sun®
microsystems

Assembly Language Reference for the Sun-2™ and Sun-3™

Part Number: 800-1773-10
Revision A, of 9 May 1988

Contents

Tables

Preface

This manual is the Programmer's Reference Manual for `as` — the assembler for Sun-2 and Sun-3 workstations running the SunOS operating system. `as` converts source programs written in *assembly language* into a form that the linker utility, *ld*(1) will turn into a runnable program.

What `as` Provides

`as` provides assembly language programmers 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, you can use the C preprocessor (see *cpp*(1)) 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, the MC68881 floating-point coprocessor, and Sun's Floating-Point Accelerator (FPA). The basic format of `as` is loosely based on the Digital Equipment Corporation's Macro-11 assembler described in DEC's publication DEC-11-0MACA-A-D. It also contains elements of the UNIX† PDP-11 `as` assembler. The instruction mnemonics and effective address format are based on 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 assumes that you are 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 also assumes that you are familiar with the relevant processors, their instruction sets and addressing modes, and especially their irregularities.

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 registered trademark of AT&T.

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

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

-k Generate position-independent code as required by

```
        cc -pic/-PIC
```

WARNING ***Don't apply this flag to hand-coded assembler programs unless they are written to be position-independent.***

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

You should also consult the *SunOS Reference Manual* 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 { and } are used for grouping so that

```
{ <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:

- The *character set* that 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.

Here are some examples of legal 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 3.4 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 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 (in the range 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 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 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, or by a backslash followed by 0x or 0X and a one- or two-digit hexadecimal constant. The table below shows the octal representation of some of the more common non-printing characters.

Character	***Octal***	***Hex***
Backspace	\010	0x8
Horizontal Tab	\011	0x9
Newline (Linefeed)	\012	0xA
Formfeed	\014	0xC
Carriage Return	\015	0xD

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	Two's complement of its argument.
~	logical negation	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 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 requires only 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 requires only 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 two 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* >] [< *opcode* > [< *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 opcode 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 with the 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. 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*. Using 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 word 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 of an *<operand field>* is:

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

The general 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 5 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 Section 6.3, and 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 on a line with a statement. The comment field is indicated by the presence of the vertical bar character (|) after 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 a 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>* in a direct assignment 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 in the same program section as the expression.

If the *<expression>* contains an external identifier, the identifier defined by direct assignment 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 below.

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.
`.bytez`	Generates a sequence of bytes in data storage initialized to zero.
`.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.

Table 5-1 *Assembler Directives— Continued*

Pseudo-Operation	*Description*
`.lcomm`	Reserves a specified amount of space in the *bss* control section.
`.skip`	Advances the location counter by a specified amount.
`.align` `.even`	Forces location counter to next one-, two- or four-byte boundary. 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 following sections.

5.1. `.ascii` — Generate 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   .ascii  "hello there"
164 150 145 162 145

127 141 162 156 151 156   .ascii  "Warning-\007\007 \012"
147 055 007 007 040 012

141 142 143 144 145 146   .ascii  "abcdefg"
147
```

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    .asciz  "Hello World!"
127 157 162 144 041 000

124 150 105 040 107 162    .asciz  "The Great PROMpkin strikes again!"
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
```

5.3. Directives to Generate Data

The `.byte`, `.word`, `.long`, `.single`, and `.double` directives reserve storage locations and initialize them with specified values.

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

```
[ <label>: ]    .byte     [ <expression> ] [ , <expression> ]...

[ <label>: ]    .bytez    [ <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 `.bytez` directive reserves one byte (8 bits) for each expression in the operand field, and initializes it to zero.

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. Directives to Switch Location Counter

These statements `.text`, `.data`, `.bss`, `.data1`, and `.data2`, change the 'control section' where assembled code is loaded.

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

Space	***Description***
text	The address space where the executable machine instructions are placed.
data	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 compilers 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	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 *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 *SunOS 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 *<constantexpression>* 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 space 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.

```
.even
```

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 `.stabn` build various types of symbol table entries.

The `.stab` directives have the following forms:

```
.stabs   name, type, 0, desc, value

.stabn   type, 0, desc, value
```

or

```
.stabd    type, 0, desc
```

The .stabs directives are used to describe types, variables, procedures, and so on, while the .stabn directives convey information about scopes and the mapping from source statements to object code.

A .stabd directive is identical in meaning to a corresponding .stabn directive with the value field set to "." (dot), which the assembler uses to mean the current location. Most of the needed information, for example symbol name and type structure, is contained in the *name* field. The *type* field identifies the type of symbolic information, for example source file, global symbol, or source line. The *desc* field specifies the number of bytes occupied by a variable or type or the nesting level for a scope symbol. The *value* field specifies an address or an offset.

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.

```
.proc
```

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 this 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. See Appendix C for information on the Sun FPA's instructions set and addressing modes.

6.1. Instruction Mnemonics

The instruction mnemonics that `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, while `beql` refers to the long version.

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

Table 6-1 below 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:

Notation	*Meaning*
a*n* d*n* r*i* f*i*	An address register. A data register. Either a data register or an address register. A floating-point register.
d	A displacement, which is a constant expression in `as`. In MC68020 mode, a length specifier (`:L`, described below) may be appended to the displacement. Any forward or external references *require* the length specifier to be `:l`. All other references permit either `:l` or `:w` or nulls.
L	The index register's length. This may be either long (`l`) or word (`w`) or null. If the only value permitted by a particular addressing mode or category is `l` or `w`, then `L` will be replaced by the appropriate value in the table notation.
s	A scale factor that may be used to multiply the index register's length. The scale factor may have a value of 1, 2, 4, or 8.

The table notation of two or three items separated by colons, such as r*i*`:L:s`, indicate items that may be optional. In that particular case, *you may not* specify `:s` unless you have specified *:L*, which you may not specify unless you have specified r*i*. The items in the list must appear in the order given in the notation of the tables that follow.

In the table where both *d* and *d'* are specified, *d* corresponds to a MC68020 outer displacement and *d'* corresponds to a MC68020 base displacement.

xxx refers to a constant expression.

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

Name	*Register*
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 instruction address register

The memory-indirect and program counter memory-indirect addressing modes listed in the following tables are usable only with the MC68020.

In each of these addressing modes, up to four user-specified values are used to generate the final operand address:

- base register
- base displacement
- index register
- outer dispacement

All four user-specified values are optional. Both base and outer displacements may be null, word or long. When a displacement is null, or an element is suppressed, its value is taken as zero in the effective address calculation.

In the case of memory-indirect addressing, an address register (a*n*) is used as a base register, and its value can be adjusted by an optional base displacement *(d')*. An index register (r*i*) specifies an index operand (r*i*:L:s) and finally, an outer displacement *(d)* can be added to the address operand, yielding the effective address.

Program counter memory-indirect mode is exactly the same. The only difference is that the program counter is used as the base register.

Some examples of these addressing modes follow:

```
an@(d':L,ri:L:s)@(d:L)
an@(d:L)@(d':L,ri:L:s)
an@@
an@(d:L)@
an@(d':L,ri:L:s)@
pc@@
pc@(d:L)@
pc@(d':L,ri:L:s)@(d:L)
pc@(d:L)@(d':L,ri:L:s)
@(d:L)@
@(d':L,ri:L:s)@(d:L)
@(d:L)@(d':L,ri:L:s)
@(d':L,ri:L:s)@
```

In the table below, note that the notation *ri/rj* means *ri* and *rj*, while *ri_rj* means *ri* through *rj*.

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@-`
FPA register	`fpai`	`fpmoves fpa1,d2`
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`
PC-Memory Indirect Pre-Indexed (68020)	`pc@(d':L,ri:L:s)@(d:L)`	`movl pc@(2:w,d4:w:4)@(14:l),d3`
PC-Memory Indirect Post-Indexed (68020)	`pc@(d:L)@(d':L,ri:L:s)`	`movl pc@(d:l)@(3:w,d2:l:4),d3`
Memory Indirect Pre-Indexed (68020)	`an@(d':L,ri:L:s)@(d:L)`	`movl a1@(d:L,d2:l:4)@(14:w)`
Memory Indirect Post-Indexed (68020)	`an@(d:L)@(d':L,ri:L:s)`	`movl a2@(2:w)@(14:w,d4:w:2)`

Table 6-1 *Addressing Modes— Continued*

Mode	*Notation*	*Example*
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 opcode required 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* in the MC68010 and 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 summarizing the grouping of the addressing modes into categories. Note that register lists can be used only by the `movem` and `fmovem` instructions.

Category	*Meaning*
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 with no explicit size specification.

Some addressing categories can be intersected 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***	***MC68020 Only***
Register Direct	`a`*n*`, d`*n*`, sp, pc, cc, sr, usp`	X			X	
A-Register Indirect	`a`*n*`@`	X	X	X	X	
A-Register Indirect with Displacement	`a`*n*`@(d:L)`	X	X	X	X	X
A-Register Indirect with Word Index	`a`*n*`@(d:L,r`*i*`:w:s)`	X	X	X	X	X
A-Register Indirect with Long Index	`a`*n*`@(d:L,r`*i*`:l:s)`	X	X	X	X	X
A-Register Indirect with Post Increment	`a`*n*`@+`	X	X		X	
A-Register Indirect with Pre Decrement	`a`*n*`@-`	X	X		X	

Table 6-2 *Addressing Categories— Continued*

Addressing Mode	*Assembler Syntax*	*Data*	*Memory*	*Control*	*Alterable*	*MC68020 Only*
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:L)@(d':L,ri:L:s)`	X	X	X	X	X
Memory-Indirect Pre-Indexed	`an@(d':L,ri:L:s)@(d:L)`	X	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-Indirect with Displacement	`pc@(d:L)`	X	X	X		X
PC-relative with Word Index	`pc@(d,ri:w)`	X	X	X		
PC-Indirect with Word Index	`pc@(d:L,ri:w:s)`	X	X	X		X
PC-relative with Long Index	`pc@(d,ri:l)`	X	X	X		
PC-Indirect with Long Index	`pc@(d:L,ri:l:s)`	X	X	X		X
PC-Memory Indirect Post-Indexed	`pc@(d:L)@(d':L,ri:L:s)`	X	X	X	X	X
PC-Memory Indirect Pre-Indexed	`pc@(d':L,ri:L:s)@(d:L)`	X	X	X	X	X
Immediate Data	`#nnn`	X	X			

A

as Error Codes

A

as 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 the specified source file. Check the spelling, that the pathname supplied is correct, and that you have read permission for the file.

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

Too many file names given
The assembler cannot 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 of this manual.

A.2. Assembler Error Messages

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

```
as: error (&lt;line_no&gt;): &lt;error_code&gt;
```

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 evaluates to some value other than 1, 2 or 4.

Invalid assignment
An attempt was made to redefine a label with a direct assignment 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 is 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 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 a direct assignment statement.
- An attempt to use, as a label, an identifier which was previously defined in a direct assignment statement.

Multiply Defined Symbol (Phase Error)

This rarely occurring message indicates an inconsistency in the assembler. Report it to Sun Microsystems Customer 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.

NOTE *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. Divide the program into smaller portions; assemble portions of the program separately, then bind them together using the linker.

Register out of range

In the FPA's dot product, matrix move and transpose instructions when the register specified does not fall within the specified range, then this error is reported. Note that for most instructions where one operand is an effective address, the register range is 0 to 15. If all operands are FPA registers, the register range is 0 to 31. For constant RAM registers, the range is 0 to 511. This type of error would probably also cause the *Invalid operand* error to be reported.

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. Divide 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 *n*$ if *n*$ has not been defined.

Unqualified forward reference

The displacement field in an MC68020 based/indexed address mode contains an unqualified forward reference. Note that the displacement in a based/indexed address mode for the MC68020 instruction set can contain a forward or external reference *only* if the length specifier is present. The length specifier should be `:l` (long). This type of error would probably also cause *Multiply defined symbol (Phase error).*

Undefined Symbol

A 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 MC680x0 processor's instructions, the second covers the MC68881 floating-point processor's instructions. For more information about floating-point programming, see the *Floating-Point Programmer's Guide.*

Each entry describes the following things:

- The mnemonics for the instruction,
- The generic name of 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`, `addw`, `addl`, *etc.*
- An operand field of `a`*n* means any A-register.
- An operand field of `d`*n* means any D-register.
- An operand field of `r`*n* means any A- or D-register.
- An operand field of `f`*n* means any floating-point register.
- An operand field of `c`*n* means any control register.
- An operand field of *ea* means an 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 *vector* means an exception vector location.
- An operand field of *#data* means an immediate operand.
- Other special registers such as `cc` (condition code register) and `sr` (status register) are specifically indicated where appropriate.

The MC68020 provides a set of bit-field manipulating instructions that don't exist on the MC68010. Their notation includes a bit field specifier of the form {*offset:width*}, where the offset denotes the beginning of the bit field in the word and the width is the number of bits in the field.

Offset values are counted from the high-order bit, as 0, to the low-order bit, as 31.

NOTE *This ordering is the reverse of the convention used in the* `bchg`, `bclr`, `bset`, *and* `btst` *instructions.*

Offset and width may be either constants or data registers. For example:

- `bfins d0,a5@(4){#0:#9}`
- `bfexta a5@(4){d0:#8},d7`

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@-,a*X*@-	
addb addw addl	add binary	add*X* *ea*,d*n* add*X* d*n*,*ea* add*X* *ea*,a*n* (except addb) add*X* *#data*,*ea*	
addqb addqw addql	add quick	addq*X* *#data*,*ea*	
addxb addxw addxl	add extended	addx*X* dy,d*X* addx*X* ay@-,a*X*@-	
andb andw andl	logical and	and*X* *ea*,d*n* and*X* d*n*,*ea* and*X* *#data*,d*n*	
aslb aslw asll	arithmetic shift left	asl*X* d*X*,dy asl*X* *#data*,dy asl*X* *ea*	

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

Mnemonic	Operation Name	Syntax	Processor
asrb asrw asrl	arithmetic shift right	asr*X* d*X*,dy asr*X* *#data*,dy asr*X* *ea*	
bcc bccl bccs	branch conditionally	bcc*X* *label*	MC68020
bchg	test a bit and change	bchg d*n*,*ea* bchg *#data*,*ea*	
bclr	test a bit and clear	bclr d*n*,*ea* bclr *#data*,*ea*	
bkpt	breakpoint	bkpt *#data*	MC68020
bset	test a bit and set	bset d*n*,*ea* bset *#data*,*ea*	
btst	test a bit	btst d*n*,*ea* btst *#data*,*ea*	
bfchg bfclr	test a bit field and change test a bit field and clear	bfchg *ea{offset:width}* bfclr *ea{offset:width}*	MC68020 MC68020
bfexts	extract a bit field signed	bfexts *ea{offset:width}*,d*n*	MC68020
bfextu	extract a bit field unsigned	bfextu *ea{offset:width}*,d*n*	MC68020
bfffo	find first one in bit field	bfffo *ea{offset:width}*,d*n*	MC68020
bfins	insert a bit field	bfins d*n*,*ea{offset:width}*	MC68020
bfset	test a bit field and set	bfset *ea{offset:width}*	MC68020
bftst	test a bit field	bftst *ea{offset:width}*	MC68020
bcs bcsl bcss	branch carry set	bcs*X* *ea*	MC68020
beq beql beqs	branch on equal	beq*X* *ea*	MC68020
bge bgel bges	branch greater or equal	bge*X* *ea*	MC68020
bgt bgtl bgts	branch greater than	bgt*X* *ea*	MC68020

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

Mnemonic	Operation Name	Syntax	Processor
bhi bhil bhis	branch higher	bhi*X* *ea*	MC68020
ble blel bles	branch less than or equal	ble*X* *ea*	MC68020
bls blsl	branch lower or same	bls*X* *ea*	MC68020
blt bltl blts	branch less than	blt*X* *ea*	
bmi bmil bmis	branch minus	bmi*X* *ea*	
bne bnel bnes	branch not equal	bne*X* *ea*	MC68020
bpl bpll bpls	branch positive	bpl*X* *ea*	MC68020
bra bral bras	branch always	bra*X* *label*	MC68020
bsr bsrl bsrs	subroutine branch	bsr*X* *label*	MC68020
bvc bvcl bvcs	branch overflow clear	bvc*X* *ea*	MC68020
bvs bvsl bvss	branch overflow set	bvs*X* *ea* bvsl	MC68020
callm	call module	callm #*data*, *ea*	MC68020
cas2b cas2l cas2w	compare & swap with operand	cas2*X* d*c*1:d*c*2,d*u*1:d*u*2,(r*n*1):(r*n*2)	MC68020 MC68020 MC68020
casb casl casw	compare & swap with operand	cas*X* d*c*,d*u*, *ea*	MC68020 MC68020 MC68020

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

Mnemonic	*Operation Name*	*Syntax*	*Processor*
chkb	check register against bounds	chkX *ea*, d*n*	MC68020
chkw			MC68020
chkl			MC68020
chk2b	check register against bounds	chk2X *ea*, rn	MC68020
chk2l			MC68020
chk2w			MC68020
clrb	clear an operand	clrX *ea*	
clrw			
clrl			
cmp2b	compare register against bounds	cmp2X *ea*, r*n*	MC68020
cmp2l			MC68020
cmp2w			MC68020
cmpmb	compare memory	cmpmX ay@+, aX@+	
cmpmw			
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		divsX *ea*, d*n*	MC68020
divsll		divsX *ea*, d*q*	MC68020
		divsX *ea*, d*r*:d*q*	MC68020
divu	unsigned divide	divu *ea*, d*n*	
divul		divuX *ea*, d*n*	MC68020

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

Mnemonic	Operation Name	Syntax	Processor
divuw		divu*X* *ea*,d*n*	MC68020
		divu*X* *ea*,d*q*	MC68020
		divu*X* *ea*,d*r*:d*q*	MC68020
divull		divull *ea*,d*r*:d*q*	MC68020
eorb eorw eorl	logical exclusive or	eor*X* d*n*,*ea* eor*X* *#data*,*ea* eorb *#data*,*cc* eorw *#data*,*sr*	
exg	exchange registers	exg r*x*,r*y*	
extbl extw extl	sign extend	extbl d*n* ext*X* d*n*	MC68020
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*	
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		linkl a*n*,*#disp*	MC68020
lslb lslw lsll	logical shift left	lsl*X* d*x*,d*y* lsl*X* *#data*,d*y* lsl*X* *ea*	
lsrb lsrw lsrl	logical shift right	lsr*X* d*x*,d*y* lsr*X* *#data*,d*y* lsr*X* *ea*	
movb movl movw	move data	mov*X* *ea*,*ea* mov*X* *#data*,d*n*	

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

Mnemonic	*Operation Name*	*Syntax*	*Processor*

movw
movw
movc
	move from condition code register
	move from status register
	move to/from control register
		movw *cc*, *ea*
		movw sr, *ea*
		movc *rn*, *cr*
		movc *cr*, *rn*	

moveml movemw	move multiple registers	movem*X* *#mask*, *ea* movem*X* *ea*, *#mask* movem*X* *ea*, *reglist* movem*X* *reglist*, *ea*	
movepl movepw	move peripheral	movep*X* d*n*, a*n*@(*d*) movep*X* a*n*@(d), d*n*	
moveq	move quick	moveq *#data*, d*n*	
movsb movsw movsl	move to/from address space	movs*X* *rn*, *ea* movs*X* *ea*, *rn*	
muls mulslw mulsll	signed multiply	muls *ea*, d*n* muls*X* *ea*, d*l* muls*X* *ea*, d*h*:d*l*	 MC68020 MC68020
mulu mulul	unsigned multiply	mulu *ea*, d*n* mulu*X* *ea*, d*l* mulu*X* *ea*, d*h*:d*l*	 MC68020 MC68020
nbcd	negate decimal with extend	nbcd *ea*	
negb negw negl	negate binary	neg*X* *ea*	
negxb negxw negxl	negate binary with extend	negx*X* *ea*	
nop	no operation	nop	
notb notw notl	logical complement	not*X* *ea*	
orb orw orl	inclusive or	or*X* *ea*, d*n* or*X* d*n*, *ea* or *#data*, *ea* orb *#data*, *cc* orw *#data*, *sr*	
pack	pack	pack a*X*@-, ay@-, *#data* pack d*X*, dy, *#data*	MC68020 MC68020
pea	push effective address	pea *ea*	

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

Mnemonic	Operation Name	Syntax	Processor
reset	reset device	reset	
rolb rolw roll	rotate left rotate left	rol*X* dx,dy rol*X* *#data*,dy rol*X* *ea*	
rorb rorw rorl	rotate right	ror*X* dx,dy ror*X* *#data*,dy ror*X* *ea*	
roxlb roxlw roxll	rotate left with extend	roxl*X* dx,dy roxl*X* *#data*,dy roxl*X* *ea*	
roxrb roxrw roxrl	rotate right with extend	roxr*X* dx,dy roxr*X* *#data*,dy roxr*X* *ea*	
rtd rte rtm rtr rts	return and deallocate parameters return from exception return from module return and restore codes return from subroutine	rtd *#data* rte rtm *rn* rtr rts rts *#n*	 MC68020
sbcd	subtract decimal with extend	sbcd dy,dx sbcd ay@-,a*X*@-	
stop	halt machine	stop *#xxx*	
subb subw subl	arithmetic subtract	sub*X* *ea*,dn sub*X* dn,*ea* sub*X* *ea*,an sub*X* *#data*,*ea*	
st sf shi sls scc scs sne seq svc svs spl smi sge slt sgt sle	set all ones set all zeros set high set lower or same set carry clear set carry set set not equal set equal set no overflow set on overflow set plus set minus set greater or equal set less than set greater than set less than or equal	st *ea* sf *ea* shi *ea* sls *ea* scc *ea* scs *ea* sne *ea* seq *ea* svc *ea* svs *ea* spl *ea* smi *ea* sge *ea* slt *ea* sgt *ea* sle *ea*	

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

Mnemonic	*Operation Name*	*Syntax*	*Processor*
subqb subqw subql	subtract quick subtract quick	subq*X* *#data, ea*	
subxb subxw subxl	subtract extended	subx*X* dy, dx subx*X* ay@-, a*X*@-	
swap	swap register halves	swap d*n*	
tas	test operand then set	tas *ea*	
trap	trap	trap *#vector*	
trapcc trapccl trapccw	trap on carry clear	trapcc*X* trapcc*X* *#data*	MC68020 MC68020 MC68020
trapcs trapcsl trapcsw	trap on carry set	trapcsx trapcs*X* *#data*	MC68020 MC68020 MC68020
trapeq trapeql trapeqw	trap on equal	trapeq*X* trapeq*X* *#data*	MC68020 MC68020 MC68020
trapf trapfl trapfw	trap on never true	trapf*X* trapf*X* *#data*	MC68020 MC68020 MC68020
trapge trapgel trapgew	trap on greater or equal	trapge*X* trapge*X* *#data*	MC68020 MC68020 MC68020
trapgt trapgtl trapgt	trap on greater	trapgt*X* trapgt*X* *#data*	MC68020 MC68020

The following table describes the MC68881 instruction 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-precision format instruction
- d indicates a double-precision format instruction

- x indicates an extended-precision format instruction
- p indicates a packed format instruction
- y indicates that any of l, s, p, w, d, or b, are acceptable.

Table B-2 *MC68881 Instructions supported by* as

Mnemonic	*Operation Name*	*Syntax*
fabsx	absolute value	fabsx *ea*, *fn*
fabsl		fabsx *fm*, *fn*
fabss		fabsy *ea*, *fn*
fabsp		
fabsw		
fabsd		
fabsb		
facosx	arc cosine	facosx *ea*, *fn*
facosl		facosx *fm*, *fn*
facoss		facosy *ea*, *fn*
facosp		
facosw		
facosd		
facosb		
faddx	add	faddx *ea*, *fn*
faddl		faddx *fm*, *fn*
fadds		faddy *ea*, *fn*
faddp		
faddw		
faddd		
faddb		
fasinx	arc sin	fasinx *ea*, *fn*
fasinl		fasinx *fm*, *fn*
fasins		fasiny *ea*, *fn*
fasinp		
fasinw		
fasind		
fasinb		
fatanx	arc tangent	fatanx *ea*, *fn*
fatanl		fatanx *fm*, *fn*
fatans		fatany *ea*, *fn*
fatanp		
fatanw		
fatand		
fatanb		
fatanhx	hyperbolic arc tangent	fatanhx *ea*, *fn*
fatanhl		fatanhx *fm*, *fn*
fatanhs		fatanhy *ea*, *fn*

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

Mnemonic	*Operation Name*	*Syntax*
fatanhp	hyperbolic arc tangent *(contd.)*	
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		
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)	

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

Mnemonic	***Operation Name***	***Syntax***
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		
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*
fcmpl		fcmpx *fm*,*fn*
fcmps		fcmpy *ea*,*fn*
fcmpp		
fcmpw		
fcmpd		
fcmpb		
fcosx	cosine	fcosx *ea*,*fn*
fcosl		fcosx *fm*,*fn*
fcoss		fcosy *ea*,*fn*
fcosp		
fcosw		
fcosd		
fcosb		
fcoshx	hyperbolic cosine	fcoshx *ea*,*fn*
fcoshl		fcoshx *fm*,*fn*
fcoshs		fcoshy *ea*,*fn*

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

Mnemonic	*Operation Name*	*Syntax*
fcoshp		
fcoshw	hyperbolic cosine *(contd.)*	
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)	
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*,f*n*
fdivl		fdivx f*m*,f*n*
fdivs		fdivy *ea*,f*n*
fdivp		
fdivw		
fdivd		
fdivb		

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

Mnemonic	*Operation Name*	*Syntax*
fetoxx fetoxl fetoxs fetoxp fetoxw fetoxd fetoxb	e^x	fetoxx *ea*,*fn* fetoxx *fm*,*fn* fetoxy *ea*,*fn*
fetoxm1x fetoxm1l fetoxm1s fetoxm1p fetoxm1w fetoxm1d fetoxm1b	e^x -1	fetoxm1x *ea*,*fn* fetoxm1x *fm*,*fn* fetoxm1y *ea*,*fn*
fgetexpx fgetexpl fgetexps fgetexpp fgetexpw fgetexpd fgetexpb	get exponent	fgetexpx *ea*,*fn* fgetexpx *fm*,*fn* fgetexpy *ea*,*fn*
fgetmanx fgetmanl fgetmans fgetmanp fgetmanw fgetmand fgetmanb	get mantissa	fgetmanx *ea*,*fn* fgetmanx *fm*,*fn* fgetmany *ea*,*fn*
fintx fintl fints fintp fintw fintd fintb	integer part	fintx *ea*,*fn* fintx *fm*,*fn* finty *ea*,*fn*
fintrx fintrzl fintrzs fintrzp fintrzw fintrzd fintrzb	integer part, round toward 0	fintrx *ea*,*fn* fintrx *fm*,*fn* fintry *ea*,*fn*

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

Mnemonic	*Operation Name*	*Syntax*
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)	
fjst	(signalling true)	
fjseq	(signalling equal)	
fjsne	(signalling not equal)	
flog10x	$\log_{10}$	flog10x *ea*,*fn*
flog10l		flog10x *fm*,*fn*
flog10s		flog10y *fn*
flog10p		
flog10w		
flog10d		
flog10b		
flog2x	$\log_2$	flog2x *ea*,*fn*
flog2l		flog2x *fm*,*fn*
flog2s		flog2y *ea*,*fn*
flog2p		

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

Mnemonic	*Operation Name*	*Syntax*
flog2w flog2d flog2b	$\log_2$ *(contd.)*	
flognx flognl flogns flognp flognw flognd flognb	$\log_e$	flognx *ea*,*fn* flognx *fm*,*fn* flogny *ea*,*fn*
flognp1x flognp1l flognp1s flognp1p flognp1w flognp1d flognp1b	$\log_e(x+1)$	flognp1x *ea*,*fn* flognp1x *fm*,*fn* flognp1y *ea*,*fn*
fmodx fmodl fmods fmodp fmodw fmodd fmodb	modulo	fmodx *ea*,*fn* fmodx *fm*,*fn* fmody *ea*,*fn*
fmovex fmovel fmoves fmovep fmovew fmoved fmoveb	move fp register	fmovex *ea*,*fn* fmovex *fm*,*ea* fmovey *ea*,*fn*
fmovecrx	move constant ROM	fmovecrx #*ccc*, *fn*
fmovemx fmoveml fmovem	move multiple data registers	fmovemy *ea*, list fmovemx list, *ea* fmoveml *ea*, d*n* fmovem d*n*, *ea*
fmulx fmull fmuls fmulp	multiply	fmulx *ea*,*fn* fmulx *fm*,*fn* fmuly *ea*,*fn*

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

Mnemonic	*Operation Name*	*Syntax*
fmulw fmuld fmulb	multiply *(contd.)*	
fnegx fnegl fnegs fnegp fnegw fnegd fnegb	negate	fnegx *ea*, *fn* fnegx *fm*, *fn* fnegy *ea*, *fn*
fnop	no operation	fnop
fremx freml frems fremp fremw fremd fremb	IEEE remainder	fremx *ea*, *fn* fremx *fm*, *fn* fremy *ea*, *fn*
frestore	restore internal state	frestore *ea*
fsave	save internal state	fsave *ea*
fscalex fscalel fscales fscalep fscalew fscaled fscaleb	scale exponent	fscalex *ea*, *fn* fscalex *fm*, *fn* fscaley *ea*, *fn*
fs*cc* fseq fsne fsneq fsgt fsngt fsge fsnge fslt fsnlt fsle fsnle fsgl fsngl fsgle	set according to condition (equal) (not equal) (not equal or equal) (greater than) (not greater than) (greater or equal) (not greater or equal) (less than) (not less than) (less or equal) (not less or equal) (greater or less) (not greater or less) (greater, less or equal)	fs*cc* *ea*

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

Mnemonic	*Operation Name*	*Syntax*
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)	
fsgldivx	single-precision divide	fsgldivx *ea*,*fn*
fsgldivs		fsgldivx *fm*,*fn*
fsgldivl		fsgldivy *ea*,*fn*
fsgldivp		
fsgldivw		
fsgldivb		
fsglmulx	single-precision multiply	fsglmulx *ea*,*fn*
fsglmuls		fsglmulx *fm*,*fn*
fsglmull		fsglmuly *ea*,*fn*
fsglmulp		
fsglmulw		
fsglmulb		
fsinx	sin	fsinx *ea*,*fn*
fsinl		fsinx *fm*,*fn*
fsins		fsiny *ea*,*fn*
fsinp		
fsinw		
fsind		
fsinb		
fsincosx	simultaneous sine and cosine	fsincosx *ea*,*fc*:*fs*
fsincosl		fsincosx *fm*,*fc*:*fs*
fsincoss		fsincosy *ea*,*fc*:*fs*
fsincosp		

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

Mnemonic	***Operation Name***	***Syntax***
fsincosw fsincosd fsincosb	simultaneous sine and cosine *(contd.)*	
fsinhx fsinhs fsinhp fsinhw fsinhd fsinhb	hyperbolic sine	fsinhx *ea*,f*n* fsinhx f*m*,f*n* fsinhy *ea*,f*n*
fsqrtx fsqrtl fsqrts fsqrtp fsqrtw fsqrtd fsqrtb	square root	fsqrtx *ea*,f*n* fsqrtx f*m*,f*n* fsqrty *ea*,f*n*
fsubx fsubl fsubs fsubp fsubw fsubd fsubb	subtract	fsubx *ea*,f*n* fsubx f*m*,f*n* fsuby *ea*,f*n*
ftanx ftanl ftans ftanp ftanw ftand ftanb	tangent	ftanx *ea*,f*n* ftanx f*m*,f*n* ftany *ea*,f*n*
ftanhx ftanhl ftanhs ftanhp ftanhw ftanhd ftanhb	hyperbolic tangent	ftanhx *ea*,f*n* ftanhx f*m*,f*n* ftanhy *ea*,f*n*
ftentoxx ftentoxl ftentoxs ftentoxp	10^x	ftentoxx *ea*,f*n* ftentoxx f*m*,f*n* ftentoxy *ea*,f*n*

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

Mnemonic	*Operation Name*	*Syntax*
`ftentoxw`	10^x *(contd.)*	
`ftentoxd`		
`ftentoxb`		
`ftrap`*cc*	trap conditionally	`ftrap`*cc*
`ftrapeq`	(equal)	`ftrap`*cc* `#`*data*
`ftrapeqw`		
`ftrapeql`		
`ftrapne`	(not equal)	
`ftrapnew`		
`ftrapnel`		
`ftrapgt`	(greater than)	
`ftrapgtw`		
`ftrapgtl`		
`ftrapngt`	(not greater than)	
`ftrapngtw`		
`ftrapngtl`		
`ftrapge`	(greater or equal)	
`ftrapgew`		
`ftrapgel`		
`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`		

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

Mnemonic	*Operation Name*	*Syntax*
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		
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		

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

Mnemonic	*Operation Name*	*Syntax*
ftrapsf	(signalling false)	
ftraptw		
ftrapsfl		
ftrapst	(signalling true)	
ftrapsfw		
ftrapstl		
ftrapseq	(signalling equal)	
ftrapseqw		
ftrapseql		
ftrapsne	(signalling not equal)	
ftrapsnew		
ftrapsnel		
ftstx	test operand	ftstx *ea*
ftstl		ftstx *fm*
ftsts		ftsty *ea*
ftstp		
ftstw		
ftstd		
ftstb		
ftwotoxx	2^x	ftwotoxx *ea*,*fn*
ftwotoxl		ftwotoxx *fm*,*fn*
ftwotoxs		ftwotoxy *ea*,*fn*
ftwotoxp		
ftwotoxw		
ftwotoxd		
ftwotoxb		

C

FPA Assembler Syntax

C

FPA Assembler Syntax

This appendix describes the Sun Floating-Point Accelerator (FPA) support extensions to `as` included in Sun software release 3.1 and later.

The extensions to `as` are described in general, with discussions of two-, three-, and four-operand instruction examples. Some instructions covered separately don't follow the formats described at the beginning of the appendix. The appendix includes restrictions and potential errors, followed by a summary of supported floating-point instructions.

C.1. Instruction Syntax

The general format for floating-point instructions is

```
fpopt@A      operands
```

where

`fp` indicates an FPA instruction.

op is the opcode name.

`t` is the operand type, either single (s) or double (d).

The `@`*A* part of the instruction is optional. When present, *A* specifies the address register which contains the base address for the FPA and can be in the range 0..7. If this form is used, a previous instruction must load the FPA address (0xe0000000) into the specified address register.

If `@`*A* is not present, then absolute long addressing is used to refer to the FPA. This form is more efficient for short routines.

Depending on the instruction, there may be from zero to four operands specified. The operands can be any of the following forms:

- Any MC68020 effective address, with the exception that absolute short addresses are not allowed for double-precision values.
- If either of the data register or the address register is used to hold a double-precision value, then the value will be in a register pair and both registers, separated by a colon, must be specified in the instruction. For example:

```
fpaddd   d0:d1, fpa0
```

The only exception to this rule is the `fpltod` instruction (convert integer to double-precision value).

- In some instructions (command register type) it is possible to specify that the register be in constant RAM. The syntax used for this case is `%`*n*, where *n* is a register number in the range 0 to 511.

C.2. Register Syntax

The 32 floating-point data registers are designated `fpa0, fpa1, ..., fpa31`. The supported control registers are:

Hardware	***Software***
`MODE3_0`	`fpamode`
`WSTATUS`	`fpastatus`

C.3. Operand Types

`as` supports three floating-point operand types:

- `s` for single-precision floating-point operands.
- `d` for double-precision floating-point operands.
- `l` for 32-bit integer operands, used for integer to floating-point conversions.

C.4. Two-Operand Instructions

Opcodes such as add, subtract, multiply, divide, negate, absolute value, square root, conversion from integer to floating-point, conversion from single to double (and vice versa) are all represented as:

```
fpopt   X, fpan
```

where *t*= `s` or `d`, and *X* is any valid MC68020 effective address for an operand or is an FPA data register.

If *X* is an FPA register which is in the constant RAM, then it can be in the range 0 to 511. If it is not in constant RAM, then it is one of the 32 FPA data registers. When *X* is an FPA register, then `fpa`*n* is one of the 32 floating-point data registers. If *X* is an effective address, then `fpa`*n* is one of the FPA registers in the range 0 to 15. The following are examples of such instructions:

Instruction		Computes
`fpnegs`	*<effective address>*, `fpa1`	
`fpsqrd`	*<effective address>*, `fpa2`	
`fpsubs`	`fpa1, fpa2`	fpa2 ← fpa2 - fpa1
`fprsubs`	`fpa1, fpa2`	fpa2 ← fpa1 - fpa2
`fpdivs`	`d0, fpa2`	fpa2 ← fpa2 / d0
`fprdivs`	`d0, fpa2`	fpa2 ← d0 / fpa2

In the above examples `fprsubs` and `fprdivs` are the reverse subtract and reverse divide operators, respectively.

The opcodes for `sine`, `cosine`, `atan`, `e^x`, `e^x -1`, `ln(x)`, `ln(1+x)`, `sqrt(x)`, and `sincos(x)` are all supported as command register type instructions:

```
fpopt  fpax, fpan
```

where *t* = `s` or `d`.

`fpax` is either a floating-point register or a register in the constant RAM (which is specified as `%`*number*). For the `sincos` instruction, the destination operand is actually a register pair:

```
fpsincost  fpax, fpac:fpas
```

where `fpac` is the cosine's destination and `fpas` is the sine's destination.

C.5. Three-Operand Instructions

The opcodes +, -, *, / are supported in extended and command register forms as

```
fpop3t    X, fpam, fpan
```

where *t* = `s` or `d` and *X* is an <*effective address*> for an extended instruction or a floating-point register for a command register type of instruction.

In the *command register form*, *X* and `fpam` can indicate a register number in the constant RAM. That is, they can either be in the range 0 to 511 or in the range 0 to 31. In the *extended instruction form*, `fpam` and `fpan` must be in the range 0 to 15. In the above format the positions of *X* and `fpam` can be exchanged for the commutative operators add and multiply (the result of the operation remains the same).

For example,

```
fpa2 ← <effective address> + fpa1
```

can be represented by either of the following forms:

```
    fpadd3s  <effective address>, fpa1, fpa2
    fpadd3s  fpa1, <effective address>, fpa2
```

The same rule applies to subtract and divide operations. However, they are not commutative, so different answers result from each order. For example,

```
fpa2 ← fpa1 - <effective address>
```

must be coded as:

```
    fpsub3s  <effective address>, fpa1, fpa2
```

whereas

```
fpa2 ← <effective address> - fpa1
```

must be coded as:

```
    fpsub3s  fpa1, <effective address>, fpa2
```

Following the same format,

```
fpa3 ← fpa2 - fpa1
```

must be coded as:

```
    fpsub3s  fpa1, fpa2, fpa3
```

C.6. Four-Operand Instructions

In the extended and command register formats there are pivot instructions of the form:

```
fpopt  X, fpax, fpay, fpan
```

where `fpan` is the destination floating-point data register, *t* = `s` or `d`, and *X* is an effective address or a floating-point register.

In the extended form, the positions of *X* and `fpay` can be exchanged for both single- and double-precision types of instructions. In single-precision extended form, it is possible for two of the four operands to be effective addresses. This is in general either the first and third or the second and third operands.

In the command register form, `fpax` and `fpay` can be replaced by `%x` and `%y` indicating register numbers *x* and *y* in the constant RAM.

For four-operand instructions, `fpax`, `fpay` and `fpan` can each be in the range 0 to 15 when *X* is an effective address. If *X* is an FPA register, then *X* and `fpan` must be in the range 0 to 31 and `fpax` and `fpay` can either be in the range 0 to 511 (designating a location in constant RAM) or else in the range 0 to 31.

These pivot instructions are rather complicated and will be dealt with completely. The following shows the forms of each operation, the assembly code equivalent to each form, a generalization of the assembly instruction and a sequence of operations equivalent to the pivot instruction.

Instruction		Meaning
fpma{s,d}	*<effective address>*, reg2, reg3, reg1	reg1 ← reg3 + (reg2 * operand)
fpma{s,d}	reg2, reg3, *<effective address>*, reg1	reg1 ← operand + (reg3 * reg2)
fpma{s,d}	reg4, reg2, reg3, reg1	reg1 ← reg3 + (reg2 * reg4)
fpmas	*<ea1>*, reg2, *<ea2>*, reg1	reg1 ← operand2 + (reg2 * operand1)

The fpma instruction, where m stands for multiply, and a stands for add, can be generalized as

```
fpmat X, fpax, fpay, fpan
```

where *t* is s or d, and *X* is an *<effective address>* or one of the floating-point data registers. In the extended type of instruction, the positions of *X* and fpay can be exchanged. Also, for single precision either the first and third operands or the second and third operands can be effective addresses. Note that, for example,

```
fpmas   d0, fpa1, fpa2, fpa3
```

is equivalent to the following sequence of instructions

```
fpmul3s     d0, fpa1, temp
fpadd3s     temp, fpa2, temp
fpmoves     temp, fpa3
```

where temp is a temporary register.

Instruction		Meaning
fpms{s,d}	*<effective address>*, reg2, reg3, reg1	reg1 ← reg3 - (reg2 * operand)
fpms{s,d}	reg2, reg3, *<effective address>*, reg1	reg1 ← operand - (reg3 * reg2)
fpms{s,d}	reg4, reg2, reg3, reg1	reg1 ← reg3 - (reg2 * reg4)
fpmss	*<ea1>*, reg2, *<ea2>*, reg1	reg1 ← operand2 - (reg2 * operand1)

The fpms instruction, where m stands for multiply, and s stands for subtract, can be generalized as

```
fpmst    X, fpax, fpay, fpan
```

where *t* is s or d, and *X* is an *<effective address>* or one of the floating-point data registers. In the extended type of instruction, the positions of *X* and fpay can be exchanged. Also, in single-precision two-memory instructions, either the first and third operands or the second and third operands can be effective addresses. Note that, for example,

```
fpmss  fpa1, fpa2, d0, fpa3
```

is equivalent to the following sequence of instructions

```
fpmul3s     fpa1, fpa2, temp
fpsub3s     temp, d0, temp
fpmoves     temp, fpa3
```

The `fpmr` instruction, where `m` stands for multiply, and `r` stands for reverse subtract, can be generalized as

fpmr*t* *X*, fpa*x*, fpa*y*, fpa*n*

where *t* is `s` or `d`, and *X* is an <*effective address*> or one of the floating-point data registers. In the extended type of instruction, the positions of *X* and `fpay` can be exchanged.

Instruction		Meaning
fpmr{s,d}	<*effective address*>, reg2, reg3, reg1	reg1 ← (-reg3) + (reg2 * operand)
fpmr{s,d}	reg2, reg3, <*effective address*>, reg1	reg1 ← (-operand) + (reg3 * reg2)
fpmr{s,d}	reg4, reg2, reg3, reg1	reg1 ← (-reg3) + (reg2 * reg4)
fpmrs	<*ea1*>, reg2, <*ea2*>, reg1	reg1 ← (-operand2) + (reg2 * operand1)

In single-precision extended form either the first and third operands or the second and third operands can be effective addresses. Note that, for example,

```
fpmrs   d0, fpa1, fpa2, fpa3
```

is equivalent to the following sequence of instructions:

```
fpmul3s     d0, fpa1, temp
fpsub3s     fpa2, temp, temp
fpmoves     temp, fpa3
```

The `fpam` instruction, where `a` stands for add, and `m` stands for multiply, can be generalized as

fpam*t* *X*, fpa*x*, fpa*y*, fpa*n*

where *t* is `s` or `d`, and *X* is an <*effective address*> or one of the floating-point data registers. In the extended type of instruction, the positions of *X* and `fpay` can be exchanged.

	Instruction	Meaning
fpam{s,d}	*<effective address>*, reg2, reg3, reg1	reg1 ← reg3 * (reg2 + operand)
fpam{s,d}	reg2, reg3, *<effective address>*, reg1	reg1 ← operand * (reg3 + reg2)
fpam{s,d}	reg4, reg2, reg3, reg1	reg1 ← reg3 * (reg2 + reg4)
fpams	*<ea1>*, reg2, *<ea2>*, reg1	reg1 ← operand2 * (reg2 + operand1)

In single-precision two-memory instructions, either the first and third operands or the second and third operands can be effective addresses. Note that, for example,

```
fpams    fpa1, fpa2, fpa3, fpa4
```

is equivalent to the following sequence of instructions:

```
fpadd3s    fpa1, fpa2, temp
fpmul3s    temp, fpa3, temp
fpmoves    temp, fpa4
```

The `fpsm` instruction, where `s` stands for subtract, and `m` stands for multiply, can be generalized as

```
fpsmt    X, fpax, fpay, fpan
```

where *t* is `s` or `d`, and *X* is an effective address or one of the floating-point data registers. In the extended type of instruction, the positions of *X* and `fpay` can be exchanged. The special cases for single-precision instructions are that either the first and third operands or the second and third operands can be effective addresses.

	Instruction	Meaning
fpsm{s,d}	*<effective address>*, reg2, reg3, reg1	reg1 ← reg3 * (reg2 - operand)
fpsm{s,d}	reg2, reg3, *<effective address>*, reg1	reg1 ← operand * (reg3 - reg2)
fpsm{s,d}	reg4, reg2, reg3, reg1	reg1 ← reg3 * (reg2 - reg4)
fpsm{s,d}	reg2, *<effective address>*, reg3, reg1	reg1 ← reg3 * (-reg2 + operand)
fpsm{s,d}	reg2, reg4, reg3, reg1	reg1 ← reg3 * (-reg2 + reg4)
fpsms	*<ea1>*, reg2, *<ea2>*, reg1	reg1 ← operand2 * (reg2 - operand1)
fpsms	reg2, *<ea1>*, *<ea2>*, reg1	reg1 ← operand2 * (-reg2 + operand1)

Note that, for example,

```
fpsms  d0, fpa1, fpa2, fpa3
```

is equivalent to the following sequence of instructions:

```
fpsub3s  d0, fpa1, temp
fpmul3s  temp, fpa2, temp
fpmoves  temp, fpa3
```

C.7. Other Instructions

Other special instructions are listed below. In each of them the last operand is also the destination, except for `tst`, `cmp` and `mcmp` where `fpastatus` is the implied destination. *X* is either an effective address or an FPA data register and *t* is either `s` or `d` for all instructions except `fpmove`*t*, where *t* can be `s`, `d`, or `l`.

Table C-1 *Other Instructions*

Mnemonic	*Operand*	*Operation Name*
fpnop		nop
fptst*t*	*X*	operand compare with zero
fpcmp*t*	*X*, fpa*m*	register *m* compare with operand
fpmcmp*t*	*X*, fpa*m*	register *m* compare magnitude with operand
fpmove*t*	fpa*m*, fpa*n*	move floating-point registers
fpmove2*t*	fpa*m*, fpa*n*	2x2 matrix move
fpmove3*t*	fpa*m*, fpa*n*	3x3 matrix move
fpmove4*t*	fpa*m*, fpa*n*	4x4 matrix move
fpdot2*t*	fpa*x*, fpa*y*, fpa*n*	fpa*n* ← fpa*x**fpa*y* + (fpa*x*+*1*) * (fpa*y*+*1*)
fpdot3*t*	fpa*x*, fpa*y*, fpa*n*	fpa*n* ← fpa*x**fpa*y* + (fpa*x*+*1*) * (fpa*y*+*1*) + (fpa*x*+2) * (fpa*y*+2)
fpdot4*t*	fpa*x*, fpa*y*, fpa*n*	fpa*n* ← fpa*x**fpa*y* + (fpa*x*+*1*)*(fpa*y*+*1*) + (fpa*x*+2)*(fpa*y*+2) + (fpa*x*+*3*)*(fpa*y*+*3*)
fptran2*t*	fpa*m*, fpa*n*	transpose 2x2 matrix
fptran3*t*	fpa*m*, fpa*n*	transpose 3x3 matrix
fptran4*t*	fpa*m*, fpa*n*	transpose 4x4 matrix
fpmove	fpamode, <*ea*>	read mode register
fpmove	<*ea*>, fpamode	write to mode register
fpmove	fpastatus, <*ea*>	read status register
fpmove	<*ea*>, fpastatus	write to status register
fpmove*t*	fpa*m*, <*ea*>	read a floating-point data register
fpmove*t*	<*ea*>, fpa*n*	write to a floating-point data register

C.8. Restrictions and Errors

In double-precision instructions, when absolute short addressing or a single data or address register is used, `as` reports an invalid operand error.

For the dot product and matrix move and transpose instructions, when the register specified does not fall within the specified range, `as` reports a register out of range error.

For most instructions where one operand is an effective address, the register range is 0 to 15. If all operands are FPA registers, then the register range is 0 to 31. For constant RAM registers, the range is 0 to 511. `as` reports an invalid operand error when any of these registers are not within the permitted range.

C.9. Instruction Set Summary

In the following table, *X* is any valid MC68020 effective address (the form (*xxx*) :w is not allowed for double) or FPA register. In some three- or four-address instructions the position of the *X* and one of the FPA register can be exchanged. This is shown in the fourth column of the following table.

Table C-2 *Floating-Point Instructions*

Instruction	*Operand*	*Operation*	*Alternative*
fpnegs fpnegd	*X*, fpa*n* *X*, fpa*n*	negate single negate double	
fpabss fpabsd	*X*, fpa*n* *X*, fpa*n*	absolute value single absolute value double	
fpltos fpltod	*X*, fpa*n* *X*, fpa*n*	convert integer to single convert integer to double	
fpstol fpdtol	*X*, fpa*n* *X*, fpa*n*	convert single to integer convert double to integer	
fpstod fpdtos	*X*, fpa*n* *X*, fpa*n*	convert single to double convert double to single	
fpsqrs fpsqrd	*X*, fpa*n* *X*, fpa*n*	square single square double	
fpadds fpadd3s	*X*, fpa*n* *X*, fpa*m*, fpa*n*	add single add single	 fpa*m*, *X*, fpa*n*
fpaddd fpadd3d	*X*, fpa*n* *X*, fpa*m*, fpa*n*	add double add double	 fpa*m*, *X*, fpa*n*
fpsubs fpsub3s fprsubs	*X*, fpa*n* *X*, fpa*m*, fpa*n* <*ea*>, fpa*n*	subtract single subtract single reverse subtract single	 fpa*m*, *X*, fpa*n*
fpsubd fpsub3d fprsubd	*X*, fpa*n* *X*, fpa*m*, fpa*n* <*ea*>, fpa*n*	subtract double subtract double reverse subtract double	 fpa*m*, *X*, fpa*n*
fpmuls fpmul3s	*X*, fpa*n* *X*, fpa*m*, fpa*n*	multiply single multiply single	 fpa*m*, *X*, fpa*n*

Table C-2 *Floating-Point Instructions— Continued*

Instruction	*Operand*	*Operation*	*Alternative*
fpmuld	*X*, fpa*n*	multiply double	
fpmul3d	*X*, fpa*m*, fpa*n*	multiply double	fpa*m*, *X*, fpa*n*
fpdivs	*X*, fpa*n*	divide single	
fpdiv3s	*X*, fpa*m*, fpa*n*	divide single	fpa*m*, *X*, fpa*n*
fprdivs	<*ea*>, fpa*n*	reverse divide single	
fpdivd	*X*, fpa*n*	divide double	
fpdiv3d	*X*, fpa*m*, fpa*n*	divide double	fpa*m*, *X*, fpa*n*
fprdivd	<*ea*>, fpa*n*	reverse divide double	
fpnop		nop	
fptsts	*X*	single compare with 0	
fptstd	*X*	double compare with 0	
fpcmps	*X*, fpa*m*	single compare	
fpcmpd	*X*, fpa*m*	double compare	
fpmcmps	*X*, fpa*m*	single magnitude compare	
fpmcmpd	*X*, fpa*m*	double magnitude compare	
fpsins	fpa*x*, fpa*n*	sine single	
fpsind	fpa*x*, fpa*n*	sine double	
fpcoss	fpa*x*, fpa*n*	cosine single	
fpcosd	fpa*x*, fpa*n*	cosine double	
fpatans	fpa*x*, fpa*n*	atan single	
fpatand	fpa*x*, fpa*n*	atan double	
fpetoxs	fpa*x*, fpa*n*	e^x single	
fpetoxd	fpa*x*, fpa*n*	e^x double	
fpetoxm1s	fpa*x*, fpa*n*	e^x-1 single	
fpetoxm1d	fpa*x*, fpa*n*	e^x-1 double	
fplogns	fpa*x*, fpa*n*	ln(x) single	
fplognd	fpa*x*, fpa*n*	ln(x) double	
fplognp1s	fpa*x*, fpa*n*	ln(1+x) single	
fplognp1d	fpa*x*, fpa*n*	ln(1+x) double	
fpsincoss	fpa*x*, fpa*c*:fpa*s*	fpa*c* ← cosine(x), fpa*s* ← sine(x)	
fpsincosd	fpa*x*, fpa*c*:fpa*s*	fpa*c* ← cosine(x), fpa*s* ← sine(x)	
fpmas	*X*, fpa*x*, fpa*y*, fpa*n*	fpa*n* ← (fpa*x* * X) + fpa*y*	fpa*x*, *X*, fpa*y*, fpa*n* fpa*y*, fpa*x*, *X*, fpa*n* *X*, fpa*x*, *X*, fpa*n* fpa*x*, *X*, *X*, fpa*n*
fpmad	*X*, fpa*x*, fpa*y*, fpa*n*	fpa*n* ← (fpa*x* * X) + fpa*y*	fpa*x*, *X*, fpa*y*, fpa*n* fpa*y*, fpa*x*, *X*, fpa*n*
fpmss	*X*, fpa*x*, fpa*y*, fpa*n*	fpa*n* ← fpa*y* - (fpa*x* * x)	fpa*x*, *X*, fpa*y*, fpa*n* fpa*y*, fpa*x*, *X*, fpa*n* *X*, fpa*x*, *X*, fpa*n* fpa*x*, *X*, *X*, fpa*n*

Table C-2 *Floating-Point Instructions— Continued*

Instruction	*Operand*	*Operation*	*Alternative*
fpmsd	X, fpax, fpay, fpan	fpan ← fpay - (fpax * x)	fpax, X, fpay, fpan
			fpay, fpax, X, fpan
fpmrs	X, fpax, fpay, fpan	fpan ← (fpax * x) - fpay	fpax, X, fpay, fpan
			fpay, fpax, X, fpan
			X, fpax, X, fpan
			fpax, X, X, fpan
fpmrd	X, fpax, fpay, fpan	fpan ← (fpax * x) - fpay	fpax, X, fpay, fpan
			fpay, fpax, X, fpan
fpams	X, fpax, fpay, fpan	fpan ← (fpax + x) * fpay	
			fpax, X, fpay, fpan
			fpay, fpax, X, fpan
			X, fpax, X, fpan
			fpax, X, X, fpan
fpamd	X, fpax, fpay, fpan	fpan ← (fpax + x) * fpay	
			fpax, X, fpay, fpan
			fpay, fpax, X, fpan
fpsms	X, fpax, fpay, fpan	fpan ← (fpax - x) * fpay	
			fpax, X, fpay, fpan
			fpay, fpax, X, fpan
			X, fpax, X, fpan
			fpax, X, X, fpan
fpsmd	X, fpax, fpay, fpan	fpan ← (fpax - x) * fpay	
			fpax, X, fpay, fpan
			fpay, fpax, X, fpan
fpmoves	<ea>, fpan	write to a register, single	
fpmoved	<ea>, fpan	write to a register, double	
fpmovel	<ea>, fpan	write to a register, integer	
fpmoves	fpam, <ea>	read a register, single	
fpmoved	fpam, <ea>	read a register, double	
fpmove2s	fpam, fpan	2x2 matrix move, single	
fpmove2d	fpam, fpan	2x2 matrix move, double	
fpmove3s	fpam, fpan	3x3 matrix move, single	
fpmove3d	fpam, fpan	3x3 matrix move, double	
fpmove4s	fpam, fpan	4x4 matrix move, single	
fpmove4d	fpam, fpan	4x4 matrix move, double	
fpdot2s	fpax, fpay, fpan	fpan ← fpax*fpay + (fpax+1) * (fpay+1)	
fpdot2d	fpax, fpay, fpan	fpan ← fpax*fpay + (fpax+1) * (fpay+1)	
fpdot3s	fpax, fpay, fpan	fpan ← fpax*fpay + (fpax+1) * (fpay+1) + (fpax+2) * (fpay+2)	
fpdot3d	fpax, fpay, fpan	fpan ← fpax*fpay + (fpax+1) * (fpay+1) + (fpax+2) * (fpay+2)	

Table C-2 *Floating-Point Instructions— Continued*

Instruction	*Operand*	*Operation*	*Alternative*
fpdot4s	fpa*x*, fpa*y*, fpa*n*	fpa*n* ← fpa*x**fpa*y* + (fpa*x*+*1*)*(fpa*y*+*1*) + (fpa*x*+2)*(fpa*y*+2) + (fpa*x*+*3*)*(fpa*y*+*3*)	
fpdot4d	fpa*x*, fpa*y*, fpa*n*	fpa*n* ← fpa*x**fpa*y* + (fpa*x*+*1*)*(fpa*y*+*1*) + (fpa*x*+2)*(fpa*y*+2) + (fpa*x*+*3*)*(fpa*y*+*3*)	
fptran2s	fpa*m*, fpa*n*	transpose 2x2 matrix, single	
fptran2d	fpa*m*, fpa*n*	transpose 2x2 matrix, double	
fptran3s	fpa*m*, fpa*n*	transpose 3x3 matrix, single	
fptran3d	fpa*m*, fpa*n*	transpose 3x3 matrix, double	
fptran4s	fpa*m*, fpa*n*	transpose 4x4 matrix, single	
fptran4d	fpa*m*, fpa*n*	transpose 4x4 matrix, double	
fpmove	fpamode, <*ea*>	read the mode register	
fpmove	<*ea*>, fpamode	write on mode register	
fpmove	fpastatus, <*ea*>	read the status register	
fpmove	<*ea*>, fpastatus	write to status register	

Index

U

W

Notes

Notes

Notes

Notes

Notes