sun microsystems

Assembly Language Reference Manual

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Credits and Acknowledgements

This *Assembly Language Reference Manual for the Sun Workstation* started life as an edited version of the MICAL Manual for the Intel 8080, written by Mike Patrick; transformed by James L. Gula and Thomas J. Teixeira, March 1980; revised by Henry McGilton at Unisoft Systems of Berkeley Corporation during March 1982; rewritten by Henry McGilton and Richard Tuck, of Sun Microsystems, during October and November 1982.

Trademarks

Multibus is a trademark of Intel Corporation.

Sun Workstation is a trademark of Sun Microsystems Incorporated.

UNIX is a trademark of Bell Laboratories.

Contents

Tables

Preface

This manual is the Programmer's Reference Manual for as – the assembler for the UNIX† system running on the Sun Workstation. as converts source programs written in *Assembly Language* into a form that the linker utility, *ld*(1) will turn into a program that is runnable on the UNIX operating system.

What as Provides

as provides the assembly language programmer with a minimal set of facilities to write programs in assembly language. Since most programming is done in high-level languages, as doesn't provide any elaborate macro facilities or conditional assembly features. It is assumed that the volume of assembly code produced is so small that these facilities aren't required. If they are needed, the C preprocessor (see *cpp* (1)) can be used to provide them.

Scope of This Manual

This manual describes the syntax and usage of the as assembler for the Motorola MC68010 and MC68020 microprocessors and the MC68881 coprocessor. The basic format of as is loosely based on the Digital Equipment Corporation's Macro-11 assembler described in DEC's publication DEC-11-OMACA-A-D but also contains elements of the UNIX PDP-11 as assembler. The instruction mnemonics and effective address format are derived from a Motorola publication on the MC68000: the *MACSS MC68000 Design Specification Instruction Set Processor* dated June 30, 1979.

Audience

This is a *reference manual* as opposed to a treatise on writing in assembly language. It is assumed that the reader is familiar with the concepts of machine architecture, the reasons for an assembler, the ideas of instruction mnemonics, operands, and effective address modes, and assembler directives. It is also assumed that the reader is familiar with the relevant processors, their instruction sets and addressing modes, and especially the irregularities in them.

Further Reading

Motorola MC68010 16-bit Microprocessor Programmer's Reference Manual.

Motorola MC68020 32-bit Microprocessor User's Manual.

Motorola MC68881 Floating-Point Coprocessor User's Manual.

† UNIX is a trademark of AT&T Bell Laboratories.

1

Introduction

1

Introduction

1.1. Using the Assembler

By convention, the assembly language source code of the program should be in one or more files with a *.s* suffix. Suppose that your program is in two files called *parts.s* and *rest.s*. To run the assembler, type the command:

```
tutorial% as parts.s rest.s
```

`as` runs silently (if there are no errors), and generates a file called *a.out*.

`as` also accepts several command line options. These are:

–o *file*
Place the output of the assembler in *file* instead of *a.out*.

–m68010 or **–10**
Accept only the MC68010 instruction set and addressing modes. This is the default on Sun-2 systems. This also puts the MC68010 machine type tag into the *a.out* file.

–m68020 or **–20**
Accept the full MC68020 and MC68881 instruction set and addressing modes. This includes the MC68010 instruction set and addressing modes as a subset, and is the default on Sun-3 systems. This also puts the MC68020 machine type tag into the *a.out* file.

–O Perform span-dependent instruction resolution over each entire file, rather than just over each procedure (see the description of the *.proc* pseudo-op in Chapter 5.

–R Make initialized data segments read-only (actually the assembler places them at the end of the .text area).

–L Keep local (compiler-generated) symbols that start with the letter L. This is a debugging feature. If the –L option is omitted, the assembler discards those symbols and does not include them in the symbol table.

–j Make all jumps to external symbols (jsr and jmp) PC-relative rather than long-absolute. This is intended for use when the programmer knows that the program is short, since it only permits jumps (forward or back) up to 32K bytes long. If there are any externals which are too far away, the loader will complain when the program is linked.

–J Suppress span-dependent instruction calculations and force all branches and calls to take the most general form. This is used when assembly time must be minimized, but program size and run time are not important.

–h Suppress span-dependent instruction calculations and force all branches to be of medium length, but all calls to take the most general form. This is used when assembly time must be minimized, but program size and run time are not important. This option results in a smaller and faster program than that produced by the –J option, but some very large programs may not be able to use it because of the limits of the medium-length branches.

–d2 This is intended for small stand-alone programs. The assembler makes all program references PC-relative and all data references short-absolute. Note that the –j option does half this job anyway.

–e Allow control sections to begin on any even-numbered byte boundary, rather than only multiples of four. Do not use this with programs intended for use with the 68020.

Readers should also consult the UNIX Programmer's Manual page for the *man* entry on `as`.

1.2. Notation

The notation used in this manual is a somewhat modified Backus-Naur Form (BNF). A string of characters on its own stands for itself, for example:

```
WIDGET
```

is an occurrence of the literal string '`WIDGET`', and:

```
1983
```

is an occurrence of the literal constant 1983. An element enclosed in < and > signs is a non-terminal symbol, and must eventually be defined in terms of some other entities. For example,

<identifier>

stands for the syntactic construct called 'identifier', which is eventually defined in terms of basic objects. A syntactic object followed by an ellipsis:

<thing> . . .

denotes one or more occurrences of *<thing>*. Syntactic objects occurring one after the other, as in:

<first thing>   *<second thing>*

simply means an occurrence of *first thing* followed by *second thing*. Syntactic elements separated by a vertical bar sign (|), as in:

```
<letter> | <digit>
```

mean an occurrence of *<letter>* or *<digit>* but not both. Brackets and braces define the order of interpretation. Brackets also indicate that the syntax described by the subexpression they enclose is optional. That is:

```
[ <thing> ]
```

denotes zero or one occurrences of *<thing>*, while:

```
{ <thing one> | <thing two> } <thing three>
```

denotes a *<thing one>* or a *<thing two>*, followed by a *<thing three>*.

2

Elements of Assembly Language

2

Elements of Assembly Language

This chapter covers the lexical elements which comprise an assembly language program. (Chapter 3 discusses the rules for expression and operand formation.) Topics covered in this chapter are:

- *Character set* which the assembler recognizes,
- Rules for *identifiers* and *labels*,
- Syntax for *numeric constants*,
- Syntax for *string constants*,
- The *assembly location counter*.

An assembly language program is ultimately constructed from characters. Characters are combined to make up *lexical elements* or *tokens* of the language. Combinations of tokens form assembly language *statements*, and sequences of statements form an assembly language program. This section describes the basic lexical elements of `as`.

2.1. Character Set

`as` recognizes the following character set:

- The *letters* `A` through `Z` and `a` through `z`.
- The *digits* `0` through `9`.
- The ASCII *graphic characters* — the printing characters other than letters and digits.
- The ASCII *non-graphics*: space, tab, carriage return, and newline (also known as linefeed).

2.2. Identifiers

Identifiers are used to tag assembler statements (where they are called *labels*), as location tags for data, and as the symbolic names of constants.

An identifier in an `as` program is a sequence of from 1 to 255 characters from the set:

- Upper case letters `A` through `Z`.
- Lower case letters `a` through `z`.
- Digits `0` through `9`.

- The characters underline (_), period (.), and dollar sign ($).

The first character of an identifier must not be numeric. Other than that restriction, there are a few other points to note:

- All characters of an identifier are significant and are checked in comparisons with other identifiers.
- Upper case letters and lower case letters are distinct, so that `kit_of_parts` and `KIT_OF_PARTS` are two different identifiers.
- Although the period (.) and dollar sign ($) characters can be used to construct identifiers, they are reserved for special purposes (pseudo-ops for instance) and should not appear in user-defined identifiers.

Examples of Identifiers

```
Grab_Hold   Widget   Pot_of_Message   MAXNAME
```

2.3. Numeric Labels

A numeric label consists of a digit (0 to 9) followed by a colon. As in the case of alphanumeric labels, a numeric label assigns the current value of the location counter to the symbol. However, several numeric labels with the same digit may be used within the same assembly. References of the form *n*`b` refer to the first numeric label named *n b*ackwards from the reference; *n*`f` symbols refer to the first numeric label named *n f*orwards from the reference.

2.4. Local Labels

Local labels are a special form of identifier which are strictly local to a control section (see Section 5.4). Local labels provide a convenient means of generating labels for branch instructions and such. Use of local labels reduces the possibility of multiply defined labels in a program, and separates entry point labels from local references, such as the top of a loop. Local labels cannot be referenced from outside the current assembly unit. Local labels are of the form *n*$ where *n* is any integer. Valid local labels include:

```
1$          27$          394$
```

2.5. Scope of Labels

The *scope* of a label is the 'distance' over which it is visible to other parts of the program which may reference it. An ordinary label which tags a location in the program or data is visible only within the current assembly. An identifier which is designated as an external identifier via a `.globl` directive is visible to other assembly units at link time.

Local labels have a scope, or span of reference, which extends between one ordinary label and the next. Every time an ordinary label is encountered, all previous local labels associated with the current location counter are discarded, and a new local label scope is created. The following example illustrates the scopes of the different kinds of labels:

```
first:   addl    d0,d1     | creates a new local label scope

100$:    addqw   #7,d3     | first appearance of 100$
         bccs    100$      | branches to the label above

second:  andl    #0x7ff,d4 | above 100$ has gone away

100$:    cmpw    d1,d3     | this is a different 100$
         beqs    100$      | branches to the previous instruction

third:   movw    d0,d7     | now 100$ has gone away again
         beqs    100$      | generates an error message if no 100$ below
```

The labels *first*, *second*, and *third* all have a scope which is the entire source file containing them. The first appearance of the local label 100$ has a scope which extends between *first* and *second.* The second appearance of the local label 100$ has a scope which extends between *second* and *third.* After the appearance of the label *third*, the branch to 100$ will generate an error message because that label is no longer defined in this scope.

2.6. Constants

There are two forms of constants available to `as` users, namely *numeric* constants and *string* constants. All constants are considered absolute quantities when they appear in an expression (see Section 3 for a discussion on absolute and relocatable expressions).

2.7. Numeric Constants

`as` assumes that any token which starts with a digit is a numeric constant. `as` accepts numeric quantities in either decimal (base 10), hexadecimal (base 16), or octal (base 8) radices. Numeric constants can represent quantities up to 32 bits in length.

Decimal numbers consist of between one and ten decimal digits (0 through 9). The range of decimal numbers is between –2,147,483,648 and 2,147,483,647. Note that you can't have commas in decimal numbers even though they are shown here for readability. Note also that decimal numbers can't be written with leading zeros, because a numeric constant starting with a zero is taken as either an octal constant or a hexadecimal constant, as described below.

Hexadecimal constants must start with the notation `0x` or `0X` (zero-ex) and can then have between one and eight hexadecimal digits. The hexadecimal digits consist of the decimal digits `0` through `9` and the hexadecimal digits `a` through `f` or `A` through `F`.

Octal constants must start with the digit `0`. There can then be from one to 11 octal digits (0 through 7) in the number. But note that 11 octal digits is 33 bits, so the largest octal number is 037777777777.

Floating-point constants must start with `#0r` or `#0R`, which may be followed by an optional sign and either a number, an infinity or a nan ("not a number"). The syntax is

{#0r | #0R} [+ | -] {<*number*> | inf | nan} where the syntax of a <*number*> is

```
{<digits> [. [<digits>]] | . <digits>} [E [+ | -] <digits>]
```

and <*digits*> *is a string of decimal digits.*

2.8. String Constants

A string is a sequence of ASCII characters, enclosed in quote signs ".

Within string constants, the quote sign is represented by a backslash character followed by a quote sign. The backslash character itself is represented by two backslash characters. Any other character can be represented by a backslash character followed by one, two, or three octal digits. The table below shows the octal representation of some of the more common non-printing characters.

Character	***Octal Representation***
Backspace	010
Horizontal Tab	011
Newline (Linefeed)	012
Formfeed	014
Carriage Return	015

2.9. Assembly Location Counter

The assembly location counter is the period character (`.`). It is colloquially known as **dot**. When used in the operand field of any statement, **dot** represents the address of the first byte of the statement. Even in assembler directives, **dot** represents the address of the start of that assembler directive. For example, if **dot** appears as the third argument in a `.long` directive, the value placed at that location is the address of the first location of the directive — **dot** is not updated until the next machine instruction or assembler directive. For example:

```
Ralph:  movl .,a0   |  load value of Ralph into a0
```

You can reserve storage by advancing **dot**. For example, the statement

```
Table:  .=.+0x100
```

reserves 256 bytes (100 hexadecimal) of storage, with the address of the first byte as the value of Table. This is exactly equivalent to using `.skip` (the preferred syntax) as follows:

```
Table:  .skip   0x100
```

The value of **dot** is always relative to the start of the current control section. For example,

```
. = 0x1000
```

doesn't set **dot** to absolute location 0x1000, but to location 0x1000 relative to the start of the current control section. This practice is not recommended.

3

Expressions

3

Expressions

Expressions are combinations of operands (numeric constants and identifiers) and operators, forming new values. The sections below define the operators which as provides, then gives the rules for combining terms into expressions.

3.1. Operators

Identifiers and numeric constants can be combined, via arithmetic operators, to form *expressions*. as provides *unary* operators and *binary* operators, as described below.

Table 3-1 *Unary Operators in Expressions*

Operator	*Function*	*Description*
–	unary minus	Returns the two's complement of its argument.
~	logical negation	Returns the one's complement (logical negation) of its argument.

Table 3-2 *Binary Operators in Expressions*

Operator	*Function*	*Description*
+	addition	Arithmetic addition of its arguments.
–	subtraction	Arithmetic subtraction of its arguments.
*	multiplication	Arithmetic multiplication of its arguments.
/	division	Arithmetic division of its arguments. Note that division in `as` is *integer* division, which truncates towards zero.

Each operator works on 32-bit numbers. If the value of a particular term occupies only 8 bits or 16 bits, it is sign extended to a full 32-bit value.

3.2. Terms

A term is a component of an expression. A term may be any one of the following:

- A numeric constant, whose 32-bit value is used. The assembly location counter, known as **dot**, is considered a number in this context.
- An identifier.
- An expression or term enclosed in parentheses `( )`. Any quantity enclosed in parentheses is evaluated before the rest of the expression. This can be used to alter the normal left-to-right evaluation of expressions — for example, differentiating between `a*b+c` and `a*(b+c)` or to apply a unary operator to an entire expression — for example, `–(a*b+c)`.
- A term preceded by a unary operator. For example, both `double_plus_ungood` and `~double_plus_ungood` are terms.

Multiple unary operators can be used in a term. For example, `--positive` has the same value as `positive`.

3.3. Expressions

Expression are combinations of terms joined together by binary operators. An expression is always evaluated to a 32-bit value.

If the operand only requires a single-byte value (a `.byte` directive or an `addq` instruction, for example) the low-order eight bits of the expression are used.

If the operand only requires a 16-bit value (a `.word` directive or a `movem` instruction, for example) the low-order 16 bits of the expression are used.

Expressions are evaluated left to right with no operator precedence. Thus

```
1 + 2 * 3
```

evaluates to 9, not 7. Unary operators have precedence over binary operators since they are considered part of a term, and both terms of a binary operator must be evaluated before the binary operator can be applied.

A missing expression or term is interpreted as having a value of zero. In this case, an *Invalid expression* error is generated.

An *Invalid Operator* error means that a valid end-of-line character or binary operator was not detected after the assembler processed a term. In particular, this error is generated if an expression contains an identifier with an illegal character, or if an incorrect comment character was used.

3.4. Absolute, Relocatable, and External Expressions

When an expression is evaluated, its value is either absolute, relocatable, or external:

An expression is absolute if its value is fixed.

- An expression whose terms are constants is absolute.
- An identifier whose value is a constant via a direct assignment statement is absolute.
- A relocatable expression minus a relocatable term is absolute, if both items belong to the same program section.

An expression is relocatable if its value is fixed relative to a base address, but will have an offset value when it is linked or loaded into memory. All labels of a program defined in relocatable sections are relocatable terms.

Expressions which contain relocatable terms must only *add or subtract constants to their value.* For example, assuming the identifiers `widget` and `blivet` were defined in a relocatable section of the program, then the following demonstrates the use of relocatable expressions:

Expression	*Description*
`widget`	is a simple relocatable term. Its value is an offset from the base address of the current control section.
`widget+5`	is a simple relocatable expression. Since the value of widget is an offset from the base address of the current control section, adding a constant to it does not change its relocatable status.
`widget*2`	Not relocatable. Multiplying a relocatable term by a constant invalidates the relocatable status.
`2-widget`	Not relocatable, since the expression cannot be linked by adding `widget`'s offset to it.
`widget-blivet`	Absolute, since the offsets added to `widget` and `blivet` cancel each other out.

An expression is external (or global) if it contains an external identifier not defined in the current program. With one exception, the same restrictions on expressions containing relocatable identifiers apply to expressions containing external identifiers. The exception is that the expression

```
widget-blivet
```

is incorrect when both `widget` and `blivet` are external identifiers — you cannot subtract external relocatable expressions. In addition, you cannot multiply or divide *any* relocatable expression.

4

Assembly Language Program Layout

4

Assembly Language Program Layout

An `as` program consists of a series of statements. Several statements can be written on one line, but statements cannot cross line boundaries. The format of a statement is:

[< *label field* >] [< *op-code* > [< *operand field* >]]

It is possible to have a statement which consists of only a label field.

The fields of a statement can be separated by spaces or tabs. There must be at least one space or tab separating the op-code field from the operand field, but spaces are unnecessary elsewhere. Spaces may appear in the operand field. Spaces and tabs are significant when they appear in a character string (for instance, as the operand of an `.ascii` pseudo-op) or in a character constant. In these cases, a space or tab stands for itself.

A line is a sequence of zero or more statements, optionally followed by a comment, ending with a < *newline* > character. A line can be up to 4096 characters long. Multiple statements on a line are separated by semicolons. Blank lines are allowed. The form of a line is:

[< *statement* > [; < *statement* > ...]] [| < *comment* >]

4.1. Label Field

Labels are identifiers which the programmer may use to tag the locations of program and data objects. The format of a <*label field*> is:

<*identifier*>: [<*identifier*>:] . . .

If present, a label *always* occurs first in a statement and *must* be terminated by a colon:

```
sticky:                    |  label defined here.
```

More than one label may appear in the same source statement, each one being terminated by a colon:

```
presson: grab: hold:    |  multiple labels defined here.
```

The collection of label definitions in a statement is called the *label field.*

When a label is encountered in the program, the assembler assigns that label the value of the current location counter. The value of a label is relocatable. The symbol's absolute value is assigned when the program is linked via the UNIX system linker *ld*(1).

4.2. Operation Code Field

The operation code field of an assembly language statement identifies the statement as either a machine instruction or an assembler directive.

One or more spaces (or tabs) must separate the operation code field from the following operand field in a statement. Spaces or tabs are unnecessary between the label and operation code fields, but they are recommended to improve readability of the program.

A machine instruction is indicated by an instruction mnemonic. The assembly language statement is intended to produce a single executable machine instruction. The operation of each instruction is described in the manufacturer's user manual. Some conventions used in `as` for instruction mnemonics are described in Chapter 6 and a complete list of the instructions is presented in Appendix B.

An assembler directive, or pseudo-op, performs some function during the assembly process. It does not produce any executable code, but it may assign space for data in a program.

Note that `as` expects that all instruction mnemonics in the op-code field should be in *lower case only.* Use of any upper case letters in instruction mnemonics gives rise to an error message.

The names of register operands must also be in lower case only. This behavior differs from the case of identifiers, where both upper and lower case letters may be used and are considered distinct.

Many MC68010 and MC68020 machine instructions can operate upon byte (8-bit), word (16-bit), or long word (32-bit) data. The size which the programmer requires is indicated as part of the instruction mnemonic. For instance, a `movb` instruction moves a `b`yte of data, a `movw` instruction moves a 16-bit `w`ord of data, and a `movl` instruction moves a 32-bit `l`ong word of data. In general, the default size for data manipulation instructions is word.

Many MC68881 machine instructions can operate on byte, word or long word integer data, on single-precision (32-bit), double-precision (64-bit) or extended-precision (96-bit) floating-point data or on packed-decimal (96-bit) data. The size required is specified as part of the instruction mnemonic by a trailing "`b`", "`w`", "`l`", "`s`", "`d`", "`x`" or "`p`", respectively.

An alternate coprocessor id can be specified for MC68881 instructions by appending `@id` to the opcode, such as `fadd@2`. If you don't do this, the coprocessor id specified by the most recent `.cpid` pseudo-operation is used. (See Chapter 5.)

Similarly, branch instructions can use a long or short offset specifier to indicate the destination. So the `beq` instruction uses a 16-bit offset, whereas the `beqs` uses a short (8-bit) offset.

Note that this implementation of `as` provides an extended set of branch instructions which start with the letter `j` instead of the letter `b`. If the programmer uses the `j` forms, the assembler computes the offset size for the instruction. See Section 1.1 for the assembler options which control this.

4.3. Operand Field

The *operand field* of an assembly language statement supplies the arguments to the machine instruction or assembler directive.

`as` makes a distinction between the *<operand field>* and individual *<operands>* in a machine instruction or assembler directive. Some machine instructions and assembler directives require two or more arguments, and each of these is referred to as an "operand".

In general, an operand field consists of zero or more operands, and in all cases, operands are separated by commas. In other words, the format for an *<operand field>* is:

```
[<operand> [,<operand>] . . . ]
```

The format of the operand field for machine instructions is the same for all instructions, and is described in Chapter 6. The format of the operand field for assembler directives depends on the directive itself, and is included in the directive's description in Chapter 6 of this manual.

Depending upon the machine instruction or assembler directive, the *operand field* consists of one or more *operands*. The kinds of objects which can form an operand are:

- Register operands
- Register pairs
- Address Operands
- String constants
- Floating-point constants
- Register lists
- Expressions

Register operands in a machine instruction refer to the machine registers of the processor or coprocessor.

Note that register names *must* be in lower case; `as` does not recognize register names in upper case or a combination of upper case and lower case.

Expressions are described in Chapter 3, address operands in Chapter 6, and floating-point constants in Chapter 2.

4.4. Comment Field

`as` provides the means for the programmer to place comments in the source code. There are two ways of representing comments.

A line whose first *non-whitespace* character is the hash character (#) is considered a comment. This feature is handy for passing C preprocessor output through the assembler. For example, these lines are comments:

```
# This is a comment line.
       # And this one is also a comment line.
```

The other way to introduce a comment is when a comment field appears as a part of a statement. The comment field is indicated by the presence of the vertical bar character (|) after the rest of the source statement.

The comment field consists of all characters on a source line following and including the comment character. The assembler ignores the comment field. Any character may appear in the comment field, with the obvious exception of the <newline> character, which starts a new line.

An assembly language source line can consist of just the comment field. For example, the two statements below are quite acceptable to the assembler:

```
| This is a comment field.
| So is this.
```

4.5. Direct Assignment Statements

A direct assignment statement assigns the value of an arbitrary expression to a specified identifier. The format of a direct assignment statement is:

```
<identifier> =  <expression>
```

Examples of direct assignments are:

```
vect_size    =  4
vectora      =  0xFFFE
vectorb      =  vectora-vect_size
CRLF         =  0x0D0A

dtemp        =  d0          |  use register d0 as temporary
```

Any identifier defined by direct assignment may be redefined later in the program, in which case its value is the result of the last such statement. This is analogous to the `SET` operation found in other assemblers.

A local identifier may be defined by direct assignment, though this doesn't make much sense.

Register identifiers may not be redefined.

An identifier which has already been used as a label may not be redefined, since this would be tantamount to redefining the address of a place in the program. In addition, an identifier which has been defined in a direct assignment statement cannot later be used as a label. Both situations give rise to assembler error messages.

If the <*expression*> is absolute, the identifier is also absolute, and may be treated as a constant in subsequent expressions. If the <*expression*> is relocatable, however, the < *identifier* > is also relocatable, and it is considered to be declared the same program section as the expression.

If the <*expression*> contains an external identifier, the identifier defined by the = statement is also considered external. For example:

```
.globl  X     | X is declared as external identifier
holder = X    | holder becomes an external identifier
```

assigns the value of `X` (zero if it is undefined) to `holder` and makes `holder` an external identifier. External identifiers may be defined by direct assignment.

5

Assembler Directives

5

Assembler Directives

Assembler directives are also known as *pseudo operations* or *pseudo-ops*. Pseudo-ops are used to direct the actions of the assembler, and to achieve effects such as generating data. The pseudo-ops available in `as` are listed in Table 5-1.

Table 5-1 *Assembler Directives*

Pseudo Operation	***Description***
`.ascii`	Generates a sequence of ASCII characters.
`.asciz`	Generates a sequence of ASCII characters, terminated by a zero byte.
`.byte`	Generates a sequence of bytes in data storage.
`.word`	Generates a sequence of words in data storage.
`.long`	Generates a sequence of long words in data storage.
`.single`	Generates a sequence of single-precision floating-point constants in data storage.
`.double`	Generates a sequence of double-precision floating-point constants in data storage.
`.text`	Specifies that generated code be placed in the *text* control section until further notice.
`.data`	Specifies that generated code be placed in the *data* control section until further notice.
`.data1`	Specifies that generated code be placed in the *data1* control section until further notice.
`.data2`	Specifies that generated code be placed in the *data2* control section until further notice.
`.bss`	Specifies that space will be reserved in the *bss* control section until further notice.
`.globl`	Declares an identifier as global (external).
`.comm`	Declares the name and size of a *common* area.
`.lcomm`	Reserves a specified amount of space in the *bss* area.

Table 5-1 *Assembler Directives— Continued*

Pseudo Operation	*Description*
`.skip`	Advances the location counter by a specified amount.
`.align`	Forces location counter to next one-, two- or four-byte boundary.
`.even`	Forces location counter to next word (even-byte) boundary.
`.stabx`	Builds special symbol table entries. These directives are included for the benefit of compilers which generate information for the symbolic debuggers *dbx* and *dbxtool*.
`.proc`	Separates procedures for faster span-dependent instruction resolution.
`.cpid`	Assigns a coprocessor number.

These assembler directives are discussed in detail in the sections following.

5.1. `.ascii` — Generate Sequence of Character Data

The `.ascii` directive translates character strings into their ASCII equivalents for use in the source program. The format of the `.ascii` directive is:

```
[<label>:] .ascii     "<character string>"
```

<character string>

contains any character or escape sequence which can appear in a character string. Obviously, a newline must not appear within the character string. A newline can be represented by the escape sequence \012.

The following examples illustrate the use of the `.ascii` directive:

Octal Code Generated:	***Statement:***
150 145 154 154 157 040 164 150 145 162 145	`.ascii "hello there"`
127 141 162 156 151 156 147 055 007 007 040 012	`.ascii "Warning-\007\007 \0`
141 142 143 144 145 146 147	`.ascii "abcdefg"`

5.2. `.asciz` — Generate Zero-Terminated Sequence of Character Data

The `.asciz` directive is equivalent to the `.ascii` directive except that a zero byte is automatically inserted as the final character of the string. This feature is intended for generating strings which C programs can use.

The following examples illustrate the use of the `.asciz` directive:

Octal Code Generated:	***Statement:***
`110 145 154 154 157 040` `127 157 162 144 041 000`	`.asciz "Hello World!"`

`124 150 105 040 107 162`
`145 141 164 040 120 122`
`117 115 160 153 151 156`
`040 163 164 162 151 153`
`145 163 040 141 147 141`
`151 156 041 000`
	`.asciz "The Great PROMpkin strikes again!"`

5.3. `.byte, .word, .long` — Generate Data

The `.byte, .word, .long, .single,` and `initialize them with specified values.`

The format of the various forms of data generation statements is:

```
[<label>:]    .byte     [<expression>] [, <expression>] ...
[<label>:]    .word     [<expression>] [, <expression>] ...
[<label>:]    .long     [<expression>] [, <expression>] ...
[<label>:]    .single   [<expression>] [, <expression>] ...
[<label>:]    .double   [<expression>] [, <expression>] ...
```

The `.byte` directive reserves one byte (8 bits) for each expression in the operand field, and initializes it to the low-order 8 bits of the corresponding expression.

The `.word` directive reserves one word (16 bits) for each expression in the operand field, and initializes it to the low-order 16 bits of the corresponding expression.

The `.long` directive reserves one long word (32 bits) for each expression in the operand field, and initializes it to the value of the corresponding expression.

The `.single` directive reserves one long word for each expression in the operand field, and initializes it to the low-order 32 bits of the corresponding expression.

The `.double` directive reserves a pair of long words for each expression in the operand field, and initializes them to the value of the corresponding expression.

Multiple expressions can appear in the operand field of the `.byte`, `.word`, `.long`, `.single`, and `.double` directives. Multiple expressions must be separated by commas.

5.4. .text, .data, .bss — Switch Location Counter

These statements change the 'control section' where assembled code will be loaded.

as (and the UNIX system linker) view programs as divided into three distinct sections or address spaces:

text is the address space where the executable machine instructions are placed.

data is the address space where initialized data is placed. The assembler actually knows about three data areas, namely, *data*, *data1*, and *data2*. The second and third data areas are mainly for the benefit of the C compiler and are of minimal interest to the assembly language programmer.

If the **–R** option is coded on the as command line, it means that the initialized data should be considered read-only. It is actually placed at the end of the *text* area.

bss is the address space where the uninitialized data areas are placed. Also, see the .lcomm directive described below.

For historical reasons, the different areas are frequently referred to as 'control sections' (csects for short).

These sections are equivalent as far as as is concerned, with the exception that no instructions or data are generated for the *bss* section — only its size is computed and its symbol values are output.

During the first pass of the assembly, as maintains a separate location counter for each section. Consider the following code fragments:

```
        .text               | place next instruction
code:   movw    d1,d2       | in text section

        .data               | now generate data in
grab:   .long   27          | data section

        .text               | now revert to text
more:   addw    d2,d1       | section

        .data               | now back to data section
hold:   .byte   4
```

During the first pass, as creates the intermediate output in two separate chunks: one for the *text* section and one for the *data* section. In the *text* section, code immediately precedes more; in the *data* section, grab immediately precedes hold. At the end of the first pass, as rearranges all the addresses so that the sections are sent to the output file in the order: *text*, *data* and *bss*.

The resulting output file is an executable image file with all addresses correctly resolved, with the exception of undefined *.globl's* and *.comm's*.

For more information on the format of the assembler's output file, consult the *a.out* (5) entry in the UNIX System Programmer's Reference Manual.

5.5. `.skip` — Advance the Location Counter

The `.skip` directive reserves storage by advancing the current location counter a specified amount. The format of the `.skip` directive is:

```
[<label>:]     .skip  <size>
```

where <size> is the number of bytes by which the location counter should be advanced. The `.skip` directive is equivalent to performing direct assignment on the location counter. For instance, a `.skip` directive like this:

```
Table   .skip  1000
```

reserves 1000 bytes of storage, with the value of Table equal to the address of the first byte.

5.6. `.lcomm` — Reserve Space in `bss` Area

The `.lcomm` directive is a compact way to get a specific amount of space reserved in the `bss` area. The format of the `.lcomm` directive is:

```
.lcomm  <name>,<size>
```

where *<name>* is the name of the area to reserve, and *<size>* is the number of bytes to reserve. The `.lcomm` directive specifically reserves the space in the `bss` area, regardless of which location counter is currently in effect.

A `.lcomm` directive like this:

```
.lcomm    lower_forty,1200
```

is equivalent to these directives:

```
         .bss           |  switch to .bss area
lower_forty:  .skip  size
revert to previous control section
```

5.7. `.globl` — Designate an External Identifier

A program may be assembled in separate modules, and then linked together to form a single executable unit. See the *ld*(1) command in the UNIX Commands Reference Manual.

External identifiers are defined in each of these separate modules. An identifier which is defined (given a value) in one module may be referenced in another module by declaring it external in *both* modules.

There are two forms of external identifiers, namely, those declared with the `.globl` and those declared with the `.comm` directive. The `.comm` directive is described in the next section.

External symbols are declared with the `.globl` assembler directive. The format is:

```
.globl    <symbol> [, <symbol>] . . .
```

For example, the following statements declare the array `TABLE` and the routine `SRCH` as external symbols, and then define them as locations in the current control section:

```
          .globl    TABLE, SRCH
TABLE:    .word     0,0,0,0,0
SRCH:     movw      TABLE,d0

          etc.
```

5.8. `.comm` — Define Name and Size of a Common Area

The `.comm` directive declares the name and size of a common area, for compatibility with FORTRAN and other languages which use common. The format of the `.comm` statement is:

```
.comm <name>, <constant expression>
```

where *name* is the name of the common area, and *constant expression* is the size of the common area. The `.comm` directive implicitly declares the identifier *name* as an external identifier.

`as` does **not** allocate storage for *common* symbols; this task is left to the linker. The linker computes the maximum declared size of each *common* symbol (which may appear in several load modules), allocates storage for it in the final *bss* section, and resolves linkages. If, however, <*name*> appears as a global symbol (label) in any module of the program, all references to <*name*> are linked to it, and no additional spaces is allocated in the *bss* area.

5.9. `.align` — Force Location Counter to Particular Byte Boundary

The `.align` directive advances the location counter to the next one-, two- or four-byte boundary, if it is not currently on such a boundary. Intervening bytes are filled with zeros. The format of the `.align` directive is:

```
.align  <size>
```

where `<size>` must be an assembler expression which evaluates to 1, 2 or 4.

This directive is necessary because word and long word data values must lie on even-byte boundaries, because machine instructions must start on even-byte boundaries, and because the MC68020 is much more efficient if word and long word data are on even-byte and four-byte boundaries, respectively.

5.10. `.even` — Force Location Counter to Even Byte Boundary

The `.even` directive advances the location counter to the next even-byte boundary, if its current value is odd. This directive is necessary because word and long word data values must lie on even-byte boundaries, and also because machine instructions must start on even-byte boundaries. `.even` is equivalent to `.align 2`.

5.11. `.stabx` — Build Special Symbol Table Entry

The `.stab`*x* directives are provided for the use of compilers which can generate information for the symbolic debuggers *dbx* and *dbxtool*. The directives `.stabs`, `.stabd`, and

5.12. `.proc` — Separate Procedures for Span-Dependent Instruction Resolution

The `.proc` directive separates procedures for span-dependent instruction resolution. In its absence the assembler does span-dependent instruction resolution over entire files. If `.proc` is used, the resolution is done between occurrences of the directive and between either end of the file and its nearest occurrences. Since the algorithm used requires more than linear time, using `.proc` can save significant time for large assemblies. Branch instructions must not cross `.proc` directives, although calls may.

5.13. `.cpid` — Name Default Coprocessor ID

The `.cpid` directive gives the assembler a coprocessor id value to use for MC68881 instructions that don't have an explicit coprocessor id given. The form of the directive is

```
.cpid  <id>
```

If no `.cpid` directive is given in a program, a value of 1 is assumed. Since no Sun systems currently have more than one coprocessor, you don't need to use the directive.

6

Instructions and Addressing Modes

6

Instructions and Addressing Modes

This chapter describes the conventions used in `as` to specify instruction mnemonics and addressing modes. The information in this chapter is specific to the machine instructions and addressing modes of the MC68010 and MC68020 microprocessors and the MC68881 coprocessor.

6.1. Instruction Mnemonics

The instruction mnemonics which `as` uses are based on the mnemonics described in the relevant Motorola processor manuals. However, `as` deviates from them in several areas.

Most of the MC68010 and MC68020 instructions can apply to byte, word or long operands. Instead of using a qualifier of `.b`, `.w`, or `.l` to indicate byte, word, or long as in the Motorola assembler, `as` appends a suffix to the normal instruction mnemonic, thereby creating a separate mnemonic to indicate which length operand was intended.

For example, there are three mnemonics for the *or* instruction: `orb`, `orw` and `orl`, meaning or byte, or word, and or long, respectively.

Instruction mnemonics for instructions with unusual opcodes may have additional suffixes. Thus in addition to the normal *add* variations, there also exist `addqb`, `addqw` and `addql` for the *add quick* instruction.

Branch instructions come in two flavors for the MC68010, byte (or short) and word, and an additional flavor, long, for the MC68020. Append the suffix *s* to the word mnemonic to specify the short version of the instruction. For example, `beq` refers to the word version of the Branch if Equal instruction, `beqs` refers to the short version of the instruction, while `beql` refers to the long version of the instruction.

6.2. Extended Branch Instruction Mnemonics

In addition to the instructions which explicitly specify the instruction length, `as` supports extended branch instructions, whose names are, in most cases, constructed from the word versions by replacing the `b` with `j`.

If the operand of the extended branch instruction is a simple address in the text segment, and the offset to that address is sufficiently small, `as` automatically generates the corresponding short branch instruction.

If the offset is too large for a short branch, but small enough for a branch, the corresponding branch instruction is generated. If the operand references an external address or is complex (see next paragraph), the extended branch instruction is implemented either by a `jmp` or `jsr` (for `jra` or `jbsr`), or (for the MC68010) by a conditional branch (with the sense of the condition inverted) around a `jmp` for the extended conditional branches and (for the MC68020) the corresponding long branch.

The extended mnemonics should only be used in the text segment — if they are used in the data segment, the most general form of the branch is generated.

In this context, a complex address is either an address which specifies other than normal mode addressing, or a relocatable expression containing more than one relocatable symbol. For instance, if *a, b* and *c* are symbols in the current segment, the expression *a+b–c* is relocatable, but not simple.

Consult Appendix B for a complete list of the instruction opcodes.

6.3. Addressing Modes

The following table describes the addressing modes that `as` recognizes. Note that certain modes are not valid for the MC68010. The notations used in this table have these meanings:

a*n*	refers to an address register,
d*n*	refers to a data register,
r*i*	refers to either a data register or an address register,
f*i*	refers to a floating-point register,
d	refers to a displacement, which is a constant expression in `as`,
xxx	refers to a constant expression.

Certain instructions, particularly *move* accept a variety of special registers including:

`sp`	the stack pointer which is equivalent to a7,
`sr`	the status register,
`cc`	the condition codes of the status register,
`usp`	the user mode stack pointer,
`pc`	the program counter,
`sfc`	the source function code register,
`dfc`	the destination function code register,
`fpcr`	the floating-point control register,
`fpsr`	the floating-point status register,
`fpiar`	the floating-point iaddr register.

Note that the 3.0 release of *as* accepts only the obsolete special register operand names `fpc` for `fpcr`, `fps` for `fpsr`, and `fpi` for `fpiar`.

Table 6-1 *Addressing Modes*

Mode	*Notation*	*Example*
Register	`an,dn,sp,pc,cc,sr,usp`	`movw a3,d2`
Register Deferred	`an@`	`movw a3@,d2`
Register List	`ri-rj or ri/rj`	`movem a0-a4, a6@-`
Floating-Point Register (MC68881 only)	`fpi`	`fmoves fp1,a3@(24)`
Postincrement	`an@+`	`movw a3@+,d2`
Predecrement	`an@-`	`movw a3@-,d2`
Displacement	`an@(d)`	`movw a3@(24),d2`
Word Index	`an@(d, ri:w)`	`movw a3@(16, d2:w),d3`
Long Index	`an@(d, ri:l)`	`movw a3@(16, d2:l),d3`
Absolute Short	`xxx:w`	`movw 14:w,d2`
Absolute Long	`xxx:l`	`movw 14:l,d2`
PC Displacement	`pc@(d)`	`movw pc@(20),d3`
PC Word Index	`pc@(d, ri:w)`	`movw pc@(14, d2:w),d3`
PC Long Index	`pc@(d, ri:l)`	`movw pc@(14, d2:l),d3`
Normal	*identifier*	`movw widget,d3`
Immediate	`#xxx`	`movw #27+3,d3`

Normal mode assembles as PC-relative if the assembler can determine that this is appropriate, otherwise it assembles as either absolute short or absolute long, under control of the **–d2** command line option.

The Motorola manuals present different mnemonics (and in fact different forms of the actual machine instructions) for instructions that use the literal effective address as data instead of using the contents of the effective address. For instance, they use the mnemonic `adda` for *add address*. `as` does not make these distinctions because it can determine the type of the operand from the form of the operand. Thus an instruction of the form:

```
avenue: .word 0
    ...
    addl      #avenue,a0
```

assembles to the *add address* instruction because `as` can determine that *a0* is an address register.

```
right_now:  =      40000
    ...
    addl      #right_now,d0
```

assembles to an *add immediate* instruction because `as` can determine that *right_now* is a constant.

Because of this determination of operand forms, some of the mnemonics listed in the Motorola manuals are missing from the set of mnemonics that `as` recognizes.

Certain classes of instructions accept only subsets of the addressing modes above. For example, the *add* instruction does not accept a PC-relative address as a destination, and register lists may be used only with the `movem` and `fmovem` instructions.

`as` tries to check all these restrictions and generates the *illegal operand* error code for instructions that do not satisfy the address mode restrictions.

The next section describes how the address modes are grouped into addressing categories.

6.4. Addressing Categories

The processors group the effective address modes into categories derived from the manner in which they are used to address operands. Note the distinction between address *modes* and address *categories*. There are 14 addressing *modes* (18 in the MC68020), and they fall into one or more of four addressing *categories*. The addressing categories are defined here, followed by a table which summarizes the grouping of the addressing modes into categories. Note that register lists can be used only by the `movem` and `fmovem` instructions.

Data means that the effective address mode is used to refer to data operands such as a `d` register or immediate data.

Memory means that the effective address mode can refer to memory operands. Examples include all the `a`-register indirect address modes and all the absolute address modes.

Alterable means that the effective address mode refers to operands which are writeable (alterable). This category takes in every addressing mode except the PC-relative addressing modes and the immediate address mode.

Control means that the effective address mode refers to memory operands without any explicit size specification.

Some addressing categories can be combined to make more restrictive ones. For example, the Motorola MC68010 manual mentions the *Data Alterable Addressing Mode* to mean that the particular instruction can only use those modes which provided data addressing and are alterable as well.

Table 6-2 *Addressing Categories*

Addressing Mode	*Assembler Syntax*	*Data*	*Memory*	*Control*	*Alterable*
Register Direct	`an, dn, sp, pc, cc, sr, usp`	X			X
A Register Indirect	`an@`	X	X	X	X
A Register Indirect with Post Increment	`an@+`	X	X		X
A Register Indirect with Pre Decrement	`an@-`	X	X		X
A Register Indirect with Displacement	`an@(d)`	X	X	X	X
A Register Indirect with Word Index	`an@(d,ri:w)`	X	X	X	X
A Register Indirect with Long Index	`an@(d,ri:l)`	X	X	X	X
Memory Indirect Post-Indexed *	`an@(d,ri:l)`	X	X	X	X
Memory Indirect Pre-Indexed *	`an@(d,ri:l)`	X	X	X	X
Absolute Short	`xxx:w`	X	X	X	X
Absolute Long	`xxx:l`	X	X	X	X
PC-relative	`pc@(d)`	X	X	X	
PC-relative with Word Index	`pc@(d,ri:w)`	X	X	X	
PC-relative with Long Index	`pc@(d,ri:w)`	X	X	X	
PC-Memory Indirect Post-Indexed *	`an@(d,ri:l)`	X	X	X	
PC-Memory Indirect Pre-Indexed *	`an@(d,ri:l)`	X	X	X	
Immediate Data	`#nnn`	X	X		

* These addressing categories are not available with the MC68010.

A

Error Codes

A

Error Codes

A.1. Usage Errors

Cannot open output file
The specified output file cannot be created. Check that the permissions allow opening this file.

Cannot open source file
The assembler cannot open a specified source file. Check the spelling, that the pathname supplied is correct, and that you have read permission for that file.

No input file
One or more input files must be specified — `as` cannot acept the output of a pipe as its input.

Too many file names given
The assembler can not cope with more than one source file. Break the job into smaller stages.

Unknown option 'x' ignored
`as` does not recognize the option *x*. Valid options are listed in Section 1.1.

A.2. Assembler Error Messages

If `as` detects any errors during the assembly process, it prints out a message of the form:

```
as: error (<line_no>): <error_code>
```

Error messages are sent to standard error. Here is a list of `as` error codes, and their possible causes.

Illegal .align
The expression following a `.align` evaluated to some value other than 1, 2 or 4.

Invalid assignment

An attempt was made to redefine a label with an = statement.

Invalid Character

An unexpected character was encountered in the program text.

Invalid Constant

An invalid digit was encountered in a number. For example, using an 8 or 9 in an octal number. Also happens when an out-of-range constant operand is found in an instruction — for example:

```
addq  #200,d0
asll  #12,d0
```

Invalid opcode

The assembler did not recognize an instruction mnemonic. Probably a misspelling.

Invalid operand

The operand used is not consistent with the instruction used — for example:

```
addqb  #1,a5
```

is an invalid combination of instruction and operand. Check the instruction set descriptions for valid combinations of instructions and operands.

Invalid Operator

Check the operand field for a bad operator. The operators that `as` recognizes are plus (+), minus (-), negate or one's complement (˜), multiply (*), and divide (/).

Invalid register expression

A register name was found where one should not appear — for example:

```
addl  #d0,_there
```

Invalid Register List

The register list in a `movem` or `fmovem` instruction was malformed. Note that the list must contain more than one register name: to express a list containing just a single register, you must write its name twice separated by a slash, e.g. "`fp0/fp0`."

Invalid string

An invalid string was encountered in an `.ascii` or `.asciz` directive.

- Make sure the string is enclosed in double quotes.
- Remember that you must use the sequence \" to represent a double quote inside a string.

Invalid symbol

An operand that should be a symbol is not — for example:

```
.globl  3
```

because the constant 3 is not a symbol.

Invalid Term

The expression evaluator could not find a valid term: a symbol, constant or *<expression>. An invalid prefix to a number or a bad symbol name in an operand generates this message.*

Line too long

A statement was found which has more than 4096 characters before the newline character.

Missing close-paren ')'

An unmatched '(' was found in an expression.

Multiply defined symbol

- An identifier appears twice as a label.
- An attempt to redefine a label using an = (direct assignment) statement.
- An attempt to use, as a label, an identifier which was previously defined in an = (direct assignment) statement.

Multiply Defined Symbol (Phase Error)

This is a rarely occurring message which indicates an inconsistency in the assembler. Report it to Sun Microsystems Technical Support if it occurs.

Non-relocatable expression

If an expression contains a relocatable symbol (a label, for instance), the only operations that can be applied to it are the addition of absolute expressions or the subtraction of another relocatable symbol (which produces an absolute result).

Odd address

The previous instruction or pseudo-op required an odd number of bytes and this instruction requires word alignment. This error can only follow an `.ascii`, an `.asciz`, a `.byte`, or a `.skip` pseudo-operation.

- Use a `.even` directive to ensure that the location counter is forced to a 16-bit boundary.

Offset too large

The instruction is a relative addressing instruction and the displacement between this instruction and the label specified is too large for the address field of the instruction.

Out of strings space

No more room is left in the assembler's internal string table. Cut the program into smaller portions; assemble portions of the program separately, then bind them together using the linker.

Stab storage exceeded

No more room is left in the assembler's symbol table for debug information. Cut the program into smaller portions; assemble portions of the program separately, then bind them together using the linker.

Symbol storage exceeded

No more room is left in the assembler's symbol table. Cut the program into smaller portions; assemble portions of the program separately, then bind them together using the linker.

Symbol Too Long

A local label reference longer than one digit was found.

Undefined L-symbol

This is a warning message. A symbol beginning with the letter 'L' was used but not defined. It is treated as an external symbol. Compiler-generated labels usually start with the letter 'L' and should be defined in this assembly. The absence of such a definition usually indicates a compiler code generation error. This message is also generated by the use of symbols such as $99 or *n*$ if *n*$ has not been defined.

Undefined Symbol

A local label reference to an undefined local label was found.

Wrong number of operands

Check Appendix B for the correct number of operands for the current instruction.

B

List of as Opcodes

B

List of as Opcodes

This appendix is a list of the instruction mnemonics accepted by as, grouped alphabetically. The list is divided into two tables, the first covers the MC600x0 processor's instuctions, the second covering the MC68881 floating-point processor's instructions.

Each entry describes the following things:

- The mnemonics for the instruction,
- The generic name for the instruction,
- The assembly language syntax and the variations on the instruction,
- Whether the instruction is specific to the MC68020, or has extended capabilities on the MC68020 compared to the MC68010.

The syntax for as machine instructions differs somewhat from the instruction layouts and categories shown in the Motorola processor manuals. For example, as provides a single set of mnemonics for add (add binary), adda (add address), and addi (add immediate), differentiated only by the length of the operands. In general, as selects the appropriate instruction from the form of the operands.

Here is a brief explanation of the notations used below.

- An instruction of the form add*x* in the assembly language syntax column means that the instruction is coded as addb or addw or addl, *etc*.
- An operand field of a*n* means a*n*y A-register.
- An operand field of d*n* means a*n*y D-register.
- An operand field of r*n* means a*n*y A- or D-register.
- An operand field of f*n* means a*n*y floating-point register.
- An operand field of *ea* means a*n* effective address designated by one of the permissible addressing modes. Consult the relevant Motorola processor manual for details of the allowed addressing modes for each instruction.
- An operand field of *#data* means a*n* immediate operand.
- Other special registers such as cc (condition code register) and sr (status register) are specifically indicated where appropriate.

In the table that follows, the processor is assumed to be the MC68010 unless specifically stated otherwise.

Table B-1 *List of MC680x0 Instruction Codes*

Mnemonic	***Operation Name***	***Syntax***	***Processor***
abcd	add decimal with extend	abcd dy, dx abcd ay@-, ax@-	

addb
addw
addl
	add binary	addx *ea*, *dn*
		addx *dn*, *ea*
		addx *ea*, *an*
		addx *#data*, *ea*	

addqb addqw addql	add quick	addqx *#data*, *ea*	
addxb addxw addxl	add extended	addxx dy, dx addxx ay@-, ax@-	
andb andw andl	logical and	andx *ea*, *dn* andx *dn*, *ea* andx *#data*, *ea*	
aslb aslw asll	arithmetic shift left	aslx dx, dy aslx *#data*, dy aslx *ea*	
asrb asrw asrl	arithmetic shift right	asrx dx, dy asrx *#data*, dy asrx *ea*	
bcc bccl bccs	branch carry clear	bccx *ea*	 68020
bchg	test a bit and change	bchg *dn*, *ea* bchg *#data*, *ea*	
bclr	test a bit and clear	bclr *dn*, *ea* bclr *#data*, *ea*	
bkpt	breakpoint	bkpt *#datafP*	68020
bset	test a bit and set	bset *dn*, *ea* bset *#data*, *ea*	
btst	test a bit	btst *dn*, *ea* btst *#data*, *ea*	
bfchg	test a bit field and change	bfchg *dn*, *ea{n,n}* bfchg *#data*, *ea{n,n}*	68020

Table B-1 *List of MC680x0 Instruction Codes— Continued*

Mnemonic	*Operation Name*	*Syntax*	*Processor*
bfclr	test a bit field and clear	bfclr *dn, ea{n,n}* bfclr *#data, ea{n,n}*	68020
bfexts	extract a bit field signed	bfexts *dn, ea{n,n}* bfexts *#data, ea{n,n}*	68020
bfextu	extract a bit field unsigned	bfextu *dn, ea{n,n}* bfextu *#data, ea{n,n}*	68020
bfffo	find first one in bit field	bfffo *dn, ea{n,n}* bfffo *#data, ea{n,n}*	68020
bfins	insert a bit field	bfins *dn, ea{n,n}* bfins *#data, ea{n,n}*	68020
bfset	test a bit field and set	bfset *dn, ea{n,n}* bfset *#data, ea{n,n}*	68020
bftst	test a bit field	bftst *dn, ea{n,n}* bftst *#data, ea{n,n}*	68020
bcs bcsl bcss	branch carry set	bcsx *ea*	68020
beq beql beqs	branch on equal	beqx *ea*	68020
bge bgel bges	branch greater or equal	bgex *ea*	68020
bgt bgtl bgts	branch greater than	bgtx *ea*	68020
bhi bhil bhis	branch higher	bhix *ea*	68020
ble blel bles	branch less than or equal	blex *ea*	68020
bls blsl	branch lower or same	blsx *ea*	68020
blt bltl blts	branch less than	bltx *ea*	
bmi	branch minus	bmix *ea*	

Table B-1 *List of MC680x0 Instruction Codes— Continued*

Mnemonic	*Operation Name*	*Syntax*	*Processor*
bmil bmis			
bne bnel bnes	branch not equal	bnex *ea*	68020
bpl bpll bpls	branch positive	bplx *ea*	68020
bra bral bras	branch always	brax *ea*	68020
bsr bsrl bsrs	subroutine branch	bsrx *ea*	68020
bvc bvcl bvcs	branch overflow clear	bvcx *ea*	68020
bvs bvsl bvss	branch overflow set	bvsx *ea* bvsl	68020
callm	call module	callm *data*, *ea*	68020
cas2b cas2l cas2w	compare & swap with operand	cas2x dc1:dc2,du1:du2,(rn1):(rn2)	68020 68020 68020
casb casl casw	compare & swap with operand	casx dc,du, *ea*	68020 68020 68020
chk chk2b chk2l chk2w	check register against bounds	chk *ea*,dn chk2x *ea*,rn	68020 68020 68020 68020
clrb clrw clrl	clear an operand	clrx *ea*	
cmp2b cmp2l cmp2w	compare register against bounds	cmp2x *ea* rn	68020 68020 68020
cmpmb cmpmw	compare memory	cmpmx ay@+,ax@+	

Table B-1 *List of MC680x0 Instruction Codes— Continued*

Mnemonic	*Operation Name*	*Syntax*	*Processor*
cmpml			
cmpb	arithmetic compare	cmpx *ea*, d*n*	
cmpw		cmpx *#data*, *ea*	
cmpl			
dbcc	decrement & branch on carry clear	dbcc d*n*, *label*	
dbcs	" on carry set	dbcs d*n*, *label*	
dbeq	" on equal	dbeq d*n*, *label*	
dbf	" on false	dbf d*n*, *label*	
dbge	" on greater than or equal	dbge d*n*, *label*	
dbgt	" on greater than	dbgt d*n*, *label*	
dbhi	" on high	dbhi d*n*, *label*	
dble	" on less than or equal	dble d*n*, *label*	
dbls	" on low or same	dbls d*n*, *label*	
dblt	" on less than	dblt d*n*, *label*	
dbmi	" on minus	dbmi d*n*, *label*	
dbne	" on not equal	dbne d*n*, *label*	
dbpl	" on plus	dbpl d*n*, *label*	
dbra	" always (same as dbf)	dbra d*n*, *label*	
dbt	" on True	dbt d*n*, *label*	
dbvc	" on overflow clear	dbvc d*n*, *label*	
dbvs	" on overflow set	dbvs d*n*, *label*	
divs	signed divide	divs *ea*, d*n*	
divsl	signed divide	divsx *ea*, d*n*	68020
divsll		divsx *ea*, d*q*	68020
		divsx *ea*, d*r*:d*q*	68020
divu	unsigned divide	divu *ea*, d*n*	
divul	unsigned divide	divux *ea*, d*n*	68020
divull		divux *ea*, d*q*	68020
		divux *ea*, d*r*:d*q*	68020
eorb	logical exclusive or	eorx d*n*, *ea*	
eorw		eorx *#data*, *ea*	
eorl		eorb *#data*, *cc*	
		eorw *#data*, *sr*	
exg	exchange registers	exg rx, ry	
extbl	sign extend	extbl d*n*	68020
extw	sign extend	ext d*n*	
extl			
jmp	jump	jmp *ea*	
jsr	jump to subroutine	jsr *ea*	
jcc	jump carry clear	jcc *ea*	
jcs	jump on carry	jcs *ea*	
jeq	jump on equal	jeq *ea*	

Table B-1 *List of MC680x0 Instruction Codes— Continued*

Mnemonic	*Operation Name*	*Syntax*	*Processor*
jge	jump greater or equal	jge *ea*	
jgt	jump greater than	jgt *ea*	
jhi	jump higher	jhi *ea*	
jle	jump less than or equal	jle *ea*	
jls	jump lower or same	jls *ea*	
jlt	jump less than	jlt *ea*	
jmi	jump minus	jmi *ea*	
jne	jump not equal	jne *ea*	
jpl	jump positive	jpl *ea*	
jra	jump always	jra *ea*	
jbsr	jump to subroutine	jbsr *ea*	
jvc	jump no overflow	jvc *ea*	
jvs	jump on overflow	jvs *ea*	
lea	load effective address	lea *ea*, a*n*	
link	link and allocate	link a*n*, #*disp*	
		linkl a*n*, #*disp*	68020
lslb	logical shift left	lsl*x* d*x*, dy	
lslw		lsl*x* #*data*, dy	
lsll		lsl*x* *ea*	
lsrb	logical shift right	lsr*x* d*x*, dy	
lsrw		lsr*x* #*data*, dy	
lsrl		lsr*x* *ea*	
movb	move data	mov*x* *ea*, *ea*	
movl			
movw		mov*x* #*data*, d*n*	
movw	move from condition code register	movw *cc*, *ea*	
movw	move from status register	movw sr, *ea*	
movc	move to control register	movc r*n*, *cr*	
	move from control register	movc *cr*, r*n*	
moveml	move multiple registers	movem*x* #*mask*, *ea*	
movemw		movem*x* *ea*, #*mask*	
		movem*x* *ea*, *reflist*	
		movem*x* *reflist*, *ea*	
movepl	move peripheral	movep*x* d*n*, a*n*@ (*d*)	
movepw		movep*x* d*n*, a*n*@ (*d*)	
moveq	move quick	moveq #*data*, d*n*	
movsb	move to address space	movs*x* r*n*, *ea*	
movsw	move from address space	movs*x* *ea*, r*n*	
movsl			
muls	signed multiply	muls *ea*, d*n*	

Table B-1 *List of MC680x0 Instruction Codes— Continued*

Mnemonic	*Operation Name*	*Syntax*	*Processor*
mulsl		mulsx *ea*,d*l*	68020
		mulsx *ea*,d*h*:d*l*	68020
mulu	unsigned multiply	mulu *ea*,d*n*	
mulul		mulux *ea*,d*l*	68020
		mulux *ea*,d*h*:d*l*	68020
nbcd	negate decimal with extend	nbcd *ea*	
negb	negate binary	negx *ea*	
negw			
negl			
negxb	negate binary with extend	negxx *ea*	
negxw			
negxl			
nop	no operation	nop	
notb	logical complement	notx *ea*	
notw			
notl			
orb	inclusive or	orx *ea*,d*n*	
orw		orx d*n*,*ea*	
orl		or #*data*,*ea*	
		orb #*data*,*cc*	
		orw #*data*,*sr*	
pack	pack	pack -(ax),-(ay),#*adjustment*	68020
		pack dx,dy,#*adjustment*	68020
pea	push effective address	pea *ea*	
reset	reset machine	reset	
rolb	rotate left	rolx dx,dy	
rolw		rolx #*data*,dy	
roll		rolx *ea*	
rorb	rotate right	rorx dx,dy	
rorw		rorx #*data*,dy	
rorl		rorx *ea*	
roxlb	rotate left with extend	roxlx dx,dy	
roxlw		roxlx #*data*,dy	
roxll		roxlx *ea*	
roxrb	rotate right with extend	roxrx dx,dy	
roxrw		roxrx #*data*,dy	
roxrl		roxrx *ea*	
rte	return from exception	rte	
rtm	return from module	rtm *rn*	68020

Table B-1 *List of MC680x0 Instruction Codes— Continued*

Mnemonic	Operation Name	Syntax	Processor
rtr	return and restore codes	rtr	
rts	return from subroutine	rts	
		rts *#n*	
sbcd	subtract decimal with extend	sbcd dy, dx	
		sbcd ay@-, ax@-	
stop	halt machine	stop *#xxx*	
subb	arithmetic subtract	sub*x* *ea*, d*n*	
subw		sub*x* d*n*, *ea*	
		sub*x* *ea*, a*n*	
subl		sub*x* *#data*, *ea*	
st	set all ones	st *ea*	
sf	set all zeros	sf *ea*	
shi	set high	shi *ea*	
sls	set lower or same	sls *ea*	
scc	set carry clear	scc *ea*	
scs	set carry set	scs *ea*	
sne	set not equal	sne *ea*	
seq	set equal	seq *ea*	
svc	set no overflow	svc *ea*	
	set on overflow	svs *ea*	
spl	set plus	spl *ea*	
smi	set minus	smi *ea*	
sge	set greater or equal	sge *ea*	
slt	set less than	slt *ea*	
sgt	set greater than	sgt *ea*	
sle	set less than or equal	sle *ea*	
subqb	subtract quick	subq*x* *#data*, *ea*	
subqw			
subql			
subxb	subtract extended	subx*x* dy, dx	
subxw		subx*x* ay@-, ax@-	
subxl			
swap	swap register halves	swap d*n*	
tas	test operand then set	tas *ea*	
trap	trap	trap *#vector*	
trapcc	trap on carry clear	trapcc*x*	68020
trapccl		trapcc*x* *#data*	68020
trapccw			68020
trapcs	trap on carry set	trapcs*x*	68020
trapcsl		trapcs*x* *#data*	68020

Table B-1 *List of MC680x0 Instruction Codes— Continued*

Mnemonic	*Operation Name*	*Syntax*	*Processor*
trapcsw			68020
trapeq	trap on equal	trapeqx	68020
trapeql		trapeqx #*data*	68020
trapeqw			68020
trapf	trap on never true	trapfx	68020
trapfl		trapfx #*data*	68020
trapfw			68020
trapge	trap on greater or equal	trapgex	68020
trapgel		trapgex #*data*	68020
trapgew			68020
trapgt	trap on greater	trapgtx	68020
trapgtl		trapgtx #*data*	68020
trapgtw			68020
traphi	trap on hi	traphix	68020
traphil		traphixx #*data*	68020
traphiw			68020
traple	trap on less or equal	traplex	68020
traplel		traplexx #*data*	68020
traplew			68020
trapls	trap on low or same	traplsx	68020
traplsl		traplsx #*data*	68020
traplsw			68020
traplt	trap on less than	trapltx	68020
trapltl		trapltx #*data*	68020
trapltw			68020
trapmi	trap on minus	trapmix	68020
trapmil		trapmix #*data*	68020
trapmiw			68020
trapne	trap on not equal	trapnex	68020
trapnel		trapnex #*data*	68020
trapnew			68020
trappl	trap on plus	trappl	68020
trappll		trapplx #*datafP*	68020
trapplw			68020
trapt	trap on always true	trapt	68020
traptl		traptx #*datafP*	68020
traptw			68020
trapv	trap on overflow	trapv	
trapvc	trap on overflow clear	trapvc	68020

Table B-1 *List of MC680x0 Instruction Codes— Continued*

Mnemonic	*Operation Name*	*Syntax*	*Processor*
trapvcl trapvcw		trapvcx #*datafP*	68020 68020
trapvs trapvsl trapvsw	trap on overflow set	trapvs trapvsx #*datafP*	68020 68020 68020
tstb tstw tstl	test operand	tstx *ea*	
unlk	unlink	unlk *an*	
unpk	unpack bcd	unpk -(ax),-ay),#*adjustment* unpk dx,dy,#*adjustment*	68020 68020

The following table describes the MC68881 instuction mnemonics supported by as. Each mnemonic indicates the data type that it operates on by the last character of the mnemonic:

- b indicates a byte format instruction
- w indicates a word format instruction
- l indicates a long format instruction
- s indicates a single-word format instruction
- d indicates a double-word format instruction
- x indicates a*n* extended format instruction
- p indicates a packed format instruction.

Table B-2 *MC68881 Instructions supported by* as

Mnemonic	*Operation Name*	*Syntax*
fabsx fabsl fabss fabsp fabsw fabsd fabsb	absolute value	fabsx *ea*, *fn* fabsx *fm*, *fn* fabsx *fn*
facosx facosl facoss facosp facosw	arc cosine	facosx *ea*, *fn* facosx *fm*, *fn* facosx, *fn*

Table B-2 *MC68881 Instructions supported by* as— *Continued*

Mnemonic	*Operation Name*	*Syntax*
facosd		
facosb		
faddx	add	faddx *ea, fn*
faddl		faddx *fm, fn*
fadds		faddx *fn*
faddp		
faddw		
faddd		
faddb		
fasinx	arc sin	fasinx *ea, fn*
fasinl		fasinx *fm, fn*
fasins		fasinx *fn*
fasinp		
fasinw		
fasind		
fasinb		
fatanx	arc tangent	fatanx *ea, fn*
fatanl		fatanx *fm, fn*
fatans		fatanx *fn*
fatanp		
fatanw		
fatand		
fatanb		
fatanhx	hyperbolic arc tangent	fatanhx *ea, fn*
fatanhl		fatanhx *fm, fn*
fatanhs		fatanhx *fn*
fatanhp		
fatanhw		
fatanhd		
fatanhb		
fb*cc*	branch conditionally	fb*cc* *label*
fbeq	(equal)	
fbeql		
fbf	(false)	
fbfl		
fbgt	(greater than)	
fbgtl		
fble	(less than or equal)	
fblel		
fblt	(less than)	
fbltl		
fbge	(greater than or equal)	
fbgel		

Table B-2 *MC68881 Instructions supported by* as*— Continued*

Mnemonic	*Operation Name*	*Syntax*
fbgl	(greater than or less)	
fbgll		
fbgle	(greater less or equal)	
fbglel		
fbgt	(greater than)	
fbne	(not equal)	
fbnel		
fbneq	(not (equal))	
fbneql		
fbnge	(not greater than or equal)	
fbngel		
fbngl	(not greater than or less)	
fbngll		
fbngle	(not greater than, less or equal)	
fbnglel		
fbngt	(not greater than)	
fbngtl		
fbnle	(not less than or equal)	
fbnlel		
fbnlt	(not less than)	
fbnltl		
fbt	(true)	
fbtl		
fbor	(ordered)	
fborl		
fboge	(ordered greater or equal)	
fbogel		
fbogl	(ordered greater or less)	
fbogll		
fbogt	(ordered greater than)	
fbogtl		
fbole	(ordered less or equal)	
fbolel		
fbolt	(ordered less than)	
fboltl		
fbseq	(signalling equal)	
fbseql		
fbsf	(signalling false)	
fbsfl		
fbsne	(signalling not equal)	
fbsnel		
fbst	(signalling true)	
fbstl		
fbueq	(unordered equal)	
fbueql		

Table B-2 *MC68881 Instructions supported by* as— *Continued*

Mnemonic	*Operation Name*	*Syntax*
fbuge	(unordered greater or equal)	
fbugel		
fbugt	(unordered greater than)	
fbugtl		
fbule	(unordered less or equal)	
fbulel		
fbult	(unordered less than)	
fbultl		
fbun	(unordered)	
fbunl		
fcmpx	compare	fcmpx *ea*, *fn*
fcmpx		fcmpx *fm*, *fn*
fcmpl		
fcmps		
fcmpp		
fcmpw		
fcmpd		
fcmpb		
fcosx	cosine	fcosx *ea*, *fn*
fcosx		fcosx *fm*, *fn*
fcosl		fcosx *fn*
fcoss		
fcosp		
fcosw		
fcosd		
fcosb		
fcoshx	hyperbolic cosine	fcoshx *ea*, *fn*
fcoshx		fcoshx *fm*, *fn*
fcoshl		fcoshx *fn*
fcoshs		
fcoshp		
fcoshw		
fcoshd		
fcoshb		
fdb*cc*	decrement & branch on condition	fdb*cc* d*n*, *label*
fdbeq	(equal)	
fdbne	(not equal)	
fdbgt	(greater than)	
fdbngt	(not greater than)	
fdbge	(greater or equal)	
fdbnge	(not greater or equal)	
fdblt	(less than)	
fdbnlt	(not less than)	
fdble	(less or equal)	

Table B-2 *MC68881 Instructions supported by* as*— Continued*

Mnemonic	*Operation Name*	*Syntax*
fdbnle	(not less or equal)	
fdbgl	(greater or less)	
fdbngl	(not greater or less)	
fdbgle	(greater, less or equal)	
fdbngle	(not greater, less or equal)	
fdbogt	(ordered greater than)	
fdbule	(unordered less or equal)	
fdboge	(unordered greater or equal)	
fdbult	(unordered less than)	
fdbolt	(ordered less than)	
fdbuge	(unordered greater or equal)	
fdbole	(ordered less or equal)	
fdbugt	(unordered greater than)	
fdbogl	(ordered greater or less)	
fdbueq	(unordered equal)	
fdbor	(ordered)	
fdbun	(unordered)	
fdbf	(false)	
fdbt	(true)	
fdbsf	(signalling false)	
fdbst	(signalling true)	
fdbseq	(signalling equal)	
fdbsne	(signalling not equal)	
fdivx	divide	fdivx *ea*, *fn*
fdivx		fdivx *fm*, *fn*
fdivl		
fdivs		
fdivp		
fdivw		
fdivd		
fdivb		
fetoxx	e^x	fetoxx *ea*, *fn*
fetoxx		fetoxx *fm*, *fn*
fetoxl		fetoxx *fn*
fetoxs		
fetoxp		
fetoxw		
fetoxd		
fetoxb		
fetoxm1x	e^x -1	fetoxm1x *ea*, *fn*
fetoxm1x		fetoxm1x *fm*, *fn*
fetoxm1l		fetoxm1x *fn*
fetoxm1s		
fetoxm1p		

Table B-2 *MC68881 Instructions supported by* as*— Continued*

Mnemonic	***Operation Name***	***Syntax***
fetoxm1w fetoxm1d fetoxm1b		
fgetexpx fgetexpx fgetexpl fgetexps fgetexpp fgetexpw fgetexpd fgetexpb	get exponent	fgetexpx *ea*, *fn* fgetexpx *fm*, *fn* fgetexpx *fn*
fgetmanx fgetmanx fgetmanl fgetmans fgetmanp fgetmanw fgetmand fgetmanb	get mantissa	fgetmanx *ea*, *fn* fgetmanx *fm*, *fn* fgetmanx *fn*
fintx fintl fints fintp fintw fintd fintb	integer part	fintx *ea*, *fn* fintx *fm*, *fn* fintx *fn*
fintrx fintrzx fintrzl fintrzs fintrzp fintrzw fintrzd fintrzb	integer part, round to 0	fintrx *ea*, *fn* fintrx *fm*, *fn* fintrx *fn*
flog10x flog10x flog10l flog10s flog10p flog10w flog10d flog10b	$\log_{10}$	flog10x *ea*, *fn* flog10x *fm*, *fn* flog10x *fn*
flog2x	$\log_2$	flog2x *ea*, *fn*

Table B-2 *MC68881 Instructions supported by* as— *Continued*

Mnemonic	*Operation Name*	*Syntax*
flog2x		flog2x *fm*, *fn*
flog2l		flog2x *fn*
flog2s		
flog2p		
flog2w		
flog2d		
flog2b		
flognx	$\log_e$	flognx *ea*, *fn*
flognx		flognx *fm*, *fn*
flognl		flognx *fn*
flogns		
flognp		
flognw		
flognd		
flognb		
flognp1x	$\log_e(x+1)$	flognp1x *ea*, *fn*
flognp1x		flognp1x *fm*, *fn*
flognp1l		flognp1x *fn*
flognp1s		
flognp1p		
flognp1w		
flognp1d		
flognp1b		
fmodx	modulo remainder	fmodx *ea*, *fn*
fmodx		fmodx *fm*, *fn*
fmodl		
fmods		
fmodp		
fmodw		
fmodd		
fmodb		
fmovex	move fp register	fmovex *ea*, *fn*
fmovex		fmovex *fm*, *ea*
fmovel		fmovex *fm*, *ea*{dn}
fmoves		fmovex *fm*, *ea*{#k}
fmovep		
fmovew		
fmoved		
fmoveb		
fmovecrx	move constant ROM	fmovecrx #*ccc*, *fn*
fmovecrx		
fmovecr		
fmovemx	move multiple data registers	fmovemx *ea*, list

Table B-2 *MC68881 Instructions supported by* as*— Continued*

Mnemonic	*Operation Name*	*Syntax*
fmovemx fmoveml fmovem		fmovemx list, *ea* fmovemx *ea*, d*n* fmovemx d*n*, *ea*
fmulx fmull fmuls fmulp fmulw fmuld fmulb	multiply	fmulx *ea*, f*n* fmulx f*m*, f*n*
fnegx fnegl fnegs fnegp fnegw fnegd fnegb	negate	fnegx *ea*, f*n* fnegx f*m*, f*n* fnegx f*n*
fnop	no operation	fnop
fremx fremx freml frems fremp fremw fremd fremb	IEEE remainder	fremx *ea*, f*n* fremx f*m*, f*n*
frestore	restore internal state	frestore *ea*
fsave	save internal state	fsave *ea*
fscalex fscalex fscalel fscales fscalep fscalew fscaled fscaleb	scale exponent	fscalex *ea*, f*n* fscalex f*m*, f*n*
fsgldivx fsgldivx fsgldivs fsgldivl fsgldivp fsgldivw	single-precision divide	fsgldivx *ea*, f*n* fsgldivx f*m*, f*n*

Table B-2 *MC68881 Instructions supported by* as*— Continued*

Mnemonic	*Operation Name*	*Syntax*
fsgldivb		
fsglmulx	single-precision multiply	fsglmulx *ea, fn*
fsglmulx		
fsglmuls		
fsglmull		
fsglmulp		
fsglmulw		
fsglmulb		
fsinx	sin	fsinx *ea, fn*
fsinx		fsinx *fm, fn*
fsinl		fsinx *fn*
fsins		
fsinp		
fsinw		
fsind		
fsinb		
fsincosx	simultaneous sine and cosine	fsincosx *ea, fc:fs*
fsincosx		fsincosx *fm, fc:fs*
fsincosl		
fsincoss		
fsincosp		
fsincosw		
fsincosd		
fsincosb		
fsinhx	hyperbolic sine	fsinhx *ea, fn*
fsinhl		fsinhx *fm, fn*
fsinhs		fsinhx *fn*
fsinhp		
fsinhw		
fsinhd		
fsinhb		
fsqrtx	square root	fsqrtx *ea, fn*
fsqrtx		fsqrtx *fm, fn*
fsqrtl		fsqrtx *fn*
fsqrts		
fsqrtp		
fsqrtw		
fsqrtd		
fsqrtb		
fsubx	subtract	fsubx *ea, fn*
fsubx		fsubx *fm, fn*
fsubl		
fsubs		

Table B-2 *MC68881 Instructions supported by* as*— Continued*

Mnemonic	*Operation Name*	*Syntax*
fsubp		
fsubw		
fsubd		
fsubb		
ftanx	tangent	ftanx *ea*, *fn*
ftanx		ftanx *fm*, *fn*
ftanl		ftanx *fn*
ftans		
ftanp		
ftanw		
ftand		
ftanb		
ftanhx	hyperbolic tangent	ftanhx *ea*, *fn*
ftanhx		ftanhx *fm*, *fn*
ftanhl		ftanhx *fn*
ftanhs		
ftanhp		
ftanhw		
ftanhd		
ftanhb		
ftentoxx	10^x	ftentoxx *ea*, *fn*
ftentoxx		ftentoxx *fm*, *fn*
ftentoxl		ftentoxx *fn*
ftentoxs		
ftentoxp		
ftentoxw		
ftentoxd		
ftentoxb		
ftrap*cc*	trap conditionally	ftrap*cc*
ftrapeq	(equal)	ftrap*cc* #*datafP*
ftrapeqw		
ftrapeql		
ftrapne	(not equal)	
ftrapnew		
ftrapnel		
ftrapgt	(greater than)	
ftrapgtw		
ftrapgtl		
ftrapngt	(not greater than)	
ftrapngtw		
ftrapngtl		
ftrapge	(greater or equal)	
ftrapgew		
ftrapgel		

Table B-2 *MC68881 Instructions supported by* as*— Continued*

Mnemonic	Operation Name	Syntax
ftrapnge	(not greater or equal)	
ftrapngew		
ftrapngel		
ftraplt	(less than)	
ftrapltw		
ftrapltl		
ftrapnlt	(not less than)	
ftrapnltw		
ftrapnltl		
ftraple	(less than or equal)	
ftraplew		
ftraplel		
ftrapnle	(not less than or equal)	
ftrapnlew		
ftrapnlel		
ftrapgl	(greater than or less)	
ftrapglw		
ftrapgll		
ftrapngl	(not greater than or less)	
ftrapnglw		
ftrapngll		
ftrapgle	(greater, less or equal)	
ftrapglew		
ftrapglel		
ftrapngle	(not greater, less or equal)	
ftrapnglew		
ftrapnglel		
ftrapogt	(ordered greater than)	
ftrapogtw		
ftrapogtl		
ftrapule	(unordered less or equal)	
ftrapulew		
ftrapulel		
ftrapoge	(ordered greater or equal)	
ftrapogew		
ftrapogel		
ftrapult	(unordered less than)	
ftrapultw		
ftrapultl		
ftrapolt	(ordered less than)	
ftrapoltw		
ftrapoltl		
ftrapuge	(unordered greater or equal)	
ftrapugew		
ftrapugel		

Table B-2 *MC68881 Instructions supported by* as— *Continued*

Mnemonic	*Operation Name*	*Syntax*
ftrapole	(ordered less or equal)	
ftrapolew		
ftrapolel		
ftrapugt	(unordered greater than)	
ftrapugtw		
ftrapugtl		
ftrapogl	(ordered greater or less)	
ftrapoglw		
ftrapogll		
ftrapueq	(unordered equal)	
ftrapueqw		
ftrapueql		
ftrapor	(ordered)	
fftraporw		
ftraporl		
trapun	(unordered)	
ftrapunw		
ftrapunl		
ftrapf	(false)	
ftrapfw		
ftrapfl		
ftrapt	(true)	
ftraptw		
ftraptl		
ftrapsf	(signalling false)	
ftraptw		
ftrapsfl		
ftrapst	(signalling true)	
ftrapsfw		
ftrapstl		
ftrapseq	(signalling equal)	
ftrapseqw		
ftrapseql		
ftrapsne	(signalling not equal)	
ftrapsnew		
ftrapsnel		
ftst*x*	test operand	ftst*x ea*
ftstx		ftst*x fm*
ftstl		
ftsts		
ftstp		
ftstw		
ftstd		
ftstb		
ftestx		

Table B-2 *MC68881 Instructions supported by* as*— Continued*

Mnemonic	*Operation Name*	*Syntax*
ftestl		
ftests		
ftestp		
ftestw		
ftestd		
ftestb		
ftwotoxx	2^x	ftwotoxx *ea, fn*
ftwotoxx		ftwotoxx *fm, fn*
ftwotoxl		ftwotoxx *fn*
ftwotoxs		
ftwotoxp		
ftwotoxw		
ftwotoxd		
ftwotoxb		
fj*cc*	jump on condition	fj*cc* *label*
fjeq	(equal)	
fjne	(not equal)	
fjneq	(not equal or equal)	
fjgt	(greater than)	
fjngt	(not greater than)	
fjge	(greater or equal)	
fjnge	(not greater or equal)	
fjlt	(less than)	
fjnlt	(not less than)	
fjle	(less or equal)	
fjnle	(not less or equal)	
fjgl	(greater or less)	
fjngl	(not greater or less)	
fjgle	(greater, less or equal)	
fjngle	(not greater, less or equal)	
fjogt	(ordered greater than)	
fjule	(unordered less or equal)	
fjoge	(ordered greater or equal)	
fjult	(unordered less than)	
fjolt	(ordered less than)	
fjuge	(unordered greater or equal)	
fjole	(ordered less or equal)	
fjugt	(unordered greater than)	
fjogl	(ordered greater or less)	
fjueq	(unordered equal)	
fjor	(ordered)	
fjun	(unordered)	
fjf	(false)	
fjt	(true)	
fjsf	(signalling false)	

Table B-2 *MC68881 Instructions supported by* as*— Continued*

Mnemonic	*Operation Name*	*Syntax*
fjst	(signalling true)	
fjseq	(signalling equal)	
fjsne	(signalling not equal)	
fs*cc*	set according to condition	fs*cc ea*
fseq	(equal)	
fsne	(not equal)	
fsneq	(not equal or equal)	
fsgt	(greater than)	
fsngt	(not greater than)	
fsge	(greater or equal)	
fsnge	(not greater or equal)	
fslt	(less than)	
fsnlt	(not less than)	
fsle	(less or equal)	
fsnle	(not less or equal)	
fsgl	(greater or less)	
fsngl	(not greater or less)	
fsgle	(greater, less or equal)	
fsngle	(greater, less or equal)	
fsogt	(not greater, less or equal)	
fsule	(unordered less or equal)	
fsoge	(ordered greater or equal)	
fsult	(unordered less than)	
fsolt	(ordered less than)	
fsuge	(unordered greater or equal)	
fsole	(ordered less or equal)	
fsugt	(unordered greater than)	
fsogl	(ordered greater or less)	
fsueq	(unordered equal)	
fsor	(ordered)	
fsun	(unordered)	
fsf	(false)	
fst	(true)	
fssf	(signalling false)	
fsst	(signalling true)	
fsseq	(signalling equal)	
fssne	(signalling not equal)	

Index

Revision History

Revision	*Date*	*Comments*
A	15 October 1982	First release of this Manual as a part of *Programming Tools for the Sun Workstation.*
B	15 May 1984	Specifics of MC68010 chip incorporated into the manual as addenda. Many minor corrections.
C	1 February 1985	Extracted from *Programming Tools for the Sun Workstation* to make a separate language manual. Folded specific MC68010 descriptions into rest of text. Many minor corrections from α to β versions.
E β	22 November 1985	Added details of the MC68020 processor and the MC68881 floating point coprocessor. Minor corrections from α to β versions.
E	17 February 1986	Minor corrections from β versions. Added missing 68020 and 68881 opcodes.

Notes

Notes

Notes

Notes

Notes