contents of one of the eight address registers to form the final address. The syntax is:

```
disp(An)
```

where 'disp' is a 16-bit signed displacement and 'n' is a number between 0 and 7. The displacement 'disp' must be an absolute expression.

Address Register Indirect with Index

The Address Register Indirect with Index addressing mode specifies that a displacement plus the contents of a specified index-register are added to the contents of a specified address register to form the final address. The syntax is:

```
disp(An,Ri)
```

where 'disp' is a signed 8-bit displacement, 'Ri' is the signed word or long word contents of any address of data register and 'An' is any of the eight address registers. The index register 'Ri' can be followed by an optional .W meaning that the low order 16 bits are to be used or .L meaning that the entire 32 bits of the register are to be used for indexing.

Special Addressing Modes

The special addressing modes include the absolute short, absolute long and program counter relative modes. These modes are indicated by a size attribute following the operand. The size attribute is W for absolute short and L for absolute long. They are covered in the subsections below.

Absolute Short Address

The Absolute Short addressing mode uses a signed 16 bit value to form the final address. The syntax is:

```
.W
```

where 'xxx.W' specifies a 16-bit value.

Absolute Long Address

The Absolute Long addressing mode uses a signed 32-bit value to form the final address. The syntax is:

```
.L
```

where 'xxxx.L' specifies a signed 32-bit value.

Program Counter with Displacement

The Program Counter with Displacement addressing mode is used for PC-relative addressing. The displacement field is indicated in the size field of the instruction itself. A size field of "B" (Byte) indicates an 8-bit displacement. If the size field is omitted or specified as "W" (Word), the displacement is assumed to be a 16-bit displacement. For example:

```
MOVE.W    CloseBy,D0    ; Obtain data relative to PC.
.....
   .....   some more instructions
      .....
CloseBy   DATA     10
```

Program Counter with Index

The Program Counter with Index addressing mode uses an index register to add an offset to the program-counter relative address. The syntax of this addressing mode is:

```
expr(Ri)
```

where 'expr' is a relocatable expression and 'Ri' is any address register or data register. The index register 'Ri' can be followed by an optional .W meaning that the low order 16 bits are to be used or .L meaning that the entire 32 bits of the register are to be used for indexing.

While this addressing mode is superficially similar to the Address Register Indirect with Displacement addressing mode, it differs in that the displacement expression 'expr' must be relocatable instead of absolute.

Immediate Data

An Immediate Data addressing mode is signalled by placing the # sign before an operand, as discussed previously. Immediate operands do not need any size attributes—immediate operands are automatically promoted to long operands if needed.

Condition Codes or Status Register

Several instructions refer to the Condition Code Register (CCR) or the Status Register (SR). The SR and CCR are both parts of the same 16-bit register, with the SR being the high order byte and the CCR being the low order byte. Only programs running in supervisor state can alter the contents of the SR, whereas a program running in either supervisor or user state can alter the CCR.

Addressing Modes for Branch Instructions

The branch instructions—Bcc (Branch on Condition), BRA (Branch Always) and BSR (Branch to Subroutine)—are all variants of the same instruction. This instruction has a PC relative displacement as part of the instruction. The displacement is either 8-bit or 16-bit. The 8-bit displacement can fit into the instruction word. The 16-bit displacement requires an extension word whose presence is signalled by the 8-bit displacement part of the instruction being zero.

The short form (8-bit displacement) of the branch instructions is signalled by a size indicator of "S" (short). The long form (16-bit displacement) is the default—there is no size indicator required.

Because of the way that the instruction works, there is one special case, namely that if the destination of the branch is to the next instruction, the short form cannot be used. Thus it is invalid to code the instruction:

```
          BRS.S        NextDoor
      NextDoor ..... the next instruction
```

To achieve the desired result—branch to the next instruction—the long form (16-bit displacement must be used).

Addressing Categories

Effective address modes are grouped into several categories which derive from the ways that they are used to address operands. The addressing categories are given below. The table on the next page summarizes which addressing modes belong to which categories.

Data	If an effective address mode is used to refer to data operands, it is called a data addressing mode.
Memory	If an effective address mode can refer to memory operands, it is called a memory addressing mode.
Alterable	If an effective address mode can refer to alterable (writable) operands, it is called an alterable addressing mode.

Control If an effective addressing mode can refer to memory operands without any size specification, it is called a control addressing mode.

The addressing modes can be combined. In the description of the operation codes, a phrase such as "data alterable addressing modes" means that the particular instruction can use either the data addressing mode or the alterable addressing mode.

Addressing Mode	Assembler Syntax	Data	Memory	Control	Alterable
D-Register Direct	Dn	X			X
A-Register Direct	An				X
A-Register Indirect	(An)	X	X	X	X
A-Register Indirect with Post-Increment	(An)+	X	X		X
A-Register Indirect with Pre-Decrement	−(An)	X	X		X
A-Register Indirect with Displacement	d(An)	X	X	X	X
A-Register Indirect with Index	d(An, Ri)	X	X	X	X
Absolute Short	xxx.W	X	X	X	X
Absolute Long	xxxxxx.L	X	X	X	X
PC Relative	<label>	X	X	X	
PC Relative with Index	<label>+Ri	X	X	X	
Immediate Data	#<data>	X	X		

Table 3-1
Addressing Categories

ASSEMBLER DIRECTIVES

Assembler Directives control assembler operation and declare data, as opposed to generating instructions for the machine to execute.

IDENT—Name an Assembly Unit

The IDENT directive is an optional component of an assembly program. When present, it serves to give a name to the unit that is being assembled so that that unit can be handled by other utility programs such as the linker.

```
[IDENT <identifier>]
```

Where the <identifier> must be a valid name in the context of the linker and other system utilities.

If an assembly unit does not have an IDENT directive, the assembler gives the unit the name NONAME by default.

END—Signal End of an Assembly Unit

The END directive signals the end of the current assembly unit.

```
END [<label>]
```

If the optional <label> is present on an END directive, it serves to identify the address at which the program should start when loaded.

If there are multiple assembly units in one run of the assembler, the label given as the operand of the last labelled END directive is the label which is considered as the transfer symbol.

GLOBAL—Define a Global Entry Point

The GLOBAL directive is used to declare labels in this assembly unit that can be referenced from outside by other units that are independently assembled or compiled.

```
GLOBAL <label>[,<label> ...]
```

All global labels appearing in a GLOBAL list must be defined as program labels in the current assembly unit.

EXTERN—Define an Externally Referenced Symbol

The EXTERN directive is used to declare labels which exist in other independently assembled or compiled units and are referenced within the current assembly unit.

```
EXTERN <label>[, <label> ...]
```

EQU—Define a Symbolic Constant

The EQU directive is used to associate a symbolic identifier with the value of an expression. Thereafter, every time that the symbolic identifier is encountered in an operand expression, the value of the associated expression is substituted.

```
<label> EQU <expression>
```

The <expression> may involve both literal constants and other symbolically defined constants, the symbol % (percent) is used to define the current location counter.

DATA—Declare Data Items

The DATA directive is used to initialize storage locations with data values. The format of the DATA statement is:

```
[<label>] DATA[.<size>]
<expression>[,<expression> ...]
```

The <label> field is optional in a DATA statement. If present, it serves to identify that storage location in memory. If a label field is present, it also forces the location counter to a word boundary for word and long data items.

The DATA statement itself may have a size indicator, which serves to define the size of the data elements being defined. The size field may be one of the following:

B	means that the data elements are byte (8-bit) quantities.
W	means that the data elements are word (16-bit) quantities. If the size field is omitted from the DATA statement, word is the default data size.
L	means that the data elements are long word (32-bit) quantities.

PAGE—Issue a Page Eject on Listing

The PAGE directive is used to cause a page eject or form feed on the assembly print file. This enables the programmer to produce a more readable printout by spacing things out.

```
<PAGE directive> : : =
[<label>] PAGE [<operand>]
```

The PAGE directive does not require either a <label> field or an <operand> field. The <label> field is ignored if present. If the <operand> field is present it must be an arithmetic expression which indicates the number of lines on a page.

LIST—Turn Listing On or Off

The LIST directive determines whether the assembler generates an output listing. The format of the LIST directive is:

```
LIST <expression>
```

If the value of <expression> evaluates to zero, the listing is turned off. If the listing is already off, or if no listing file was specified on the assembler command line, the LIST directive has no effect.

If the value of <expression> evaluates to non-zero, the listing is turned on. If the listing is already on, the LIST directive has no effect.

```
ON    equ   1
OFF   equ   0

      LIST  OFF   ; turn off the listing
      LIST  ON    ; turn listing
                    on again
```

Include—To insert other files

The Include directive will cause the content of different text files to be included into the current assembly file.

Include expression

The expression is in the form '/volume/filename'. The /volume/ is optional, the single quote is required.

ASSEMBLER OPERATION CODES

This chapter describes the instruction mnemonics and the forms of the operands which ASM68K accepts. There is a concise alphabetically ordered summary of all the instruction codes in Appendix A.

This manual is not intended as a detailed blow-by-blow description of generated object code and so on. For a detailed description of operation codes, object codes, affected status flags and so on, the reader is referred to the Motorola publication "MC68000 16-bit Microprocessor User's Manual."

ABCD—Add Decimal with Extend

Syntax: ABCD Dy,Dx
ABCD −(Ay), −(Ax)

Size: Byte

Condition Codes:

N—Undefined
Z—Cleared if the result is non-zero, otherwise unchanged.
V—Undefined.
C—Set if a (decimal) carry was generated, otherwise cleared.
X—Set the same as the carry bit.

The source operand plus the X (extend) bit is added to the destination operand and the result is stored in the destination. The operands are addressed in one of two ways:

- Data register to data register. The operands are contained in the specified data registers.
- Memory to memory. The operands are addressed via the Address Register Indirect with Pre-decrement addressing mode, using the address registers specified in the instruction.

ADD—Add Binary

Syntax: ADD <ea>,Dn
ADD Dn,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a carry is generated, otherwise cleared.
X—Set the same as the carry bit.

The source operand is added to the destination operand and the result is stored in the destination operand.

If the <ea> field is the source, all addressing modes may be used. The Address Register Direct addressing mode may not be used for a byte size operation. If the <ea> field is the destination, only alterable memory addressing modes may be used.

ADDA—Add Address

Syntax: ADDA <ea>,An

Size: Word or Long

The ADDA instruction does not affect any condition codes.

The source operand is added to the destination address register and the result is stored in the address register. All addressing modes can be used for the source operand.

ADDI—Add Immediate

Syntax: ADDI #<data>,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a carry is generated, otherwise cleared.
X—Set the same as the carry bit.

The immediate data field is added to the destination operand and the result is stored in the destination location. Only data alterable addressing modes can be used.

ADDQ—Add Quick

Syntax: ADDQ #<data>,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a carry is generated, otherwise cleared.
X—Set the same as the carry bit.

The immediate data field is added to the destination operand and the result is stored in the destination location. The data field must lie in the range 1 .. 8. Only data alterable addressing modes can be used. The Address Register Direct addressing mode may not be used for byte size operations.

ADDX—Add Extended

Syntax: ADDX Dy,Dx
ADDX −(Ay), −(Ax)

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a carry is generated, otherwise cleared.
X—Set the same as the carry bit.

The source operand plus the X (extend) bit is added to the destination operand and the result is stored in the destination location. The operands can be addressed in one of two ways:

- Data register to data register. The operands are contained in the specified data registers.

- Memory to memory. The operands are addressed via the Address Register Indirect with Pre-decrement addressing mode, using the address registers specified in the instruction.

AND—Logical AND

Syntax: AND <ea>,Dn
AND Dn,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the most significant bit of the result is set, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.
X—Not affected.

The source and destination operands are logically ANDed and the result is stored in the destination location. Address registers may not be used as operands. If the <ea> field is a source operand, only data addressing modes can be used. If the <ea> field is a destination operand, only alterable memory addressing modes can be used.

ANDI—Logical AND Immediate

Syntax: ANDI #<data>,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the most significant bit of the result is set, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.

The immediate data field is logically ANDed with the destination operand. The result is stored in the destination location.

ANDI to CCR or SR

The effective address field <ea> in the ANDI instruction can refer to either the condition codes register (CCR) or the status register (SR). Access to the status register is in one of two modes:

- If the size field of the instruction is byte, the operation only affects the condition codes register—the low order bits of the status register.
- If the size field of the instruction is word, the operation affects the entire status register. This is a privileged operation and can only be issued by a program running in supervisor state.

ASL and ASR—Arithmetic Shifts

Syntax: ASd Dx,Dy
ASd #<data>,Dy
ASd <ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the most significant bit of the result is set, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if the most significant bit changes at any time during the shift, otherwise cleared.
C—Set according to the last bit shifted out of the operand. It is cleared for a shift count of zero.
X—Set according to the last bit shifted out of the operand. Unaffected by a shift count of zero.

Bits of the operand are arithmetically shifted in the direction specified by 'd'—R for right and L for left.

Either a data register or a memory location can be shifted. If the operand is a memory location, the shift count is 1. Only memory alterable addressing modes can be used for this form.

If the operand is a data register, the shift count can be specified in one of two ways:

Immediate	The shift count is specified by the immediate <data> field in the instruction. In this case, the shift count must lie in the range 1 .. 8. This is the second form of the instruction given in the syntax above.
Register	The shift count is contained in data register specified in the instruction. This is the first form of the instruction given in the syntax above.

Bcc—Branch Conditionally

Syntax: Bcc <label>

Size: Short (8-bit displacement) or default (16-bit displacement)

The Bcc instruction does not affect any condition codes.

If the condition specified by 'cc' is satisfied, a branch is made to the location specified by <label>. The 'cc' is one of the following:

CC—Carry clear
CS —Carry set
EQ—Equal
GE—greater than or equal (S)
GT—greater than (S)
HI —High (U)
HS—High or same (U)
LE —Less than or equal (S)
LO—Lower (U)
LS —Low or same (U)
LT —Less than (S)
MI —Minus
NE—Not equal
PL —Plus
VC—No overflow
VS —Overflow

The notations (U) and (S) mean that the condition codes apply the Unsigned and Signed operations, respectively.

If the destination of the branch is to the next instruction, the short form of the instruction must not be used.

BCHG—Test a Bit and Change

Syntax: BCHG Dn,<ea>
BCLR #<data>,<ea>

Size: Byte or Long

Condition Codes:

N—Unaffected.
Z—Set if the tested bit is zero, otherwise cleared.
V—Unaffected.
C—Unaffected.
X—Unaffected.

A bit in the destination operand is tested. If the bit is zero, the Z condition code is set. If the bit is non-zero, the Z condition code is cleared. The bit is then inverted in the destination operand.

If the destination operand is a data register, bit numbering is modulo 32.

If the destination operand is a memory location, the size of the operation is implicitly byte, and bit numbering is modulo 8.

The bit number for the operation can be specified in one of two ways:

Immediate	The immediate <data> field in the instruction specifies the bit number.
Register	The bit number is contained in a data register specified in the instruction.

Only data alterable addressing modes can be used in this instruction.

BCLR—Test a Bit and Clear

Syntax: BCLR Dn,<ea>
BCLR #<data>,<ea>

Size: Byte or Long

Condition Codes:

N—Unaffected.
Z—Set if the tested bit is zero, otherwise cleared.
V—Unaffected.
C—Unaffected.
X—Unaffected.

A bit in the destination operand is tested. If the bit is zero, the Z condition code is set. If the bit is non-zero, the Z condition code is cleared. The bit is then cleared in the destination operand.

If the destination operand is a data register, bit numbering is modulo 32.

If the destination operand is a memory location, the size of the operation is implicitly byte, and bit numbering is modulo 8.

The bit number for the operation can be specified in one of two ways:

Immediate	The immediate <data> field in the instruction specifies the bit number.
Register	The bit number is contained in a data register specified in the instruction.

Only data alterable addressing modes can be used in this instruction.

BRA—Branch Always

Syntax: BRA <label>

Size: Byte or Word

The BRA instruction does not affect any condition codes.

Program execution continues at the location specified by <label>.

Note that the BRA instruction cannot perform a short-offset branch to the next location.

BSET—Test a Bit and Set

Syntax: BSET Dn,<ea>
BSET #<data>,<ea>

Size: Byte or Long

Condition Codes:

N—Unaffected.
Z—Set if the tested bit is zero, otherwise cleared.
V—Unaffected.
C—Unaffected.
X—Unaffected.

A bit in the destination operand is tested. If the bit is zero, the Z condition code is set. If the bit is non-zero, the Z condition code is cleared. The bit is then set to one in the destination operand.

If the destination operand is a data register, bit numbering is modulo 32.

If the destination operand is a memory location, the size of the operation is implicitly byte, and bit numbering is modulo 8.

The bit number for the operation can be specified in one of two ways:

Immediate	The immediate <data> field in the instruction specifies the bit number.
Register	The bit number is contained in a data register specified in the instruction.

Only data alterable addressing modes can be used in this instruction.

BSR—Branch to Subroutine

Syntax: BSR <label>

Size: Byte or Word

The BSR instruction does not affect any condition codes.

The address of the instruction immediately following the BSR instruction is pushed onto the stack. Program execution then continues at the location specified by <label>.

Note that the BSR instruction cannot perform a short-offset branch to the next instruction.

BTST—Test a Bit

Syntax: BTST Dn,<ea>
BTST #<data>,<ea>

Size: Byte or Long

Condition Codes:

N—Unaffected.
Z—Set if the tested bit is zero, otherwise cleared.
V—Unaffected.
C—Unaffected.
X—Unaffected.

A bit in the destination operand is tested. If the bit is zero, the Z condition code is set. If the bit is non-zero, the Z condition code is cleared.

If the destination operand is a data register, bit numbering is modulo 32.

If the destination operand is a memory location, the size of the operation is implicitly byte, and bit numbering is modulo 8.

The bit number for the operation can be specified in one of two ways:

Immediate	The immediate <data> field in the instruction specifies the bit number.
Register	The bit number is contained in a data register specified in the instruction.

Only data alterable addressing modes can be used in this instruction.

CHK—Check Register against Bounds

Syntax: CHK <ea>,Dn

Size: Word

Condition Codes:

N—Set if Dn < 0, cleared if Dn > (<ea), otherwise undefined.
Z—Undefined.
V—Undefined.
C—Undefined.
X—Unaffected.

The CHK instruction checks the contents of a data register against boundaries. The lower bound is always considered zero. The upper bound is contained in the <ea>. If the contents of Dn are less than zero or greater than the contents of <ea>, a trap is generated to the CHK vector.

Only data addressing modes can be used for the <ea> field of the CHK instruction.

CLR—Clear an Operand

Syntax: CLR <ea>

Size: Byte, Word or Long

Condition Codes:

N—Always cleared.
Z—Always set.
V—Always cleared.
C—Always cleared.
X—Unaffected.

The destination operand specified by <ea> is cleared to all zeros. Only data alterable addressing modes can be used.

CMP—Compare

Syntax: CMP <ea>,Dn

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a borrow is generated, otherwise cleared.
X—Unaffected.

The source operand is subtracted from the destination operand without changing the destination operand. The condition codes are set appropriately. All addressing modes are allowed for the <ea> field. The Address Register Direct addressing mode may not be used for byte size operands.

CMPA—Compare Address

Syntax: CMPA <ea>,An

Size: Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a borrow is generated, otherwise cleared.
X—Unaffected.

The source operand is subtracted from the destination address register without changing the destination address register. The condition codes are set appropriately. All addressing modes are allowed for the <ea> field.

CMPI—Compare Immediate

Syntax: CMPI #<data>,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise.cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a borrow is generated, otherwise cleared.
X—Unaffected.

The immediate <data> field is subtracted from the destination operand without changing the destination operand. The condition codes are set appropriately. Only data alterable addressing modes are allowed for the <ea> field.

CMPM—Compare Memory

Syntax: CMPM (Ay)+,(Ax)+

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a borrow is generated, otherwise cleared.
X—Unaffected.

The source operand is subtracted from the destination operand without changing the destination operand. The condition codes are set appropriately. Both operands are always addressed using the Address Register Indirect with Post-increment addressing mode, using the address registers specified in the instruction.

DBcc—Test Condition, Decrement and Branch

Syntax: DBcc Dn,<label>

Size: Word

The DBcc instruction does not affect any condition codes.

The DBcc instruction is a loop-control primitive. There are three parts to the instruction: A condition to be tested (specified by 'cc'), a data register used as a counter (specified by Dn) and a label used as the target of a branch (specified by <label>).

The condition specified by 'cc' is tested first to see if the termination condition has been met. If it has, program execution continues with the next instruction in sequence. If the termination condition has not been satisfied, the low order 16 bits of the data register 'Dn' are decremented by one. If the result is −1, program execution continues with the next instruction in sequence. If the result is not equal to −1, program execution continues at the location specified by <label>. The condition 'cc' is one of the following:

CC—Carry clear	LS —Low or same (U)
CS —Carry set	LT —Less than (S)
EQ—Equal	MI —Minus
F —False (never true)	NE—Not equal
GE—greater than or equal (S)	PL —Plus
GT—greater than (S)	RA—Always (same as F)
HI —High (U)	T —Always True
HS—High or same (U)	VC—No overflow
LE —Less than or equal (S)	VS —Overflow
LO—Lower (U)	

The notations (U) and (S) denote the condition codes for operations which are Unsigned and Signed, respectively.

DIVS—Signed Divide

Syntax: DIVS <ea>,Dn

Size: Word

Condition Codes:

N—Set if the quotient is negative, otherwise cleared.
Z—Set if the quotient is zero, otherwise cleared.
X—Undefined if overflow occurs.
V—Set if division overflow is detected, otherwise cleared.
C—Always cleared.
X—Unaffected.

The destination data register is divided by the source operand and the result is stored in the destination data register. The destination data register is treated as a long (32-bit) operand and the source operand is a word (16-bit) operand. The operation is performed using signed division.

The result appears in the specified data register with the quotient in the least significant 16 bits and the remainder in the most significant 16 bits.

The sign of the remainder is always the same as the sign of the dividend, unless the remainder is zero. Two special cases must be noted:

- Division by zero causes a trap.
- If overflow is detected during the division, the overflow condition code is set and the operands are unchanged.

Only data addressing modes can be used to specify the <ea> field in the DIVS instruction.

DIVU—Unsigned Divide

Syntax: DIVU <ea>,Dn

Size: Word

Condition Codes:

N—Set if the most significant bit of the quotient is set, otherwise cleared.
Z—Set if the quotient is zero, otherwise cleared.
V—Set if division overflow is detected, otherwise cleared.
C—Always cleared.
X—Unaffected.

The destination data register is divided by the source operand and the result is stored in the destination data register. The destination data register is treated as a long (32-bit) operand and the source operand is a word (16-bit) operand. The operation is performed using unsigned division.

The result appears in the specified data register with the quotient in the least significant 16 bits and the remainder in the most significant 16 bits.

The sign of the remainder is always the same as the sign of the dividend, unless the remainder is zero. Two special cases must be noted:

- Division by zero causes a trap.
- If overflow is detected during the division, the overflow condition code is set and the operands are unchanged.

Only data addressing modes can be used to specify the <ea> field for the DIVU instruction.

EOR—Logical Exclusive OR

Syntax: EOR Dn,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the most significant bit of the result is set, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.
X—Unaffected.

The source data register is exclusive ORed with the destination operand and the result is stored in the destination operand. Note that this instruction restricts the source operand to a data register.

Only data alterable addressing modes can be used to specify the destination operand.

EORI—Logical Exclusive OR Immediate

Syntax: EORI #<data>,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the most significant bit of the result is set, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.
X—Unaffected.

The immediate <data> field is exclusive ORed with the destination operand and the result is stored in the destination operand.

Only data alterable addressing modes can be used to specify the destination operand.

EORI to CCR or SR

The effective address field <ea> in the EORI instruction can refer to either the condition codes register (CCR) or the status register (SR). Access to the status register is in one of two modes:

- If the size field of the instruction is byte, the operation only affects the condition codes register—the low order bits of the status register.
- If the size field of the instruction is word, the operation affects the entire status register. This is a privileged operation and can only be issued by a program running in supervisor state.

EXG—Exchange a Pair of Registers

Syntax: EXG Rx,Ry

Size: Long

The EXG instruction does not affect any condition codes.

The EXG instruction can work in one of three ways:

- exchange a pair of data registers,
- exchange a pair of address registers, or,
- exchange a data register with an address register.

The Rx field specifies either a data register or an address register. If the exchange is between a data register and an address register, the Rx field must be a data register.

The Ry field specifies either a data register or an address register. If the exchange is between a data register and an address register, the Ry field must be an address register.

EXT—Sign Extend

Syntax: EXT Dn

Size: Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.
X—Unaffected.

The specified data register is sign extended from a byte to a word or from a word to a long word, depending on the size field of the instruction. If the size field is W, bit 7 of the specified data register is copied into bits 8 thru 15. If the size field is L, bit 15 is copied into bits 16 thru 31.

JMP—Jump

Syntax: JMP <ea>

Size: Unsized

The JMP instruction does not affect any condition codes.

Program execution continues at the location specified by the effective address field <ea>.

LEA—Load Effective Address

Syntax: LEA <ea>,An

Size: Long

The LEA instruction does not affect any condition codes.

The effective address is loaded into the specified address register.

LINK—Allocate Stack Space

Syntax: LINK An,#<Displacement>

The LINK instruction does not affect any condition codes.

The contents of the specified address register are pushed onto the stack. The updated stack pointer is then moved to the address register. The sign-extended displacement is then added to the stack pointer. See the UNLK instruction.

LSL and LSR—Logical Shifts

Syntax: LSd Dx,Dy
LSd #<data>,Dy
LSd <ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Set according to the last bit shifted out of the operand. It is cleared for a shift count of zero.
X—Set according to the last bit shifted out of the operand. Unaffected by a shift count of zero.

Bits of the operand are logically shifted in the direction specified by 'd'—R for right and L for left.

Either a data register or a memory location can be shifted. If the operand is a memory location, the shift count is 1. Only memory alterable addressing modes can be used for this form.

If the operand is a data register, the shift count can be specified in one of two ways:

Immediate	The shift count is specified by the immediate <data> field in the instruction. In this case, the shift count must lie in the range 1 .. 8. This is the second form of the instruction given in the syntax above.
Register	The shift count is contained in data register specified in the instruction. This is the first form of the instruction given in the syntax above.

MOVE—Move Data from Source to Destination

Syntax: MOVE <source ea>,<destination ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.
X—Unaffected.

The operand at <source ea> is moved to the location at <destination ea>. The data is examined as it is moved and the condition codes set appropriately.

All addressing modes can be used for the <source ea> with the exception that Address Register Direct addressing cannot be used for byte size operations.

Only data alterable addressing modes can be used for the <destination ea>.

MOVE to CCR—Move to Condition Codes

Syntax: MOVE <ea>,CCR

Size: Word

Condition Codes:

N—Set according to the source operand.
Z—Set according to the source operand.
V—Set according to the source operand.
C—Set according to the source operand.
X—Set according to the source operand.

The contents of the source operand are moved to the condition codes register. Although the source operand is a word, only the low order eight bits are used to update the condition codes register. Only data addressing modes can be used in this instruction.

MOVE to SR—Move to Status Register (Privileged)

Syntax: MOVE <ea>,SR

Size: Word

Condition Codes:

N—Set according to the source operand.
Z—Set according to the source operand.
V—Set according to the source operand.
C—Set according to the source operand.
X—Set according to the source operand.

The contents of the source operand are moved to the condition codes register. The source operand is a word—all bits of the status register are affected. Only data addressing modes can be used in this instruction.

This is a privileged instruction and can only be issued by a program running in supervisor state.

MOVE from SR—Move from Status Register

Syntax: MOVE SR,<ea>

Size: Word

The MOVE from SR instruction does not affect any condition codes.

The contents of the status register are moved to the destination location specified by the <ea> field. The operand size is a word. Only data alterable addressing modes can be used in this instruction.

MOVE USP—Move User Stack Pointer (Privileged)

Syntax: MOVE USP,An
MOVE An,USP

Size: Long

The MOVE USP instruction does not affect any condition codes.

The contents of the user stack pointer are moved to or from the specified address register.

The MOVE USP instruction is privileged and may only be issued by a program running in supervisor state.

MOVEA—Move Address

Syntax: MOVEA <ea>,An

Size: Word or Long

The MOVEA instruction does not affect any condition codes.

The contents of the source operand are moved to the specified address register. All addressing modes can be used in this instruction.

MOVEM—Move Multiple Registers

MOVEM moves multiple registers to memory or moves multiple words of memory to registers. It is used as a high speed register save and restore mechanism.

Syntax: MOVEM <Register List>,<ea>
MOVEM <ea>,<Register List>

Size: Word or Long

The MOVEM instruction does not affect any condition codes.

Selected registers are moved to or from consecutive memory locations starting at the location specified by the effective address. Registers to be moved are selected by a register selection mask which is described below. The size field of the instruction selects how much of a register is to be moved. Either the entire register is moved or just the low order word. If a word sized transfer is being made to the registers, each word is sign-extended to 32 bits and the resulting long word is moved to the register.

MOVEM can use control addressing mode, post-increment mode or pre-decrement mode. If the effective address is in one of the control modes, the registers are moved starting at the effective address and up through higher addresses. The registers are transferred in the order D0 through D7, then A0 through A7.

If the effective address is the post-increment mode, only memory to register moves are allowed. The order of transfer is the same as for the control modes as described in the previous paragraph. The incremented address register is updated to contain the address of the last word loaded plus two.

If the effective address is the pre-decrement mode, only register to memory moves are allowed. The registers are moved starting at the specified address minus two, and down through lower addresses. The order of storing the registers is from A7 down to A0, then from D7 down to D0. The decremented address register is updated to contain the address of the last word stored.

The register list mask list is a bit map which controls which registers are to be moved. The low order bit corresponds to the first register to be moved, while the high order bit corresponds to the last register to be moved. For control and post-increment addressing modes, the mask correspondence is:

```
bit→ 15                                                0
 ------------------------------------------------------
  A7  A6  A5  A4  A3  A2  A1  A0  D7  D6  D5  D4  D3  D2  D1  D0
 ------------------------------------------------------
```

For the pre-decrement address mode, the mask correspondence is:

```
bit→ 15                                                0
 ------------------------------------------------------
  D0  D1  D2  D3  D4  D5  D6  D7  A0  A1  A2  A3  A4  A5  A6  A7
 ------------------------------------------------------
```

The register list is specified by giving lists of register names separated by slashes. A range of registers can be specified by giving two register names separated by a hyphen.

MOVEP—Move Peripheral Data

Syntax: MOVEP Dx,d(Ay)
        MOVEP d(Ay),Dx

Size: Word or Long

The MOVEP instruction does not affect any condition codes.

Data is moved between a data register and alternate bytes of memory, starting at the specified location and incrementing by two. The high order byte of the data register is moved first and the low order byte is moved last. The memory address is specified using the Address Register Indirect with Displacement addressing mode.

If the address is even, all data transfers are made on the high order half of the data bus. If the address is odd, all data transfers are made on the low order half of the data bus.

MOVEQ—Move Quick

Syntax: MOVEQ #<data>,Dn

Size: Long

Move <data> to a data register. The data must be in the range −128 .. +127. The data is sign-extended to a long operand and all 32 bits are moved to the data register.

Condition Codes:

N—Set if result is negative; cleared otherwise.
Z—Set if result is zero; cleared otherwise.
V and C—always cleared.
X—not affected.

MULS—Signed Multiply

Syntax: MULS <ea>,Dn

Size: Word

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.
X—Unaffected.

The low order word of the destination data register and the source operand are multiplied together using signed arithmetic. The 32-bit product is stored in the destination data register. Only Data addressing modes can be used for the source operand specified by the <ea> field.

MULU—Unsigned Multiply

Syntax: MULU <ea>,Dn

Size: Word

Condition Codes:

N—Set if the most significant bit of the result is set, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.
X—Unaffected.

The low order word of the destination data register, and the source operand indicated by the <ea> field are multiplied together using unsigned arithmetic. The 32-bit product is stored in the destination data register. Only data addressing modes can be used for the source <ea> field.

NBCD—Negate Decimal with Extend

Syntax: NBCD <ea>

Size: Byte

Condition Codes:

N—Undefined.
Z—Cleared if the result is non-zero, otherwise unchanged.
V—Undefined.
C—Set if a (decimal) borrow was generated, otherwise cleared.
X—Set the same as the carry bit.

The destination operand specified by the <ea> field and the extend bit are subtracted from zero and the result is stored in the destination location. The subtraction is done using binary-coded-decimal (BCD) arithmetic. If the extend bit is clear, the operation produces the ten's complement of the operand. If the extend bit is set, the operation generates the nine's complement of the operand. Only data alterable addressing modes can be used in this instruction.

NEG—Negate

Syntax: NEG <ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a borrow is generated, otherwise cleared.
X—Set the same as the carry bit.

The destination operand specified by the <ea> field is subtracted from zero and the result is stored in the destination location. Only data alterable addressing modes can be used in this instruction.

NEGX—Negate with Extend

Syntax: NEGX <ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a borrow is generated, otherwise cleared.
X—Set the same as the carry bit.

The destination operand specified by the <ea> field and the extend bit are subtracted from zero and the result is stored in the destination location. Only data alterable addressing modes can be used in this instruction.

NOP—No Operation

Syntax: NOP

Size: Unsized

The NOP instruction does not affect any condition codes.

This instruction performs no operation. No processor state other than the program counter are affected.

NOT—Logical Complement

Syntax: NOT <ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.
X—Unaffected.

The destination operand specified by the <ea> field is logically complemented and the result is stored in the destination location. Only data alterable addressing modes can be used in this instruction.

OR—Logical Inclusive OR

Syntax: OR <ea>,Dn
OR Dn,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the most significant bit of the result is set, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.
X—Unaffected.

The source and destination operands are logically ORed and the result is stored in the destination location. Address registers may not be used as operands. If the <ea> field is a source operand, only data addressing modes can be used. If the <ea> field is a destination operand, only alterable memory addressing modes can be used.

ORI—Logical Inclusive OR Immediate

Syntax: ORI #<data>,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the most significant bit of the result is set, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.

The immediate data field is logically ORed with the destination operand. The result is stored in the destination location.

ORI to CCR or SR

The effective address field <ea> in the ORI instruction can refer to either the condition codes register (CCR) or the status register (SR). Access to the status register is in one of two modes:

- If the size field of the instruction is byte, the operation only affects the condition codes register—the low order bits of the status register.
- If the size field of the instruction is word, the operation affects the entire status register. This is a privileged operation and can only be issued by a program running in supervisor state.

PEA—Push Effective Address

Syntax: PEA <ea>

Size: Long

The PEA instruction does not affect any condition codes.

The effective (long word) address is computed, then pushed onto the top of the stack.

RESET—Reset External Devices

Syntax: RESET

Size: Unsized

The RESET instruction does not affect any condition codes.

The RESET instruction asserts the reset line on the processor bus, causing all external devices to be reset. This instruction is a privileged instruction and can only be issued by a program running in supervisor state.

ROL and ROR—Rotate without Extend

Syntax: ROd Dx,Dy
ROd #<data>,Dy
ROd <ea>

Size: Byte, Word or Long

Condition Codes:

- N—Set if the most significant bit of the result is set, otherwise cleared.
- Z—Set if the result is zero, otherwise cleared.
- V—Always cleared.
- C—Set according to the last bit shifted out of the operand. It is cleared for a rotate count of zero.
- X—Unaffected.

Bits of the operand are rotated in the direction specified by 'd'—R for right and L for left. Bits shifted out one end of the operand are shifted back in at the other end. The extend bit is not included in the rotate.

Either a data register or a memory location can be rotated. If the operand is a memory location, the rotate count is 1. Only memory alterable addressing modes can be used for this form.

If the operand is a data register, the rotate count can be specified in one of two ways:

Immediate The rotate count is specified by the immediate <data> field in the instruction. In this case, the rotate count must lie in the range 1 .. 8. This is the second form of the instruction given in the syntax above.

Register The rotate count is contained in data register specified in the instruction. This is the first form of the instruction given in the syntax above.

ROXL and ROXR—Rotate with Extend

Syntax: ROXd Dx,Dy
ROXd #<data>,Dy
ROXd <ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the most significant bit of the result is set, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Always cleared.
C—Set according to the last bit shifted out of the operand. Set to the value of the extend bit for a rotate count of zero.
X—Set according to the last bit shifted out of the operand. Unaffected by a rotate count of zero.

Bits of the operand are rotated in the direction specified by 'd'—R for right and L for left. Bits shifted out one end of the operand go into the carry and extend bits. The previous value of the extend bit is shifted into the other end of the operand.

Either a data register or a memory location can be rotated. If the operand is a memory location, the rotate count is 1. Only memory alterable addressing modes can be used for this form.

If the operand is a data register, the rotate count can be specified in one of two ways:

Immediate The rotate count is specified by the immediate <data> field in the instruction. In this case, the rotate count must lie in the range 1 .. 8. This is the second form of the instruction given in the syntax above.

Register The rotate count is contained in data register specified in the instruction. This is the first form of the instruction given in the syntax above.

RTE—Return from Exception (Privileged)

Syntax: RTE

Size: Unsized

Condition Codes:

N—Set per the contents of the word on the stack.
Z—Set per the contents of the word on the stack.
V—Set per the contents of the word on the stack.
C—Set per the contents of the word on the stack.
X—Set per the contents of the word on the stack.

The RTE instruction is used to effect a return to the previous program state after processing any form of exception (interrupt, trap and such). The status register and program counter are popped from the system stack. The previous contents of the status register and program counter are overwritten. All bits of the status register are affected by this instruction.

This is a privileged instruction and may only be issued by a program running in supervisor state.

RTR—Return and Restore Condition Codes

Syntax: RTR

Size: Unsized

Condition Codes:

N—Set per the contents of the word on the stack.
Z—Set per the contents of the word on the stack.
V—Set per the contents of the word on the stack.
C—Set per the contents of the word on the stack.
X—Set per the contents of the word on the stack.

This instruction is used to effect a return from a subroutine. The main difference from the RTS instruction is that the condition codes are popped from the stack and are loaded into the condition codes register, overwriting the previous contents.

RTS—Return from Subroutine

Syntax: RTS

Size: Unsized

The RTS instruction does not affect any of the condition codes.

The RTS instruction is used to effect a return from a subroutine. The program counter is popped from the top of the stack. The previous value of the program counter is lost.

SBCD—Subtract Decimal with Extend

Syntax: SBCD Dy,Dx

Size: Byte

Condition Codes:

N—Undefined.
Z—Cleared if the result is non-zero, otherwise unchanged.
V—Undefined.
C—Set if a (decimal) borrow was generated, otherwise cleared.
X—Set the same as the carry bit.

The source operand plus the X (extend) bit is subtracted from the destination operand and the result is stored in the destination. The subtraction is done using binary-coded-decimal (BCD) arithmetic. The operands are addressed in one of two ways:

- Data register to data register. The operands are contained in the specified data registers.
- Memory to memory. The operands are addressed via the Address Register Indirect with Pre-decrement addressing mode, using the address registers specified in the instruction.

Scc—Set According to Condition

Syntax: Scc <ea>

Size: Byte

The Scc instruction does not affect any condition codes.

The Scc instruction tests the condition code specified by 'cc'. If the condition is true, the byte specified by the effective address <ea> is set to all ones. If the condition is false, the destination byte is set to all zeros. Only data alterable addressing modes can be used in the Scc instruction. The condition specified by 'cc' can be one of the following:

CC—Carry clear	LO—Lower (U)
CS —Carry set	LS —Low or same (U)
EQ—Equal	LT —Less than (S)
F —False (never true)	MI —Minus
GE—greater than or equal (S)	NE—Not equal
GT—greater than (S)	PL —Plus
HI —High (U)	T —Always True
HS—High or same (U)	VC—No overflow
LE —Less than or equal (S)	VS —Overflow

The notations (U) and (S) apply to operations which are Unsigned and Signed, respectively.

STOP—Load Status Register and Stop (Privileged)

Syntax: STOP #<data>

Size: Unsized

Condition Codes:

N—Set per the immediate operand.
Z—Set per the immediate operand.
V—Set per the immediate operand.
C—Set per the immediate operand.
X—Set per the immediate operand.

The STOP instruction executes a "dynamic halt". The immediate <data> operand is treated as a 16-bit value, is moved to the entire status register and the CPU stops fetching and executing instructions. Program execution resumes when a trace, interrupt or reset exception occurs.

If the STOP instruction is executed during trace state, a trace exception is raised.

Interrupts are accepted if the priority of the interrupt is higher than that of the current processor priority.

If the bit of the immediate <data> field which corresponds to the S (supervisor state) bit is off, a privilege violation interrupt occurs.

External reset always initiates reset-exception processing.

This is a privileged instruction and may only be issued by a program running in supervisor state.

SUB—Subtract Binary

Syntax: SUB <ea>,Dn
SUB Dn,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a borrow is generated, otherwise cleared.
X—Set the same as the carry bit.

The source operand is subtracted from the destination operand and the result is stored in the destination operand.

If the <ea> field is the source, all addressing modes may be used. The Address Register Direct addressing mode may not be used for a byte size operation. If the <ea> field is the destination, only alterable memory addressing modes may be used.

SUBA—Subtract Address

Syntax: SUBA <ea>,An

Size: Word or Long

The SUBA instruction does not affect any condition codes.

The source operand is subtracted from the destination address register and the result is stored in the address register. All addressing modes can be used for the source operand.

SUBI—Subtract Immediate

Syntax: SUBI #<data>,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a borrow is generated, otherwise cleared.
X—Set the same as the carry bit.

The immediate data field is subtracted from the destination operand and the result is stored in the destination location. Only data alterable addressing modes can be used.

SUBQ—Subtract Quick

Syntax: SUBQ #<data>,<ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a borrow is generated, otherwise cleared.
X—Set the same as the carry bit.

The immediate data field is subtracted from the destination operand and the result is stored in the destination location. The data field must lie in the range 1 .. 8. Only data alterable addressing modes can be used. The Address Register Direct addressing mode may not be used for byte size operations.

SUBX—Subtract Extended

Syntax: SUBX Dy,Dx
SUBX −(Ay), −(Ax)

Size: Byte, Word or Long

Condition Codes:

N—Set if the result is negative, otherwise cleared.
Z—Set if the result is zero, otherwise cleared.
V—Set if an overflow is generated, otherwise cleared.
C—Set if a borrow is generated, otherwise cleared.
X—Set the same as the carry bit.

The source operand plus the X (extend) bit are subtracted from the destination operand and the result is stored in the destination location. The operands can be addressed in one of two ways:

- Data register to data register. The operands are contained in the specified data registers.
- Memory to memory. The operands are addressed via the Address Register Indirect with Pre-decrement addressing mode, using the address registers specified in the instruction.

SWAP—Swap Register Halves

Syntax: SWAP Dn

Size: Word

Condition Codes:

N—Set if the most significant bit of the 32-bit result is set, otherwise cleared.
Z—Set if the 32-bit result is zero, otherwise cleared.
V and C—Always cleared.
X—Not affected.

SWAP exchanges the 16-bit words in a 32-bit data register. It is useful for exchanging the quotient and remainder after a divide instruction.

TAS—Test and Set

Syntax: TAS <ea>

Size: Byte

Condition Codes:

N—Set if the most significant bit of the operand is set, otherwise cleared.
Z—Set if the operand is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.
X—Unaffected.

The byte operand specified by the <ea> field is tested, and then the high order bit of the operand is set, all in one indivisible (uninterruptible) operation. The N and Z condition code bits are set according to the state of the operand when the test is performed. Only data alterable addressing modes can be used in the TAS instruction.

TRAP—Trap

Syntax: TRAP #<vector>

Size: Unsized

The TRAP instruction does not affect any condition codes.

The TRAP instruction initiates exception processing. The specific exception vector selected is determined by the trap vector number supplied as the immediate <vector> operand of the instruction. The <vector> number is a value in the range 0 .. 15.

TRAPV—Trap on Overflow

Syntax: TRAPV

Size: Unsized

The TRAPV instruction does not affect any condition codes.

If the overflow condition is set, the TRAPV instruction initiates exception processing to the TRAPV exception vector. If the overflow condition is off, the TRAPV instruction acts as a no-op.

TST—Test an Operand

Syntax: TST <ea>

Size: Byte, Word or Long

Condition Codes:

N—Set if the operand is negative, otherwise cleared.
Z—Set if the operand is zero, otherwise cleared.
V—Always cleared.
C—Always cleared.
X—Unaffected.

The operand specified by <ea> is compared with zero and the condition codes set as a result of the test. Only data alterable addressing modes can be used by the TST instruction.

UNLK—Deallocate Stack Space

Syntax: UNLK An

The UNLK instruction does not affect any condition codes.

The stack pointer is loaded from the specified address register. The long word on the top of the stack is then popped into the specified address register. See the LINK instruction.

USING THE ASSEMBLER

ASM68K is a relocatable assembler for the Motorola MC68000 processor. This chapter is a description of how to run the assembler on MERLIN.

ASM68K accepts source programs written in the Assembler Language as defined in this manual and generates self-relocatable object-code output. An optional listing file can be generated that contains the source statements, hexadecimal object-code, error diagnostics (if any) and a symbol cross-reference. The Assembler is run by giving the following command line:

```
asm68k sfile -llfile -oofile
```

where "sfile" is the name of a file containing the assembler source program. The source is expected to be on a file with a ".text" suffix. If the user omits the ".text" suffix from the source file name on the command list, ASM68K appends a ".text" suffix to the name before accessing the source file.

Command line *options* are designated by a " – " sign followed by a letter (shown underlined in the command line above). The argument for that option immediately follows the option letter as shown. The options are:

"lfile" Is the optional name of the file to receive the assembler generated listing. No suffix is appended to the listing filename. If omitted, a listing is not generated. In this case, errors are directed to the standard output and the standard input is polled for a "continue" or an "abort" response.

"ofile" Is the optional name of the file to receive the generated relocatable object-code. The generated object-code is placed on a file with an ".obj" suffix. If omitted from the command line, the object code output is written to a file with the same name as the source file (minus the ".text" suffix) with a suffix of ".obj" appended.

Examples of Running the Assembler

```
asm68k bookshelf
```

In this example, ASM68K assembles the source in the file "bookshelf.text". There is no listing file, so any errors are directed to the standard output. The generated object-code is placed on the file "bookshelf.obj".

```
asm68k cabinet -ldrawers.list -omodules
```

This example illustrates the use of the command line options. The source text is on a file called "cabinet.text", the listing appears on a file called "drawers.list" and the relocatable object code appears on a file called "modules. obj".

The object-code that ASM68K generates is not directly executable, but must be linked, using the linker, before it can be executed.

ASSEMBLER OUTPUT AND MESSAGES

This chapter describes the output files that the assembler generates, and contains the error messages that are generated when errors are encountered in the assembler source code.

Object Code File

ASM68K generates an object-code file containing relocatable object-code. The object-code file is not directly executable on the computer, but must be linked, using the linker, before it can be executed. Alternatively, the object-code file can be an input to the library utility, for incorporation into an object code library. Details of the linker and library utilities can be found in the linker manual.

Listing File

ASM68K generates an optional listing file which contains the source code as read, plus generated object code, error messages (if any) and a cross reference of the symbols in the assembly unit. The listing is paginated with page headings at the top of each page. Each line in the listing has a line number attached to it. A fragment of a listing file (not to scale) is shown here:

```
          001F0000   46*  IOBase68    EQU    $lf0000          ;Base address of 68K I/O.
          001F0080   47*  MSCBase     EQU    IOBase68+$80
          .....
                 ...... lots more assembler statements
                              .......
001E  7400          245*  NextTrak    moveq  #0,dl              ; Cylinder Number.
                    246*;
0020  1602          247*  NextHead    move.b d2,d3              ; Shift head number..
          .......
                 ........ then a line with an error message
                              ..........
***** Error 7—                                      v Undefined Symbol
0044  41FA  FFBA  262*  leaSuccess,a0  ; Finished—display message.
  1       2          3   4       5        6 ←Field Reference.
```

The significance of the fields in the above illustration are:

1. Shows the address of this code relative to the start of the assembly unit.

2. Shows the generated object code for the instruction. In the case of an EQU directive, the code field shows the value of the operand for the equate.
3. Shows the line number in that assembly unit.
4. Shows the opcode field.
5. Is the operand field for the instruction.
6. Is the comment field.

With the exception of the line number, all numbers are in hexadecimal.

At the end of the assembly listing, ASM68K generates an alphabetical list of all symbols defined in the assembly:

- Relocatable symbols are marked with a + sign.
- Multiply defined symbols are flagged with the word "DOUBLE".
- External symbols are marked by a row of asterisks: ******.
- Global symbols are preceded by a single asterisk character.
- Undefined symbols do not appear in the list.

```
^     ^     ^     ^     ^     ^
.     .     .     .     .     .
```

Assembler Error Messages

Illegal Character in Label

A label contains a character that is not valid in the context of a label. Remember that labels can only contain the letters A thru Z, a through z, the digits 0 through 9, and the characters _ and %. All other characters are invalid in labels.

Illegal Character

The assembler read a character that was unexpected at that point. Check if a string literal or data item was correctly written.

Opcode Expected

The assembler read something that was not an operation code, in a place where an operation code or assembler directive was expected. Check the opcode field to see if it is a valid opcode. Maybe there is a label that does not start in column one.

Absolute Value(s) Expected

The operand expression is either relocatable or it references an external symbol where the assembler expected an absolute expression. For example, the ADDI (Add Immediate) instruction expects an absolute expression for an operand.

')' Expected

A close parenthesis was expected to be read, and something else was found instead.

Illegal Symbol in Expression

An expression contains characters that are not part of the valid set of symbols that can appear in an expression. Check the rules for operand expressions.

Undefined Symbol

A symbol used as an operand is undefined. Check to see if it is defined, or check for possible spelling mistakes.

Absolute or Relocatable Value Expected

An external symbol is referenced when the assembler expects only an absolute or relocatable expression.

Illegal Extension

The 'size' extension of an operation or operand is not a valid extension, or is not valid in the particular context. Valid extensions are B, W, L and S. For example, the B size attribute cannot be used as the extension of a BRA instruction.

',' Expected

Missing comma in an argument list.

Data Register Expected

A data register designator in the range D0 through D7 is expected.

Label Required

The particular operation or directive requires a label field. For example, the EQU directive must have a label field.

Multiply Defined Symbol

A label has been encountered that is already defined previously. Check for possible spelling mistakes.

.W or .L Expected

The specific operation code requires a size designator of W or L.

Address Register Expected

An address register in the range A0 through A7 (or SP) is expected as the operand of the instruction.

'(' Expected

An opening parenthesis is expected at this point in the operand field.

Register Expected

The instruction operand should be either an address register, a data register, or one of the system special registers.

Value Must be in Range −128 .. 127

Some instructions (such as MOVEQ—Move Quick), can only accept single byte operands. Check the operand field or expression for validity.

Relocatable Value Expected

An absolute or external-valued expression occurs where the assembler is expecting to find a relocatable expression. For example, the BRA instruction expects a relocatable operand, not an absolute or external operand.

Strings Not Allowed

A string constant cannot be used as the operand of this instruction. For example, the JMP instruction expects an address not a string.

'#' Expected

A literal or immediate operand must be signalled by placing a # sign in front of it.

Value Must be in Range 1 .. 8

Instructions such as ASL (arithmetic shift left) only accept literal or immediate operands in the range 1 through 8. If a greater shift count is required, multiple shift instructions must be used or the shift count must be placed in a register.

Illegal Operand Mode

The operand used in this instruction is the wrong kind. For example, in the CMPM (Compare Memory) instruction, the operands must be addressed using the Address Register Indirect with Post-increment addressing mode.

Unterminated String

A character string data item or literal does not have a correct closing string delimiter.

Object File Full

There is no more room on the output storage device for the file that is accumulating the generated object code. At this juncture, consider splitting the program into smaller modules.

Cannot EQU to an External

An EQU directive cannot have an external label as its operand.

Identifier Expected

An identifier is the only valid object which can occur at this point. For example, the GLOBAL directive expects only identifiers and not expressions.

Value Must be in Range 0 .. 15

The TRAP instruction only accepts a trap number in the range 0 through 15.

Extra Junk at End of Line

There is text on the statement line, after the instruction is considered 'complete', that is not required in the context of the particular instruction. Check to see if there is a comment on the line without a comment delimiter in front of it.

Missing END Statement

The assembler read the end-of-file in the assembly unit before an END statement was found.

Cannot Have Zero Offset

The BRA (Branch Always), Bcc (Branch on Condition) and BSR (Branch to Subroutine) instructions must not have an offset which is zero. In practical terms this means that these instructions must not branch to the location immediately following the branch instruction itself.

Hex Constants Begin with $

The assembler started to read what it assumed was a decimal number and the number turned out to have hexadecimal digits in it (72A for example).

Appendix A

ALPHABETICAL INSTRUCTION SUMMARY

Here is an alphabetical summary of the instructions, their operand forms and the condition codes affected. The entries under the condition code field have the following meanings:

– Unaffected.	* Set or cleared.
U Undefined.	0 Always cleared.

			Condition Codes				
Mnemonic	Operation	Assembler Syntax	X	N	Z	V	C
ABCD	Add Decimal with Extend	ABCD Dy,Dx ABCD –(Ay),–(Ax)	*	U	*	U	*
ADD	Add Binary	ADD <ea>,Dn ADD Dn,<ea>	*	*	*	*	*
ADDA	Add Address	ADDA <ea>,An	–	–	–	–	–
ADDI	ADD Immediate	ADDI #<data>,<ea>	*	*	*	*	*
ADDQ	Add Quick	ADDQ #<data>,<ea>	*	*	*	*	*
ADDX	Add Extended	ADDX Dy,Dx ADDX –(Ay),–(Ax)	*	*	*	*	*
AND	AND Logical	AND <ea>,Dn AND Dn,<ea>	–	*	*	0	0
ANDI	AND Immediate	ANDI #<data>,<ea>	–	*	*	0	0
ASL, ASR	Arithmetic Shift	ASd Dx,Dy ASd #data,Dy ASd <ea>	*	*	*	*	*

Mnemonic	Operation	Assembler Syntax	Condition Codes				
			X	N	Z	V	C
Bcc	Branch Conditionally	Bcc <label>	–	–	–	–	–
BCHG	Test a Bit and Change	BCHG Dn,<ea> BCHG #<data>,<ea>	–	–	*	–	–
BCLR	Test a Bit and Clear	BCLR Dn,<ea> BCLR #<data>,<ea>	–	–	*	–	–
BRA	Branch Always	BRA <label>	–	–	–	–	–
BSET	Test a Bit and Set	BSET Dn,<ea> BSET #<data>,<ea>	–	–	*	–	–
BSR	Branch to Subroutine	BSR <label>	–	–	–	–	–
BTST	Test a Bit	BTST Dn,<ea> BTST #<data>,<ea>	–	–	*	–	–
CHK	Check Register Against Bounds	CHK <ea>,Dn	–	*	U	U	U
CLR	Clear an Operand	CLR <ea>	–	0	1	0	0
CMP	Arithmetic Compare	CMP <ea>,Dn	–	*	*	*	*
CMPA	Arithmetic Compare Address	CMPA <ea>,An	–	*	*	*	*
CMPI	Compare Immediate	CMPI #<data>,<ea>	–	*	*	*	*
CMPM	Compare Memory	CMPM (Ay)+,(Ax)+	–	*	*	*	*
DBcc	Test Condition, Decrement and Branch	DBcc Dn,<label>	–	–	–	–	–
DIVS	Signed Divide	DIVS <ea>,Dn	–	*	*	*	0
DIVU	Unsigned Divide	DIVU <ea>,Dn	–	*	*	*	0

Mnemonic	Operation	Assembler Syntax	Condition Codes				
			X	N	Z	V	C
EOR	Logical Exclusive OR	EOR Dn,<ea>	–	*	*	0	0
EORI	Exclusive OR Immediate	EORI #<data>,<ea>	–	*	*	0	0
EXG	Exchange Registers	EXG Rx,Ry	–	–	–	–	–
EXT	Sign Extend	EXT Dn	–	*	*	0	0
JMP	Jump	JMP <ea>	–	–	–	–	–
JSR	Jump to Subroutine	JSR <ea>	–	–	–	–	–
LEA	Load Effective Address	LEA <ea>,An	–	–	–	–	–
LINK	Link and Allocate	LINK An,#<disp>	–	–	–	–	–
LSL, LSR	Logical Shift	LSd Dx,Dy LSd #<data>,Dy LSd <ea>	*	*	*	0	*
MOVE	Move Data from Source to Destination	MOVE <ea>,<ea>	–	*	*	0	0
MOVE to CCR	Move to Condition Codes	MOVE <ea>,CCR	*	*	*	*	*
MOVE to SR	Move to Status Register	MOVE <ea>,SR	*	*	*	*	*
MOVE from SR	Move from Status Register	MOVE SR,<ea>	–	–	–	–	–
MOVE USP	Move User Stack Pointer	MOVE USP,An	–	–	–	–	–
MOVEA	Move Address	MOVEA <ea>,An	–	–	–	–	–
MOVEM	Move Multiple Registers	MOVEM<rlist>,<ea> MOVEM<ea>,<rlist>	–	–	–	–	–
MOVEP	Move Peripheral Data	MOVEP Dx,d(Ay) MOVEP d(Ay),Dx	–	–	–	–	–
MOVEQ	Move Quick	MOVEQ #<data>,Dn	–	*	*	0	0

Mnemonic	Operation	Assembler Syntax	Condition Codes				
			X	N	Z	V	C
MULS	Signed Multiply	MULS <ea>,Dn	–	*	*	0	0
MULU	Unsigned Multiply	MULU <ea>,Dn	–	*	*	0	0
NBCD	Negate Decimal with Expand	NBCD <ea>	*	U	*	U	
NEG	Two's Complement Negate	NEG <ea>	*	*	*	*	*
NEGX	Negate with Extend	NEGX <ea>	*	*	*	*	*
NOP	No Operation	NOP	–	–	–	–	–
NOT	Logical Complement	NOT <ea>	–	*	*	0	0
OR	Logical Inclusive OR	OR <ea>,Dn OR Dn,<ea>	–	*	*	0	0
ORI	Inclusive OR Immediate	ORI #<data>,<ea>	–	*	*	0	0
PEA	Push Effective Address	PEA <ea>	–	–	–	–	–
RESET	Reset External Devices	RESET	–	–	–	–	–
ROL, ROR	Rotate without Extend	ROd Dx,Dy ROd #<data>,Dy ROd <ea>	–	*	*	0	*
ROXL, ROXR	Rotate with Extend	ROXd Dx,Dy ROXd #<data>,Dy ROXd <ea>	*	*	*	0	*
RTE	Return From Exception	RTE	*	*	*	*	*
RTR	Return and Restore Condition Codes	RTR	*	*	*	*	*
RTS	Return From Subroutine	RTS	–	–	–	–	–

Mnemonic	Operation	Assembler Syntax	Condition Codes X	N	Z	V	C
SBCD	Subtract Decimal with Extend	SBCD Dy,Dx SBCD – (Ay), – (Ax)	*	U	*	U	*
Scc	Set According to Codes	Scc <ea>	–	–	–	–	–
STOP	Stop Program Execution	STOP #<data>	–	–	–	–	–
SUB	Subtract Binary	SUB <ea>,Dn SUB Dn,<ea>	*	*	*	*	*
SUBA	Subtract Address	SUBA <ea>,An	–	–	–	–	–
SUBI	Subtract Immediate	SUBI #<data>,<ea>	*	*	*	*	*
SUBQ	Subtract Quick	SUBQ #<data>,<ea>	*	*	*	*	*
SUBX	Subtract with Extend	SUBX Dy,Dx SUBX – (Ay), – (Ax)	*	*	*	*	*
SWAP	Swap Register Halves	SWAP Dn –	*	*	0	0	
TAS	Test and Set an Operand	TAS <ea>	–	*	*	0	0
TRAP	Trap	TRAP #<vector>	–	–	–	–	–
TRAPV	Trap on Overflow	TRAPV	–	–	–	–	–
TST	Test an Operand	TST <ea>	–	*	*	0	0
UNLK	Unlink	UNLK An	–	–	–	–	–

<rlist> in the MOVEM instruction specifies the registers selected for transfer to or from memory. <rlist> may be:

- a single register,
- a range of consecutive registers indicated by low-high.
- Any combination of the above two items. Each element in the list is separated from the next by a slash.

Appendix B

ALPHABETICAL LIST OF DIRECTIVES

DATA	defines and intializes data items.
END	signals the end of an assembly unit.
EQU	associates a value with an identifier.
EXTERN	defines an identifier external to the current assembly unit that is referenced in that unit.
GLOBAL	defines an identifier in the current assembly unit that is referenced from outside that unit.
IDENT	specifies the name of an assembly unit.
LIST	controls assembler listing on or off.
PAGE	produces a page eject on the assembler listing.

Appendix C

LIST OF OPERANDS

An	Address Register. 'n' may be 0 .. 7. A7 is also the Stack Pointer.
Dn	Data Register. 'n' may be 0 .. 7.
SP	Stack Pointer, A7.
USP	User Stack Pointer.
CCR	Condition Codes Register. The CCR is an 8-bit register. It is actually the low order eight bits of the status register.
SR	Status Register. The SR is a 16-bit register which includes the CCR. The status bits (most significant) portion of the SR can only be changed by a program running in supervisor state.

Writing Characters to the Screen

```
CONOUT   MOVE.W   #CONSOLE,-(SP)    ; push unit # of device
         PEA      OUTSTR            ; push address of string to write
         MOVE.W   #STRLEN,-(SP)     ; push # of chars to write
         CLR.W    -(SP)             ; push block # (N/A for console)
         CLR.W    -(SP)             ; push mode
         CLR.W    D0                ; D0 = offset of unitwrite vector
         MOVE.L   PSYSCOM.W,A0      ; A0==> syscom
         MOVE.L   SYSVECT(A0),A0    ; A0==> system vector table
         MOVE.L   0(A0,D0.W),A0     ; A0==> unitwrite routine
         JSR      (A0)              ; call unitwrite
         RTS                        ; done

CONSOLE  EQU      1                 ; unit number of /CONSOLE device
OUTSTR   DATA.B   'This is a test.' ; string to write
         DATA.B   $0D,$0A           ; carriage return, linefeed
STRLEN   EQU      %-OUTSTR          ; length of string
PSYSCOM  EQU      $180              ; address of syscom pointer
SYSVECT  EQU      8                 ; offset of vector table pointer

         END      CONOUT
```

Making Unitxxxx System Calls From 68K Assembler

function	data to push,	length	vector offset
UnitRead,	unit number	W	4 (read)
UnitWrite	buffer address	L	0 (write)
	buffer length	W	
	block number	W	
	mode	W	
UnitStatus	unit number	W	100
	buffer address	L	
	command	W	
UnitClear	unit number	W	8
UnitBusy	unit number	W	12 ***returns byte on stack (boolean T=1,F=0)