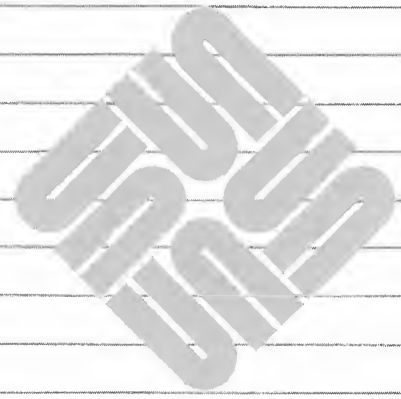
sun microsystems

Assembler Language Reference Manual *for the* Sun Workstation



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

Part No: 800-1179-01
Revision B of 15 May, 1985

Credits and Acknowledgements

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

Trademarks

Revision History

Revision	Date	Comments
A	15th July 1983	First release of this Manual.
B	15th August 1983	Second Release of this manual entailed a complete reorganization and some rewriting of the individual articles.
C	1st November 1983	Third Release of this manual entailed minor corrections and updates.
D	7th January 1984	Added chapter on Shell Programming. Added chapter on ADB. Many minor corrections and updates.
E	15 May 1985	Extracted from ***Programming Tools*** manual to make a separate language manual. Minor corrections and updates. Folded specific MC68010 descriptions into rest of text. Many minor corrections from α to β versions.

Preface

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

As provides the assembly language programmer with a minimal set of facilities to write programs in assembly language. Since the majority of 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.

This manual describes the syntax and usage of the *as* assembler for the Motorola MC68010 microprocessor. The basic format of *as* is loosely based on the Digital Equipment Corp Macro-11 assembler described in DEC's publication DEC-11-0MACA-A-D but also contains elements of the UNIX PDP-11 *as*(1) assembler. The instruction mnemonics and effective address format are derived from a Motorola publication on the MC68000: the *MACSS MC68000 Design Specification Instruction Set Processor* dated June 30, 1979.

This is a *reference manual* as opposed to a treatise on writing in assembly language. It is assumed that the reader is familiar with the concepts of machine architecture, the reasons for an assembler, the ideas of instruction mnemonics, operands, and effective address modes, and assembler directives. It is also assumed that the reader is familiar with the MC68010 processor, its instruction set, its addressing modes, and especially the irregularities in them.

† UNIX is a trademark of Bell Laboratories.

Contents

Chapter 1 Introduction **1-1**

Chapter 2 Elements of Assembly Language **2-1**

Chapter 3 Expressions **3-1**

Chapter 4 Layout of an Assembly Language Source Program **4-1**

Chapter 5 Assembler Directives **5-1**

Chapter 6 Instructions and Addressing Modes **6-1**

Appendix A Error Codes **A-1**

Appendix B List of As Opcodes **B-1**

Contents

Preface iv

Chapter 1 Introduction **1-1**
1.1. How to Use the Assembler 1-1
1.2. Notation 1-2
1.3. Further Reading 1-3

Chapter 2 Elements of Assembly Language **2-1**
2.1. Character Set Which the Assembler Recognizes 2-1
2.2. Identifiers 2-1
2.3. Numeric Labels 2-2
2.4. Local Labels 2-2
2.5. Scope of Labels 2-2
2.6. Constants 2-3
2.7. Numeric Constants 2-3
2.8. String Constants 2-4
2.9. Assembly Location Counter 2-4

Chapter 3 Expressions **3-1**
3.1. Operators 3-1
3.2. Terms 3-2
3.3. Expressions 3-2
3.4. Absolute, Relocatable, and External Expressions 3-2

Chapter 4 Layout of an Assembly Language Source Program **4-1**
4.1. Label Field 4-1
4.2. Operation Code Field 4-2
4.3. Operand Field 4-2
4.3.1. Register Operands 4-3
4.4. Comment Field 4-3
4.5. Direct Assignment Statements 4-4

Chapter 5 Assembler Directives **5-1**
5.1. `.ascii` — Generate Sequence of Character Data 5-2
5.2. `.asciz` — Generate Zero Terminated Sequence of Character Data 5-3
5.3. `.byte`, `.word`, `.long` — Generate Data 5-3

5.4. `.text`, `.data`, `.bss` — Switch Location Counter 5-4
5.5. `.skip` — Advance the Location Counter 5-5
5.6. `.lcomm` — Reserve Space in .bss Area 5-5
5.7. `.globl` — Designate an External Identifier 5-6
5.8. `.comm` — Define Name and Size of a Common Area 5-6
5.9. `.even` — Force Location Counter to Even Byte Boundary 5-6

Chapter 6 Instructions and Addressing Modes **6-1**
6.1. Instruction Mnemonics 6-1
6.2. Extended Branch Instruction Mnemonics 6-1
6.3. Addressing Modes 6-2
6.4. Addressing Categories 6-4

Appendix A Error Codes **A-1**
A.1. Usage Errors A-1
A.2. Assembler Error Messages A-1

Appendix B List of As Opcodes **B-1**

Tables

Table 5-1 Assembler Directives 5-2
Table 6-1 Addressing Modes 6-3
Table 6-2 Addressing Categories 6-5

Chapter 1

Introduction

1.1. How to Use the Assembler

By convention, the assembly language source code of the program should be in a file with a *.s* suffix. Suppose that your program is in a file called *parts.s*. To run the assembler, type the command:

```
tutorial% as parts.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*.

−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. If there are any externals which are too far away, the loader will complain when the program is linked.

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

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

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

this job anyway.

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

1.2. Notation

The notation used in this chapter 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 occuring 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>*

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

[*<thing>*]

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

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

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

1.3. Further Reading

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

Chapter 2

Elements of Assembly Language

This chapter covers the lexical elements which comprise an assembly language program. The next chapter discusses the rules for expressions and operand formation. Topics covered in this chapter are:

- ***Character set*** which the assembler recognizes,
- Rules for ***identifiers***,
- Syntax for ***numeric constants***,
- Syntax for ***string constants***,
- Rules for ***comments***,
- Layout of an assembly language ***source statement***.

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 then form assembly language ***statements***, and sequences of statements then form an assembly language program. This section describes the basic lexical elements of ***as***.

2.1. Character Set Which the Assembler Recognizes

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 line feed).

2.2. Identifiers

Identifiers are used to tag assembler statements (where they are called ***labels***), as the location tag 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 255 characters of an identifier are significant and are checked in comparisons with other identifiers.
- Upper case letters and lower case letters are considered distinct, so that `kit_of_parts` and `KIT_OF_PARTS` are two different identifiers.
- Although the period (`.`) and dollar sign (`$`) characters can be used to construct identifiers, they are reserved for special purposes (pseudo-ops for instance) and should not appear in user-defined identifiers.

Examples of Identifiers

```
Grab_Hold   Widget   Pot_of_Message   MAXNAME
```

2.3. Numeric Labels

A numeric label consists of a digit 0 to 9 followed by a colon. As in the case of name 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 ***n*** ***b***ackwards from the reference; ***n*f** symbols refer to the first numeric label ***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. 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 of 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 want to reference it. An ordinary label which tags a location in the program or data is visible only within the current assembly. An identifier which is designated as an external identifier via a `.globl` directive is visible to other assembly units at link time.

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

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

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

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

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

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

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

2.6. Constants

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

2.7. Numeric Constants

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

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

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

Octal constants must start with the digit 0. There can then be from one to 11 octal digits (0 through 7) in the number. But note that 11 octal digits is 33 bits, so the largest octal number is 037777777777. The assembler generates an error message if the decimal digits 8 and 9 appear in an octal constant.

2.8. String Constants

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

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

Character	*Octal Representation*
Backspace	010
Horizontal Tab	011
Newline (Line-Feed)	012
Form-Feed	014
Carriage-Return	015

2.9. Assembly Location Counter

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

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

At the beginning of each assembly pass, the assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of generated code. However, the location where the code is stored may be changed by a direct assignment altering the location counter:

```
. = <expression>
```

This *<expression>* must not contain any forward references, and must not change value from one pass to another. Storage may also be reserved be advancing **dot**. For example, if the current value of **dot** is 1000, the direct assignment statement:

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

reserves 256 bytes (100 hexadecimal) of storage, with the address of the first byte as the value of Table. The next instruction is stored at address `0x1100`. Also see the `.skip` assembler directive for another means of achieving the same effect.

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

```
. = 0x1000
```

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

Chapter 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, described below.

Unary Operators

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

Binary operators

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 is assumed to work on a 32-bit number. If the value of a particular term occupies only 8 bits or 16 bits, the short quantity is sign extended to a full 32-bit value.

3.2. Terms

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

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

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

3.3. Expressions

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

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

If the operand only requires a single 16-bit word value (a `.word` directive or an `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, where 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 an external relocatable expression. In addition, you cannot multiply or divide ***any*** relocatable expression.

Chapter 4

Layout of an Assembly Language Source Program

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

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

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

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

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

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

4.1. Label Field

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

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

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

```
sticky:                          |  there is a label defined here.
```

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

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

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

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

4.2. Operation Code Field

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

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

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

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

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

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

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

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

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

4.3. Operand Field

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

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

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

[<*operand*>[,<*operand*>]...]

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

These forms of operands are described in the subsections following.

4.3.1. Register Operands

Register operands in a machine instruction refer to the machine registers of the MC68010 processor.

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.

4.4. Comment Field

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

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

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

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

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

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

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

4.5. Direct Assignment Statements

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

<identifier> = *<expression>*

Examples of direct assignments are:

```
vect_size   =  4
vectora     =  OxFFFE
vectorb     =  vectora-vect_size
CRLF        =  OxODOA

dtemp       =  dO                  |  use register dO as a 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 an assembler error message.

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

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

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

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

Chapter 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 following pseudo-ops are available in ***as***:

Table 5-1: Assembler Directives

Pseudo Operation	*Description*
`.ascii`	Generates a sequence of ASCII characters.
`.asciz`	Generates a sequence of ASCII characters, terminated by a zero byte.
`.byte`	Generates a sequence of bytes in data storage.
`.word`	Generates a sequence of words in data storage.
`.long`	Generates a sequence of long words in data storage.
`.text`	Specifies that generated code be placed in the *text* control section until further notice.
`.data`	Specifies that generated code be placed in the *data* control section until further notice.
`.data1`	Specifies that generated code be placed in the *data1* control section until further notice.
`.data2`	Specifies that generated code be placed in the *data2* control section until further notice.
`.bss`	Specifies that space will be reserved in the *bss* control section until further notice.
`.globl`	Declares an identifier as global (external).
`.comm`	Declares the name and size of a *common* area.
`.lcomm`	reserves a specified amount of space in the *bss* area.
`.skip`	advances the location counter by a specified amount.
`.even`	forces location counter to next word (even byte) boundary.
`.stab`*x*	Builds special symbol table entries. These directives are for the benefit of compilers which generate information for the symbolic debugger *dbx*.

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

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

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

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

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

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

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

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

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

Octal Code Generated:	*Statement:*
110 145 154 154 157 040 127 157 162 144 041 000	`.asciz "Hello World!"`
124 150 105 040 107 162 145 141 164 040 120 122 117 115 160 153 151 156 040 163 164 162 151 153 145 163 040 141 147 141 151 156 041 000	`.asciz "The Great PROMpkin strikes again!"`

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

The `.byte`, `.word` and `.long` directives reserve bytes, words, and long words, and initialize them with specified values.

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

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

[<label>:]    .word    [<expression>][,<expression>]...

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

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

The `.word` directive reserves one word (16 bits) for each expression in the operand field, and initializes the word 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 the long word to the low-order 32 bits of the corresponding expression.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        .data               |  and now back to the 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 UNIX System Programmer's Reference Manual for the entry on ***a.out***(5).

5.5. .skip — Advance the Location Counter

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

```
.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:

```
.skip  1000
```

is equivalent to the direct assignment statement:

```
. = . + 1000
```

5.6. .lcomm — Reserve Space in .bss Area

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

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

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

A `.lcomm` directive like this:

```
.lcomm    lower_forty,1200
```

is equivalent to these directives:

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

5.7. .globl — Designate an External Identifier

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

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

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

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

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

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

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

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

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

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

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

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

5.9. .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 data values must lie on even byte boundaries, and also because machine instructions must start on even byte boundaries.

Chapter 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 processor.

6.1. Instruction Mnemonics

The instruction mnemonics which *as* uses are based on the mnemonics described in the Motorola MC68010 processor manual. *As* deviates from the Motorola manual in several areas.

Most of the MC68010 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, byte (or short) and word. 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, while `beqs` refers to the short version of that instruction.

6.2. Extended Branch Instruction Mnemonics

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

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 by a conditional branch (with the sense of the conditional inverted) around a `jmp` for the extended conditional branches.

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 op-codes.

6.3. Addressing Modes

The following table describes the addressing modes that *as* recognizes. The notations used in this table have these meanings:

`a`*n* refers to an address register,

`d`*n* refers to a data register,

`r`*i* refers to either a data register or an address register,

d refers to a displacement, which is a constant expression in *as*,

xxx refers to a constant expression.

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

`sp` the stack pointer which is equivalent to a7,

`sr` the status register,

`cc` the condition codes of the status register,

`usp` the user mode stack pointer,

`pc` the program counter,

`sfc` the source function code register,

`dfc` the destination function code register,

Note that register `a7` and the stack pointer (`sp`) are the same register. The only place where this is important is when the supervisor must explicitly use `usp` to refer to the user stack pointer.

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`
Postincrement	`an@+`	`movw a3@+,d2`
Predecrement	`an@-`	`movw a3@-,d2`
Displacement	`an@(d)`	`movw a3@(24),d2`
Word Index	`an@(d, Ri:W)`	`movw a3@(16, d2:W),d3`
Long Index	`an@(d, Ri:L)`	`movw a3@(16, d2:L),d3`
Absolute Short	`xxx:W`	`movw 14:W,d2`
Absolute Long	`xxx:L`	`movw 14:L,d2`
PC Displacement	`pc@(d)`	`movw pc@(20),d3`
PC Word Index	`pc@(d, Ri:W)`	`movw pc@(14, d2:W),d3`
PC Long Index	`pc@(d, Ri:L)`	`movw pc@(14, d2:L),d3`
Normal	*identifier*	`movw widget,d3`
Immediate	`#xxx`	`movw #27+3,d3`

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

The Motorola manual presents 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, the Motorola manual uses the mnemonic `adda` for *add address*. *as* does not make these distinctions because it can determine the type of the operand from the form of the operand. Thus an instruction of the form:

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

assembles to the *add address* instruction because *as* can determine that *avenue* is an address.

```
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 manual are missing from the set of mnemonics that *as* recognizes.

The MC68010 is restrictive in that certain classes of instructions accept only subsets of the address modes above. For example, the *add* instruction does not accept a PC-relative address as a destination.

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 below describes how the address modes are grouped into address categories.

6.4. Addressing Categories

The MC68010 groups 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***, and they fall into one or more of four addressing ***categories***. The addressing categories are defined here, followed by a table which summarizes the grouping of the addressing modes into the categories.

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

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

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

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

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

Table 6-2: Addressing Categories

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

Appendix A

Error Codes

A.1. Usage Errors

Unknown option 'x' ignored
As does not recognize the option *x*. Valid options are listed in section 1.1.

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

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

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

No input file
Exactly one input file must be specified — *as* cannot acept the output of a pipe as its input.

A.2. Assembler Error Messages

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

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

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

Invalid Character
An unexpected character was encountered in the program text.

Multiply defined symbol

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

Symbol storage exceeded

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

Out of strings space

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

Stab storage exceeded

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

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 Term

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

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 (/).

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).

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

```
.globl  3
```

because the constant 3 is not a symbol.

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

Invalid op-code
The assembler did not recognize an instruction mnemonic. Probably a misspelling.

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

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

Wrong number of operands
Check appendix B for the correct number of operands for the current instruction.

Line too long
A statement was found which has more than 4096 characters before the newline.

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

```
add  #d0,_there
```

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.

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

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

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

as $99 or *n*$ if *n*$ has not been defined.

Missing close-paren ')'
An unmatched '(' was found in an expression.

Appendix B

List of As Opcodes

This appendix is a list of the instruction mnemonics, grouped alphabetically.

Each entry describes the following things:

- The mnemonics for the instruction,
- The generic name for the instruction,
- The assembly language syntax and the variations on the instruction,
- The condition codes that this instruction affects.

The syntax for *as* machine instructions differs somewhat from the instruction layouts and categories shown in the Motorola MC68010 manual. For example, *as* provides a single set of mnemonics for `add` (add binary), `adda` (add address), and `addi` (add immediate). 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 `addx` in the assembly language syntax column means that the instruction is coded as `addb` or `addw` or `addl`, *etcetera*.
- 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 *ea* means an effective address designated by one of the permissible addressing modes for the MC68010. Consult the Motorola MC68010 manual for details of the allowed addressing modes for each instruction.
- 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 condition code register has the following flags, with the following meanings.
  - N Set if the most significant bit of the result is set. Cleared otherwise.
  - Z Set if the result is zero. Cleared otherwise.
  - V Set if there was an arithmetic overflow. Cleared otherwise.
  - C Set if a carry is generated (for addition) or a borrow is generated (for a subtraction) out of the most significant bit of the operand. Cleared otherwise.
  - X This condition code is transparent to data movement instructions. When it is affected it is set the same as the C (carry) condition.

- The notations under ***condition codes*** in the tables below have these meanings:
  - * set according to the result of the instruction.
  - — this instruction does not affect this condition code.
  - 0 this instruction clears this condition code.
  - 1 this instruction sets this condition code.
  - U this condition code is undefined after the instruction.
  - ? this condition code is set according to the status register pulled off the stack, or according to the immediate operand.

Mnemonics	*Operation*	*Assembly Language Syntax*		*Condition Codes* X	N	Z	V	C
abcd	add decimal with extend	abcd	dy,dx	*	U	*	U	*
		abcd	ay@-,ax@-					
addb addw addl	add binary	addx	ea,dn	*	*	*	*	*
		addx	dn,ea					
		addx	ea,an					
		addx	#data,ea					
addqb addqw addql	add quick	addqx	#data,ea	*	*	*	*	*
addxb addxw addxl	add extended	addxx	dy,dx	*	*	*	*	*
		addxx	ay@-,ax@-					
andb andw andl	logical and	andx	ea,dn	*	*	*	*	*
		andx	dn,ea					
		andx	#data,ea					
aslb aslw asll	arithmetic shift left	aslx	dx,dy	*	*	*	*	*
		aslx	#data,dy					
		aslx	ea					
asrb asrw asrl	arithmetic shift right	asrx	dx,dy	*	*	*	*	*
		asrx	#data,dy					
		asrx	ea					
bcc bccs	branch carry clear	bccx	ea	—	—	—	—	—
bchg	test a bit and change	bchg	dn,ea	—	—	*	—	—
		bchg	#data,ea					
bclr	test a bit and clear	bclr	dn,ea	—	—	*	—	—
		bclr	#data,ea					
bset	test a bit and set	bset	dn,ea	—	—	*	—	—
		bset	#data,ea					
		btst	dn,ea					

Mnemonics	*Operation*	*Assembly Language Syntax*		*Condition Codes* X	N	Z	V	C
		`btst`	*`#data,ea`*					
`bcs` `bcss`	branch carry set	`bcs`*`x`*	*`ea`*	–	–	–	–	–
`beq` `beqs`	branch on equal	`beq`*`x`*	*`ea`*	–	–	–	–	–
`bge` `bges`	branch greater or equal	`bge`*`x`*	*`ea`*	–	–	–	–	–
`bgt` `bgts`	branch greater than	`bgt`*`x`*	*`ea`*	–	–	–	–	–
`bhi` `bhis`	branch higher	`bhi`*`x`*	*`ea`*	–	–	–	–	–
`ble` `bles`	branch less than or equal	`ble`*`x`*	*`ea`*	–	–	–	–	–
`bls` `blss`	branch lower or same	`bls`*`x`*	*`ea`*	–	–	–	–	–
`blt` `blts`	branch less than	`blt`*`x`*	*`ea`*	–	–	–	–	–
`bmi` `bmis`	branch minus	`bmi`*`x`*	*`ea`*	–	–	–	–	–
`btst`	test a bit			–	–	*	–	–
`bne` `bnes`	branch not equal	`bne`*`x`*	*`ea`*	–	–	–	–	–
`bpl` `bpls`	branch positive	`bpl`*`x`*	*`ea`*	–	–	–	–	–
`bra` `bras`	branch always	`bra`*`x`*	*`ea`*	–	–	–	–	–
`bsr` `bsrs`	subroutine branch	`bsr`*`x`*	*`ea`*	–	–	–	–	–
`bvc` `bvcs`	branch overflow clear	`bvc`*`x`*	*`ea`*	–	–	–	–	–
`bvs` `bvss`	branch overflow set	`bvs`*`x`*	*`ea`*	–	–	–	–	–
`chk`	check register against bounds	`chk`	*`ea,dn`*	–	*	U	U	U

`clrb`
`clrw`
`clrl`
	clear an operand	`clr`*`x`*	*`ea`*	–	0	1	0	0

`cmpmb`
`cmpmw`
`cmpml`
	compare memory	`cmpm`*`x`*	*`ay@+,Ax@+`*	–	*	*	*	*

Mnemonics	*Operation*	*Assembly Language Syntax*		*Condition Codes* X	N	Z	V	C
cmpb cmpw cmpl	arithmetic compare	cmp*x* cmp*x*	*ea*, dn #*data*, *ea*	–	*	*	*	*
dbcc	decrement & branch on carry clear	dbcc	dn, *label*	–	–	–	–	–
dbcs	decrement & branch on carry set	dbcs	dn, *label*	–	–	–	–	–
dbeq	decrement & branch on equal	dbeq	dn, *label*	–	–	–	–	–
dbf	decrement & branch on false	dbf	dn, *label*	–	–	–	–	–
dbge	decrement & branch on greater than or equal	dbge	dn, *label*	–	–	–	–	–
dbgt	decrement & branch on greater than	dbgt	dn, *label*	–	–	–	–	–
dbhi	decrement & branch on high	dbhi	dn, *label*	–	–	–	–	–
dble	decrement & branch on less than or equal	dble	dn, *label*	–	–	–	–	–
dbls	decrement & branch on low or same	dbls	dn, *label*	–	–	–	–	–
dblt	decrement & branch on less than	dblt	dn, *label*	–	–	–	–	–
dbmi	decrement & branch on minus	dbmi	dn, *label*	–	–	–	–	–
dbne	decrement & branch on not equal	dbne	dn, *label*	–	–	–	–	–
dbpl	decrement & branch on plus	dbpl	dn, *label*	–	–	–	–	–
dbra	decrement & branch always (same as dbf)	dbra	dn, *label*	–	–	–	–	–
dbt	decrement & branch on True	dbt	dn, *label*	–	–	–	–	–
dbvc	decrement & branch on overflow clear	dbvc	dn, *label*	–	–	–	–	–
dbvs	decrement & branch on overflow set	dbvs	dn, *label*	–	–	–	–	–
divs	signed divide	divs	*ea*, dn	–	*	*	*	0
divu	unsigned divide	divs	*ea*, dn	–	*	*	*	0
eorb eorw eorl	logical exclusive or	eor*x* eor*x* eorb eorw	dn, *ea* #*data*, *ea* #*data*, *cc* #*data*, *sr*	–	*	*	0	0

Mnemonics	*Operation*	*Assembly Language Syntax*		*Condition Codes* X	N	Z	V	C
exg	exchange registers	exg	rx,ry	–	–	–	–	–
extw extl	sign extend	ext	dn	–	*	*	0	0
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,an	–	–	–	–	–
link	link and allocate	link	an,#disp	–	–	–	–	–
lslb lslw lsll	logical shift left	lslx lslx lslx	dx,dy #data,dy ea	*	*	*	0	*
lsrb lsrw lsrl	logical shift right	lsrx lsrx lsrx	dx,dy #data,dy ea	*	*	*	0	*
movb movw movl	move data	movx movx	ea,ea #data,dn	–	*	*	0	0
movw	move from condition code register	movw	cc,ea	–	–	–	–	–
movw	move from status register	movw	sr,ea	–	–	–	–	–
movc	move to control register	movc	rn,cr	–	–	–	–	–
movc	move from control register	movc	cr,rn	–	–	–	–	–
moveml movemw	move multiple registers	movemx movemx	#mask,ea ea,#mask	–	–	–	–	–
movepl movepw	move peripheral	movepx movepx	dn,an@(d) dn,an@(d)	–	–	–	–	–

Mnemonics	*Operation*	*Assembly Language Syntax*		*Condition Codes* X	N	Z	V	C
moveq	move quick	moveq	#*data*,dn	–	*	*	0	0
movsb movsw movsl	move to address space move from address space	movs*x* movs*x*	rn,*ea* *ea*,rn	–	–	–	–	–
muls mulu	signed multiply unsigned multiply	muls mulu	*ea*,dn *ea*,dn	0 0	0 0	* *	* *	0 0
nbcd	negate decimal with extend	nbcd	*ea*	*	U	*	U	*
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*	–	*	*	0	0
orb orw orl	inclusive or	or*x* or*x* or orb orw	*ea*,dn dn,*ea* #*data*,*ea* #*data*,*cc* #*data*,*sr*	–	*	*	0	0
pea	push effective address	pea	*ea*	–	–	–	–	–
reset	reset machine	reset		–	–	–	–	–
rolb rolw roll	rotate left	rol*x* rol*x* rol*x*	d*x*,d*y* #*data*,dy *ea*	0	*	*	0	*
rorb rorw rorl	rotate right	ror*x* ror*x* ror*x*	d*x*,d*y* #*data*,dy *ea*	0	*	*	0	*
roxlb roxlw roxll	rotate left with extend	roxl*x* roxl*x* roxl*x*	d*x*,d*y* #*data*,dy *ea*	*	*	*	0	*
roxrb roxrw roxrl	rotate right with extend	roxr*x* roxr*x* roxr*x*	d*x*,d*y* #*data*,dy *ea*	*	*	*	0	*
rte rtr rts rts	return from exception return and restore codes return from subroutine return from subroutine	rte rtr rts rts	 #*n*	? ? – –	? ? – –	? ? – –	? ? – –	? ? – –

Mnemonics	*Operation*	*Assembly Language Syntax*		*Condition Codes* X	N	Z	V	C
sbcd	subtract decimal with extend	sbcd	dy,dx	*	U	*	U	*
		sbcd	ay@-,ax@-					
stop	halt machine	stop	#xxx	?	?	?	?	?
subb	arithmetic subtract	subx	ea,dn	*	*	*	*	*
subw		subx	dn,ea					
		subx	ea,an					
subl		subx	#data,ea					
st	set all ones	st	ea	—	—	—	—	—
sf	set all zeros	sf	ea	—	—	—	—	—
shi	set high	shi	ea	—	—	—	—	—
sls	set lower or same	sls	ea	—	—	—	—	—
scc	set carry clear	scc	ea	—	—	—	—	—
scs	set carry set	scs	ea	—	—	—	—	—
sne	set not equal	sne	ea	—	—	—	—	—
seq	set equal	seq	ea	—	—	—	—	—
svc	set no overflow	svc	ea	—	—	—	—	—
svs	set on overflow	svs	ea	—	—	—	—	—
spl	set plus	spl	ea	—	—	—	—	—
smi	set minus	smi	ea	—	—	—	—	—
sge	set greater or equal	sge	ea	—	—	—	—	—
slt	set less than	slt	ea	—	—	—	—	—
sgt	set greater than	sgt	ea	—	—	—	—	—
sle	set less than or equal	sle	ea	—	—	—	—	—
subqb	subtract quick	subqx	#data,ea	*	*	*	*	*
subqw								
subql								
subxb	subtract extended	subxx	dy,dx	*	*	*	*	*
subxw		subxx	ay@-,ax@-					
subxl								
swap	swap register halves	swap	dn	*	*	*	*	*
tas	test operand then set	tas	ea	—	*	*	0	0
trap	trap	trap	#vector	—	—	—	—	—
trapv	trap on overflow	trapv		—	—	—	—	—
tstb	test operand	tstx	ea	—	*	*	0	0
tstw								
tstl								
unlk	unlink	unlk	an	—	—	—	—	—

Index

A

absolute expressions, 3-2 *thru* 3-3
adressing categories, 6-4 *thru* 6-5
    alterable, 6-4
    control, 6-4
    data, 6-4
    memory, 6-4
adressing modes, 6-2 *thru* 6-4
`.ascii` directive, 5-2
`.asciz` directive, 5-3
assembler directives, 5-1 *thru* 5-6
    `.ascii`, 5-2
    `.asciz`, 5-3
    `.bss`, 5-4
    `.byte`, 5-3
    `.comm`, 5-6
    `.data`, 5-4
    `.even`, 5-6
    `.globl`, 5-6
    `.lcomm`, 5-5
    `.long`, 5-3
    `.skip`, 5-5
    `.text`, 5-4
    `.word`, 5-3
assembler options, 1-1 *thru* 1-2
    –d2, 1-1
    –h, 1-1
    –j, 1-1, 1-1
    –L, 1-1
    –o, 1-1
    –R, 1-1
assignment statements, 4-4

B

basic elements, 2-1 *thru* 2-5
`.bss` directive, 5-4
`.byte` directive, 5-3

C

character set, 2-1
`.comm` directive, 5-6
comment field, 4-3 *thru* 4-4
constants, 2-3 *thru* 2-4
    decimal, 2-3
    hexadecimal, 2-3
    numeric, 2-3
    octal, 2-3
constants, *continued*
    string, 2-4

D

–d2 option, 1-1
`.data` directive, 5-4
decimal constants, 2-3
direct assignment, 4-4
directives, 5-1 *thru* 5-6
    `.ascii`, 5-2
    `.asciz`, 5-3
    `.bss`, 5-4
    `.byte`, 5-3
    `.comm`, 5-6
    `.data`, 5-4
    `.even`, 5-6
    `.globl`, 5-6
    `.lcomm`, 5-5
    `.long`, 5-3
    `.skip`, 5-5
    `.text`, 5-4
    `.word`, 5-3

E

`.even` directive, 5-6
expressions, 3-1 *thru* 3-3
    absolute, 3-2 *thru* 3-3
    external, 3-2 *thru* 3-3
    operators, 3-1
    relocatable, 3-2 *thru* 3-3
    terms, 3-2
external expressions, 3-2 *thru* 3-3

G

`.globl` directive, 5-6

H

–h option, 1-1
hexadecimal constants, 2-3

I

identifiers, 2-1 *thru* 2-2

J

–j option, 1-1, 1-1

L

–L option, 1-1
label field, 4-1 *thru* 4-2
labels, 2-2 *thru* 2-3
    local, 2-2
    numeric, 2-2
    scope, 2-2
`.lcomm` directive, 5-5
lexical elements, 2-1 *thru* 2-5
lines, 4-1
local labels, 2-2
location counter, 2-4
`.long` directive, 5-3

N

notation, 1-2
numeric constants, 2-3
numeric labels, 2-2

O

–o option, 1-1
octal constants, 2-3
operand field, 4-2 *thru* 4-3
operation code field, 4-2
options, 1-1 *thru* 1-2
    –d2, 1-1
    –h, 1-1
    –j, 1-1, 1-1
    –L, 1-1
    –o, 1-1
    –R, 1-1

P

program layout, 4-1 *thru* 4-4
pseudo-ops, 5-1 *thru* 5-6
    `.ascii`, 5-2
    `.asciz`, 5-3
    `.bss`, 5-4
    `.byte`, 5-3
    `.comm`, 5-6
    `.data`, 5-4
    `.even`, 5-6
    `.globl`, 5-6
    `.lcomm`, 5-5
    `.long`, 5-3
    `.skip`, 5-5
    `.text`, 5-4
    `.word`, 5-3

R

–R option, 1-1
register operands, 4-3
    address registers, 6-2
    data registers, 6-2
    special registers, 6-2
relocatable expressions, 3-2 *thru* 3-3

S

scope of labels, 2-2
`.skip` directive, 5-5
special register operands
    `cc`, 6-2
    `dfc`, 6-2
    `pc`, 6-2
    `sfc`, 6-2
    `sp`, 6-2
    `sr`, 6-2
    `usp`, 6-2
statements, 4-1
    comment field, 4-3 *thru* 4-4
    direct assignment, 4-4
    label field, 4-1 *thru* 4-2
    operand field, 4-2 *thru* 4-3
    operation code field, 4-2
string constants, 2-4

T

`.text` directive, 5-4

W

`.word` directive, 5-3