SYSTEM V/68
ASSEMBLER USER'S GUIDE

Third Edition

First Edition June 1984
Second Edition November 1984

This manual is reprinted in its entirety by ICON INTERNATIONAL with permission of Motorola Inc., 1987

Trademarks

UNIX is a registered trademark of AT&T.

EXORmacs, EXORterm, and SYSTEM V/68 are trademarks of Motorola, Inc.

VAX is a trademark of Digital Equipment Corporation.

1. INTRODUCTION

This is a reference manual for the MPS/UX resident assembler, *as*. Programmers familiar with the M68000 family of processors should be able to program in *as* by referring to this manual, but this is not a manual for the processor itself. Details about the effects of instructions, meanings of status register bits, handling of interrupts, and many other issues are not dealt with here. This manual, therefore, should be used in conjunction with the following reference manuals:

- M68000 16/32-Bit Microprocessor Programmer's Reference Manual, Fourth Edition; Englewood Cliffs, NJ: PRENTICE-HALL, 1984. This manual is also available from the Motorola Literature Distribution Center, P.O. Box 20912, Phoenix, AZ 85036, part number M68000UM.
- MC68020 32-Bit Microprocessor User's Manual; Englewood Cliffs, NJ: PRENTICE-HALL, 1984. This manual is also available from the Motorola Literature Distribution Center, part number MC68020UM.
- M68000 Family Resident Structured Assembler Reference Manual, part number M68KMASM.
- SYSTEM V/68 User's Manual, part number M68KUNUM.
- SYSTEM V/68 VM04 System Manual, part number M68KVM4SYS. This document includes user manual pages to support the MC68881 floating point co-processor provided in SYSTEM V/68 Release 2, Version 2 from Motorola Corp.

This guide also contains information for users of the SGS M68020 Cross Compilation System. For these users, references to *as*(1) and *cc*(1) should be read as *as20*(1) and *cc20*(1). Information about these commands is provided in the SGS M68020 Cross Compilation System Reference Manual, part number M68KUNASX.

2. WARNINGS

A few important warnings to the *as* user should be emphasized at the outset. Though for the most part there is a direct correspondence between *as* notation and the notation used in the documents listed in the preceding section, several exceptions exist that could lead the unsuspecting user to write incorrect code. In addition to the exceptions described in the following paragraphs, refer also to sections 10 and 11 for information about address mode syntax and machine instructions.

2.1. Comparison Instructions

First, the order of the operands in *compare* instructions follows one convention in the M68000 Programmer's Reference Manual and the opposite convention in *as*. Using the convention of the M68000 Programmer's Reference Manual, one might write

```
CMP.W     D5,D3      Is D3 less than D5?
BLE       IS_LESS    Branch if less.
```

Using the *as* convention, one would write

```
cmp.w     %d3,%d5    # Is d3 less than d5?
ble       is_less    # Branch if less.
```

As follows the convention used by other assemblers supported in the UNIX® operating system (both the 3B20S and the VAX follow this convention). This convention makes for straightforward reading of *compare*-and-*branch* instruction sequences, but does nonetheless lead to the peculiarity that if a *compare* instruction is replaced by a *subtract* instruction, the effect on the condition codes will be entirely different. This may be confusing to programmers who are used to thinking of a comparison as a subtraction whose result is not stored. Users of *as* who become accustomed to the convention will find that both the *compare* and *subtract* notations make sense in their respective contexts.

2.2. Overloading of Opcodes

Another issue that users must be aware of arises from the M68000 processors' use of several different instructions to do more or less the same thing. For example, the M68000 Programmer's Reference Manual lists the instructions **SUB, SUBA, SUBI,** and **SUBQ**, which all have the effect of subtracting their source operand from their destination operand. *As* provides the convenience of allowing all these operations to be specified by a single assembly instruction **sub**. On the basis of the operands given to the **sub** instruction, the *as* assembler selects the appropriate M68000 operation code. The danger created by this convenience is that it could leave the misleading impression that all forms of the **SUB** operation are semantically identical. In fact, they are not. The careful reader of the M68000 Programmer's Reference Manual will notice that whereas **SUB, SUBI,** and **SUBQ** all affect the condition codes in a consistent way, **SUBA** does not affect the condition codes at all. Consequently, the *as* user must be aware that when the destination of a **sub** instruction is an address register (which causes the **sub** to be mapped into the operation code for **SUBA**), the condition codes will not be affected.

3. USE OF THE ASSEMBLER

The SYSTEM V/68 command *as* invokes the assembler and has the following syntax:

as [**-o** *output*] *file*

When *as* is invoked with the **-o** *output* flag, the output of the assembly is put in the file *output*. If the -o flag is not specified, the output is left in a file whose name is formed by removing the **.s** suffix, if there is one, from the input filename and appending a **.o** suffix.

The M68020 cross assembler, *as20*(1), is invoked with the same syntax as *as*(1). For information about additional options for these commands, refer to the *SYSTEM V/68 User's Manual* for *as*(1) and the *SGS M68020 Cross Compilation System Reference Manual* for *as20*(1).

UNIX is a registered trademark of AT&T.

4. GENERAL SYNTAX RULES

4.1. Format of Assembly Language Line

Typical lines of *as* assembly code look like these:

```
# Clear a block of memory at location %a3

        text    2
        move.w  &const,%d1
loop:   clr.l   (%a3)+
        dbf     %d1,loop        # go back for const
                                # repetitions

init2:
        clr.l count;  clr.l credit;  clr.l debit;
```

These general points about the example should be noted:

— An identifier occurring at the beginning of a line and followed by a colon (:) is a *label*. One or more *labels* may precede any assembly language instruction or pseudo-operation. Refer to Section 5.2, "Location Counters and Labels."

— A line of assembly code need not include an instruction. It may consist of a comment alone (introduced by #), a label alone (terminated by :), or it may be entirely blank.

— It is good practice to use tabs to align assembly language operations and their operands into columns, but this is not a requirement of the assembler. An opcode may appear at the beginning of the line, if desired, and spaces may precede a label. A single blank or tab suffices to separate an opcode from its operands. Additional blanks and tabs are ignored by the assembler.

— It is permissible to write several instructions on one line separating them by semicolons. The semicolon is syntactically equivalent to a newline character; however, a semicolon inside a comment is ignored.

4.2. Comments

Comments are introduced by the character # and continue to the end of the line. Comments may appear anywhere and are completely disregarded by the assembler.

4.3. Identifiers

An identifier is a string of characters taken from the set **a-z, A-Z, _, ~, %,** and **0-9**. The first character of an identifier must be a letter (uppercase or lowercase) or an underscore. Uppercase and lowercase letters are distinguished; for example, **con35** and **CON35** are two distinct identifiers.

There is no limit on the length of an identifier.

The value of an identifier is established by the **set** pseudo-operation (refer to Section 8.2, "Symbol Definition Operations") or by using it as a label. Refer to Section 5.2, "Location Counters and Labels".

The tilde character (~) has special significance to the assembler. A ~ used alone, as an identifier, means "the current location". A ~ used as the first character in an identifier

becomes a period (.) in the symbol table, allowing symbols such as **.eos** and **.Ofake** to be entered into the symbol table, as required by the Common Object File format (COFF). Information about file formats is provided in the *SYSTEM V/68 User's Manual,* Section 4.

4.4. Register Identifiers

A register identifier is an identifier preceded by the character **%**, and represents one of the MC68000 processor's registers. The predefined register identifiers are;

%d0	%d4	%a0	%a4	%acc	%usp
%d1	%d5	%a1	%a5	%pc	%fp
%d2	%d6	%a2	%a6	%sp	
%d3	%d7	%a3	%a7	%sr	

Note: The identifiers **%a7** and **%sp** represent the same machine register. Likewise, **%a6** and **%fp** are equivalent. Use of both **%a7** and **%sp**, or **%a6** and **%fp**, in the same program may result in confusion.

The current version of the assembler will correctly assemble instructions intended for the M68010. There will be a warning message issued. The following additions will be flagged with warnings:

REGISTERS ADDED FOR THE MC68010	
NAME	DESCRIPTION
%sfc	Source Function Code Register
%dfc	Destination Function Code Register
%vbr	Vector Base Register

The entire register set of the MC68000 and MC68010 is included in the MC68020 register set. The following are new control registers for the MC68020:

MC68020 REGISTERS	
NAME	DESCRIPTION
%caar	Cache Address Register
%cacr	Cache Control Register
%isp	Interrupt Stack Pointer
%msp	Master Stack Pointer

The following are suppressed registers (zero registers) used in various MC68020 addressing modes:

MC68020 ZERO REGISTERS		
SUPPRESSED ADDRESS REGISTERS	SUPPRESSED DATA REGISTERS	SUPPRESSED PROGRAM COUNTER
%za0	%zd0	%zpc
%za1	%zd1	
%za2	%zd2	
%za3	%zd3	
%za4	%zd4	
%za5	%zd5	
%za6	%zd6	
%za7	%zd7	

4.5. Constants

As deals only with integer constants. They may be entered in decimal, octal, or hexadecimal, or they may be entered as character constants. Internally, *as* treats all constants as 32-bit binary two's complement quantities.

4.5.1. Numerical Constants. A decimal constant is a string of digits beginning with a non-zero digit. An octal constant is a string of digits beginning with zero. A hexadecimal constant consists of the characters **0x** or **0X** followed by a string of characters from the set **0-9, a-f,** and **A-F**. In hexadeximal constants, uppercase and lowercase letters are not distinguished.

Examples:

```
set     const,35      # Decimal 35
mov.w   &035,%d1      # Octal 35 (decimal 29)
set     const, 0x35   # Hex 35 (decimal 53)
mov.w   &0xff, %d1    # Hex ff (decimal 255)
```

4.5.2. Character Constants. An ordinary character constant consists of a single-quote character (') followed by an arbitrary ASCII character other than the backslash (\). The value of the constant is equal to the ASCII code for the character. Special meanings of characters are overridden when used in character constants; for example, if '# is used, the # is not treated as introducing a comment.

A special character constant consists of '\ followed by another character. All the special character constants and examples of ordinary character constants are listed in the following table.

CONSTANT	VALUE	MEANING
'\ b	0x08	Backspace
'\ t	0z09	Horizontal Tab
'\ n	0x0a	Newline (Line Feed)
'\ v	0x0b	Vertical Tab
'\ f	0x0c	Form Feed
'\ r	0x0d	Carriage Return
'\\	0x5c	Backslash
' '	0x27	Single Quote
'0	0x30	Zero
'A	0x41	Uppercase A
'a	0x61	Lowercase a

4.6. Other Syntactic Details

A discussion of expression syntax appears in Section 7 of this guide. Information about the syntax of specific components of *as* instructions and pseudo-operations is given in Sections 8, 9, and 10.

5. SEGMENTS, LOCATION COUNTERS, AND LABELS

5.1. Segments

A program in *as* assembly language may be broken into segments known as *text, data* and *bss* segments. The convention regarding the use of these segments is to place instructions in *text* segments, initialized data in *data* segments, and uninitialized data in *bss* segments. However, the assembler does not enforce this convention; for example, it permits intermixing of instructions and data in a *text* segment.

Primarily to simplify compiler code generation, the assembler permits up to four separate *text* segments and four separate *data* segments named **0, 1, 2,** and **3**. The assembly language program may switch freely between them by using assembler pseudo-operations (refer to Section 8.3, "Location Counter Control Operations"). When generating the object file, the assembler concatenates the *text* segments to generate a single *text* segment, and the *data* segments to generate a single *data* segment. Thus, the object file contains only one *text* segment and only one *data* segment. There is always only one *bss* segment and it maps directly into the object file.

Because the assembler keeps together everything from a given segment when generating the object file, the order in which information appears in the object file may not be the same as in the assembly language file. For example, if the data for a program consisted of

```
data    1           # segment 1
short   0x1111
data    0           # segment 0
long    0xff ff ff ff
data    1           # segment 1
byte    0xff
```

then equivalent object code would be generated by

```
data    0
long    0xffffffff
short   0x1111
byte    0xff
```

5.2. Location Counters and Labels

The assembler maintains separate *location counters* for the *bss* segment and for each of the *text* and *data* segments. The location counter for a given segment is incremented by one for each byte generated in that segment.

The location counters allow values to be assigned to labels. When an identifier is used as a label in the assembly language input, the current value of the current location counter is assigned to the identifier. The assembler also keeps track of which segment the label appeared in. Thus, the identifier represents a memory location relative to the beginning of a particular segment. Any label relative to the location counter should be within the text segment.

6. TYPES

Identifiers and expressions may have values of different types.

— In the simplest case, an expression (or identifier) may have an *absolute* value, such as 29, -5000, or 262143.

— An expression (or identifier) may have a value relative to the start of a particular segment. Such a value is known as a *relocatable* value. The memory location represented by such an expression cannot be known at assembly time, but the relative values of two such expressions (i.e., the difference between them) can be known if they refer to the same segment.

  Identifiers which appear as labels have *relocatable* values.

— If an identifier is never assigned a value, it is assumed to be an *undefined external.* Such identifiers may be used with the expectation that their values will be defined in another program, and therefore known at load time; but the relative values of *undefined externals* cannot be known.

7. EXPRESSIONS

For conciseness, the following abbreviations are useful:

abs	absolute expression
rel	relocatable expression
ext	undefined external

All constants are absolute expressions. An identifier may be thought of as an expression having the identifier's type. Expressions may be built up from lesser expressions using the operators **+**, **-**, *****, and **/**, according to the following type rules:

abs + abs = abs
abs + rel = rel + abs = rel
abs + ext = ext + abs = ext

abs — abs = abs
rel — abs = rel
ext — abs = ext
rel — rel = abs (provided that
the two relocatable expressions are relative to the same segment)

abs * abs = abs

abs / abs = abs

— abs = abs

Note: **rel — rel** expressions are permitted only within the context of a switch statement (refer to Section 8.5, "Switch Table Operation".) Use of a **rel — rel** expression is dangerous, particularly when dealing with identifiers from *text* segments. The problem is that the assembler will determine the value of the expression before it has resolved all questions concerning span-dependent optimizations.

The unary minus operator takes the highest precedence; the next highest precedence is given to * and /, and lowest precedence is given to + and -. Parentheses may be used to coerce the order of evaluation.

If the result of a division is a positive non-integer, it will be truncated toward zero. If the result is a negative non-integer, the direction of truncation cannot be guaranteed.

8. PSEUDO-OPERATIONS

8.1. Data Initialization Operations

byte *abs,abs,...*

One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce successive bytes in the assembly output.

short *abs,abs,...*

One or more arguments, separated by commas, may be given. The values of the arguments are computed to produce successive 16-bit words in the assembly output.

long *expr,expr,...*

One or more arguments, separated by commas, may be given. Each expression may be *absolute*, *relocatable*, or *undefined external*. A 32-bit quantity is generated for each such argument (in the case of *relocatable* or *undefined external* expressions, the actual value may not be filled in until load time).

Alternatively, the arguments may be bit-field expressions. A bit-field expression has the form

n : *value*

where both *n* and *value* denote *absolute* expressions. The quantity *n* represents a field width; the low-order *n* bits of *value* become the contents of the bit-field.

Successive bit-fields fill up 32-bit long quantities starting with the high-order part. If the sum of the lengths of the bit-fields is less than 32 bits, the assembler creates a 32-bit long with zeroes filling out the low-order bits. For example,

```
long        4: -1, 16: 0x7f, 12:0, 5000
```

and

```
long        4: -1, 16: 0x7f, 5000
```

are equivalent to

```
long        0xf007f000, 5000
```

Bit-fields may not span pairs of 32-bit longs. Thus,

```
long        24: 0xa, 24: 0xb, 24:0xc
```

yields the same thing as

```
long        0x00000a00, 0x00000b00, 0x00000c00
```

space *abs*

The value of *abs* is computed, and the resultant number of bytes of zero data is generated. For example,

```
space       6
```

is equivalent to

```
byte        0,0,0,0,0,0
```

8.2. Symbol Definition Operations

set *identifier,expr*

The value of *identifier* is set equal to *expr*, which may be absolute or relocatable.

comm *identifier,abs*

The named identifier is to be assigned to a common area of size *abs* bytes. If *identifier* is not defined by another program, the loader will allocate space for it.

lcomm *identifier,abs*

The named *identifier* is assigned to a *local common* of size *abs* bytes. This results in allocation of space in the *bss* segment.

The type of *identifier* becomes *relocatable*.

global *identifier*

This causes *identifier* to be externally visible. If *identifier* is defined in the current program, then declaring it global allows the loader to resolve references to *identifier* in other programs.

If *identifier* is not defined in the current program, the assembler expects an external resolution; in this case, therefore, *identifier* is global by default.

8.3. Location Counter Control Operations

data *abs*

The argument, if present, must evaluate to 0, 1, 2, or 3; this indicates the number of the data segment into which assembly is to be directed. If no argument is present, assembly is directed into *data* segment 0.

text *abs*

The argument, if present, must evaluate to 0, 1, 2, or 3; this indicates the number of the *text* segment into which assembly is to be directed. If no argument is present, assembly is directed into *text* segment 0.

Before the first **text** or **data** operation is encountered, assembly is by default directed into *text* segment 0.

org *expr*

The current location counter is set to *expr*. *Expr* must represent a value in the current segment, and must not be less than the current location counter.

even

The current location counter is rounded up to the next even value.

8.4. Symbolic Debugging Operations

The assembler allows for symbolic debugging information to be placed into the object code file with special pseudo-operations. The information typically includes line numbers and information about C language symbols, such as their type and storage class. The C compiler (*cc*(1)) generates symbolic debugging information when the **-g** option is used. Assembler programmers may also include such information in source files.

8.4.1. file and ln. The **file** pseudo-operation passes the name of the source file into the object file symbol table. It has the form

file *filename*

where *filename* consists of one to 14 characters enclosed in quotation marks.

The **ln** pseudo-operation makes a line number table entry in the object file. That is, it associates a line number with a memory location. Usually the memory location is the current location in text. The format is

ln *line*[,*value*]

where *line* is the line number. The optional value is the address in *text, data,* or *bss* to associate with the line number. The default when *value* is omitted (which is usually the case) is the current location in *text*.

8.4.2. Symbol Attribute Operations. The basic symbolic testing pseudo-operations are **def** and **endef**. These operations enclose other pseudo-operations that assign attributes to a symbol and must be paired.

```
def     name
.               # Attribute
.               # Assigning
.               # Operations
endef
```

NOTES

- **def** does not define the symbol, although it does create a symbol table entry. Because an undefined symbol is treated as external, a symbol which appears in a **def**, but which never acquires a value, will ultimately result in an error at link edit time.
- to allow the assembler to calculate the sizes of functions for other tools, each **def/endef** pair that defines a function name must be matched by a **def/endef** pair after the function in which a storage class of **—1** is assigned.

The paragraphs below describe the attribute-assigning operations. Keep in mind that all of these operations apply to symbol *name* which appeared in the opening **def** pseudo-operation.

val *expr*

Assigns the value *expr* to *name*. the type of the expression *expr* determines with which section *name* is associated. If value is ~, the current location in the *text* section is used.

scl *expr*

Declares the C language type of *name*. The expression *expr* must yield an ABSOLUTE value that corresponds to the C compiler's internal representation of a storage class. The special value *—1* designates the physical end of a function.

type *expr*

Declares the C language type of *name*. The expression *expr* must yield an ABSOLUTE value that corresponds to the C compiler's internal representation of a basic or derived type.

tag *str*

Associates *name* with the structure, enumeration, or union named *str* which must have already been declared with a **def/endef** pair.

line *expr*

Provides the line number of *name*, where *name* is a block symbol. The expression *expr* should yield an ABSOLUTE value that represents a line number.

size *expr*

Gives a size for *name*. The expression *expr* must yield an ABSOLUTE value. When *name* is a structure or an array with a predetermined extent, *expr* gives the size in bytes. For bit fields, the size is in bits.

dim *expr1,expr2,...*

Indicates that *name* is an array. Each of the expressions must yield an ABSOLUTE value that provides the corresponding array dimension.

8.5. Switch Table Operation

The C compiler generates a compact set of instructions for the C language *switch* construct. An example is shown below.

```
                sub.l   &1,%d0
                cmp.l   %d0,&4
                bhi     L%21
                add.w   %d0,%d0
                mov.w   10(%pc,%d0.w),%d0
                jmp     6(%pc,%d0.w)
                swbeg   &5
L%22:
                short   L%15-L%22
                short   L%21-L%22
                short   L%16-L%22
                short   L%21-L%22
                short   L%17-L%22
```

The special **swbeg** pseudo-operation communicates to the assembler that the lines following it contain **rel-rel** subtractions. Remember that ordinarily such subtractions are risky because of span-dependent optimization. In this case, however, the assembler makes special allowances for the subtraction because the compiler guarantees that both symbols will be defined in the current assembler file, and that one of the symbols is a fixed distance away from the current location.

The **swbeg** pseudo-operation takes an argument that looks like an immediate operand. The argument is the number of lines that follow **swbeg** and that contain switch table entries. **Swbeg** inserts two words into text. The first is the ILLEGAL instruction code. The second is the number of table entries that follow. The disassembler *dis*(1) needs the ILLEGAL instruction as a hint that what follows is a switch table. Otherwise, it would get confused when it tried to decode the table entries, differences between two symbols, as instructions.

9. SPAN-DEPENDENT OPTIMIZATION

The assembler makes certain choices about the object code it generates based on the distance between an instruction and its operand(s). Span-dependent optimization occurs most obviously in the choice of object code for branches and jumps. It also occurs when an operand may be represented by the program counter relative address mode instead of as an absolute 2-word (**long**) address. The span-dependent optimization capability is normally enabled; the **-n** command line flag disables it. When this capability is disabled,

the assembler makes worst-case assumptions about the types of object code that must be generated. Span-dependent optimizations are performed only within **text** segment 0. Any reference outside **text** segment 0 is assumed to be worst-case.

The C compiler (*cc*(1)) generates branch instructions without a specific offset size. When the optimizer is used, it identifies branches which could be represented by the short form, and it changes the operation accordingly. The assembler chooses only between long and very-long representations for branches.

For the MC68000 and MC68010 processors, branch instructions, e.g., **bra, bsr,** or **bgt**, can have either a byte or a word pc-relative address operand. A byte or word size specification should be used only when the user is sure that the address intended can be represented in the byte or word allowed. The assembler will take one of these instructions with a size specification and generate the byte or word form of the instruction without asking questions.

Although the largest offset specification allowed for the M68000 and M68010 is a word,* large programs could conceivably have need for a branch to location not reachable by a word displacement. Therefore, equivalent long forms of these instructions might be needed. When the Assembler encounters a branch instructions without a size specification, it tries to choose between the long and very-long forms of the instruction. If the operand can be represented in a word, then the word form of the instruction will be generated. Otherwise, the very-long form will be generated. For unconditional branches, e.g., **br, bra,** and **bsr,** the very-long form is just the equivalent jump (**jmp** and **jsr**) with an absolute address operand (instead of pc-relative). For conditional branches, the equivalent very-long form is a conditional branch around a jump, where the conditional test has been reversed.

The following table summarizes span-dependent optimizations. The assembler chooses only between the long form and the very-long form, while the optimizer chooses between the short and long forms for branches (but not **bsr**).

ASSEMBLER SPAN-DEPENDENT OPTIMIZATIONS

Instruction	Short Form	Long Form	Very-Long Form
br, bra, bsr	**byte offset**	word offset (See footnote for information about M68020.)	**jmp** or **jsr** with absolute long address
conditional branch	byte offset	word offset (See footnote for information about M68020.)	short conditional branch with reversed condition around **jmp** with absolute long addres
jmp, jsr		pc-relative address	absolute long address
lea.l, pea.l		pc-relative address	absolute long address

* The M68020 allows long word offset, as shown by the syntax for the branch instructions.

For the MC68020 processor, branch instructions can have either a byte, word, or long pc-relative address operand. The assembler still chooses between word and long representations for branches if no byte size specification is given; however, the long form is replaced by a branch long with pc-relative address instead of a jump with absolute long address.

10. ADDRESS MODE SYNTAX

The following table summarizes the *as* syntax for MC68000, MC68010, and MC68020 addressing modes. New addressing modes for the MB68020 are shown with "MC68020 Only" in parentheses beneath the MC6800 notation; modes not specified in this way are for all three processors.

In the table, the following abbreviations are used:

an Address register, where *n* is any digit from 0 through 7.

dn Data register, where *n* is any digit from 0 through 7.

ri Index register *i* may be any address or data register with an optional size designation (i.e., **ri.w** for 16 bits or **ri.l** for 32 bits); default size is **.w**.

scl Optional scale factor that may be multipled time index register in some modes. Values for *scl* are 1, 2, 4, or 8; default is 1.

bd Two's complement base displacement that is added before indirection takes place; size can be 16 or 32 bits.

od Outer displacement that is added as a part of effective address calculation after memory indirection; size can be 16 or 32 bits.

d Two's complement or sign-extended displacement that is added as part of effective address calculation; size may be 8 or 16 bits; when omitted, assembler uses value of zero.

pc Program counter

[ ] Grouping characters used to enclose an indirect expression; required characters. Addressing arguments can occur in any order within the brackets.

( ) Grouping characters used to enclose an entire effective address; required characters. Addressing arguments can occur in any order within the parentheses.

{ } Indicate that a scale factor is optional; not required characters.

It is important to note that expressions used for the **absolute** addressing modes need not be *absolute expressions* in the sense described in Section 6. Although the addresses used in those addressing modes must ultimately be filled in with constants, that can be done later by the loader. There is no need for the assembler to be able to compute them. Indeed, the **Absolute Long** addressing mode is commonly used for accessing *undefined external* addresses.

EFFECTIVE ADDRESS MODES		
M68000 Family Notation	*as* Notation	Effective Address Mode
Dn	%dn	Data Register Direct
An	%an	Address Register Direct
(An)	(%an)	Address Register Indirect
(An)+	(%an)+	Address Register Indirect With Postincrement
—(An)	—(%an)	Address Register Indirect With Predecrement
d(An)	d(%an)	Address Register Indirect With Displacement (*d* signifies a signed 16-bit absolute displacement)
d(An,Ri)	d(%an,%ri.w) d(%an,%ri.l)	Address Register Indirect With Index Plus Displacement (*d* signifies a signed 8-bit absolute displacement)
(bd,An,Ri{*scl}) (MC68020 Only)	(bd,%an,%ri{*ri})	Address Register Direct With Index Plus Base Displacement
([bd,An,Ri{*scl}],od) (MC68020 Only)	(bd,%an,%ri{*scl}],od)	Memory Indirect With Preindexing Plus Base and Outer Displacement
([bd,An],Ri{*scl},od) (MC68020 Only)	([bd,%an],%ri{*scl}],od)	Memory Indirect With Postindexing Plus Base and Outer Displacement
d(PC)	d(%pc)	Program Counter Indirect With Displacement (d signifies 16-bit displacement)
d(PC,Ri)	d(%pc,%rn.l) d(%pc,%rn.w)	Program Counter Direct With Index and Displacement (d signifies 8-bit displacement)
(bd,PC,Ri{*scl}) (MC68020 Only)	(bd,%pc,%ri{*scl})	Program Counter Direct With Index and Base Displacement
([bd,PC],Ri{*scl},od) (MC68020 Only)	([bd,%pc],%ri{*scl},od)	Program counter Memory Indirect With Postindexing Plus Base and Outer Displacement
([bd,PC,Ri{*scl}],od) (MC68020 Only)	([bd,%pc,%ri{*scl}],od)	Program Counter Memory Indirect With Preindexing Plus Base and Outer Displacement

EFFECTIVE ADDRESS MODES		
M68000 Family Notation	*as* Notation	Effective Address Mode
d,PC,Ri*scl],od) (MCC68020 Only)	d,pc,ri*scl],od)	Program Counter Memory Indirect With Preindexing Plus Base and Outer Displacement
xxx.W	xxx	Absolute Short Address (*xxx* signifies an expression yielding a 16-bit memory address)
xxx.L	xxx	Absolute Long Address (*xxx* signifies an expression yielding a 32-bit memory address)
#xxx	&xxx	Immediate Data (*xxx* signifies an absolute constant expression)

In the table above, the index register notation should be understood as **ri.size*scale**, where both size and scale are optional. Refer to Chapter 2 of the *M68000 Family Resident Structured Assembler Reference Manual* for additional information about effective address modes. Section 2 of the *MC68020 32-Bit Microprocessor User's Manual* also provides information about generating effective addresses and assembler syntax.

Note that suppressed address register **%zan** can be used in place of **%an**, suppressed PC register **%zpc** can be used in place of **%pc**, and suppressed data register **%zdn** can be used in place of **%dn**, if suppression is desired.

The new address modes for the MB68020 use two different formats of extension. The brief format provides fast indexed addressing, while the full format provides a number of options in size of displacement and indirection. The assembler will generate the brief format if the effective address expression is not memory indirect, value of displacement is within a byte, and no base or index suppression is specified; otherwise, the assembler will generate the full format.

Some source code variations of the new modes may be redundant with the MC68000 address register indirect, address register indirect with displacement, and program counter with displacement modes. The assembler will select the more efficient mode when redundancy occurs. For example, when the assembler sees the form (**An**), it will generate address register indirect mode (mode 2). The assembler will generate address register indirect with displacement (mode 5) when seeing any of the following forms (as long as bd fits in 16 bits or less):

bd(An)
(bd,An)
(An,bd)

11. MACHINE INSTRUCTIONS

11.1. Instructions For The MC68000/MC68010/MC68020

The following table shows how MC68000/MC68010/MC68020 instructions should be written in order to be understood correctly by the *as* assembler. The entire instruction set can be used for the MC68020. Instructions that are MC68010/MC68020-only or MC68020-only are noted as such in the "OPERATION" column.

Several abbreviations are used in the table:

S The letter **S**, as in **add.S**, stands for one of the operation size attribute letters **b**, **w**, or **l**, representing a byte, word, or long operation.

A The letter **A**, as in **add.A**, stands for one of the address operation size attribute letters **w** or **l**, representing a word or long operation.

CC In the contexts **bCC**, **dbCC**, and **sCC**, the letters CC represent any of the following condition code designations (except that **f** and **t** may not be used in the **bCC** instruction):

cc	carry clear	ls	low or same
cs	carry set	lt	less than
eq	equal	mi	minus
f	false	ne	not equal
ge	greater or equal	pl	plus
gt	greater than	t	true
hi	high	vc	overflow clear
hs	high or same (=cc)	vs	overflow set
le	less or equal		
lo	low (=cs)		

EA This represents an arbitrary effective address.

I An absolute expression, used as an immediate operand.

Q An absolute expression evaluating to a number from 1 to 8.

L A label reference, or any expression representing a memory address in the current segment.

d Two's complement or sign-extended displacement that is added as part of effective address calculation; size may be 8 by 16 bits; when omitted, assembler uses value of zero.

%dx, %dy, %dn Represent data registers.

%ax, %ay, %an Represent address registers.

%rx, %ry, %rn Represent either data or address registers.

%rc Represents control register (**%sfc, %dfc, %cacr, %usr, %vbr, %caar, %msp, %isp**).

offset Either an immediate operand or a data register.

width Either an immediate operand or a data register.

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
ABCD	abcd.d	%dy, %dx —(%ay),—(%ax)	Add Decimal with Extend
ADD	add.S	EA,%dn %dn,EA	Add Binary
ADDA	add.A	EA,%an	Add Address
ADDI	add.S	&I,EA	Add Immediate
ADDQ	add.S	&Q,EA	Add Quick
ADDX	addx.S	%dy,%dx —(%ay),—(%ax)	Add Extended
AND	and.S	EA,%dn %dn,EA	AND Logical
ANDI	and.S	&I,EA	AND Immediate
ANDI to CCR	and.b	&I,%cc	AND Immediate to Condition Codes
ANDI to SR	and.w	&I,%sr	AND Immediate to the Status Register
ASL	asl.S	%dx,%dy &Q,%dy	Arithmetic Shift (Left)
	als.w	&1,EA	
	als.w	EA	
ASR	asr.S	%dx,%dy &Q,%dy	Arithmetic Shift (Right)
	asr.w	&1,EA	
	asr.w	EA	
Bcc	bCC	L	Branch Conditionally (16-bit Displacement)
	bCC.b	L	Branch Conditionally (Short) (8-bit Displacement)
	bCC.l	L	Branch Conditionally (Long) (32-bit Displacement) (MC68020 Only)

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
BCHG	bchg	%dn,EA &I,EA	Test a Bit and Change NOTE: **bchg** should be written with no suffix. If the second operand is a data register, **.l** is assumed; otherwise, **.b** is.
BCLR	bclr	%dn,EA &I,EA	Test a Bit and Clear NOTE: **bclr** should be written with no suffix. If the second operand is a data register, **.l** is assumed; otherwise, **.b** is.
BFCHG	bfchg	EA{offset:width}	Complement Bit Field (MC68020 Only)
BFCLR	bfclr	EA{offset:width}	Clear Bit Field (MC68020 Only)
BFEXTS	bfexts	EA{offset:width},%dn	Extract Bit Field (Signed) (MC68020 Only)
BFEXTU	bfextu	EA{offset:width},%dn	Extract Bit Field (Unsigned) (MC68020 Only)
BFFFO	bfffo	EA{offset:width},%dn	Find First One in Bit Field (MC68020 Only)
BFINS	bfins	%dn,EA{offset:width}	Insert Bit Field (MC68020 Only)
BFSET	bfset	EA{offset:width}	Set Bit Field (MC68020 Only)
BFTST	bftst	EA{offset:width}	Test Bit Field (MC68020 Only)
BKPT	bkpt	&I	Breakpoint (MC68020 Only)

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
BRA	bra	L	Branch Always (16-bit Displacement)
	bra.b	L	Branch Always (Short) (8-bit Displacement)
	br.l	L	Branch Always (Long) (32-bit Displacement) (MC68020 Only)
	br br.b	L L	Same as **bra** Same as **bra.b**
BSET	bset	%dn,EA &I,EA	Test a Bit and Set NOTE: **bset** should be written with no suffix. If the second operand is a data register, **.l** is assumed; otherwise **.b** is.
BSR	bsr	L	Branch to Subroutine (16-bit Displacement)
	bsr.b	L	Branch to Subroutine (Short) (8-bit Displacement)
	bsr.l	L	Branch to Subroutine (Long) (32-bit Displacement) (MC68020 Only)
BTST	btst	%dn,EA &I,EA	Test a Bit and Set NOTE: **btst** should be written with no suffix. If the second operand is a data register, **.l** is assumed; otherwise **.b** is.
CALLM	callm	&I,EA	Call Module (MC68020 Only)
CAS	cas	%ds,%dy,EA	Compare and Swap Operands (MC68020 Only)
CAS2	cas2	%dx:%dy,%dx:%dy,%rx:%ry	Compare and Swap Dual Operands (MC68020 Only)

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
CHK	chk.w	EA,%dn	Check Register Against Bounds
	chk.l	EA,%dn	Check Register Against Bounds (Long) (MC68020 Only)
CHK2	chk2.S	EA,%rn	Check Register Against Bounds (MC68020 Only)
CLR	clr.S	EA	Clear an Operand
CMP	cmp.S	%dn,EA	Compare
CMPA	cmp.A	%an,EA	Compare Address
CMPI	cmp.S	EA,&I	Compare Immediate
CMPM	cmp.S	(%ax)+,(%ay)+	Compare Memory
CMP2	cmp.S	%rn,EA	Compare Register Against Bounds (MC68020 Only)*
DBcc	dbCC	%dn,L	Test Condition, Decrement, and Branch
	dbra	%dn,L	Decrement and Branch Always
	dbr	%dn,L	Same as **dbra**
DIVS	divs.w	EA,%dx	Signed Divide 32/16 → 32
	tdivs.l divs.l	EA,%dx EA,%dx	Signed Divide (Long) 32/32 → 32 (MC68020 Only)
	tdivs.l	EA,%dx:%dy	Signed Divide (Long) 32/32 → 32r:32q (MC68020 Only)
	divs.l	EA,%dx:%dy	Signed Divide (Long) 64/32 → 32r:32q (MC68020 Only)

* Note: The order of operands in *as* is the reverse of that in the M68000 Programmer's Reference Manual.

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
DIVU	divu.w	EA,%dn	Unsigned Divide 32/16 → 32
	tdivu.l divu.l	EA,%dx EA,%dx	Unsigned Divide (Long) 32/32 → 32 (MC68020 Only)
	tdivu.l	EA,%dx:%dy	Unsigned Divide (Long) 32/32 → 32r:32q (MC68020 Only)
	divu.l	EA,%dx:%dy	Unsigned Divide (Long) 64/32 → 32r:32q (MC68020 Only)
EOR	eor.S	%dn,EA	Exclusive OR Logical
EORI	eor.S	&I,EA	Exclusive OR Immediate
EORI to CCR	eor.b	&I,%cc	Exclusive OR Immediate to Condition Code Register
EORI to SR	eor.w	&I,%sr	Exclusive OR Immediate to the Status Register
EXG	exg	%rx,%ry	Exchange Registers
EXT	ext.w	%dn	Sign-Extend Low-Order Byte of Data to Word
	ext.l	%dn	Sign-Extend Low-Order Word of Data to Long
	extb.l	%dn	Sign-Extend Low-Order Byte of Data to Long (MC68020 Only)
	extw.l	%dn	Same as **ext.l** (MC68020 Only)
JMP	jmp	EA	Jump
JSR	jsr	EA	Jump to Subroutine
LEA	lea.l	EA,%an	Load Effective Address
LINK	link	%an,&I	Link and Allocate

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
LSL	lsl.S	%dx,%dy &Q,%dy	Logical Shift (Left)
	lsl.w	&1,EA	
	lsl.w	EA	
LSR	lsr.S	%dx,%dy &Q,&dy	Logical Shift (Right)
	lsr.w	&1,EA	
	lsr.w	EA	
MOVE	mov.S	EA,EA	Move Data from Source to Destination NOTE: If the destination is an address register, the instruction generated is **MOVEA.**
MOVE to CCR	mov.w	EA,%cc	Move to Condition Codes
MOVE from CCR	mov.w	%cc,EA (MC68010/MC68020 Only)	Move from Condition Codes
MOVE to SR	mov.w	EA,%sr	Move to the Status Register
MOVE from SR	mov.w	%sr,EA	Move from the Status Register
MOVE USP	mov.l	%usp,%an %an,%usp	Move User Stack Pointer
MOVEA	mov.A	EA,%an	Move Address
MOVEC to CCR	mov.l	%rn,%rc	Move to Control Register (MC68010/MC68020 Only)
MOVEC from CCR	mov.l	%rc,%rn (MC68010/MC68020 Only)	Move from Control Register

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
MOVEM	movm.A	&I,EA EA,&I	Move Multiple Registers* (See footnote)
MOVEP	movp.A	%dx,d(%ay) d(%ay),%dx	Move Peripheral Data
MOVEQ	mov.l	&I,%dn	Move Quick
MOVES	movs.S movs.S	%rn,EA EA,%rn	Move to/form Address Space (MC68010/MC68020 Only)
MULS	muls.w	EA,%dx	Signed Multiply 16*16 → 32
	tmuls.l muls.l	EA,%dx EA,%dx	Signed Multiply (Long) 32*32 → 32 (MC68020 Only)
	muls.l	EA,%dx:%dy	Signed Multiply (Long) 32*32 → 64 (MC68020 Only)
MULU	mulu.w	EA,%dx	Unsigned Multiply 16*16 → 32
	tmulu.l mulu.l	EA,%dx EA,%dx	Unsigned Multiply (Long) 32*32 → 32 (MC68020 Only)
	mulu.l	EA,%dx:%dy	Unsigned Multiply (Long) 32*32 → 64 (MC68020 Only)
NBCD	nbcd.b	EA	Negate Decimal with Extend
NEG	neg.S	EA	Negate
NEGX	negx.S	EA	Negate with Extend
NOP	nop		No Operation
NOT	not.S	EA	Logical Complement

* The immediate operand is a mask designating which registers are to be moved to memory or which registers are to receive memory data. not all addressing modes are permitted, and the correspondence between mask bits and register numbers depends on the addressing mode used. Refer to the MC68000 Programmer's Reference Manual for details.

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
OR	or.S	EA,%dn %dn,EA	Inclusive OR Logical
ORI	or.S	&I,EA	Inclusive OR Immediate
ORI to CCR	or.b	&I,%cc	Inclusive OR Immediate to Condition Codes
ORI to SR	or.w	&I,%sr	Inclusive OR Immediate to the Status Register
PACK	pack pack	—(%ax),—(%ay),&I %dx,%dy,&I	Pack BCD (MC68020 Only)
PEA	pea.l	EA	Push Effective Address
RESET	reset		Reset External Devices
ROL	rol.S	%dx,%dy &Q,%dy	Rotate (without Extend) (Left)
	rol.w rol.w	&1,EA EA	
ROR	ror.S	%dx,%dy &Q,%dy	Rotate (without Extend) (Right)
	ror.w ror.w	&1,EA EA	
ROXL	roxl.S	%dx,%dy &Q,%dy	Rotate with Extend (Left)
	roxl.w roxl.w	&1,EA EA	
ROXR	roxr.S	%dx,%dy &Q,%dy	Rotate with Extend (Right)
	roxr.w roxr.w	&1,EA EA	
RTD	rtd	&I	Return and Deallocate Parameters (MC68010/MC68020 Only)
RTE	rte		Return from Exception
RTM	rtm	%rn	Return from Module (MC68020 Only)

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
RTR	rtr		Return and Restore Condition Codes
RTS	rts		Return from Subroutine
SBCD	sbcd.b	%dy,%dx —(%ay),—(%ax)	Subtract Decimal with Extend
Scc	sCC.b	EA	Set According to Condition
STOP	stop	&I	Load Status Register and Stop
SUB	sub.S	EA,%dn %dn,EA	Subtract Binary
SUBA	sub.A	EA,%an	Subtract Address
SUBI	sub.S	&I,EA	Subtract Immediate
SUBQ	sub.S	&Q,EA	Subtract Quick
SUBX	subx.S	%dy,%dx —(%ay),—(%ax)	Subtract with Extend
SWAP	swap.w	%dn	Swap Register Halves
TAS	tas.b	EA	Test and Set an Operand
TRAP	trap	&I	Trap
TRAPV	trapv		Trap on Overflow
TRAPcc	tCC tpCC.A	&I	Trap on Condition (MC68020 Only)
TST	tst.S	EA	Test an Operand
UNLK	unlk	%an	Unlink
UNPK	unpk	—(%ax),—(%ay),&I %dx,%dy,&I	Unpack BCD (MC68020 Only)

11.2. Instructions For the MC68881

The following table shows how the floating point co-processor (MC68881) instructions should be written to be understood by the *as* assembler.

In the table, *fpcc* represents any of the following floating point condition code designations:

TRAP ON UNORDERED	
fpcc	MEANING
ge	greater than or equal
gl	greater or less than
gle	greater or less than or equal
gt	greater than
le	less than or equal
lt	less than
ngt	not greater than
nge	not greater than or equal
nlt	not less than
ngl	not greater or less than
nle	not greater or less than or equal
ngle	not greater or less than or equal
sneq	not equal
sf	never
seq	equal
st	always

NO TRAP ON UNORDERED	
fpcc	MEANING
eq	equal
oge	greater than or equal
ogl	greater or less than
ogt	greater than
ole	less than or equal
olt	less than
or	ordered
t	always
ule	unordered or less or equal
ult	unordered less than
uge	unordered greater than or equal
ueq	unordered equal
ugt	unordered greater than
un	unordered
neq	unordered ore greater or less
f	never

The designation *ccc* represents a group of constants in MC68881 constant ROM which have the following values:

ccc	VALUE
00	pi
0B	log10(2)
0C	e
0D	log2(e)
0D	log10(e)
0F	0.0
10	logn(2)
11	logn(10)
12	10**0
13	10**1
14	10**2
15	10**4
16	10**8
17	10**16
18	10**32
19	10**64
1A	10**128
1B	10**256
1C	10**512
1D	10**1024
1E	10**2048
1F	10**4096

Additional abbreviations used in the table are:

EA	represents and effective address
L	a label reference or any expression representing a memory address in the current segment
I	represents an absolute expression, used as an immediate operand
%dn	represents data register
%fpm,%fpn,%fpq	represents floating point data registers
%control	represents floating point control register
%status	represents floating point status register
%iaddr	represents floating point instruction address register
SF	represents source format letters: b byte integer w word integer l long word integer s single precision d double precision x extended precision p packed binary code decimal
A	represents source format letters w or l
B	represents source format letters b, w, l, s, or p

NOTE: The source format must be specified if more than one source format is permitted or a default source format *x* is assumed. Source format need not be specified if only one format is permitted by the operation.

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FABS	fabs.SF fabs.x fabs.x	EA,%fpn %fpm,%fpn %fpn	absolute value function
FACOS	facos.SF facos.x facos.x	EA,%fpn %fpm,%fpn %fpn	arccosine function
FADD	fadd.SF fadd.x	EA,%fpn %fpm,%fpn	floating point add
FASIN	fasin.SF fasin.x fasin.x	EA,%fpn %fpm,%fpn %fpn	arcsine function
FATAN	fatan.SF fatan.x fatan.x	EA,%fpn %fpm,%fpn %fpn	arctangent function
FATANH	fatanh.SF fatanh.x fatanh.x	EA,%fpn %fpm,%fpn %fpn	hyperbolic arctangent function
FBfpcc	fbfpcc.A	L	co-processor branch conditionally
FCMP	fcmp.SF fcmp.x	%fpn,EA %fpn,%fpm	floating point compare
FCOS	fcos.SF fcos.x fcos.x	EA,%fpn %fpm,%fpn %fpn	cosine function
FCOSH	fcosh.SF fcosh.x fcosh.x	EA,%fpn %fpm,%fpn %fpn	hyperbolic cosine function
FDBfpcc	fdbfpcc.w	%dn,L	decrement and branch on condition
FDIV	fdiv.SF fdiv.x	EA,%fpn %fpm,%fpn	floating point divided

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FETOX	fetox.SF	EA,%fpn	e**x function
	fetox.x	%fpm,%fpn	
	fatan.x	%fpn	
FETOXM1	fetoxm1.SF	EA,%fpn	e**x(x-1) function
	fetoxm1.x	%fpm,%fpn	
	fetoxm1.x	%fpn	
FGETEXP	fgetexp.SF	EA,%fpn	get the exponent function
	fgetexp.x	%fpm,%fpn	
	fgetexp.x	%fpn	
FGETMAN	fgetman.SF	EA,%fpn	get the mantissa function
	fgetman.x	%fpm,%fpn	
	fgetman.x	%fpn	
FINT	fint.SF	EA,%fpn	integer part function
	fint.x	%fpm,%fpn	
	fint.x	%fpn	
FLOG2	flog2.SF	EA,%fpn	binary log function
	flog2.x	%fpm,%fpn	
	flog2.x	%fpn	
FLOG10	flog10.SF	EA,%fpn	common log function
	flog10.x	%fpm,%fpn	
	flog10.x	%fpn	
FLOGN	flogn.SF	EA,%fpn	natural log function
	flogn.x	%fpm,%fpn	
	flogn.x	%fpn	
FLOGNP1	flognp1.SF	EA,%fpn	natural log (x+1) function
	flognp1.x	%fpm,%fpn	
	flognp1.x	%fpn	
FMOD	fmod.SF	EA,%fpn	floating point module
	fmod.x	%fpm,%fpn	

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FMOVE	fmov.SF fmov.x	EA,%fpn %fpm,%fpn	move to floating point register
	fmov.SF fmov.p fmov.p	%fpn,EA %fpn,EA{&I} %fpn,EA{%dn}	move from floating point register to memory
	fmov.l fmov.l fmov.l	EA,%control EA,%status EA,%iaddr	move from memory to special register
	fmov.l fmov.l fmov.l	%control,EA %statsu,EA %iaddr,EA	move to memory from special register
FMOVECR	fmovcr.x	&ccc,%fpn	move a ROM-stored to a floating point register
FMOVEM	fmovm.x	EA,&I	move to multiple floating point register
	fmovm.x	&I,EA	move from multiple registers to memory
	fmovm.x	EA,%dn	move to a data register
	fmovm.x	%dn,EA	move a data register to memory
	fmovm.l	EA,%control/%status/%iaddr	move to special registers
	fmovm.l	%control/%status/%iaddr,EA	move from special registers

NOTE: The immediate operand is a mask designating which registers are to be moved to memory or which registers are to receive memory data. Not all addressing modes are permitted and the correspondence between mask bvits and register numbers depends on the addressing mode used.

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FMUL	fmul.SF fmul.x	EA,%fpn %fpm,%fpn	floating point multiply
FNEG	fneg.SF fneg.x fneg.x	EA,%fpn %fpm,%fpn %fpn	negate function
FNOP	fnop		floating point no-op
FREM	frem.SF frem.x	EA,%fpn %fpm,%fpn	floating point remainder
FRESTORE	frestore	EA	restore internal state of co-processor
FSAVE	fsave	EA	co-processor save
FSCALE	fscale.SF fscale.x	EA,%fpn %fpm,%fpn	floating point scale exponent
FSfpcc	fsfpcc.b	EA	set on condition
FSGLDIV	fsgldiv.B fsgldiv.x	EA,%fpn %fpm,%fpn	floating point single precision divide
FSGLMUL	fsglmul.B fsglmul.s	EA,%fpn %fpm,%fpn	floating point single precision multiply
FSIN	fsin.SF fsin.x fsin.x	EA,%fpn %fpm,%fpn %fpn	sine function
FSINCOS	fsincos.SF fsincos.x	EA,%fpn %fpm,%fpn:%fpq	sine/cosine function
FSINH	fsinh.SF fsinh.x fsinh.x	EA,%fpn %fpm,%fpn %fpn	hyperbolic sine function
FSQRT	fsqrt.SF fsqrt.x fsqrt.x	EA,%fpn %fpm,%fpn %fpn	square root function

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FSUB	fsub.SF fsub.x	EA,%fpn %fpm,%fpn	square root function
FTAN	ftan.SF ftan.x ftan.x	EA,%fpn %fpm,%fpn %fpn	tangent function
FTANH	ftanh.SF ftanh.x ftanh.x	EA,%fpn %fpm,%fpn %fpn	hyperbolic tangent function
FTENTOX	ftentox.SF ftentox.x ftentox.x	EA,%fpn %fpm,%fpn %fpn	10**x function
FTfpcc	ftfpcc		trap on condition without a parameter
FTPfpcc	ftpfpcc.A	&I	trap on condition with a parameter
FTST	ftest.SF ftest.x	EA %fpm	floating point test an operand
FTWOTOX	ftwotox.SF ftwotox.x ftwotox.x	EA,%fpn %fpm,%fpn %fpn	2**x function
FYTOX	fytox.SF fytox.x	EA,%fpn %fpm,%fpn	floating point y**x