18. SYSTEM ASSEMBLER

Introduction

This is a reference manual for the SYSTEM V/68 resident assembler, *as*. Programmers familiar with the M68XXX family of processors should be able to program in *as* by referring to this manual, but this is not a manual for the processors. Details about the effects of instructions, the meanings of status register bits, the 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 8-/16-/32-Bit Microprocessors Programmer's Reference Manual, Fifth Edition; Englewood Cliffs, NJ: PRENTICE-HALL, 1986. This manual is also available from the Motorola Literature Distribution Center, part number M68000UM/AD REV 4.
- 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/AD REV 1.
- MC68030 Enhanced 32-Bit Microprocessor User's Manual; MOTOROLA, 1987. This manual is available from the Motorola Literature Distribution Center, part number MC68030UM/AD.
- MC68881 Floating Point Coprocessor User's Manual, MC68881UM/AD; MOTOROLA, 1985. This manual is available from the Motorola Literature Distribution Center, part number MC68881UM/AD.
- MC68851 Paged Memory Management Unit User's Manual, MC68851UM/AD; Englewood Cliffs, NJ: PRENTICE-HALL, 1986. This manual is also available from the Motorola Literature Distribution Center, part number MC68851UM/AD.
- M68000 Family Resident Structured Assembler Reference Manual, M68KMASM.
- SYSTEM V/68 User's Reference Manual, MU43810UR/D2.

For users of the SGS M68020 Cross Compilation System, references to *as*(1) and *cc*(1) should be read as *as20*(1) and *cc20*(1) if you have a MC68020 processor system or *as30*(1) and *cc30*(1) if you have a MC68030 processor system. Information about the MC68020 commands is provided in the SGS

M68020 Cross Compilation System Reference Manual, M68KUNASX.

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 the "Address Mode Syntax" and "Machine Instructions" sections later in this chapter for further information.

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.
```

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.

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**. Based on 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.

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 Programmer's Reference Manual* for *as*(1) and the *SGS M68020 Cross Compilation System Reference Manual* for *as20*(1).

General Syntax Rules

Format of Assembly Language Line

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

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

      text        2
      mov.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 "Location Counters and Labels" later in this chapter.

— 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.

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.

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 "Symbol Definition Operations") or by using it as a label (refer to "Location Counters and Labels").

The tilde character (˜) has special significance to the assembler. A ˜ used alone, as an identifier, means "the current location." More specifically, a ˜ used in an instruction means the value of the program counter at the beginning of that instruction and a tilde used in a pseudo–instruction means the current value of the location counter for the current section. A ˜ used as the first character in an identifier becomes a period (.) in the symbol table, allowing symbols such as **.eos** and **.0fake** to be entered into the symbol table, as required by the Common Object File Format (COFF). Information about file formats is provided in the *Programmer's Reference Manual*.

Register Identifiers

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

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

Notes:

%cc and **%ccr** are equivalent.

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.

With the proper option, the assembler will correctly assemble instructions intended for the M68010. The entire register set of the MC68000 is included in the MC68010 register set. The following are new control registers for the MC68010.

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

Notes:

%sfc and **%sfcr** are equivalent.
%dfc and **%dfcr** are equivalent.

The entire register set of the 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 entire register set of the MC68020 is included in the MC68030 register set. The following are control registers for the MC68030:

MC68030 REGISTERS	
NAME	DESCRIPTION
%crp	Cpu Root Pointer Register
%srp	Supervisor Root Pointer Register
%tc	Translation Control Register
%tt0	Transparent Translation Register 0
%tt1	Transparent Translation Register 1
%mmusr	Memory Management Unit Status Register

Notes:

The new MC68030 registers are dedicated to memory management.

%mmusr is equivalent to the **%psr** on the MC68851.

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	

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.

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 hexadecimal 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)
```

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	0x09	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

Other Syntactic Details

For a discussion of expression syntax, see "Expressions" in this chapter. For information about the syntax of specific components of *as* instructions and pseudo-operations, see "Pseudo-Operations" and "Address Mode Syntax."

Segments, Location Counters, And Labels

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 "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    0xffffffff
data    1           # segment 1
byte    0xff
```


then equivalent object code would be generated by:

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

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.

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.

18

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 that **rel – rel** expressions are permitted only within the context of a switch statement (refer to "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 binary –. 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.

Pseudo-Operations

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 zeros 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
```

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.

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 directed by default 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.

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.

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*.

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 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 the value is ˜ , the current location in the *text* section is used.

scl *expr*
Declares a storage class for *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.

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
        mov.w   (%d0.w*2,L%22),%d0
        jmp     (%d0.w,L%22)
        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.

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). Choosing the smallest, fastest form is called span-dependent optimization. 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 word (16 bits) and long–word (32 bits) 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 MC68000 and MC68010 processors is a word, large programs could conceivably have need for a branch to location not reachable by a word displacement. Therefore, equivalent long–word forms of these instructions might be needed. When the assembler encounters a branch instruction without a size specification, it tries to choose between the word

and long–word 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 long–word form will be generated. For unconditional branches, e.g., **br**, **bra**, and **bsr**, the long-word form is just the equivalent jump (**jmp** and **jsr**) with an absolute address operand (instead of pc-relative). For conditional branches, the equivalent long-word form is a conditional branch around a jump, where the conditional test has been reversed.

The following table summarizes span-dependent optimizations. The optimizer chooses only between the word form and long–word forms for branches (but not **bsr**).

Assembler Span-Dependent Optimizations

Instruction	Word Form	Long-word Form
br, bra, bsr	word offset	**jmp** or **jsr** with absolute long address
conditional branch	word offset	short conditional branch with reversed condition around **jmp** with absolute long address

For the MC68020 and MC68030 processors, branch instructions can have either a byte, word, or long-word pc-relative address operand.

Address Mode Syntax

The following table summarizes the *as* syntax for MC68000, MC68010, MC68020 and MC68030 addressing modes. Addressing modes for the MC68020 and MC68030 are shown with "MC68020 Only" in parentheses beneath the MC68000 notation; modes not specified in this way are for all four 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 multiplied times index register in some modes. Values for *scl* are 1, 2, 4, or 8; default is 1. Only MC68020 and MC68030 instructions can have scale factors.

bd Two's complement base displacement that is added before indirection takes place; size can be 16 or 32 bits. Only MC68020 and MC68030 instructions can have scale factors.

od Outer displacement that is added as part of effective address calculation after memory indirection; size can be 16 or 32 bits. Only MC68020 and MC68030 instructions can have scale factors.

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 earlier under "Types." 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/MC68030 Only)	(bd,%an,%ri{*scl})	Address Register Direct With Index Plus Base Displacement
([bd,An,Ri{*scl}],od) (MC68020/MC68030 Only)	(bd,%an,%ri{*scl}],od)	Memory Indirect With Preindexing Plus Base and Outer Displacement
([bd,An],Ri{*scl},od) (MC68020/MC68030 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/MC68030 Only)	(bd,%pc,%ri{*scl})	Program Counter Direct With Index and Base Displacement
([bd,PC],Ri{*scl},od) (MC68020/MC68030 Only)	([bd,%pc],%ri{*scl},od)	Program Counter Memory Indirect With Post-indexing Plus Base and Outer Displacement

M68000 FAMILY NOTATION	*as* NOTATION	EFFECTIVE ADDRESS MODE
([bd,PC,Ri{*scl}],od) (MC68020/MC68030 Only)	([bd,%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 address modes for the MC68020 and MC68030 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)

Machine Instructions

Instructions For The MC68000/MC68010/MC68020/MC68030

The following table shows how MC68000/MC68010/MC68020/MC68030 instructions should be written in order to be understood correctly by the *as* assembler. The entire instruction set for the MC68030 can be used. Instructions that are MC68010/MC68020/MC68030-only, MC68020-only or MC68020/MC68030-only are noted as such in the "OPERATION" column. Additional MC68030-only instructions which deal specifically with memory management are listed separately as a subset of the MC68851 instructions.

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 or 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, %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.b**	**%dy,%dx** **–(%ay),–(%ax)**	Add Decimal with Extend
ADD	**add.S**	**EA,%dn** **%dn,EA**	Add Binary
ADDA	**add.A** **adda.A**	**EA,%an** **EA,%an**	Add Address. Second form is PMMU-supported *as20* only.
ADDI	**add.S** **addi.S**	**&I,EA** **&I,EA**	Add Immediate. Second form is PMMU-supported *as20* only.
ADDQ	**add.S** **addq.S**	**&Q,EA** **&Q,EA**	Add Quick. Second form is PMMU-supported *as20* only.
ADDX	**addx.S**	**%dy,%dx** **–(%ay),–(%ax)**	Add Extended
AND	**and.S**	**EA,%dn** **%dn,EA**	AND Logical
ANDI	**and.S** **andi.S**	**&I,EA** **&I,EA**	AND Immediate Second form is PMMU-supported *as20* only.
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)
	asl.w	**&1,EA**	
ASR	**asr.S**	**%dx,%dy** **&Q,%dy**	Arithmetic Shift (Right)
	asr.w	**&1,EA**	

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
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/MC68030 Only)
BCHG	**bchg**	**%dn,EA** **&l,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** **&l,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/MC68030 Only)
BFCLR	**bfclr**	**EA{offset:width}**	Clear Bit Field (MC68020/MC68030 Only)
BFEXTS	**bfexts**	**EA{offset:width},%dn**	Extract Bit Field (Signed) (MC68020/MC68030 Only)
BFEXTU	**bfextu**	**EA{offset:width},%dn**	Extract Bit Field (Unsigned) (MC68020/MC68030 Only)
BFFFO	**bfffo**	**EA{offset:width},%dn**	Find First One in Bit Field (MC68020/MC68030 Only)
BFINS	**bfins**	**%dn,EA{offset:width}**	Insert Bit Field (MC68020/MC68030 Only)
BFSET	**bfset**	**EA{offset:width}**	Set Bit Field (MC68020/MC68030 Only)

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
BFTST	**bftst**	**EA{offset:width}**	Test Bit Field (MC68020/MC68030 Only)
BKPT	**bkpt**	**&I**	Breakpoint (MC68020/MC68030 Only)
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/MC68030 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/MC68030 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)

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
CAS	**cas**	**%dx,%dy,EA**	Compare and Swap Operands (MC68020/MC68030 Only)
CAS2	**cas2**	**%dx:%dy,%dx:%dy,%rx:%ry**	Compare and Swap Dual Operands (MC68020/MC68030 Only)
CHK	**chk.w**	**EA,%dn**	Check Register Against Bounds
	chk.l	**EA,%dn**	Check Register Against Bounds (Long) (MC68020/MC68030 Only)
CHK2	**chk2.S**	**EA,%rn**	Check Register Against Bounds (MC68020/MC68030 Only)
CLR	**clr.S**	**EA**	Clear an Operand
CMP	**cmp.S**	**%dn,EA**	Compare
CMPA	**cmp.A** **cmpa.A**	**%an,EA** **%an,EA**	Compare Address. Second form is PMMU-supported *as20* only.
CMPI	**cmp.S** **cmpi.S**	**EA,&I** **EA,&I**	Compare Immediate. Second form is PMMU-supported *as20* only.
CMPM	**cmp.S** **cmpm.S**	**(%ax)+,(%ay)+** **(%ax)+,(%ay)+**	Compare Memory. Second form is PMMU-supported *as20* only.
CMP2	**cmp.S** **cmp2.A**	**%rn,EA** **%rn,EA**	Compare Register Against Bounds (MC68020/MC68030 Only).[1] Second form is PMMU-supported *as20* only.
DBcc	**dbCC**	**%dn,L**	Test Condition, Decrement, and Branch
	dbra	**%dn,L**	Decrement and Branch Always
	dbr	**%dn,L**	Same as **dbra**

1. 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
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/MC68030 Only)
	tdivs.l **divsl.l**	**EA,%dx:%dy** **EA,%dx:%dy**	Signed Divide (Long) 32/32 –> 32r:32q[2] (MC68020/MC68030 Only). Second form is PMMU-supported *as20* only.
	divs.l	**EA,%dx:%dy**	Signed Divide (Long) 64/32 –> 32r:32q[3] (MC68020/MC68030 Only)
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/MC68030 Only)
	tdivu.l **divul.l**	**EA,%dx:%dy** **EA,%dx:%dy**	Unsigned Divide (Long) 32/32 –> 32r:32q (MC68020/MC68030 Only)[4] Second form is PMMU-supported *as20* only.
	divu.l	**EA,%dx:%dy**	Unsigned Divide (Long) 64/32 –> 32r:32q (MC68020/MC68030 Only)[5]

2. Whenever %dx and %dy are the same register, the form is equivalent to the tdivs.l EA,%dx form (PMMU-supported *as20* only).
3. Whenever %dx and %dy are the same register, the form is equivalent to the divs.l EA,%dx form (PMMU-supported *as20* only).
4. Whenever %dx and %dy are the same register, then the form is equivalent to the tdivu.l EA,%dx form.
5. Whenever %dx and %dy are the same register, then the form is equivalent to the divu.l EA,%dx form.

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
EOR	**eor.S**	**%dn,EA**	Exclusive OR Logical
EORI	**eor.S** **eori.S**	**&I,EA** **&I,EA**	Exclusive OR Immediate. Second form is PMMU-supported *as20* only.
EORI to CCR	**eor.b** **eori.b** **eori.b**	**&I,%cc** **&I,%cc** **&I,%ccr**	Exclusive OR Immediate to Condition Code Register. Second and third forms PMMU-supported *as20* only.
EORI to SR	**eor.w** **eori.w**	**&I,%sr** **&I,%sr**	Exclusive OR Immediate to the Status Register. Second form is PMMU-supported *as20* only.
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/MC68030 Only)
	extw.l	**%dn**	Same as **ext.l** (MC68020/MC68030 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	**&l,EA**	
LSR	**lsr.S**	**%dx,%dy** **&Q,&dy**	Logical Shift (Right)
	lsr.w	**&l,EA**	
MOVE	**mov.S** **move.S**	**EA,EA** **EA,EA**	Move Data from Source to Destination. move.S form is PMMU-supported *as20* only. Note: If the destination is an address register, the instruction generated is **MOVEA.**
MOVE to CCR	**mov.w** **move.w**	**EA,%cc** **EA,%ccr**	Move to Condition Codes. move.w form is PMMU-supported *as20* only.
MOVE from CCR	**mov.w** **move.w**	**%cc,EA** **%ccr,EA**	Move from Condition Codes. (MC68010/MC68020/MC68030 Only) move.w form is PMMU-supported *as20* only.
MOVE to SR	**mov.w** **move.w**	**EA,%sr** **EA,%sr**	Move to the Status Register. move.w form is PMMU-supported *as20* only.
MOVE from SR	**mov.w** **move.w**	**%sr,EA** **%sr,EA**	Move from the Status Register. move.w form is PMMU-supported *as20* only.
MOVE USP	**mov.l** **move.l**	**%usp,%an** **%an,%usp** **%usp,%an** **%an,%usp**	Move User Stack Pointer. move.l form is PMMU-supported *as20* only.

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
MOVEA	**mov.A**	**EA,%an**	Move Address.
	mova.A	**EA,%an**	movea.A and movea.A PMMU-
	movea.A	**EA,%an**	supported *as20* only.
MOVEC	**mov.l**	**&l,EA**	Move to Control Register.
to CR	**movc.l**	**%rn,%rc**	(MC68010/MC68020/MC68030 Only)
	movec.l	**%rn,%rc**	movc.l and movec.l PMMU-
		%rn,%rc	supported *as20* only.
MOVEC	**mov.l**	**%rc,%rn**	Move from Control Register.
from CR	**movc.l**	**%rc,%rn**	(MC68010/MC68020/MC68030 Only)
	movec.l	**%rc,%rn**	movc.l and movec.l PMMU-
			supported *as20* only.
MOVEM	**movm.A**	**EA,&l**	Move Multiple Registers[6]
			(See footnote).
	movem.A	**&l,EA**	movem.A form is PMMU-
		EA,&l	supported *as20* only.
MOVEP	**movp.A**	**%dx,d(%ay)**	Move Peripheral Data.
		d(%ay),%dx	movep.A form is PMMU-
	movep.A	**%dx,d(%ay)**	supported *as20* only.
		d(%ay),%dx	
			Move Quick.
MOVEQ	**mov.l**	**&l,%dn**	movq.l and moveq.l forms
	movq.l	**&l,%dn**	PMMU-supported *as20* only.
	moveq.l	**&l,%dn**	
			Move to/from Address
MOVES	**movs.S**	**%rn,EA**	Space
	movs.S	**EA,%rn**	(MC68010/MC68020/MC68030 Only).
	moves.S	**%rn,EA**	moves.S forms PMMU-
	moves.S	**EA,%rn**	supported *as20* only.

6. The immediate operand is a mask designating which registers are to be moved to memory or which 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. Unlike the other M68000 family of assemblers, only a mask is allowed for the *as* assembler (PMMU-supported *as20* only).

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
MULS	**muls.w**	**EA,%dx**	Signed Multiply 16*16 –> 32
	tmuls.l	**EA,%dx**	Signed Multiply (Long) 32*32 –> 32 (MC68020/MC68030 Only)
	muls.l	**EA,%dx**	
	muls.l	**EA,%dx:%dy**	Signed Multiply (Long) 32*32 –> 64 (MC68020/MC68030 Only)[7]
MULU	**mulu.w**	**EA,%dx**	Unsigned Multiply 16*16 –> 32
	tmulu.l	**EA,%dx**	Unsigned Multiply (Long) 32*32 –> 32 (MC68020/MC68030 Only)
	mulu.l	**EA,%dx**	
	mulu.l	**EA,%dx:%dy**	Unsigned Multiply (Long) 32*32 –> 64 (MC68020/MC68030 Only)[8]
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
OR	**or.S**	**EA,%dn** **%dn,EA**	Inclusive OR Logical
ORI	**or.S** **ori.S**	**&I,EA** **&I,EA**	Inclusive OR Immediate. ori.S form is PMMU-supported *as20* only.
ORI to CCR	**or.b** **ori.b** **ori.b**	**&I,%cc** **&I,%cc** **&I,%ccr**	Inclusive OR Immediate to Condition Codes. ori.b forms are PMMU-supported *as20* only.

7. Whenever %dx and %dy are the same register, the form is equivalent to the muls.l EA,%dx form (PMMU-supported *as20* only).

8. Whenever %dx and %dy are the same register, the form is equivalent to the mulu.l DA,%dx form (PMMU-supported *as20* only).

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
ORI to SR	**or.w** **ori.w**	**&I,%sr** **&I,%sr**	Inclusive OR Immediate to the Status Register. ori.w form is PMMU-supported *as20* only.
PACK	**pack** **pack**	**–(%ax),–(%ay),&I** **%dx,%dy,&I**	Pack BCD (MC68020/MC68030 Only)
PEA	**pea.l**	**EA**	Push Effective Address
RESET	**reset**		Reset External Devices
ROL	**rol.S** **rol.w**	**%dx,%dy** **&Q,%dy** **&I,EA**	Rotate (without Extend) (Left)
ROR	**ror.S** **ror.w**	**%dx,%dy** **&Q,%dy** **&I,EA**	Rotate (without Extend) (Right)
ROXL	**roxl.S** **roxl.w**	**%dx,%dy** **&Q,%dy** **&I,EA**	Rotate with Extend (Left)
ROXR	**roxr.S** **roxr.w**	**%dx,%dy** **&Q,%dy** **&I,EA**	Rotate with Extend (Right)
RTD	**rtd**	**&I**	Return and Deallocate Parameters (MC68010/MC68020/MC68030 Only)
RTE	**rte**		Return from Exception
RTM	**rtm**	**%rn**	Return from Module (MC68020 Only)
RTR	**rtr**		Return and Restore Condition Codes
RTS	**rts**		Return from Subroutine

MC68000 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
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** **suba.A**	**EA,%an** **EA,%an**	Subtract Address. suba.A form is PMMU-supported *as20* only.
SUBI	**sub.S** **subi.S**	**&I,EA** **&I,EA**	Subtract Immediate. subi.S form is PMMU-supported *as20* only.
SUBQ	**sub.S** **subq.S**	**&Q,EA** **&Q,EA**	Subtract Quick. subq.S form is PMMU-supported *as20* only.
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 trapCC** **tpCC.A** **trapCC.A**	 **&I** **&I**	Trap on Condition (MC68020/MC68030 Only)
TST	**tst.S**	**EA**	Test an Operand
UNLK	**unlk**	**%an**	Unlink
UNPK	**unpk**	**–(%ax),–(%ay),&I** **%dx,%dy,&I**	Unpack BCD (MC68020/MC68030 Only)

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, *cc* represents any of the following floating point condition code designations:

TRAP ON UNORDERED	
cc	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 (less than or equal)
ngle	not (greater or less than or equal)
sneq	signaling not equal
sf	signaling false
seq	signaling equal
st	signaling true

NO TRAP ON UNORDERED	
cc	MEANING
eq	equal
oge	ordered greater than or equal
ogl	ordered greater or less than
ogt	ordered greater than
ole	ordered less than or equal
olt	ordered less than
or	ordered
t	true
ule	unordered or less or equal
ult	unordered or less than
uge	unordered or greater than or equal
ueq	unordered or equal
ugt	unordered or greater than
un	unordered
neq	not equal
f	false

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

ccc	VALUE	*ccc*	VALUE
00	pi	35	10**4
0B	log10(2)	36	10**8
0C	e	37	10**16
0D	log2(e)	38	10**32
0D	log10(e)	39	10**64
0F	0.0	3A	10**128
10	ln(2)	3B	10**256
11	ln(10)	3C	10**512
32	10**0	3D	10**1024
33	10**1	3E	10**2048
34	10**2	3F	10**4096

Additional abbreviations used in the table are:

EA	represents an effective addresss
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	represent floating point data registers
%control	represents floating point control register
%fpcr	represents floating point control register (PMMU-supported *as20* only)
%status	represents floating point status register
%fpsr	represents floating point status register (PMMU-supported *as20* only)
%iaddr	represents floating point instruction address register
%fpiar	represents floating point instruction address register (PMMU-supported *as20* only)
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.

MC68881 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FABS	**fabs.SF**	**EA,%fpn**	absolute value function
	fabs.x	**%fpm,%fpn**	
	fabs.x	**%fpn**	
FACOS	**facos.SF**	**EA,%fpn**	arccosine function
	facos.x	**%fpm,%fpn**	
	facos.x	**%fpn**	
FADD	**fadd.SF**	**EA,%fpn**	floating point add
	fadd.x	**%fpm,%fpn**	
ftst form.sp			
FASIN	**fasin.SF**	**EA,%fpn**	arcsine function
	fasin.x	**%fpm,%fpn**	
	fasin.x	**%fpn**	
FATAN	**fatan.SF**	**EA,%fpn**	arctangent function
	fatan.x	**%fpm,%fpn**	
	fatan.x	**%fpn**	
FATANH	**fatanh.SF**	**EA,%fpn**	hyperbolic arctangent function
	fatanh.x	**%fpm,%fpn**	
	fatanh.x	**%fpn**	
FBcc	**fbcc.A**	**L**	co-processor branch conditionally
FCMP	**fcmp.SF**	**%fpn,EA**	floating point compare
	fcmp.x	**%fpn,%fpm**	
FCOS	**fcos.SF**	**EA,%fpn**	cosine function
	fcos.x	**%fpm,%fpn**	
	fcos.x	**%fpn**	
FCOSH	**fcosh.SF**	**EA,%fpn**	hyperbolic cosine function
	fcosh.x	**%fpm,%fpn**	
	fcosh.x	**%fpn**	
FDBcc	**fdbcc.w**	**%dn,L**	decrement and branch on condition
FDIV	**fdiv.SF**	**EA,%fpn**	floating point divide
	fdiv.x	**%fpm,%fpn**	

MC68881 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FETOX	**fetox.SF**	**EA,%fpn**	e**x function
	fetox.x	**%fpm,%fpn**	
	fetox.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
	fgetexp.x	**%fpm,%fpn**	function
	fgetexp.x	**%fpn**	
FGETMAN	**fgetman.SF**	**EA,%fpn**	get the mantissa
	fgetman.x	**%fpm,%fpn**	function
	fgetman.x	**%fpn**	
FINT	**fint.SF**	**EA,%fpn**	integer part function
	fint.x	**%fpm,%fpn**	
	fint.x	**%fpn**	
FINTRZ	**fintrz.SF**	**EA,%fpn**	integer part, round-to-zero
	fintrz.x	**%fpm,%fpn**	function
	fintrz.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**	

MC68881 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
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**	
FMOVE	**fmov.SF**	**EA,%fpn**	move to floating point register (fmove.SF and fmove.x forms PMMU-supported *as20* only)
	fmov.x	**%fpm,%fpn**	
	fmove.SF	**EA,%fpn**	
	fmove.x	**%fpm,%fpn**	
	fmov.SF	**%fpn,EA**	move from floating point register to memory (fmove.SF and fmove.p forms PMMU-supported *as20* only)
	fmov.p	**%fpn,EA{&l}**	
	fmov.p	**%fpn,EA{%dn}**	
	fmove.SF	**%fpn,EA**	
	fmove.p	**%fpn,EA{&l}**	
	fmove.p	**%fpn,EA{%dn}**	
	fmov.l	**EA,%control**	move from memory to special register (fmove.l forms PMMU-supported *as20* only)
	fmov.l	**EA,%status**	
	fmov.l	**EA,%iaddr**	
	fmove.l	**EA,%control**	
	fmove.l	**EA,%status**	
	fmove.l	**EA,%iaddr**	
	fmov.l	**%control,EA**	move to memory from special register (fmove.l forms PMMU-supported *as20* only)
	fmov.l	**%status,EA**	
	fmov.l	**%iaddr,EA**	
	fmove.l	**%control,EA**	
	fmove.l	**%status,EA**	
	fmove.l	**%iaddr,EA**	
FMOVECR	**fmovcr.x**	**&ccc,%fpn**	move a ROM-stored to a floating point register

MC68881 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FMOVEM	**fmovm.x** **fmovem.x**	**EA,&l** **EA,&l**	move to multiple floating point register (fmovem.x form PMMU-supported *as20* only)
	fmovm.x **fmovem.x**	**&l,EA** **&l,EA**	move from multiple registers to memory (fmovem.x PMMU-supported *as20* only)
	fmovm.x **fmovem.x**	**EA,%dn** **EA,%dn**	move to a data register (fmovem.x form PMMU-supported *as20* only)
	fmovm.x **fmovem.x**	**%dn,EA** **%dn,EA**	move a data register to memory (fmovem.x PMMU-supported *as20* only)
	fmovm.l **fmovem.l**	**EA,%control/%status/%laddr** **EA,%control/%status/%laddr**	move to special registers (fmovem.l form PMMU-supported *as20* only)
	fmovm.l **fmovem.l**	**%control/%status/%laddr,EA** **%control/%status/%laddr,EA**	move from special registers (fmovem.l form PMMU-supported *as20* only)
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

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 bits and register numbers depends on the addressing mode used.

MC68881 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
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
FScc	**fscc.b**	**EA**	set on condition
FSGLDIV	**fsgldiv.B** **fsgldiv.s**	**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:%fpq** **%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
FSUB	**fsub.SF** **fsub.x**	**EA,%fpn** **%fpm,%fpn**	square root function

MC68881 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
FTAN	**ftan.SF**	**EA,%fpn**	tangent function
	ftan.x	**%fpm,%fpn**	
	ftan.x	**%fpn**	
FTANH	**ftanh.SF**	**EA,%fpn**	hyperbolic tangent function
	ftanh.x	**%fpm,%fpn**	
	ftanh.x	**%fpn**	
FTENTOX	**ftentox.SF**	**EA,%fpn**	10**x function
	ftentox.x	**%fpm,%fpn**	
	ftentox.x	**%fpn**	
FTcc	**ftcc**		trap on condition without a parameter[9]
FTRAPcc	**ftrapcc**		trap on condition without a parameter (PMMU-supported *as20* only)
FTPcc	**ftpcc.A**	**&I**	trap on condition with a parameter
FTRAPcc	**ftrapcc.A**	**&I**	trap on condition with a parameter (PMMU-supported *as20* only)
FTST	**ftest.SF**	**EA**	floating point test an operand (ftst.SF and ftst.x forms PMMU-supported *as20* only)
	ftest.x	**%fpm**	
	ftst.SF	**EA**	
	ftst.x	**%fpm**	
FTWOTOX	**ftwotox.SF**	**EA,%fpn**	2**x function
	ftwotox.x	**%fpm,%fpn**	
	ftwotox.x	**%fpn**	

9. The ftst form (floating point trap on signal true) is no longer supported due to a conflict with the FTST (floating point test and operand instruction) - PMMU-supported *as20* only.

Instructions For The MC68851

The following table shows how the paged memory management unit (PMMU) (MC68851) instructions should be written to be understood by the *as* assembler. Instructions that are MC68030-only or MC68851-only are noted as such in the "OPERATION" column. Additional MC68030 instructions which do not deal with memory management are listed separately with the MC68000 instructions.

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

SET PSR BIT	
cc/CC	MEANING
bs	bus error
ls	limit violation
ss	supervisor violation
as	access level violation
ws	write protected
is	invalid
gs	gate
cs	globally shared

CLEAR PSR BIT	
cc/CC	MEANING
bc	bus error
lc	limit violation
sc	supervisor violation
ac	access level violation
wc	write protected
ic	invalid
gc	gate
cc	globally shared

Additional abbreviations used in the table are:

EA	represents an effective addresss
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

FC	represents one of the following function codes: I represents an absolute expression used as an immediate operand %dn represents a data register %sfc represents the source function code register %sfcr represents the source function code register %dfc represents the destination function code register %dfcr represents the destination function code register
M	represents an absolute expression used as an immediate operand mask in the PFLUSH/PFLUSHS instructions where $0 \leq M \leq 15$
D	represents an absolute expression used as an immediate operand depth level in the PTESTR/PTESTW instructions where $0 \leq D \leq 7$
PMRn	represents any of the MC68881 registers
MRn	represents any of the MC68030 memory management registers
%dn	represents a data register 0 through 7
%an	represents an address register 0 through 7
%ac	represents pmmu access control register (MC68851 only)
%bac	represents pmmu breakpoint acknowledge control register 0 through 7 (MC68851 only)
%bad	represents pmmu breakpoint acknowledge data register 0 through 7 (MC68851 only)
%cal	represents pmmu current access level register (MC68851 only)
%crp	represents pmmu CPU root pointer register
%drp	represents pmmu DMA root pointer register (MC68851 only)
%mmusr	represents pmmu status register
%pcsr	represents pmmu cache status register (MC68851 only)
%psr	represents pmmu status register
%scc	represents pmmu stack change control register (MC68851 only)
%srp	represents pmmu supervisor root pointer register
%tc	represents pmmu translation control register
%tt	represents pmmu transparent translation control registers 0 or 1 (MC68030 only)
%val	represents pmmu validate access level register (MC68851 only)

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

MC68851 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
PBcc	**pbCC.A**	**L**	Branch on PMMU Condition (MC68851 only)
PDBcc	**pdbCC.w**	**%dn,L**	Test, decrement, branch (MC68851 only)
PFLUSH	**pflush**	**FC,&M**	Invalidate entries in ATC
	pflush	**FC,&M,EA**	
PFLUSHA	**pflusha**		Invalidate all ATC entries
PFLUSHS	**pflushs**	**FC,&M**	Invalidate entries in ATC including shared entries (MC68851 only)
	pflushs	**FC,&M,EA**	
PFLUSHR	**pflushr**	**EA**	Invalidate ATC and RPT entries
PLOADR	**ploadr**	**FC,EA**	Load an entry into ATC
PLOADW	**ploadw**	**FC,EA**	Load an entry into ATC
PMOVE			Move to/from MMU register
	pmove	**EA,PMRn**	(MC68851 only)
	pmove	**PMRn,EA**	(MC68851 only)[10]
	pmove	**EA,MRn**	(MC68030 only)
	pmove	**MRn,EA**	(MC68030 only)
	pmove.d	**%crp,EA**	
	pmove.d	**EA,%crp**	
	pmove.d	**%srp,EA**	
	pmove.d	**EA,%srp**	
	pmove.d	**%drp,EA**	(MC68881 only)
	pmove.d	**EA,%drp**	(MC68881 only)
	pmove.l	**%tc,EA**	
	pmove.l	**EA,%tc**	
	pmove.l	**%tt,EA**	(MC68881 only)
	pmove.l	**EA,%tt**	(MC68881 only)
	pmove.w	**%bac,EA**	(MC68881 only)
	pmove.w	**EA,%bac**	(MC68881 only)
	pmove.w	**%bad,EA**	(MC68881 only)
	pmove.w	**EA,%bad**	(MC68881 only)
	pmove.w	**%ac,EA**	
	pmove.w	**EA,%ac**	
	pmove.w	**%psr,EA**	(MC68881 only)
	pmove.w	**EA,%psr**	
	pmove.w	**%pcsr,EA**	

10. Cannot move to %pcsr register.

MC68851 INSTRUCTION FORMATS			
MNEMONIC	ASSEMBLER SYNTAX		OPERATION
PMOVE (cont'd)			
	pmove.b	**%cal,EA**	(MC68881 only)
	pmove.b	**EA,%cal**	(MC68881 only)
	pmove.b	**%val,EA**	(MC68881 only)
	pmove.b	**EA,%val**	(MC68881 only)
	pmove.b	**%scc,EA**	(MC68881 only)
	pmove.b	**EA,%scc**	(MC68881 only)
PMOVEFD			Move to MMU register, flush disabled
	pmovefd	**EA,MRn**	(MC68030 only)
	pmovefd.d	**EA,%crp**	(MC68030 only)
	pmovefd.d	**EA,%srp**	(MC68030 only)
	pmovefd.l	**EA,%tc**	(MC68030 only)
	pmovefd.l	**EA,%tt**	(MC68030 only)
PRESTORE	**prestore**	**EA**	PMMU restore function (MC68881 only)
PSAVE	**psave**	**EA**	PMMU save function (MC68881 only)
PScc	**psCC**	**EA**	Set on PMMU condition (MC68881 only)
PTESTR	**ptestr**	**FC,EA,&D**	Get information about
	ptestr	**FC,EA,&D,%an**	logical address
PTESTW	**ptestr**	**FC,EA,&D**	Get information about
	ptestr	**FC,EA,&D,%an**	logical address
PTRAPcc	**ptrapCC**		Trap on PMMU condition
	ptrapCC.A	**&I**	(MC68881 only)
PVALID	**pvalid**	**%val,EA**	Validate a pointer
	pvalid	**%an,EA**	(MC68881 only)