as

COLLABORATORS			
	TITLE : as		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 28, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

as

1.1 as.guide

```
                Using as
********

   This file is a user guide to the GNU assembler as.


                 Overview
                                    Overview

                 Invoking
                                    Command-Line Options

                 Syntax
                                      Syntax

                 Sections
                                    Sections and Relocation

                 Symbols
                                     Symbols

                 Expressions
                                  Expressions

                 Pseudo Ops
                                   Assembler Directives

                 Machine Dependencies
                         Machine Dependent Features


                 Copying
                                     GNU GENERAL PUBLIC LICENSE


                 Acknowledgements
                             Who Did What
```

Index

Index

1.2 as.guide/Overview

```
                Overview
********

   Here is a brief summary of how to invoke as.  For details, see

                Comand-Line Options
                .

       as [ -a[dhlns] ] [ -D ] [ -f ]
        [ -I path ] [ -K ] [ -L ]
        [ -o objfile ] [ -R ] [ -v ] [ -w ]
        [ -Av6 | -Av7 | -Av8 | -Asparclite | -bump ]
        [ -ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC ]
        [ -b ] [ -norelax ]
        [ -l ] [ -m68000 | -m68010 | -m68020 | ... ]
        [ -- | files ... ]

-a[dhlns]
     Turn on listings; -ad, omit debugging pseudo-ops from listing,
     -ah, include high-level source, -al, assembly listing, -an, no
     forms processing, -as, symbols.  These options may be combined;
     e.g., -aln for assembly listing without forms processing.  By
     itself, -a defaults to -ahls -- that is, all listings turned on.

-D
     This option is accepted only for script compatibility with calls to
     other assemblers; it has no effect on as.

-f
     "fast"--skip whitespace and comment preprocessing (assume source is
     compiler output)

-I path
     Add path to the search list for .include directives

-K
     Issue warnings when difference tables altered for long
     displacements.

-L
     Keep (in symbol table) local symbols, starting with L

-o objfile
     Name the object-file output from as

-R
```

```
     Fold data section into text section

-v
     Announce as version

-W
     Suppress warning messages

- | files ...
     Standard input, or source files to assemble.

   The following options are available when as is configured for the
Intel 80960 processor.

-ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC
     Specify which variant of the 960 architecture is the target.

-b
     Add code to collect statistics about branches taken.

-norelax
     Do not alter compare-and-branch instructions for long
     displacements; error if necessary.

   The following options are available when as is configured for the
Motorola 68000 series.

-l
     Shorten references to undefined symbols, to one word instead of
     two.

-m68000 | -m68008 | -m68010 | -m68020 | -m68030 | -m68040
| -m68302 | -m68331 | -m68332 | -m68333 | -m68340 | -mcpu32
     Specify what processor in the 68000 family is the target.  The
     default is normally the 68020, but this can be changed at
     configuration time.

-m68881 | -m68882 | -mno-68881 | -mno-68882
     The target machine does (or does not) have a floating-point
     coprocessor.  The default is to assume a coprocessor for 68020,
     68030, and cpu32.  Although the basic 68000 is not compatible with
     the 68881, a combination of the two can be specified, since it's
     possible to do emulation of the coprocessor instructions with the
     main processor.

-m68851 | -mno-68851
     The target machine does (or does not) have a memory-management
     unit coprocessor.  The default is to assume an MMU for 68020 and
     up.

   The following options are available when as is configured for the
SPARC architecture:

-Av6 | -Av7 | -Av8 | -Asparclite
     Explicitly select a variant of the SPARC architecture.

-bump
```

Warn when the assembler switches to another architecture.

1.3 as.guide/Manual

Structure of this Manual
========================

This manual is intended to describe what you need to know to use GNU as. We cover the syntax expected in source files, including notation for symbols, constants, and expressions; the directives that as understands; and of course how to invoke as.

This manual also describes some of the machine-dependent features of various flavors of the assembler.

On the other hand, this manual is not intended as an introduction to programming in assembly language--let alone programming in general! In a similar vein, we make no attempt to introduce the machine architecture; we do not describe the instruction set, standard mnemonics, registers or addressing modes that are standard to a particular architecture. You may want to consult the manufacturer's machine architecture manual for this information.

1.4 as.guide/GNU Assembler

```
                as, the GNU Assembler
=====================

   GNU as is really a family of assemblers.  If you use (or have used)
the GNU assembler on one architecture, you should find a fairly similar
environment when you use it on another architecture.  Each version has
much in common with the others, including object file formats, most
assembler directives (often called pseudo-ops) and assembler syntax.

   as is primarily intended to assemble the output of the GNU C
compiler gcc for use by the linker ld.  Nevertheless, we've tried to
make as assemble correctly everything that other assemblers for the same
machine would assemble.  Any exceptions are documented explicitly (see

                Machine Dependencies
                ).  This doesn't mean as always uses the same
syntax as another assembler for the same architecture; for example, we
know of several incompatible versions of 680x0 assembly language syntax.

   Unlike older assemblers, as is designed to assemble a source program
in one pass of the source file.  This has a subtle impact on the .org
directive (see
                .org
                ).
```

1.5 as.guide/Object Formats

```
                Object File Formats
===================

   The GNU assembler can be configured to produce several alternative
object file formats.  For the most part, this does not affect how you
write assembly language programs; but directives for debugging symbols
are typically different in different file formats.  See

                Symbol Attributes
                .
```

1.6 as.guide/Command Line

```
Command Line
============

   After the program name as, the command line may contain options and
file names.  Options may appear in any order, and may be before, after,
or between file names.  The order of file names is significant.
```

- (two hyphens) by itself names the standard input file explicitly, as one of the files for as to assemble.

Except for - any command line argument that begins with a hyphen (-) is an option. Each option changes the behavior of as. No option changes the way another option works. An option is a - followed by one or more letters; the case of the letter is important. All options are optional.

Some options expect exactly one file name to follow them. The file name may either immediately follow the option's letter (compatible with older assemblers) or it may be the next command argument (GNU standard). These two command lines are equivalent:

```
as -o my-object-file.o mumble.s
as -omy-object-file.o mumble.s
```

1.7 as.guide/Input Files

```
                Input Files
===========
```

We use the phrase source program, abbreviated source, to describe the program input to one run of as. The program may be in one or more files; how the source is partitioned into files doesn't change the meaning of the source.

The source program is a concatenation of the text in all the files, in the order specified.

Each time you run as it assembles exactly one source program. The source program is made up of one or more files. (The standard input is also a file.)

You give as a command line that has zero or more input file names. The input files are read (from left file name to right). A command line argument (in any position) that has no special meaning is taken to be an input file name.

If you give as no file names it attempts to read one input file from the as standard input, which is normally your terminal. You may have to type ctl-D to tell as there is no more program to assemble.

Use - if you need to explicitly name the standard input file in your command line.

If the source is empty, as will produce a small, empty object file.

```
Filenames and Line-numbers
--------------------------
```

There are two ways of locating a line in the input file (or files) and either may be used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a

```
"logical" file.  See
                Error and Warning Messages
                .

   Physical files are those files named in the command line given to as.

   Logical files are simply names declared explicitly by assembler
directives; they bear no relation to physical files.  Logical file names
help error messages reflect the original source file, when as source is
itself synthesized from other files.  See
                .app-file
                .
```

1.8 as.guide/Object

```
Output (Object) File
====================

   Every time you run as it produces an output file, which is your
assembly language program translated into numbers.  This file is the
object file, named b.out, if as is configured for the Intel 80960, or
unless you tell as to give it another name by using the -o option.
Conventionally, object file names end with .o.  The default name of
a.out is used for historical reasons:  older assemblers were capable of
assembling self-contained programs directly into a runnable program.
(For some formats, this isn't currently possible, but it can be done for
a.out format.)

   The object file is meant for input to the linker ld.  It contains
assembled program code, information to help ld integrate the assembled
program into a runnable file, and (optionally) symbolic information for
the debugger.
```

1.9 as.guide/Errors

```
                Error and Warning Messages
==========================

   as may write warnings and error messages to the standard error file
(usually your terminal).  This should not happen when  a compiler runs
as automatically.  Warnings report an assumption made so that as could
keep assembling a flawed program; errors report a grave problem that
stops the assembly.

   Warning messages have the format

     file_name:NNN:Warning Message Text

(where NNN is a line number).  If a logical file name has been given
```

(see
.app-file
) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (see

.line
) then it is used to calculate the number printed, otherwise the actual line in the current source file is printed. The message text is intended to be self explanatory (in the grand Unix tradition).

Error messages have the format
file_name:NNN:FATAL:Error Message Text
The file name and line number are derived as for warning messages. The actual message text may be rather less explanatory because many of them aren't supposed to happen.

1.10 as.guide/Invoking

Command-Line Options
****

This chapter describes command-line options available in all versions of the GNU assembler; see
Machine Dependencies
, for options specific to particular machine architectures.

If you are invoking as via the GNU C compiler (version 2), you can use the -Wa option to pass arguments through to the assembler. The assembler arguments must be separated from each other (and the -Wa) by commas. For example:

```
gcc -c -g -O -Wa,-alh,-L file.c
```

will cause a listing to be emitted to standard output with high-level and assembly source.

Many compiler command-line options, such as -R and many machine-specific options, will be automatically be passed to the assembler by the compiler, so usually you do not need to use this -Wa mechanism.

a
-a[dhlns] enable listings

D
-D for compatibility

f
-f to work faster

```
I
            -I for .include search path

K
            -K for difference tables

L
            -L to retain local labels

o
            -o to name the object file

R
            -R to join data and text sections

v
            -v to announce version

W
            -W to suppress warnings
```

1.11 as.guide/a

```
Enable Listings: -a[dhlns]
==========================

   These options enable listing output from the assembler.  By itself,
-a requests high-level, assembly, and symbols listing.  Other letters
may be used to select specific options for the list: -ah requests a
high-level language listing, -al requests an output-program assembly
listing, and -as requests a symbol table listing.  High-level listings
require that a compiler debugging option like -g be used, and that
assembly listings (-al) be requested also.

   The -ad option may be used to omit debugging pseudo-ops from the
listing.

   Once you have specified one of these options, you can further control
listing output and its appearance using the directives .list, .nolist,
.psize, .eject, .title, and .sbttl.  The -an option turns off all forms
processing.  If you do not request listing output with one of the -a
options, the listing-control directives have no effect.

   The letters after -a may be combined into one option, e.g., -aln.
```

1.12 as.guide/D

```
-D
==

   This option has no effect whatsoever, but it is accepted to make it
more likely that scripts written for other assemblers will also work
with as.
```

1.13 as.guide/f

```
                Work Faster: -f
===============

   -f should only be used when assembling programs written by a
(trusted) compiler.  -f stops the assembler from doing whitespace and
comment pre-processing on the input file(s) before assembling them.
See
                Pre-processing
                .

     Warning: if the files actually need to be pre-processed (if they
     contain comments, for example), as will not work correctly if -f
     is used.
```

1.14 as.guide/I

```
                .include search path: -I path
============================

   Use this option to add a path to the list of directories as will
search for files specified in .include directives (see
                .include
                ).  You
may use -I as many times as necessary to include a variety of paths.
The current working directory is always searched first; after that, as
searches any -I directories in the same order as they were specified
(left to right) on the command line.
```

1.15 as.guide/K

```
                Difference Tables: -K
====================

   as sometimes alters the code emitted for directives of the form
.word sym1-sym2; see
```

.word
. You can use the -K option if you want a
warning issued when this is done.

1.16 as.guide/L

Include Local Labels: -L
==========

Labels beginning with L (upper case only) are called local labels.
See
Symbol Names
. Normally you don't see such labels when debugging,
because they are intended for the use of programs (like compilers) that
compose assembler programs, not for your notice. Normally both as and
ld discard such labels, so you don't normally debug with them.

This option tells as to retain those L... symbols in the object
file. Usually if you do this you also tell the linker ld to preserve
symbols whose names begin with L.

1.17 as.guide/o

Name the Object File: -o
==========

There is always one object file output when you run as. By default
it has the name a.out (or b.out, for Intel 960 targets only). You use
this option (which takes exactly one filename) to give the object file
a different name.

Whatever the object file is called, as will overwrite any existing
file of the same name.

1.18 as.guide/R

Join Data and Text Sections: -R
==========

-R tells as to write the object file as if all data-section data
lives in the text section. This is only done at the very last moment:
your binary data are the same, but data section parts are relocated
differently. The data section part of your object file is zero bytes
long because all its bytes are appended to the text section. (See

```
                Sections and Relocation
                .)
```

When you specify -R it would be possible to generate shorter address displacements (because we don't have to cross between text and data section). We refrain from doing this simply for compatibility with older versions of as. In future, -R may work this way.

When as is configured for COFF output, this option is only useful if you use sections named .text and .data.

1.19 as.guide/v

```
Announce Version: -v
====================
```

You can find out what version of as is running by including the option -v (which you can also spell as -version) on the command line.

1.20 as.guide/W

```
Suppress Warnings: -W
=====================
```

as should never give a warning or error message when assembling compiler output. But programs written by people often cause as to give a warning that a particular assumption was made. All such warnings are directed to the standard error file. If you use this option, no warnings are issued. This option only affects the warning messages: it does not change any particular of how as assembles your file. Errors, which stop the assembly, are still reported.

1.21 as.guide/Syntax

```
                Syntax
******
```

This chapter describes the machine-independent syntax allowed in a source file. as syntax is similar to what many other assemblers use; it is inspired by the BSD 4.2 assembler, except that as does not assemble Vax bit-fields.

```
 Pre-processing
              Pre-processing
```

1.22 as.guide/Pre-processing

==========

The as internal pre-processor:
* adjusts and removes extra whitespace. It leaves one space or tab before the keywords on a line, and turns any other whitespace on the line into a single space.

* removes all comments, replacing them with a single space, or an appropriate number of newlines.

* converts character constants into the appropriate numeric values.

Note that it does not do macro processing, include file handling, or anything else you may get from your C compiler's pre-processor. You can do include file processing with the .include directive (see
.include
).
Other "CPP" style pre-processing can be done with the GNU C compiler, by giving the input file a .S suffix; see the compiler documentation for details.

Excess whitespace, comments, and character constants cannot be used in the portions of the input text that are not pre-processed.

If the first line of an input file is #NO_APP or the -f option is given, the input file will not be pre-processed. Within such an input file, parts of the file can be pre-processed by putting a line that says #APP before the text that should be pre-processed, and putting a line that says #NO_APP after them. This feature is mainly intend to support asm statements in compilers whose output normally does not need to be pre-processed.

1.23 as.guide/Whitespace

```
                Whitespace
==========

   Whitespace is one or more blanks or tabs, in any order.  Whitespace
is used to separate symbols, and to make programs neater for people to
read.  Unless within character constants (see
                Character Constants
                ), any
whitespace means the same as exactly one space.
```

1.24 as.guide/Comments

```
                Comments
========

   There are two ways of rendering comments to as.  In both cases the
comment is equivalent to one space.

   Anything from /* through the next */ is a comment.  This means you
may not nest these comments.

     /*
       The only way to include a newline ('\n') in a comment
       is to use this sort of comment.
     */

     /* This sort of comment does not nest. */

   Anything from the line comment character to the next newline is
considered a comment and is ignored.  The line comment character is #
on the Vax; # on the i960; ! on the SPARC; | on the 680x0; ; for the
AMD 29K family; ; for the H8/300 family; ! for the H8/500 family; ! for
the Hitachi SH; ! for the Z8000; see
                Machine Dependencies
                .

   On some machines there are two different line comment characters.
One will only begin a comment if it is the first non-whitespace
character on a line, while the other will always begin a comment.

   To be compatible with past assemblers, a special interpretation is
given to lines that begin with #.  Following the # an absolute
expression (see
                Expressions
                ) is expected:  this will be the logical
line number of the next line.  Then a string (See
                Strings
                .) is allowed:
if present it is a new logical file name.  The rest of the line, if any, should be
whitespace.
```

```
   If the first non-whitespace characters on the line are not numeric,
the line is ignored.  (Just like a comment.)
                               # This is an ordinary comment.
     # 42-6 "new_file_name"    # New logical file name
                               # This is logical line # 36.
   This feature is deprecated, and may disappear from future versions
of as.
```

1.25 as.guide/Symbol Intro

```
                Symbols
=======

   A symbol is one or more characters chosen from the set of all
letters (both upper and lower case), digits and the three characters
_.$.  On most machines, you can also use $ in symbol names;
exceptions are noted in
                Machine Dependencies
                .  No symbol may begin with
a digit.  Case is significant.  There is no length limit: all
characters are significant.  Symbols are delimited by characters not in
that set, or by the beginning of a file (since the source program must
end with a newline, the end of a file is not a possible symbol
delimiter).  See
                Symbols
                .
```

1.26 as.guide/Statements

```
                Statements
==========

   A statement ends at a newline character (\n) or line separator
character.  (The line separator is usually ;, unless this conflicts
with the comment character; see
                Machine Dependencies
                .)  The newline or
separator character is considered part of the preceding statement.
Newlines and separators within character constants are an exception:
they don't end statements.

   It is an error to end any statement with end-of-file:  the last
character of any input file should be a newline.

   You may write a statement on more than one line if you put a
backslash (\ ) immediately in front of any newlines within the
statement.  When as reads a backslashed newline both characters are
```

ignored. You can even put backslashed newlines in the middle of symbol names without changing the meaning of your source program.

An empty statement is allowed, and may include whitespace. It is ignored.

A statement begins with zero or more labels, optionally followed by a key symbol which determines what kind of statement it is. The key symbol determines the syntax of the rest of the statement. If the symbol begins with a dot . then the statement is an assembler directive: typically valid for any computer. If the symbol begins with a letter the statement is an assembly language instruction: it will assemble into a machine language instruction. Different versions of as for different computers will recognize different instructions. In fact, the same symbol may represent a different instruction in a different computer's assembly language.

A label is a symbol immediately followed by a colon (:). Whitespace before a label or after a colon is permitted, but you may not have whitespace between a label's symbol and its colon. See
Labels
.

```
label:     .directive    followed by something
another_label:           # This is an empty statement.
           instruction   operand_1, operand_2, ...
```

1.27 as.guide/Constants

Constants
=========

A constant is a number, written so that its value is known by inspection, without knowing any context. Like this:

```
.byte  74, 0112, 092, 0x4A, 0X4a, 'J, '\J # All the same value.
.ascii "Ring the bell\7"                  # A string constant.
.octa  0x123456789abcdef0123456789ABCDEF0 # A bignum.
.float 0f-314159265358979323846264338327\
95028841971.693993751E-40                 # - pi, a flonum.
```

Characters
Character Constants

Numbers
Number Constants

1.28 as.guide/Characters

```
                Character Constants
-------------------
```

There are two kinds of character constants. A character stands for one character in one byte and its value may be used in numeric expressions. String constants (properly called string literals) are potentially many bytes and their values may not be used in arithmetic expressions.

Strings

Strings

Chars

Characters

1.29 as.guide/Strings

```
Strings
.......
```

A string is written between double-quotes. It may contain double-quotes or null characters. The way to get special characters into a string is to escape these characters: precede them with a backslash \ character. For example \ represents one backslash: the first \ is an escape which tells as to interpret the second character literally as a backslash (which prevents as from recognizing the second \ as an escape character). The complete list of escapes follows.

\b
: Mnemonic for backspace; for ASCII this is octal code 010.

\f
: Mnemonic for FormFeed; for ASCII this is octal code 014.

\n
: Mnemonic for newline; for ASCII this is octal code 012.

\r
: Mnemonic for carriage-Return; for ASCII this is octal code 015.

\t
: Mnemonic for horizontal Tab; for ASCII this is octal code 011.

\ digit digit digit
: An octal character code. The numeric code is 3 octal digits. For compatibility with other Unix systems, 8 and 9 are accepted as digits: for example, \008 has the value 010, and \009 the value 011.

```
\
     Represents one \  character.

\"
     Represents one " character.  Needed in strings to represent this
     character, because an unescaped " would end the string.

\ anything-else
     Any other character when escaped by \  will give a warning, but
     assemble as if the \  was not present.  The idea is that if you
     used an escape sequence you clearly didn't want the literal
     interpretation of the following character.  However as has no
     other interpretation, so as knows it is giving you the wrong code
     and warns you of the fact.
```

Which characters are escapable, and what those escapes represent, varies widely among assemblers. The current set is what we think the BSD 4.2 assembler recognizes, and is a subset of what most C compilers recognize. If you are in doubt, don't use an escape sequence.

1.30 as.guide/Chars

```
Characters
..........
```

A single character may be written as a single quote immediately followed by that character. The same escapes apply to characters as to strings. So if you want to write the character backslash, you must write '\ where the first \ escapes the second \ . As you can see, the quote is an acute accent, not a grave accent. A newline immediately following an acute accent is taken as a literal character and does not count as the end of a statement. The value of a character constant in a numeric expression is the machine's byte-wide code for that character. as assumes your character code is ASCII: 'A means 65, 'B means 66, and so on.

1.31 as.guide/Numbers

```
                Number Constants
----------------
```

as distinguishes three kinds of numbers according to how they are stored in the target machine. Integers are numbers that would fit into an int in the C language. Bignums are integers, but they are stored in more than 32 bits. Flonums are floating point numbers, described below.

```
                 Integers
```

1.32 as.guide/Integers

Integers
.....

A binary integer is 0b or 0B followed by zero or more of the binary digits 01.

An octal integer is 0 followed by zero or more of the octal digits (01234567).

A decimal integer starts with a non-zero digit followed by zero or more digits (0123456789).

A hexadecimal integer is 0x or 0X followed by one or more hexadecimal digits chosen from 0123456789abcdefABCDEF.

Integers have the usual values. To denote a negative integer, use the prefix operator - discussed under expressions (see

Prefix Operators
).

1.33 as.guide/Bignums

Bignums
.....

A bignum has the same syntax and semantics as an integer except that the number (or its negative) takes more than 32 bits to represent in binary. The distinction is made because in some places integers are permitted while bignums are not.

1.34 as.guide/Flonums

Flonums
....

A flonum represents a floating point number. The translation is indirect: a decimal floating point number from the text is converted by as to a generic binary floating point number of more than sufficient precision. This generic floating point number is converted to a particular computer's floating point format (or formats) by a portion of as specialized to that computer.

A flonum is written by writing (in order)
* The digit 0.

* A letter, to tell as the rest of the number is a flonum. e is recommended. Case is not important.

  On the H8/300, H8/500, Hitachi SH, and AMD 29K architectures, the letter must be one of the letters DFPRSX (in upper or lower case).

  On the Intel 960 architecture, the letter must be one of the letters DFT (in upper or lower case).

* An optional sign: either + or -.

* An optional integer part: zero or more decimal digits.

* An optional fractional part: . followed by zero or more decimal digits.

* An optional exponent, consisting of:

    * An E or e.

    * Optional sign: either + or -.

    * One or more decimal digits.

At least one of the integer part or the fractional part must be present. The floating point number has the usual base-10 value.

as does all processing using integers. Flonums are computed independently of any floating point hardware in the computer running as.

1.35 as.guide/Sections

Sections and Relocation
****

Secs Background
Background

1.36 as.guide/Secs Background

Background
==========

Roughly, a section is a range of addresses, with no gaps; all data "in" those addresses is treated the same for some particular purpose. For example there may be a "read only" section.

The linker ld reads many object files (partial programs) and combines their contents to form a runnable program. When as emits an object file, the partial program is assumed to start at address 0. ld will assign the final addresses the partial program occupies, so that different partial programs don't overlap. This is actually an over-simplification, but it will suffice to explain how as uses sections.

ld moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a section. Assigning run-time addresses to sections is called relocation. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses. For the H8/300 and H8/500, and for the Hitachi SH, as pads sections if needed to ensure they end on a word (sixteen bit) boundary.

An object file written by as has at least three sections, any of which may be empty. These are named text, data and bss sections.

When it generates COFF output, as can also generate whatever other named sections you specify using the .section directive (see
.section
).
If you don't use any directives that place output in the .text or .data sections, these sections will still exist, but will be empty.

Within the object file, the text section starts at address 0, the data section follows, and the bss section follows the data section.

To let ld know which data will change when the sections are

relocated, and how to change that data, as also writes to the object file details of the relocation needed. To perform relocation ld must know, each time an address in the object file is mentioned:

* Where in the object file is the beginning of this reference to an address?

* How long (in bytes) is this reference?

* Which section does the address refer to? What is the numeric value of
  (address) - (start-address of section)?

* Is the reference to an address "Program-Counter relative"?

In fact, every address as ever uses is expressed as
(section) + (offset into section)

Further, every expression as computes is of this section-relative nature. Absolute expression means an expression with section "absolute" (see
Ld Sections
). A pass1 expression means an expression with section "pass1" (see
as Internal Sections
). In this manual we use the notation {secname N } to mean "offset N into section secname".

Apart from text, data and bss sections you need to know about the absolute section. When ld mixes partial programs, addresses in the absolute section remain unchanged. For example, address {absolute 0} is "relocated" to run-time address 0 by ld. Although two partial programs' data sections will not overlap addresses after linking, by definition their absolute sections will overlap. Address {absolute 239} in one partial program will always be the same address when the program is running as address {absolute 239} in any other partial program.

The idea of sections is extended to the undefined section. Any address whose section is unknown at assembly time is by definition rendered {undefined U }--where U will be filled in later. Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has section undefined.

By analogy the word section is used to describe groups of sections in the linked program. ld puts all partial programs' text sections in contiguous addresses in the linked program. It is customary to refer to the text section of a program, meaning all the addresses of all partial program's text sections. Likewise for data and bss sections.

Some sections are manipulated by ld; others are invented for use of as and have no meaning except during assembly.

1.37 as.guide/Ld Sections

```
ld Sections
===========

   ld deals with just four kinds of sections, summarized below.

named sections
text section
data section
     These sections hold your program.  as and ld treat them as
     separate but equal sections.  Anything you can say of one section
     is true another.  When the program is running, however, it is
     customary for the text section to be unalterable.  The text
     section is often shared among processes: it will contain
     instructions, constants and the like.  The data section of a
     running program is usually alterable: for example, C variables
     would be stored in the data section.

bss section
     This section contains zeroed bytes when your program begins
     running.  It is used to hold unitialized variables or common
     storage.  The length of each partial program's bss section is
     important, but because it starts out containing zeroed bytes there
     is no need to store explicit zero bytes in the object file.  The
     bss section was invented to eliminate those explicit zeros from
     object files.

absolute section
     Address 0 of this section is always "relocated" to runtime address
     0.  This is useful if you want to refer to an address that ld must
     not change when relocating.  In this sense we speak of absolute
     addresses being "unrelocatable": they don't change during
     relocation.

undefined section
     This "section" is a catch-all for address references to objects
     not in the preceding sections.

   An idealized example of three relocatable sections follows.  The
example uses the traditional section names .text and .data.  Memory
addresses are on the horizontal axis.

                          +-----+----+--+
     partial program # 1: |ttttt|dddd|00|
                          +-----+----+--+

                          text   data bss
                          seg.   seg. seg.

                          +---+---+---+
     partial program # 2: |TTT|DDD|000|
                          +---+---+---+

                          +--+---+-----+--+----+---+-----+~~
     linked program:      |  |TTT|ttttt|  |dddd|DDD|00000|
```

```
                        +--+---+-----+--+----+---+-----+~~

         addresses:     0 ...
```

1.38 as.guide/As Sections

```
as Internal Sections
====================

   These sections are meant only for the internal use of as.  They have
no meaning at run-time.  You don't really need to know about these
sections for most purposes; but they can be mentioned in as warning
messages, so it might be helpful to have an idea of their meanings to
as.  These sections are used to permit the value of every
expression in your assembly language program to be a section-relative
address.

ASSEMBLER-INTERNAL-LOGIC-ERROR!
     An internal assembler logic error has been found.  This means
     there is a bug in the assembler.

expr section
     The assembler stores complex expression internally as combinations
     of symbols.  When it needs to represent an expression as a symbol,
     it puts it in the expr section.
```

1.39 as.guide/Sub-Sections

```
                Sub-Sections
============

   Assembled bytes conventionally fall into two sections: text and data.
You may have separate groups of data in named sections that you want to
end up near to each other in the object file, even though they are not
contiguous in the assembler source.  as allows you to use subsections
for this purpose.  Within each section, there can be numbered
subsections with values from 0 to 8192.  Objects assembled into the
same subsection will be grouped with other objects in the same
subsection when they are all put into the object file.  For example, a
compiler might want to store constants in the text section, but might
not want to have them interspersed with the program being assembled.
In this case, the compiler could issue a .text 0 before each section of
code being output, and a .text 1 before each group of constants being
output.

   Subsections are optional.  If you don't use subsections, everything
will be stored in subsection number zero.

   Each subsection is zero-padded up to a multiple of four bytes.
(Subsections may be padded a different amount on different flavors of
```

```
as.)

   Subsections appear in your object file in numeric order, lowest
numbered to highest.  (All this to be compatible with other people's
assemblers.) The object file contains no representation of subsections;
ld and other programs that manipulate object files will see no trace of
them.  They just see all your text subsections as a text section, and
all your data subsections as a data section.

   To specify which subsection you want subsequent statements assembled
into, use a numeric argument to specify it, in a .text expression or a
.data expression statement.  When generating COFF output, you can also
use an extra subsection argument with arbitrary named sections:
.section name, expression.  Expression should be an absolute expression.
(See
                Expressions
                .)  If you just say .text then .text 0 is assumed.
Likewise .data means .data 0.  Assembly begins in text 0.  For instance:
     .text 0     # The default subsection is text 0 anyway.
     .ascii "This lives in the first text subsection. *"
     .text 1
     .ascii "But this lives in the second text subsection."
     .data 0
     .ascii "This lives in the data section,"
     .ascii "in the first data subsection."
     .text 0
     .ascii "This lives in the first text section,"
     .ascii "immediately following the asterisk (*)."

   Each section has a location counter incremented by one for every
byte assembled into that section.  Because subsections are merely a
convenience restricted to as there is no concept of a subsection
location counter.  There is no way to directly manipulate a location
counter--but the .align directive will change it, and any label
definition will capture its current value.  The location counter of the
section that statements are being assembled into is said to be the
active location counter.
```

1.40 as.guide/bss

```
                bss Section
===========

   The bss section is used for local common variable storage.  You may
allocate address space in the bss section, but you may not dictate data
to load into it before your program executes.  When your program starts
running, all the contents of the bss section are zeroed bytes.

   Addresses in the bss section are allocated with special directives;
you may not assemble anything directly into the bss section.  Hence
there are no bss subsections. See
                .comm
                , see
```

```
.lcomm
.
```

1.41 as.guide/Symbols

```
Symbols
*******
```

Symbols are a central concept: the programmer uses symbols to name things, the linker uses symbols to link, and the debugger uses symbols to debug.

> Warning: as does not place symbols in the object file in the same order they were declared. This may break some debuggers.

```
Labels
                        Labels

Setting Symbols
              Giving Symbols Other Values

Symbol Names
                 Symbol Names

Dot
                           The Special Dot Symbol

Symbol Attributes
            Symbol Attributes
```

1.42 as.guide/Labels

```
Labels
======
```

A label is written as a symbol immediately followed by a colon :. The symbol then represents the current value of the active location counter, and is, for example, a suitable instruction operand. You are warned if you use the same symbol to represent two different locations: the first definition overrides any other definitions.

1.43 as.guide/Setting Symbols

```
                Giving Symbols Other Values
============================

   A symbol can be given an arbitrary value by writing a symbol,
followed by an equals sign =, followed by an expression (see

                Expressions
                ).  This is equivalent to using the .set directive.  See

                .set
                .
```

1.44 as.guide/Symbol Names

```
                Symbol Names
============

   Symbol names begin with a letter or with one of ._.  On most
machines, you can also use $ in symbol names; exceptions are noted in

                Machine Dependencies
                .  That character may be followed by any string of
digits, letters, dollar signs (unless otherwise noted in

                Machine Dependencies
                ), and underscores.  For the AMD 29K family, ? is
also allowed in the body of a symbol name, though not at its beginning.

   Case of letters is significant: foo is a different symbol name than
Foo.

   Each symbol has exactly one name. Each name in an assembly language
program refers to exactly one symbol. You may use that symbol name any
number of times in a program.

Local Symbol Names
------------------

   Local symbols help compilers and programmers use names temporarily.
There are ten local symbol names, which are re-used throughout the
program.  You may refer to them using the names 0 1 ... 9.  To define a
local symbol, write a label of the form N: (where N represents any
digit).  To refer to the most recent previous definition of that symbol
write Nb, using the same digit as when you defined the label.  To refer
to the next definition of a local label, write Nf--where N gives you a
choice of 10 forward references.  The b stands for "backwards" and the
f stands for "forwards".

   Local symbols are not emitted by the current GNU C compiler.

   There is no restriction on how you can use these labels, but
remember that at any point in the assembly you can refer to at most 10
```

prior local labels and to at most 10 forward local labels.

Local symbol names are only a notation device. They are immediately transformed into more conventional symbol names before the assembler uses them. The symbol names stored in the symbol table, appearing in error messages and optionally emitted to the object file have these parts:

L
All local labels begin with L. Normally both as and ld forget symbols that start with L. These labels are used for symbols you are never intended to see. If you give the -L option then as will retain these symbols in the object file. If you also instruct ld to retain these symbols, you may use them in debugging.

digit
If the label is written 0: then the digit is 0. If the label is written 1: then the digit is 1. And so on up through 9:.

This unusual character is included so you don’t accidentally invent a symbol of the same name. The character has ASCII value \001.

ordinal number
This is a serial number to keep the labels distinct. The first 0: gets the number 1; The 15th 0: gets the number 15; etc.. Likewise for the other labels 1: through 9:.

For instance, the first 1: is named L!A1, the 44th 3: is named L!A44.

1.45 as.guide/Dot

The Special Dot Symbol
==========

The special symbol . refers to the current address that as is assembling into. Thus, the expression melvin: .long . will cause melvin to contain its own address. Assigning a value to . is treated the same as a .org directive. Thus, the expression .=.+4 is the same as saying .space 4.

1.46 as.guide/Symbol Attributes

Symbol Attributes
==========

Every symbol has, as well as its name, the attributes "Value" and "Type". Depending on output format, symbols can also have auxiliary attributes.

If you use a symbol without defining it, as assumes zero for all these attributes, and probably won't warn you. This makes the symbol an externally defined symbol, which is generally what you would want.

Symbol Value
Value

Symbol Type
Type

a.out Symbols
Symbol Attributes: a.out

COFF Symbols
Symbol Attributes for COFF

1.47 as.guide/Symbol Value

Value

The value of a symbol is (usually) 32 bits. For a symbol which labels a location in the text, data, bss or absolute sections the value is the number of addresses from the start of that section to the label. Naturally for text, data and bss sections the value of a symbol changes as ld changes section base addresses during linking. Absolute symbols' values do not change during linking: that is why they are called absolute.

The value of an undefined symbol is treated in a special way. If it is 0 then the symbol is not defined in this assembler source program, and ld will try to determine its value from other programs it is linked with. You make this kind of symbol simply by mentioning a symbol name without defining it. A non-zero value represents a .comm common declaration. The value is how much common storage to reserve, in bytes (addresses). The symbol refers to the first address of the allocated storage.

1.48 as.guide/Symbol Type

Type

The type attribute of a symbol contains relocation (section) information, any flag settings indicating that a symbol is external, and (optionally), other information for linkers and debuggers. The exact format depends on the object-code output format in use.

1.49 as.guide/a.out Symbols

Symbol Attributes: a.out

Symbol Desc
Descriptor

Symbol Other
Other

1.50 as.guide/Symbol Desc

Descriptor
.....

This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a .desc statement (see .desc). A descriptor value means nothing to as.

1.51 as.guide/Symbol Other

Other
.....

This is an arbitrary 8-bit value. It means nothing to as.

1.52 as.guide/COFF Symbols

Symbol Attributes for COFF

The COFF format supports a multitude of auxiliary symbol attributes; like the primary symbol attributes, they are set between .def and .endef directives.

Primary Attributes
....

The symbol name is set with .def; the value and type, respectively, with .val and .type.

Auxiliary Attributes
....

The as directives .dim, .line, .scl, .size, and .tag can generate auxiliary symbol table information for COFF.

1.53 as.guide/Expressions

Expressions
****

An expression specifies an address or numeric value. Whitespace may precede and/or follow an expression.

Empty Exprs
Empty Expressions

Integer Exprs
Integer Expressions

1.54 as.guide/Empty Exprs

Empty Expressions
=================

An empty expression has no value: it is just whitespace or null. Wherever an absolute expression is required, you may omit the expression and as will assume a value of (absolute) 0. This is compatible with other assemblers.

1.55 as.guide/Integer Exprs

```
                Integer Expressions
====================

   An integer expression is one or more arguments delimited by
operators.



                Arguments
                                  Arguments

                Operators
                                  Operators

                Prefix Ops
                                 Prefix Operators

                Infix Ops
                                  Infix Operators
```

1.56 as.guide/Arguments

```
Arguments
---------

   Arguments are symbols, numbers or subexpressions.  In other contexts
arguments are sometimes called "arithmetic operands".  In this manual,
to avoid confusing them with the "instruction operands" of the machine
language, we use the term "argument" to refer to parts of expressions
only, reserving the word "operand" to refer only to machine instruction
operands.

   Symbols are evaluated to yield {section NNN } where section is one
of text, data, bss, absolute, or undefined.  NNN is a signed, 2's
complement 32 bit integer.

   Numbers are usually integers.

   A number can be a flonum or bignum.  In this case, you are warned
that only the low order 32 bits are used, and as pretends these 32 bits
are an integer.  You may write integer-manipulating instructions that
act on exotic constants, compatible with other assemblers.

   Subexpressions are a left parenthesis ( followed by an integer
expression, followed by a right parenthesis ); or a prefix operator
followed by an argument.
```

1.57 as.guide/Operators

Operators

Operators are arithmetic functions, like + or %. Prefix operators are followed by an argument. Infix operators appear between their arguments. Operators may be preceded and/or followed by whitespace.

1.58 as.guide/Prefix Ops

Prefix Operator
----

as has the following prefix operators. They each take one argument, which must be absolute.

-
: Negation. Two's complement negation.

~
: Complementation. Bitwise not.

1.59 as.guide/Infix Ops

Infix Operators

Infix operators take two arguments, one on either side. Operators have precedence, but operations with equal precedence are performed left to right. Apart from + or -, both arguments must be absolute, and the result is absolute.

1. Highest Precedence

*
: Multiplication.

/
: Division. Truncation is the same as the C operator /

%
: Remainder.

<
<<
: Shift Left. Same as the C operator <<.

>
>>
: Shift Right. Same as the C operator >>.

2. Intermediate precedence

|
Bitwise Inclusive Or.

&
Bitwise And.

^
Bitwise Exclusive Or.

!
Bitwise Or Not.

3. Lowest Precedence

+
Addition. If either argument is absolute, the result has the section of the other argument. If either argument is pass1 or undefined, the result is pass1. Otherwise + is illegal.

-
Subtraction. If the right argument is absolute, the result has the section of the left argument. If either argument is pass1 the result is pass1. If either argument is undefined the result is difference section. If both arguments are in the same section, the result is absolute--provided that section is one of text, data or bss. Otherwise subtraction is illegal.

The sense of the rule for addition is that it's only meaningful to add the offsets in an address; you can only have a defined section in one of the two arguments.

Similarly, you can't subtract quantities from two different sections.

1.60 as.guide/Pseudo Ops

Assembler Directives
****

All assembler directives have names that begin with a period (.). The rest of the name is letters, usually in lower case.

This chapter discusses directives that are available regardless of the target machine configuration for the GNU assembler. Some machine configurations provide additional directives. See
Machine Dependencies
.

Abort

```
                        .abort

ABORT
                        .ABORT

Align
                        .align abs-expr , abs-expr

App-File
                    .app-file string

Ascii
                        .ascii "string" ...

Asciz
                        .asciz "string" ...

Byte
                         .byte expressions

Comm
                         .comm symbol , length

Data
                         .data subsection

Def
                          .def name

Desc
                         .desc symbol, abs-expression

Dim
                          .dim

Double
                   .double flonums

Eject
                        .eject

Else
                          .else

Endef
                        .endef

Endif
                        .endif
```

```
Equ
                            .equ symbol, expression

Extern
                          .extern

File
                            .file string

Fill
                            .fill repeat , size , value

Float
                           .float flonums

Global
                          .global symbol, .globl symbol

hword
                           .hword expressions

Ident
                           .ident

If
                              .if absolute expression

Include
                         .include "file"

Int
                              .int expressions

Lcomm
                           .lcomm symbol , length

Lflags
                          .lflags

Line
                            .line line-number

Ln
                              .ln line-number

List
                            .list

Long
                            .long expressions
```

```
Nolist
                        .nolist

Octa
                          .octa bignums

Org
                           .org new-lc , fill

Psize
                         .psize lines, columns

Quad
                          .quad bignums

Sbttl
                         .sbttl "subheading"


Scl
                           .scl class


Section
                       .section name, subsection


Set
                            .set symbol, expression

Short
                         .short expressions

Single
                        .single flonums


Size
                          .size


Space
                         .space size , fill


Stab
                          .stabd, .stabn, .stabs


Tag
                           .tag structname


Text
                          .text subsection

Title
```

.title "heading"

Type
.type int

Val
.val addr

Word
.word expressions

Deprecated
Deprecated Directives

1.61 as.guide/Abort

.abort
======

This directive stops the assembly immediately. It is for compatibility with other assemblers. The original idea was that the assembly language source would be piped into the assembler. If the sender of the source quit, it could use this directive tells as to quit also. One day .abort will not be supported.

1.62 as.guide/ABORT

.ABORT
======

When producing COFF output, as accepts this directive as a synonym for .abort.

When producing b.out output, as accepts this directive, but ignores it.

1.63 as.guide/Align

.align abs-expr , abs-expr
==========================

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the number of low-order zero bits the location counter will have after

advancement. For example .align 3 will advance the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are zero.

1.64 as.guide/App-File

.app-file string
==========

.app-file (which may also be spelled .file) tells as that we are about to start a new logical file. string is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes "; but if you wish to specify an empty file name is permitted, you must give the quotes-"". This statement may go away in future: it is only recognized to be compatible with old as programs.

1.65 as.guide/Ascii

.ascii "string" ...
==========

.ascii expects zero or more string literals (see Strings) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

1.66 as.guide/Asciz

.asciz "string" ...
==========

.asciz is just like .ascii, but each string is followed by a zero byte. The "z" in .asciz stands for "zero".

1.67 as.guide/Byte

```
.byte expressions
=================

   .byte expects zero or more expressions, separated by commas.  Each
expression is assembled into the next byte.
```

1.68 as.guide/Comm

```
.comm symbol , length
=====================

   .comm declares a named common area in the bss section.  Normally ld
reserves memory addresses for it during linking, so no partial program
defines the location of the symbol.  Use .comm to tell ld that it must
be at least length bytes long.  ld will allocate space for each .comm
symbol that is at least as long as the longest .comm request in any of
the partial programs linked.  length is an absolute expression.
```

1.69 as.guide/Data

```
.data subsection
================

   .data tells as to assemble the following statements onto the end of
the data subsection numbered subsection (which is an absolute
expression).  If subsection is omitted, it defaults to zero.
```

1.70 as.guide/Def

```
.def name
=========

   Begin defining debugging information for a symbol name; the
definition extends until the .endef directive is encountered.

   This directive is only observed when as is configured for COFF
format output; when producing b.out, .def is recognized, but ignored.
```

1.71 as.guide/Desc

```
                .desc symbol, abs-expression
============================

   This directive sets the descriptor of the symbol (see

                Symbol Attributes
                ) to the low 16 bits of an absolute expression.

   The .desc directive is not available when as is configured for COFF
output; it is only for a.out or b.out object format.  For the sake of
compatibility, as will accept it, but produce no output, when
configured for COFF.
```

1.72 as.guide/Dim

```
.dim
====

   This directive is generated by compilers to include auxiliary
debugging information in the symbol table.  It is only permitted inside
.def/.endef pairs.

   .dim is only meaningful when generating COFF format output; when as
is generating b.out, it accepts this directive but ignores it.
```

1.73 as.guide/Double

```
                .double flonums
===============

   .double expects zero or more flonums, separated by commas.  It
assembles floating point numbers.  The exact kind of floating point
numbers emitted depends on how as is configured.  See

                Machine Dependencies
                .
```

1.74 as.guide/Eject

```
.eject
======

   Force a page break at this point, when generating assembly listings.
```

1.75 as.guide/Else

```
                .else
=====

   .else is part of the as support for conditional assembly; see
                .if
                .
It marks the beginning of a section of code to be assembled if the
condition for the preceding .if was false.
```

1.76 as.guide/Endef

```
.endef
======

   This directive flags the end of a symbol definition begun with .def.

   .endef is only meaningful when generating COFF format output; if as
is configured to generate b.out, it accepts this directive but ignores
it.
```

1.77 as.guide/Endif

```
                .endif
======

   .endif is part of the as support for conditional assembly; it marks
the end of a block of code that is only assembled conditionally.  See

                .if
                .
```

1.78 as.guide/Equ

```
                .equ symbol, expression
=======================

   This directive sets the value of symbol to expression.  It is
synonymous with .set; see
                .set
                .
```

1.79 as.guide/Extern

```
.extern
=======
```

.extern is accepted in the source program--for compatibility with other assemblers--but it is ignored. as treats all undefined symbols as external.

1.80 as.guide/File

```
                .file string
============
```

.file (which may also be spelled .app-file) tells as that we are about to start a new logical file. string is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes "; but if you wish to specify an empty file name, you must give the quotes-"". This statement may go away in future: it is only recognized to be compatible with old as programs. In some configurations of as, .file has already been removed to avoid conflicts with other assemblers. See

```
                Machine Dependencies
                .
```

1.81 as.guide/Fill

```
.fill repeat , size , value
===========================
```

result, size and value are absolute expressions. This emits repeat copies of size bytes. Repeat may be zero or more. Size may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The contents of each repeat bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are value rendered in the byte-order of an integer on the computer as is assembling for. Each size bytes in a repetition is taken from the lowest order size bytes of this number. Again, this bizarre behavior is compatible with other people's assemblers.

size and value are optional. If the second comma and value are absent, value is assumed zero. If the first comma and following tokens are absent, size is assumed to be 1.

1.82 as.guide/Float

```
                .float flonums
==============
```

   This directive assembles zero or more flonums, separated by commas.
It has the same effect as .single. The exact kind of floating point
numbers emitted depends on how as is configured. See

```
                Machine Dependencies
                .
```

1.83 as.guide/Global

```
.global symbol, .globl symbol
============================
```

   .global makes the symbol visible to ld. If you define symbol in
your partial program, its value is made available to other partial
programs that are linked with it. Otherwise, symbol will take its
attributes from a symbol of the same name from another partial program
it is linked with.

   Both spellings (.globl and .global) are accepted, for compatibility
with other assemblers.

1.84 as.guide/hword

```
.hword expressions
==================
```

   This expects zero or more expressions, and emits a 16 bit number for
each.

   This directive is a synonym for .short; depending on the target
architecture, it may also be a synonym for .word.

1.85 as.guide/Ident

```
.ident
======
```

   This directive is used by some assemblers to place tags in object
files. as simply accepts the directive for source-file compatibility
with such assemblers, but does not actually emit anything for it.

1.86 as.guide/If

```
                .if absolute expression
=======================

   .if marks the beginning of a section of code which is only
considered part of the source program being assembled if the argument
(which must be an absolute expression) is non-zero.  The end of the
conditional section of code must be marked by .endif (see
                .endif
                );
optionally, you may include code for the alternative condition, flagged
by .else (see
                .else
                .

   The following variants of .if are also supported:
.ifdef symbol
     Assembles the following section of code if the specified symbol
     has been defined.

.ifndef symbol
ifnotdef symbol
     Assembles the following section of code if the specified symbol
     has not been defined.  Both spelling variants are equivalent.
```

1.87 as.guide/Include

```
                .include "file"
===============

   This directive provides a way to include supporting files at
specified points in your source program.  The code from file is
assembled as if it followed the point of the .include; when the end of
the included file is reached, assembly of the original file continues.
You can control the search paths used with the -I command-line option
(see
                Command-Line Options
                ).  Quotation marks are required around file.
```

1.88 as.guide/Int

```
.int expressions
================

   Expect zero or more expressions, of any section, separated by
commas.  For each expression, emit a 32-bit number that will, at run
time, be the value of that expression.  The byte order of the
```

expression depends on what kind of computer will run the program.

1.89 as.guide/Lcomm

```
                .lcomm symbol , length
=====================
```

Reserve length (an absolute expression) bytes for a local common denoted by symbol. The section and value of symbol are those of the new local common. The addresses are allocated in the bss section, so at run-time the bytes will start off zeroed. Symbol is not declared global (see

```
                .global
```

), so is normally not visible to ld.

1.90 as.guide/Lflags

```
.lflags
=======
```

as accepts this directive, for compatibility with other assemblers, but ignores it.

1.91 as.guide/Line

```
.line line-number
=================
```

Tell as to change the logical line number. line-number must be an absolute expression. The next line will have that logical line number. So any other statements on the current line (after a statement separator character) will be reported as on logical line number line-number - 1. One day this directive will be unsupported: it is used only for compatibility with existing assembler programs.

Warning: In the AMD29K configuration of as, this command is only available with the name .ln, rather than as either .line or .ln.

Even though this is a directive associated with the a.out or b.out object-code formats, as will still recognize it when producing COFF output, and will treat .line as though it were the COFF .ln if it is found outside a .def/.endef pair.

Inside a .def, .line is, instead, one of the directives used by compilers to generate auxiliary symbol information for debugging.

1.92 as.guide/Ln

```
.ln line-number
===============

   .ln is a synonym for .line.
```

1.93 as.guide/List

```
                .list
=====

   Control (in conjunction with the .nolist directive) whether or not
assembly listings are generated.  These two directives maintain an
internal counter (which is zero initially).   .list increments the
counter, and .nolist decrements it.  Assembly listings are generated
whenever the counter is greater than zero.

   By default, listings are disabled.  When you enable them (with the
-a command line option; see
                Command-Line Options
                ), the initial
value of the listing counter is one.
```

1.94 as.guide/Long

```
                .long expressions
=================

   .long is the same as .int, see
                .int
                .
```

1.95 as.guide/Nolist

```
.nolist
=======

   Control (in conjunction with the .list directive) whether or not
assembly listings are generated.  These two directives maintain an
internal counter (which is zero initially).   .list increments the
counter, and .nolist decrements it.  Assembly listings are generated
whenever the counter is greater than zero.
```

1.96 as.guide/Octa

```
.octa bignums
=============
```

   This directive expects zero or more bignums, separated by commas.
For each bignum, it emits a 16-byte integer.

   The term "octa" comes from contexts in which a "word" is two bytes;
hence octa-word for 16 bytes.

1.97 as.guide/Org

```
.org new-lc , fill
==================
```

   .org will advance the location counter of the current section to
new-lc. new-lc is either an absolute expression or an expression with
the same section as the current subsection. That is, you can't use
.org to cross sections: if new-lc has the wrong section, the .org
directive is ignored. To be compatible with former assemblers, if the
section of new-lc is absolute, as will issue a warning, then pretend
the section of new-lc is the same as the current subsection.

   .org may only increase the location counter, or leave it unchanged;
you cannot use .org to move the location counter backwards.

   Because as tries to assemble programs in one pass new-lc may not be
undefined. If you really detest this restriction we eagerly await a
chance to share your improved assembler.

   Beware that the origin is relative to the start of the section, not
to the start of the subsection. This is compatible with other people's
assemblers.

   When the location counter (of the current subsection) is advanced,
the intervening bytes are filled with fill which should be an absolute
expression. If the comma and fill are omitted, fill defaults to zero.

1.98 as.guide/Psize

```
.psize lines , columns
======================
```

   Use this directive to declare the number of lines--and, optionally,
the number of columns--to use for each page, when generating listings.

   If you don't use .psize, listings will use a default line-count of
60. You may omit the comma and columns specification; the default
width is 200 columns.

as will generate formfeeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using .eject).

If you specify lines as 0, no formfeeds are generated save those explicitly specified with .eject.

1.99 as.guide/Quad

```
.quad bignums
=============
```

.quad expects zero or more bignums, separated by commas. For each bignum, it emits an 8-byte integer. If the bignum won't fit in 8 bytes, it prints a warning message; and just takes the lowest order 8 bytes of the bignum.

The term "quad" comes from contexts in which a "word" is two bytes; hence quad-word for 8 bytes.

1.100 as.guide/Sbttl

```
.sbttl "subheading"
===================
```

Use subheading as the title (third line, immediately after the title line) when generating assembly listings.

This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

1.101 as.guide/Scl

```
.scl class
==========
```

Set the storage-class value for a symbol. This directive may only be used inside a .def/.endef pair. Storage class may flag whether a symbol is static or external, or it may record further symbolic debugging information.

The .scl directive is primarily associated with COFF output; when configured to generate b.out output format, as will accept this directive but ignore it.

1.102 as.guide/Section

```
.section name, subsection
=========================

   Assemble the following code into end of subsection numbered
subsection in the COFF named section name.  If you omit subsection, as
uses subsection number zero.  .section .text is equivalent to the .text
directive; .section .data is equivalent to the .data directive.
```

1.103 as.guide/Set

```
                .set symbol, expression
=======================

   This directive sets the value of symbol to expression.  This will
change symbol's value and type to conform to expression.  If symbol was
flagged as external, it remains flagged. (See
                Symbol Attributes
                .)

   You may .set a symbol many times in the same assembly.  If the
expression's section is unknowable during pass 1, a second pass over
the source program will be forced.  The second pass is currently not
implemented.  as will abort with an error message if one is required.

   If you .set a global symbol, the value stored in the object file is
the last value stored into it.
```

1.104 as.guide/Short

```
                .short expressions
==================

   .short is normally the same as .word.  See
                .word
                .

   In some configurations, however, .short and .word generate numbers
of different lengths; see
                Machine Dependencies
                .
```

1.105 as.guide/Single

```
                .single flonums
===============
```

   This directive assembles zero or more flonums, separated by commas.
It has the same effect as .float. The exact kind of floating point
numbers emitted depends on how as is configured. See

```
                Machine Dependencies
                .
```

1.106 as.guide/Size

```
.size
=====
```

   This directive is generated by compilers to include auxiliary
debugging information in the symbol table. It is only permitted inside
.def/.endef pairs.

   .size is only meaningful when generating COFF format output; when as
is generating b.out, it accepts this directive but ignores it.

1.107 as.guide/Space

```
                .space size , fill
==================
```

   This directive emits size bytes, each of value fill. Both size and
fill are absolute expressions. If the comma and fill are omitted, fill
is assumed to be zero.

   On the AMD 29K, this directive is ignored; it is accepted for
compatibility with other AMD 29K assemblers.

```
     Warning: In most versions of the GNU assembler, the directive
     .space has the effect of .block  See
                Machine Dependencies
                .
```

1.108 as.guide/Stab

```
                .stabd, .stabn, .stabs
======================
```

There are three directives that begin .stab. All emit symbols (see

Symbols
), for use by symbolic debuggers. The symbols are not entered in the as hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required:

string
This is the symbol's name. It may contain any character except \000, so is more general than ordinary symbol names. Some debuggers used to code arbitrarily complex structures into symbol names using this field.

type
An absolute expression. The symbol's type is set to the low 8 bits of this expression. Any bit pattern is permitted, but ld and debuggers will choke on silly bit patterns.

other
An absolute expression. The symbol's "other" attribute is set to the low 8 bits of this expression.

desc
An absolute expression. The symbol's descriptor is set to the low 16 bits of this expression.

value
An absolute expression which becomes the symbol's value.

If a warning is detected while reading a .stabd, .stabn, or .stabs statement, the symbol has probably already been created and you will get a half-formed symbol in your object file. This is compatible with earlier assemblers!

.stabd type , other , desc
The "name" of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings.

The symbol's value is set to the location counter, relocatably. When your program is linked, the value of this symbol will be where the location counter was when the .stabd was assembled.

.stabn type , other , desc , value
The name of the symbol is set to the empty string "".

.stabs string , type , other , desc , value
All five fields are specified.

1.109 as.guide/Tag

```
.tag structname
===============

   This directive is generated by compilers to include auxiliary
debugging information in the symbol table.  It is only permitted inside
.def/.endef pairs.  Tags are used to link structure definitions in the
symbol table with instances of those structures.

   .tag is only used when generating COFF format output; when as is
generating b.out, it accepts this directive but ignores it.
```

1.110 as.guide/Text

```
.text subsection
================

   Tells as to assemble the following statements onto the end of the
text subsection numbered subsection, which is an absolute expression.
If subsection is omitted, subsection number zero is used.
```

1.111 as.guide/Title

```
.title "heading"
================

   Use heading as the title (second line, immediately after the source
file name and pagenumber) when generating assembly listings.

   This directive affects subsequent pages, as well as the current page
if it appears within ten lines of the top of a page.
```

1.112 as.guide/Type

```
.type int
=========

   This directive, permitted only within .def/.endef pairs, records the
integer int as the type attribute of a symbol table entry.

   .type is associated only with COFF format output; when as is
configured for b.out output, it accepts this directive but ignores it.
```

1.113 as.guide/Val

```
.val addr
=========
```

   This directive, permitted only within .def/.endef pairs, records the address addr as the value attribute of a symbol table entry.

   .val is used only for COFF output; when as is configured for b.out, it accepts this directive but ignores it.

1.114 as.guide/Word

```
                .word expressions
=================
```

   This directive expects zero or more expressions, of any section, separated by commas.

   The size of the number emitted, and its byte order, depends on what kind of computer will run the program.

     Warning: Special Treatment to support Compilers

   Machines with a 32-bit address space, but that do less than 32-bit addressing, require the following special treatment. If the machine of interest to you does 32-bit addressing (or doesn't require it; see

               Machine Dependencies
               ), you can ignore this issue.

   In order to assemble compiler output into something that will work, as will occasionlly do strange things to .word directives. Directives of the form .word sym1-sym2 are often emitted by compilers as part of jump tables. Therefore, when as assembles a directive of the form .word sym1-sym2, and the difference between sym1 and sym2 does not fit in 16 bits, as will create a secondary jump table, immediately before the next label. This secondary jump table will be preceded by a short-jump to the first byte after the secondary table. This short-jump prevents the flow of control from accidentally falling into the new table. Inside the table will be a long-jump to sym2. The original .word will contain sym1 minus the address of the long-jump to sym2.

   If there were several occurrences of .word sym1-sym2 before the secondary jump table, all of them will be adjusted. If there was a .word sym3-sym4, that also did not fit in sixteen bits, a long-jump to sym4 will be included in the secondary jump table, and the .word directives will be adjusted to contain sym3 minus the address of the long-jump to sym4; and so on, for as many entries in the original jump table as necessary.

1.115 as.guide/Deprecated

```
Deprecated Directives
=====================

   One day these directives won't work.  They are included for
compatibility with older assemblers.
.abort
.app-file
.line
```

1.116 as.guide/Machine Dependencies

```
                Machine Dependent Features
**************************

   The machine instruction sets are (almost by definition) different on
each machine where as runs.  Floating point representations vary as
well, and as often supports a few additional directives or command-line
options for compatibility with other assemblers on a particular
platform.  Finally, some versions of as support special
pseudo-instructions for branch optimization.

   This chapter discusses most of these differences, though it does not
include details on any machine's instruction set.  For details on that
subject, see the hardware manufacturer's manual.



                Vax-Dependent
                               VAX Dependent Features


                AMD29K-Dependent
                            AMD 29K Dependent Features


                H8-300-Dependent
                            Hitachi H8/300 Dependent Features


                H8-500-Dependent
                            Hitachi H8/500 Dependent Features


                SH-Dependent
                                Hitachi SH Dependent Features


                i960-Dependent
                              Intel 80960 Dependent Features
```

1.117 as.guide/Vax-Dependent

VAX Dependent Features
==========

1.118 as.guide/Vax-Opts

VAX Command-Line Options

The Vax version of as accepts any of the following options, gives a warning message that the option was ignored and proceeds. These options are for compatibility with scripts designed for other people's assemblers.

-D (Debug)
-S (Symbol Table)
-T (Token Trace)
These are obsolete options used to debug old assemblers.

-d (Displacement size for JUMPs)
This option expects a number following the -d. Like options that expect filenames, the number may immediately follow the -d (old standard) or constitute the whole of the command line argument that follows -d (GNU standard).

-V (Virtualize Interpass Temporary File)
Some other assemblers use a temporary file. This option commanded them to keep the information in active memory rather than in a disk file. as always does this, so this option is redundant.

-J (JUMPify Longer Branches)
Many 32-bit computers permit a variety of branch instructions to do the same job. Some of these instructions are short (and fast) but have a limited range; others are long (and slow) but can branch anywhere in virtual memory. Often there are 3 flavors of branch: short, medium and long. Some other assemblers would emit short and medium branches, unless told by this option to emit short and long branches.

-t (Temporary File Directory)
Some other assemblers may use a temporary file, and this option takes a filename being the directory to site the temporary file. Since as does not use a temporary disk file, this option makes no difference. -t needs exactly one filename.

The Vax version of the assembler accepts two options when compiled for VMS. They are -h, and -+. The -h option prevents as from modifying the symbol-table entries for symbols that contain lowercase characters (I think). The -+ option causes as to print warning messages if the FILENAME part of the object file, or any symbol name is larger than 31 characters. The -+ option also insertes some code following the _main symbol so that the object file will be compatible with Vax-11 "C".

1.119 as.guide/VAX-float

VAX Floating Point

Conversion of flonums to floating point is correct, and compatible with previous assemblers. Rounding is towards zero if the remainder is exactly half the least significant bit.

D, F, G and H floating point formats are understood.

Immediate floating literals (e.g. S`$6.9) are rendered correctly. Again, rounding is towards zero in the boundary case.

The .float directive produces f format numbers. The .double directive produces d format numbers.

1.120 as.guide/VAX-directives

Vax Machine Directives

The Vax version of the assembler supports four directives for generating Vax floating point constants. They are described in the table below.

.dfloat
This expects zero or more flonums, separated by commas, and assembles Vax d format 64-bit floating point constants.

.ffloat
This expects zero or more flonums, separated by commas, and assembles Vax f format 32-bit floating point constants.

.gfloat
This expects zero or more flonums, separated by commas, and assembles Vax g format 64-bit floating point constants.

.hfloat
This expects zero or more flonums, separated by commas, and assembles Vax h format 128-bit floating point constants.

1.121 as.guide/VAX-opcodes

VAX Opcodes

All DEC mnemonics are supported. Beware that case... instructions have exactly 3 operands. The dispatch table that follows the case... instruction should be made with .word statements. This is compatible with all unix assemblers we know of.

1.122 as.guide/VAX-branch

VAX Branch Improvement

Certain pseudo opcodes are permitted. They are for branch instructions. They expand to the shortest branch instruction that will reach the target. Generally these mnemonics are made by substituting j for b at the start of a DEC mnemonic. This feature is included both for compatibility and to help compilers. If you don't need this feature, don't use these opcodes. Here are the mnemonics, and the code they can expand into.

jbsb
Jsb is already an instruction mnemonic, so we chose jbsb.
(byte displacement)
bsbb ...

(word displacement)
bsbw ...

(long displacement)
jsb ...

jbr
jr
Unconditional branch.
(byte displacement)
brb ...

(word displacement)
brw ...

(long displacement)
jmp ...

jCOND
COND may be any one of the conditional branches neq, nequ, eql, eqlu, gtr, geq, lss, gtru, lequ, vc, vs, gequ, cc, lssu, cs. COND may also be one of the bit tests bs, bc, bss, bcs, bsc, bcc, bssi, bcci, lbs, lbc. NOTCOND is the opposite condition to COND.
(byte displacement)
bCOND ...

(word displacement)
bNOTCOND foo ; brw ... ; foo:

(long displacement)
bNOTCOND foo ; jmp ... ; foo:

jacbX
X may be one of b d f g h l w.
(word displacement)
OPCODE ...

(long displacement)
OPCODE ..., foo ;
brb bar ;

```
               foo: jmp ... ;
               bar:

jaobYYY
     YYY may be one of lss leq.

jsobZZZ
     ZZZ may be one of geq gtr.
    (byte displacement)
          OPCODE ...

    (word displacement)
               OPCODE ..., foo ;
               brb bar ;
               foo: brw destination ;
               bar:

    (long displacement)
               OPCODE ..., foo ;
               brb bar ;
               foo: jmp destination ;
               bar:

aobleq
aoblss
sobgeq
sobgtr
    (byte displacement)
          OPCODE ...

    (word displacement)
               OPCODE ..., foo ;
               brb bar ;
               foo: brw destination ;
               bar:

    (long displacement)
               OPCODE ..., foo ;
               brb bar ;
               foo: jmp destination ;
               bar:
```

1.123 as.guide/VAX-operands

```
VAX Operands
------------

   The immediate character is $ for Unix compatibility, not # as DEC
writes it.

   The indirect character is * for Unix compatibility, not  @ as DEC
writes it.

   The displacement sizing character is ` (an accent grave) for Unix
```

compatibility, not ^ as DEC writes it. The letter preceding ` may have either case. G is not understood, but all other letters (b i l s w) are understood.

Register names understood are r0 r1 r2 ... r15 ap fp sp pc. Any case of letters will do.

For instance
```
tstb *w`$4(r5)
```

Any expression is permitted in an operand. Operands are comma separated.

1.124 as.guide/VAX-no

```
Not Supported on VAX
--------------------
```

Vax bit fields can not be assembled with as. Someone can add the required code if they really need it.

1.125 as.guide/AMD29K-Dependent

```
                AMD 29K Dependent Features
==========================
```

```
                AMD29K Options
                              Options

                AMD29K Syntax
                               Syntax

                AMD29K Floating Point
                       Floating Point

                AMD29K Directives
                           AMD 29K Machine Directives

                AMD29K Opcodes
                              Opcodes
```

1.126 as.guide/AMD29K Options

```
Options
-------

   as has no additional command-line options for the AMD 29K family.
```

1.127 as.guide/AMD29K Syntax

```
                Syntax
------


                 AMD29K-Chars
                                Special Characters

                 AMD29K-Regs
                                 Register Names
```

1.128 as.guide/AMD29K-Chars

```
Special Characters
..................

   ; is the line comment character.

    @ can be used instead of a newline to separate statements.

   The character ? is permitted in identifiers (but may not begin an
identifier).
```

1.129 as.guide/AMD29K-Regs

```
Register Names
..............

   General-purpose registers are represented by predefined symbols of
the form GRnnn (for global registers) or LRnnn (for local registers),
where nnn represents a number between 0 and 127, written with no
leading zeros.  The leading letters may be in either upper or lower
case; for example, gr13 and LR7 are both valid register names.

   You may also refer to general-purpose registers by specifying the
register number as the result of an expression (prefixed with %% to
flag the expression as a register number):
     %%expression
```

```
--where expression must be an absolute expression evaluating to a
number between 0 and 255.  The range [0, 127] refers to global
registers, and the range [128, 255] to local registers.

   In addition, as understands the following protected special-purpose
register names for the AMD 29K family:

       vab    chd    pc0
       ops    chc    pc1
       cps    rbp    pc2
       cfg    tmc    mmu
       cha    tmr    lru

   These unprotected special-purpose register names are also recognized:
       ipc    alu    fpe
       ipa    bp     inte
       ipb    fc     fps
       q      cr     exop
```

1.130 as.guide/AMD29K Floating Point

```
Floating Point
--------------

   The AMD 29K family uses IEEE floating-point numbers.
```

1.131 as.guide/AMD29K Directives

```
AMD 29K Machine Directives
--------------------------

.block size , fill
     This directive emits size bytes, each of value fill.  Both size
     and fill are absolute expressions.  If the comma and fill are
     omitted, fill is assumed to be zero.

     In other versions of the GNU assembler, this directive is called
     .space.

.cputype
     This directive is ignored; it is accepted for compatibility with
     other AMD 29K assemblers.

.file
     This directive is ignored; it is accepted for compatibility with
     other AMD 29K assemblers.

          Warning: in other versions of the GNU assembler, .file is
          used for the directive called .app-file in the AMD 29K
```

```
          support.

.line
     This directive is ignored; it is accepted for compatibility with
     other AMD 29K assemblers.

.sect
     This directive is ignored; it is accepted for compatibility with
     other AMD 29K assemblers.

.use section name
     Establishes the section and subsection for the following code;
     section name may be one of .text, .data, .data1, or .lit.  With
     one of the first three section name options, .use is equivalent to
     the machine directive section name; the remaining case, .use .lit,
     is the same as .data 200.
```

1.132 as.guide/AMD29K Opcodes

```
Opcodes
-------

   as implements all the standard AMD 29K opcodes.  No additional
pseudo-instructions are needed on this family.

   For information on the 29K machine instruction set, see ‘Am29000
User’s Manual’, Advanced Micro Devices, Inc.
```

1.133 as.guide/H8-300-Dependent

```
                H8/300 Dependent Features
=========================



                 H8-300 Options
                              Options

                 H8-300 Syntax
                               Syntax

                 H8-300 Floating Point
                        Floating Point

                 H8-300 Directives
                            H8/300 Machine Directives

                 H8-300 Opcodes
                              Opcodes
```

1.134 as.guide/H8-300 Options

```
Options
-------

   as has no additional command-line options for the Hitachi H8/300
family.
```

1.135 as.guide/H8-300 Syntax

```
                Syntax
------



                 H8-300-Chars
                                 Special Characters

                 H8-300-Regs
                                  Register Names

                 H8-300-Addressing
                             Addressing Modes
```

1.136 as.guide/H8-300-Chars

```
Special Characters
..................

   ; is the line comment character.

   $ can be used instead of a newline to separate statements.
Therefore you may not use $ in symbol names on the H8/300.
```

1.137 as.guide/H8-300-Regs

```
Register Names
..............

   You can use predefined symbols of the form  rnh and  rnl to refer to
the H8/300 registers as sixteen 8-bit general-purpose registers.  n is
```

a digit from 0 to 7); for instance, both r0h and r7l are valid register names.

You can also use the eight predefined symbols rn to refer to the H8/300 registers as 16-bit registers (you must use this form for addressing).

On the H8/300H, you can also use the eight predefined symbols ern (er0 ... er7) to refer to the 32-bit general purpose registers.

The two control registers are called pc (program counter; a 16-bit register, except on the H8/300H where it is 24 bits) and ccr (condition code register; an 8-bit register). r7 is used as the stack pointer, and can also be called sp.

1.138 as.guide/H8-300-Addressing

Addressing Modes
.....

as understands the following addressing modes for the H8/300:

rn
    Register direct

@rn
    Register indirect

@(d, rn)
@(d:16, rn)
@(d:24, rn)
    Register indirect: 16-bit or 24-bit displacement d from register n. (24-bit displacements are only meaningful on the H8/300H.)

@rn+
    Register indirect with post-increment

@-rn
    Register indirect with pre-decrement

@ aa
@ aa:8
@ aa:16
@ aa:24
    Absolute address aa. (The address size :24 only makes sense on the H8/300H.)

#xx
#xx:8
#xx:16
#xx:32
    Immediate data xx. You may specify the :8, :16, or :32 for clarity, if you wish; but as neither requires this nor uses it--the data size required is taken from context.

```
 @  @ aa
 @  @ aa:8
     Memory indirect.  You may specify the :8 for clarity, if you wish;
     but as neither requires this nor uses it.
```

1.139 as.guide/H8-300 Floating Point

```
Floating Point
--------------

   The H8/300 family has no hardware floating point, but the .float
directive generates IEEE floating-point numbers for compatibility with
other development tools.
```

1.140 as.guide/H8-300 Directives

```
H8/300 Machine Directives
-------------------------

   as has only one machine-dependent directive for the H8/300:

.h300h
     Recognize and emit additional instructions for the H8/300H
     variant, and also make .int emit 32-bit numbers rather than the
     usual (16-bit) for the H8/300 family.

   On the H8/300 family (including the H8/300H) .word directives
generate 16-bit numbers.
```

1.141 as.guide/H8-300 Opcodes

```
Opcodes
-------

   For detailed information on the H8/300 machine instruction set, see
‘H8/300 Series Programming Manual’ (Hitachi ADE-602-025).  For
information specific to the H8/300H, see ‘H8/300H Series Programming
Manual’ (Hitachi).

   as implements all the standard H8/300 opcodes.  No additional
pseudo-instructions are needed on this family.

   The following table summarizes the H8/300 opcodes, and their
arguments.  Entries marked * are opcodes used only on the H8/300H.

               Legend:
                  Rs   source register
```

```
            Rd   destination register
            abs  absolute address
            imm  immediate data
         disp:N  N-bit displacement from a register
        pcrel:N  N-bit displacement relative to program counter

   add.b #imm,rd              *  andc #imm,ccr
   add.b rs,rd                   band #imm,rd
   add.w rs,rd                   band #imm,@rd
*  add.w #imm,rd                 band #imm,@abs:8
*  add.l rs,rd                   bra  pcrel:8
*  add.l #imm,rd              *  bra  pcrel:16
   adds #imm,rd                  bt   pcrel:8
   addx #imm,rd               *  bt   pcrel:16
   addx rs,rd                    brn  pcrel:8
   and.b #imm,rd              *  brn  pcrel:16
   and.b rs,rd                   bf   pcrel:8
*  and.w rs,rd                *  bf   pcrel:16
*  and.w #imm,rd                 bhi  pcrel:8
*  and.l #imm,rd              *  bhi  pcrel:16
*  and.l rs,rd                   bls  pcrel:8

*  bls  pcrel:16                 bld  #imm,rd
   bcc  pcrel:8                  bld  #imm,@rd
*  bcc  pcrel:16                 bld  #imm,@abs:8
   bhs  pcrel:8                  bnot #imm,rd
*  bhs  pcrel:16                 bnot #imm,@rd
   bcs  pcrel:8                  bnot #imm,@abs:8
*  bcs  pcrel:16                 bnot rs,rd
   blo  pcrel:8                  bnot rs,@rd
*  blo  pcrel:16                 bnot rs,@abs:8
   bne  pcrel:8                  bor  #imm,rd
*  bne  pcrel:16                 bor  #imm,@rd
   beq  pcrel:8                  bor  #imm,@abs:8
*  beq  pcrel:16                 bset #imm,rd
   bvc  pcrel:8                  bset #imm,@rd
*  bvc  pcrel:16                 bset #imm,@abs:8
   bvs  pcrel:8                  bset rs,rd
*  bvs  pcrel:16                 bset rs,@rd
   bpl  pcrel:8                  bset rs,@abs:8
*  bpl  pcrel:16                 bsr  pcrel:8
   bmi  pcrel:8                  bsr  pcrel:16
*  bmi  pcrel:16                 bst  #imm,rd
   bge  pcrel:8                  bst  #imm,@rd
*  bge  pcrel:16                 bst  #imm,@abs:8
   blt  pcrel:8                  btst #imm,rd
*  blt  pcrel:16                 btst #imm,@rd
   bgt  pcrel:8                  btst #imm,@abs:8
*  bgt  pcrel:16                 btst rs,rd
   ble  pcrel:8                  btst rs,@rd
*  ble  pcrel:16                 btst rs,@abs:8
   bclr #imm,rd                  bxor #imm,rd
   bclr #imm,@rd                 bxor #imm,@rd
   bclr #imm,@abs:8              bxor #imm,@abs:8
   bclr rs,rd                    cmp.b #imm,rd
   bclr rs,@rd                   cmp.b rs,rd
   bclr rs,@abs:8                cmp.w rs,rd
```

```
   biand #imm,rd                  cmp.w rs,rd
   biand #imm,@rd             *   cmp.w #imm,rd
   biand #imm,@abs:8          *   cmp.l #imm,rd
   bild #imm,rd               *   cmp.l rs,rd
   bild #imm,@rd                  daa  rs
   bild #imm,@abs:8               das  rs
   bior #imm,rd                   dec.b rs
   bior #imm,@rd              *   dec.w #imm,rd
   bior #imm,@abs:8           *   dec.l #imm,rd
   bist #imm,rd                   divxu.b rs,rd
   bist #imm,@rd              *   divxu.w rs,rd
   bist #imm,@abs:8           *   divxs.b rs,rd
   bixor #imm,rd              *   divxs.w rs,rd
   bixor #imm,@rd                 eepmov
   bixor #imm,@abs:8          *   eepmovw

*  exts.w rd                      mov.w rs,@abs:16
*  exts.l rd                  *   mov.l #imm,rd
*  extu.w rd                  *   mov.l rs,rd
*  extu.l rd                  *   mov.l @rs,rd
   inc  rs                    *   mov.l @(disp:16,rs),rd
*  inc.w #imm,rd              *   mov.l @(disp:24,rs),rd
*  inc.l #imm,rd              *   mov.l @rs+,rd
   jmp  @rs                   *   mov.l @abs:16,rd
   jmp  abs                   *   mov.l @abs:24,rd
   jmp  @@abs:8               *   mov.l rs,@rd
   jsr  @rs                   *   mov.l rs,@(disp:16,rd)
   jsr  abs                   *   mov.l rs,@(disp:24,rd)
   jsr  @@abs:8               *   mov.l rs,@-rd
   ldc  #imm,ccr              *   mov.l rs,@abs:16
   ldc  rs,ccr                *   mov.l rs,@abs:24
*  ldc  @abs:16,ccr               movfpe @abs:16,rd
*  ldc  @abs:24,ccr               movtpe rs,@abs:16
*  ldc  @(disp:16,rs),ccr         mulxu.b rs,rd
*  ldc  @(disp:24,rs),ccr     *   mulxu.w rs,rd
*  ldc  @rs+,ccr              *   mulxs.b rs,rd
*  ldc  @rs,ccr               *   mulxs.w rs,rd
*  mov.b @(disp:24,rs),rd         neg.b rs
*  mov.b rs,@(disp:24,rd)     *   neg.w rs
   mov.b @abs:16,rd           *   neg.l rs
   mov.b rs,rd                    nop
   mov.b @abs:8,rd                not.b rs
   mov.b rs,@abs:8            *   not.w rs
   mov.b rs,rd                *   not.l rs
   mov.b #imm,rd                  or.b #imm,rd
   mov.b @rs,rd                   or.b rs,rd
   mov.b @(disp:16,rs),rd     *   or.w #imm,rd
   mov.b @rs+,rd              *   or.w rs,rd
   mov.b @abs:8,rd            *   or.l #imm,rd
   mov.b rs,@rd               *   or.l rs,rd
   mov.b rs,@(disp:16,rd)         orc  #imm,ccr
   mov.b rs,@-rd                  pop.w rs
   mov.b rs,@abs:8            *   pop.l rs
   mov.w rs,@rd                   push.w rs
*  mov.w @(disp:24,rs),rd     *   push.l rs
*  mov.w rs,@(disp:24,rd)         rotl.b rs
*  mov.w @abs:24,rd           *   rotl.w rs
```

```
     *  mov.w rs,@abs:24           *  rotl.l rs
        mov.w rs,rd                   rotr.b rs
        mov.w #imm,rd              *  rotr.w rs
        mov.w @rs,rd               *  rotr.l rs
        mov.w @(disp:16,rs),rd        rotxl.b rs
        mov.w @rs+,rd              *  rotxl.w rs
        mov.w @abs:16,rd           *  rotxl.l rs
        mov.w rs,@(disp:16,rd)        rotxr.b rs
        mov.w rs,@-rd              *  rotxr.w rs

     *  rotxr.l rs                 *  stc  ccr,@(disp:24,rd)
        bpt                        *  stc  ccr,@-rd
        rte                        *  stc  ccr,@abs:16
        rts                        *  stc  ccr,@abs:24
        shal.b rs                     sub.b rs,rd
     *  shal.w rs                     sub.w rs,rd
     *  shal.l rs                  *  sub.w #imm,rd
        shar.b rs                  *  sub.l rs,rd
     *  shar.w rs                  *  sub.l #imm,rd
     *  shar.l rs                     subs #imm,rd
        shll.b rs                     subx #imm,rd
     *  shll.w rs                     subx rs,rd
     *  shll.l rs                  *  trapa #imm
        shlr.b rs                     xor  #imm,rd
     *  shlr.w rs                     xor  rs,rd
     *  shlr.l rs                  *  xor.w #imm,rd
        sleep                      *  xor.w rs,rd
        stc  ccr,rd                *  xor.l #imm,rd
     *  stc  ccr,@rs               *  xor.l rs,rd
     *  stc  ccr,@(disp:16,rd)        xorc #imm,ccr
```

   Four H8/300 instructions (add, cmp, mov, sub) are defined with variants using the suffixes .b, .w, and .l to specify the size of a memory operand. as supports these suffixes, but does not require them; since one of the operands is always a register, as can deduce the correct size.

   For example, since r0 refers to a 16-bit register,

```
     mov    r0,@foo
```

is equivalent to

```
     mov.w  r0,@foo
```

   If you use the size suffixes, as issues a warning when the suffix and the register size do not match.

1.142 as.guide/H8-500-Dependent

```
                H8/500 Dependent Features
=========================
```

```
                 H8-500 Options
                              Options
```

1.143 as.guide/H8-500 Options

Options
----

as has no additional command-line options for the Hitachi H8/500 family.

1.144 as.guide/H8-500 Syntax

Syntax

1.145 as.guide/H8-500-Chars

Special Characters
....

! is the line comment character.

; can be used instead of a newline to separate statements.

Since $ has no special meaning, you may use it in symbol names.

1.146 as.guide/H8-500-Regs

Register Names
....

You can use the predefined symbols r0, r1, r2, r3, r4, r5, r6, and r7 to refer to the H8/500 registers.

The H8/500 also has these control registers:

cp
code pointer

dp
data pointer

bp
base pointer

tp
stack top pointer

ep
extra pointer

sr
status register

ccr
condition code register

All registers are 16 bits long. To represent 32 bit numbers, use two adjacent registers; for distant memory addresses, use one of the segment pointers (cp for the program counter; dp for r0-r3; ep for r4 and r5; and tp for r6 and r7.

1.147 as.guide/H8-500-Addressing

Addressing Modes
....

as understands the following addressing modes for the H8/500:

Rn
Register direct

@Rn
Register indirect

```
@(d:8, Rn)
    Register indirect with 8 bit signed displacement

@(d:16, Rn)
    Register indirect with 16 bit signed displacement

@-Rn
    Register indirect with pre-decrement

@Rn+
    Register indirect with post-increment

@aa:8
    8 bit absolute address

@aa:16
    16 bit absolute address

#xx:8
    8 bit immediate

#xx:16
    16 bit immediate
```

1.148 as.guide/H8-500 Floating Point

```
Floating Point
--------------

   The H8/500 family uses IEEE floating-point numbers.
```

1.149 as.guide/H8-500 Directives

```
H8/500 Machine Directives
-------------------------

   as has no machine-dependent directives for the H8/500.  However, on
this platform the .int and .word directives generate 16-bit numbers.
```

1.150 as.guide/H8-500 Opcodes

```
Opcodes
-------

   For detailed information on the H8/500 machine instruction set, see
‘H8/500 Series Programming Manual’ (Hitachi M21T001).
```

as implements all the standard H8/500 opcodes. No additional pseudo-instructions are needed on this family.

The following table summarizes H8/500 opcodes and their operands:

```
Legend:
abs8      8-bit absolute address
abs16     16-bit absolute address
abs24     24-bit absolute address
crb       ccr, br, ep, dp, tp, dp
disp8     8-bit displacement
ea        rn,  @rn,  @(d:8, rn),  @(d:16, rn),
           @-rn,  @rn+,  @aa:8,  @aa:16,
          #xx:8, #xx:16
ea_mem     @rn,  @(d:8, rn),  @(d:16, rn),
           @-rn,  @rn+,  @aa:8,  @aa:16
ea_noimm  rn,  @rn,  @(d:8, rn),  @(d:16, rn),
           @-rn,  @rn+,  @aa:8,  @aa:16
fp        r6
imm4      4-bit immediate data
imm8      8-bit immediate data
imm16     16-bit immediate data
pcrel8    8-bit offset from program counter
pcrel16   16-bit offset from program counter
qim       -2, -1, 1, 2
rd        any register
rs        a register distinct from rd
rlist     comma-separated list of registers in parentheses;
          register ranges rd-rs are allowed
sp        stack pointer (r7)
sr        status register
sz        size; .b or .w.  If omitted, default .w

ldc[.b] ea,crb                bcc[.w] pcrel16
ldc[.w] ea,sr                 bcc[.b] pcrel8
add[:q] sz qim,ea_noimm       bhs[.w] pcrel16
add[:g] sz ea,rd              bhs[.b] pcrel8
adds sz ea,rd                 bcs[.w] pcrel16
addx sz ea,rd                 bcs[.b] pcrel8
and sz ea,rd                  blo[.w] pcrel16
andc[.b] imm8,crb             blo[.b] pcrel8
andc[.w] imm16,sr             bne[.w] pcrel16
bpt                           bne[.b] pcrel8
bra[.w] pcrel16               beq[.w] pcrel16
bra[.b] pcrel8                beq[.b] pcrel8
bt[.w] pcrel16                bvc[.w] pcrel16
bt[.b] pcrel8                 bvc[.b] pcrel8
brn[.w] pcrel16               bvs[.w] pcrel16
brn[.b] pcrel8                bvs[.b] pcrel8
bf[.w] pcrel16                bpl[.w] pcrel16
bf[.b] pcrel8                 bpl[.b] pcrel8
bhi[.w] pcrel16               bmi[.w] pcrel16
bhi[.b] pcrel8                bmi[.b] pcrel8
bls[.w] pcrel16               bge[.w] pcrel16
bls[.b] pcrel8                bge[.b] pcrel8
```

```
blt[.w] pcrel16
blt[.b] pcrel8
bgt[.w] pcrel16
bgt[.b] pcrel8
ble[.w] pcrel16
ble[.b] pcrel8
bclr sz imm4,ea_noimm
bclr sz rs,ea_noimm
bnot sz imm4,ea_noimm
bnot sz rs,ea_noimm
bset sz imm4,ea_noimm
bset sz rs,ea_noimm
bsr[.b] pcrel8
bsr[.w] pcrel16
btst sz imm4,ea_noimm
btst sz rs,ea_noimm
clr sz ea
cmp[:e][.b] imm8,rd
cmp[:i][.w] imm16,rd
cmp[:g].b imm8,ea_noimm
cmp[:g][.w] imm16,ea_noimm
Cmp[:g] sz ea,rd
dadd rs,rd
divxu sz ea,rd
dsub rs,rd
exts[.b] rd
extu[.b] rd
jmp @rd
jmp @(imm8,rd)
jmp @(imm16,rd)
jmp abs16
jsr @rd
jsr @(imm8,rd)
jsr @(imm16,rd)
jsr abs16
ldm @sp+,(rlist)
link fp,imm8
link fp,imm16
mov[:e][.b] imm8,rd
mov[:i][.w] imm16,rd
mov[:l][.w] abs8,rd
mov[:l].b abs8,rd
mov[:s][.w] rs,abs8
mov[:s].b rs,abs8
mov[:f][.w] @(disp8,fp),rd
mov[:f][.w] rs,@(disp8,fp)
mov[:f].b @(disp8,fp),rd
mov[:f].b rs,@(disp8,fp)
mov[:g] sz rs,ea_mem
mov[:g] sz ea,rd
mov[:g][.b] imm8,ea_mem
mov[:g][.w] imm16,ea_mem
movfpe[.b] ea,rd
movtpe[.b] rs,ea_noimm
mulxu sz ea,rd
neg sz ea
nop
not sz ea
or sz ea,rd
orc[.b] imm8,crb
orc[.w] imm16,sr
pjmp abs24
pjmp @rd
pjsr abs24
pjsr @rd
prtd imm8
prtd imm16
prts
rotl sz ea
rotr sz ea
rotxl sz ea
rotxr sz ea
rtd imm8
rtd imm16
rts
scb/f rs,pcrel8
scb/ne rs,pcrel8
scb/eq rs,pcrel8
shal sz ea
shar sz ea
shll sz ea
shlr sz ea
sleep
stc[.b] crb,ea_noimm
stc[.w] sr,ea_noimm
stm (rlist),@-sp
sub sz ea,rd
subs sz ea,rd
subx sz ea,rd
swap[.b] rd
tas[.b] ea
trapa imm4
trap/vs
tst sz ea
unlk fp
xch[.w] rs,rd
xor sz ea,rd
xorc.b imm8,crb
xorc.w imm16,sr
```

1.151 as.guide/SH-Dependent

```
                Hitachi SH Dependent Features
============================


                SH Options
                              Options

                SH Syntax
                               Syntax

                SH Floating Point
                       Floating Point

                SH Directives
                           SH Machine Directives

                SH Opcodes
                              Opcodes
```

1.152 as.guide/SH Options

```
Options
-------

   as has no additional command-line options for the Hitachi SH family.
```

1.153 as.guide/SH Syntax

```
                Syntax
------


                SH-Chars
                                Special Characters

                SH-Regs
                                 Register Names

                SH-Addressing
                           Addressing Modes
```

1.154 as.guide/SH-Chars

```
Special Characters
..................

   ! is the line comment character.

   You can use ; instead of a newline to separate statements.

   Since $ has no special meaning, you may use it in symbol names.
```

1.155 as.guide/SH-Regs

```
Register Names
..............

   You can use the predefined symbols r0, r1, r2, r3, r4, r5, r6, r7,
r8, r9, r10, r11, r12, r13, r14, and r15 to refer to the SH
registers.

   The SH also has these control registers:

pr
     procedure register (holds return address)

pc
     program counter

mach
macl
     high and low multiply accumulator registers

sr
     status register

gbr
     global base register

vbr
     vector base register (for interrupt vectors)
```

1.156 as.guide/SH-Addressing

```
Addressing Modes
................

   as understands the following addressing modes for the SH.   Rn in
the following refers to any of the numbered registers, but not the
control registers.

 Rn
```

```
     Register direct

 @Rn
     Register indirect

 @-Rn
     Register indirect with pre-decrement

 @Rn+
     Register indirect with post-increment

 @(disp, Rn)
     Register indirect with displacement

 @(R0, Rn)
     Register indexed

 @(disp, GBR)
     GBR offset

 @(R0, GBR)
     GBR indexed

addr
 @(disp, PC)
     PC relative address (for branch or for addressing memory).  The as
     implementation allows you to use the simpler form addr anywhere a
     PC relative address is called for; the alternate form is supported
     for compatibility with other assemblers.

 #imm
     Immediate data
```

1.157 as.guide/SH Floating Point

```
Floating Point
--------------

   The SH family uses IEEE floating-point numbers.
```

1.158 as.guide/SH Directives

```
SH Machine Directives
---------------------

   as has no machine-dependent directives for the SH.
```

1.159 as.guide/SH Opcodes

```
Opcodes
-------

   For detailed information on the SH machine instruction set, see
'SH-Microcomputer User's Manual' (Hitachi Micro Systems, Inc.).

   as implements all the standard SH opcodes.  No additional
pseudo-instructions are needed on this family.  Note, however, that
because as supports a simpler form of PC-relative addressing, you may
simply write (for example)

     mov.l  bar,r0

where other assemblers might require an explicit displacement to bar
from the program counter:

     mov.l  @(disp, PC)

   Here is a summary of SH opcodes:

     Legend:
     Rn         a numbered register
     Rm         another numbered register
     #imm       immediate data
     disp       displacement
     disp8      8-bit displacement
     disp12     12-bit displacement

     add #imm,Rn                    lds.l @Rn+,PR
     add Rm,Rn                      mac.w @Rm+,@Rn+
     addc Rm,Rn                     mov #imm,Rn
     addv Rm,Rn                     mov Rm,Rn
     and #imm,R0                    mov.b Rm,@(R0,Rn)
     and Rm,Rn                      mov.b Rm,@-Rn
     and.b #imm,@(R0,GBR)           mov.b Rm,@Rn
     bf disp8                       mov.b @(disp,Rm),R0
     bra disp12                     mov.b @(disp,GBR),R0
     bsr disp12                     mov.b @(R0,Rm),Rn
     bt disp8                       mov.b @Rm+,Rn
     clrm                           mov.b @Rm,Rn
     clrt                           mov.b R0,@(disp,Rm)
     cmp/eq #imm,R0                 mov.b R0,@(disp,GBR)
     cmp/eq Rm,Rn                   mov.l Rm,@(disp,Rn)
     cmp/ge Rm,Rn                   mov.l Rm,@(R0,Rn)
     cmp/gt Rm,Rn                   mov.l Rm,@-Rn
     cmp/hi Rm,Rn                   mov.l Rm,@Rn
     cmp/hs Rm,Rn                   mov.l @(disp,Rn),Rm
     cmp/pl Rn                      mov.l @(disp,GBR),R0
     cmp/pz Rn                      mov.l @(disp,PC),Rn
     cmp/str Rm,Rn                  mov.l @(R0,Rm),Rn
     div0s Rm,Rn                    mov.l @Rm+,Rn
     div0u                          mov.l @Rm,Rn
     div1 Rm,Rn                     mov.l R0,@(disp,GBR)
     exts.b Rm,Rn                   mov.w Rm,@(R0,Rn)
```

```
exts.w Rm,Rn                    mov.w Rm,@-Rn
extu.b Rm,Rn                    mov.w Rm,@Rn
extu.w Rm,Rn                    mov.w @(disp,Rm),R0
jmp @Rn                         mov.w @(disp,GBR),R0
jsr @Rn                         mov.w @(disp,PC),Rn
ldc Rn,GBR                      mov.w @(R0,Rm),Rn
ldc Rn,SR                       mov.w @Rm+,Rn
ldc Rn,VBR                      mov.w @Rm,Rn
ldc.l @Rn+,GBR                  mov.w R0,@(disp,Rm)
ldc.l @Rn+,SR                   mov.w R0,@(disp,GBR)
ldc.l @Rn+,VBR                  mova @(disp,PC),R0
lds Rn,MACH                     movt Rn
lds Rn,MACL                     muls Rm,Rn
lds Rn,PR                       mulu Rm,Rn
lds.l @Rn+,MACH                 neg Rm,Rn
lds.l @Rn+,MACL                 negc Rm,Rn

nop                             stc VBR,Rn
not Rm,Rn                       stc.l GBR,@-Rn
or #imm,R0                      stc.l SR,@-Rn
or Rm,Rn                        stc.l VBR,@-Rn
or.b #imm,@(R0,GBR)             sts MACH,Rn
rotcl Rn                        sts MACL,Rn
rotcr Rn                        sts PR,Rn
rotl Rn                         sts.l MACH,@-Rn
rotr Rn                         sts.l MACL,@-Rn
rte                             sts.l PR,@-Rn
rts                             sub Rm,Rn
sett                            subc Rm,Rn
shal Rn                         subv Rm,Rn
shar Rn                         swap.b Rm,Rn
shll Rn                         swap.w Rm,Rn
shll16 Rn                       tas.b @Rn
shll2 Rn                        trapa #imm
shll8 Rn                        tst #imm,R0
shlr Rn                         tst Rm,Rn
shlr16 Rn                       tst.b #imm,@(R0,GBR)
shlr2 Rn                        xor #imm,R0
shlr8 Rn                        xor Rm,Rn
sleep                           xor.b #imm,@(R0,GBR)
stc GBR,Rn                      xtrct Rm,Rn
stc SR,Rn
```

1.160 as.guide/i960-Dependent

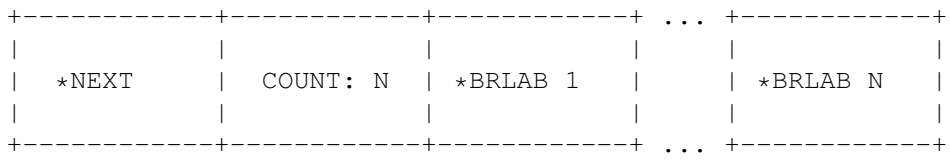
```
                Intel 80960 Dependent Features
==============================


                Options-i960
                                i960 Command-line Options

                Floating Point-i960
```

```
                        Floating Point

                Directives-i960
                             i960 Machine Directives

                Opcodes for i960
                            i960 Opcodes
```

1.161 as.guide/Options-i960

```
i960 Command-line Options
-------------------------

-ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC
     Select the 80960 architecture.  Instructions or features not
     supported by the selected architecture cause fatal errors.

     -ACA is equivalent to -ACA_A; -AKC is equivalent to -AMC.
     Synonyms are provided for compatibility with other tools.

     If none of these options is specified, as will generate code for
     any instruction or feature that is supported by some version of the
     960 (even if this means mixing architectures!).  In principle, as
     will attempt to deduce the minimal sufficient processor type if
     none is specified; depending on the object code format, the
     processor type may be recorded in the object file.  If it is
     critical that the as output match a specific architecture, specify
     that architecture explicitly.

-b
     Add code to collect information about conditional branches taken,
     for later optimization using branch prediction bits.  (The
     conditional branch instructions have branch prediction bits in the
     CA, CB, and CC architectures.)  If BR represents a conditional
     branch instruction, the following represents the code generated by
     the assembler when -b is specified:

                  call    increment routine
                  .word   0       # pre-counter
          Label:  BR
                  call    increment routine
                  .word   0       # post-counter

     The counter following a branch records the number of times that
     branch was not taken; the differenc between the two counters is the
     number of times the branch was taken.

     A table of every such Label is also generated, so that the
     external postprocessor gbr960 (supplied by Intel) can locate all
     the counters.  This table is always labelled __BRANCH_TABLE__;
     this is a local symbol to permit collecting statistics for many
     separate object files.  The table is word aligned, and begins with
     a two-word header.  The first word, initialized to 0, is used in
```

maintaining linked lists of branch tables. The second word is a count of the number of entries in the table, which follow immediately: each is a word, pointing to one of the labels illustrated above.



__BRANCH_TABLE__ layout

The first word of the header is used to locate multiple branch tables, since each object file may contain one. Normally the links are maintained with a call to an initialization routine, placed at the beginning of each function in the file. The GNU C compiler will generate these calls automatically when you give it a -b option. For further details, see the documentation of gbr960.

-norelax
Normally, Compare-and-Branch instructions with targets that require displacements greater than 13 bits (or that have external targets) are replaced with the corresponding compare (or chkbit) and branch instructions. You can use the -norelax option to specify that as should generate errors instead, if the target displacement is larger than 13 bits.

This option does not affect the Compare-and-Jump instructions; the code emitted for them is always adjusted when necessary (depending on displacement size), regardless of whether you use -norelax.

1.162 as.guide/Floating Point-i960

Floating Point
---

as generates IEEE floating-point numbers for the directives .float, .double, .extended, and .single.

1.163 as.guide/Directives-i960

i960 Machine Directives

.bss symbol, length, align
Reserve length bytes in the bss section for a local symbol, aligned to the power of two specified by align. length and align must be positive absolute expressions. This directive differs from .lcomm only in that it permits you to specify an alignment.

```
     See
                .lcomm
                .

.extended flonums
     .extended expects zero or more flonums, separated by commas; for
     each flonum, .extended emits an IEEE extended-format (80-bit)
     floating-point number.

.leafproc call-lab, bal-lab
     You can use the .leafproc directive in conjunction with the
     optimized callj instruction to enable faster calls of leaf
     procedures.  If a procedure is known to call no other procedures,
     you may define an entry point that skips procedure prolog code
     (and that does not depend on system-supplied saved context), and
     declare it as the bal-lab using .leafproc.  If the procedure also
     has an entry point that goes through the normal prolog, you can
     specify that entry point as call-lab.

     A .leafproc declaration is meant for use in conjunction with the
     optimized call instruction callj; the directive records the data
     needed later to choose between converting the callj into a bal or
     a call.

     call-lab is optional; if only one argument is present, or if the
     two arguments are identical, the single argument is assumed to be
     the bal entry point.

.sysproc name, index
     The .sysproc directive defines a name for a system procedure.
     After you define it using .sysproc, you can use name to refer to
     the system procedure identified by index when calling procedures
     with the optimized call instruction callj.

     Both arguments are required; index must be between 0 and 31
     (inclusive).
```

1.164 as.guide/Opcodes for i960

```
                i960 Opcodes
------------

   All Intel 960 machine instructions are supported; see

                i960 Command-line Options
                 for a discussion of selecting the instruction
subset for a particular 960 architecture.

   Some opcodes are processed beyond simply emitting a single
corresponding instruction: callj, and Compare-and-Branch or
Compare-and-Jump instructions with target displacements larger than 13
bits.
```

1.165 as.guide/callj-i960

callj
.....

You can write callj to have the assembler or the linker determine the most appropriate form of subroutine call: call, bal, or calls. If the assembly source contains enough information--a .leafproc or .sysproc directive defining the operand--then as will translate the callj; if not, it will simply emit the callj, leaving it for the linker to resolve.

1.166 as.guide/Compare-and-branch-i960

Compare-and-Branch
....

The 960 architectures provide combined Compare-and-Branch instructions that permit you to store the branch target in the lower 13 bits of the instruction word itself. However, if you specify a branch target far enough away that its address won't fit in 13 bits, the assembler can either issue an error, or convert your Compare-and-Branch instruction into separate instructions to do the compare and the branch.

Whether as gives an error or expands the instruction depends on two choices you can make: whether you use the -norelax option, and whether you use a "Compare and Branch" instruction or a "Compare and Jump" instruction. The "Jump" instructions are always expanded if necessary; the "Branch" instructions are expanded when necessary unless you specify -norelax--in which case as gives an error instead.

These are the Compare-and-Branch instructions, their "Jump" variants, and the instruction pairs they may expand into:

Branch	Compare and Jump	Expanded to
bbc		chkbit; bno
bbs		chkbit; bo
cmpibe	cmpije	cmpi; be
cmpibg	cmpijg	cmpi; bg

```
cmpibge   cmpijge       cmpi; bge
 cmpibl    cmpijl       cmpi; bl
cmpible   cmpijle       cmpi; ble
cmpibno   cmpijno       cmpi; bno
cmpibne   cmpijne       cmpi; bne
 cmpibo    cmpijo       cmpi; bo
 cmpobe    cmpoje       cmpo; be
 cmpobg    cmpojg       cmpo; bg
cmpobge   cmpojge       cmpo; bge
 cmpobl    cmpojl       cmpo; bl
cmpoble   cmpojle       cmpo; ble
cmpobne   cmpojne       cmpo; bne
```

1.167 as.guide/M68K-Dependent

```
                M680x0 Dependent Features
=========================



                M68K-Opts
                                  M680x0 Options

                M68K-Syntax
                                 Syntax

                M68K-Moto-Syntax
                            Motorola Syntax

                M68K-Float
                                  Floating Point

                M68K-Directives
                             680x0 Machine Directives

                M68K-opcodes
                                Opcodes
```

1.168 as.guide/M68K-Opts

```
M680x0 Options
--------------
```

The Motorola 680x0 version of as has two machine dependent options. One shortens undefined references from 32 to 16 bits, while the other is used to tell as what kind of machine it is assembling for.

You can use the -l option to shorten the size of references to undefined symbols. If the -l option is not given, references to

undefined symbols will be a full long (32 bits) wide. (Since as cannot know where these symbols will end up, as can only allocate space for the linker to fill in later. Since as doesn't know how far away these symbols will be, it allocates as much space as it can.) If this option is given, the references will only be one word wide (16 bits). This may be useful if you want the object file to be as small as possible, and you know that the relevant symbols will be less than 17 bits away.

The 680x0 version of as is most frequently used to assemble programs for the Motorola MC68020 microprocessor. Occasionally it is used to assemble programs for the mostly similar, but slightly different MC68000 or MC68010 microprocessors. You can give as the options -m68000, -mc68000, -m68010, -mc68010, -m68020, and -mc68020 to tell it what processor is the target.

1.169 as.guide/M68K-Syntax

Syntax

This syntax for the Motorola 680x0 was developed at MIT.

The 680x0 version of as uses syntax similar to the Sun assembler. Intervening periods are now ignored; for example, movl is equivalent to move.l.

In the following table apc stands for any of the address registers (a0 through a7), nothing, (), the Program Counter (pc), or the zero-address relative to the program counter (zpc).

The following addressing modes are understood:

Immediate
#digits

Data Register
d0 through d7

Address Register
a0 through a7

Address Register Indirect
a0@ through a7@
a7 is also known as sp, i.e. the Stack Pointer. a6 is also known as fp, the Frame Pointer.

Address Register Postincrement
a0@+ through a7@+

Address Register Predecrement
a0@- through a7@-

Indirect Plus Offset
apc @(digits)

```
Index
     apc @(digits,register:size:scale)

     or apc @(register:size:scale)

Postindex
     apc @(digits)@(digits,register:size:scale)

     or apc @(digits)@(register:size:scale)

Preindex
     apc @(digits,register:size:scale)@(digits)

     or apc @(register:size:scale)@(digits)

Memory Indirect
     apc @(digits)@(digits)

Absolute
     symbol, or digits
```

   For some configurations, especially those where the compiler normally does not prepend an underscore to the names of user variables, the assembler requires a % before any use of a register name. This is intended to let the assembler distinguish between user variables and registers named a0 through a7, et cetera. The % is always accepted, but is only required for some configurations, notably m68k-coff.

1.170 as.guide/M68K-Moto-Syntax

```
                Motorola Syntax
---------------
```

   The standard Motorola syntax for this chip differs from the syntax already discussed (see
                Syntax
). as can accept both kinds of syntax, even within a single instruction. The syntaxes are fully compatible, because the Motorola syntax never uses the @ character and the MIT syntax always does, except in cases where the syntaxes are identical.

   In particular, you may write or generate M68K assembler with the following conventions:

   (In the following table apc stands for any of the address registers (a0 through a7), nothing, (), the Program Counter (pc), or the zero-address relative to the program counter (zpc).)

   The following additional addressing modes are understood:

```
Address Register Indirect
     a0 through a7
     a7 is also known as sp, i.e. the Stack Pointer.  a6 is also known
     as fp, the Frame Pointer.
```

```
Address Register Postincrement
     (a0)+ through (a7)+

Address Register Predecrement
     -(a0) through -(a7)

Indirect Plus Offset
     digits(apc)

Index
     digits(apc,(register.size*scale)
     or  (apc,register.size*scale)
     In either case, size and scale are optional (scale defaults to 1,
     size defaults to l).   scale can be 1, 2, 4, or 8.   size can be w
     or l.  scale is only supported on the 68020 and greater.
```

1.171 as.guide/M68K-Float

```
Floating Point
--------------

   The floating point code is not too well tested, and may have subtle
bugs in it.

   Packed decimal (P) format floating literals are not supported.  Feel
free to add the code!

   The floating point formats generated by directives are these.

.float
     Single precision floating point constants.

.double
     Double precision floating point constants.

   There is no directive to produce regions of memory holding extended
precision numbers, however they can be used as immediate operands to
floating-point instructions.  Adding a directive to create extended
precision numbers would not be hard, but it has not yet seemed
necessary.
```

1.172 as.guide/M68K-Directives

```
680x0 Machine Directives
------------------------

   In order to be compatible with the Sun assembler the 680x0 assembler
understands the following directives.
```

.data1
     This directive is identical to a .data 1 directive.

.data2
     This directive is identical to a .data 2 directive.

.even
     This directive is identical to a .align 1 directive.

.skip
     This directive is identical to a .space directive.

1.173 as.guide/M68K-opcodes

Opcodes

M68K-Branch
Branch Improvement

M68K-Chars
Special Characters

1.174 as.guide/M68K-Branch

Branch Improvement
....

Certain pseudo opcodes are permitted for branch instructions. They expand to the shortest branch instruction that will reach the target. Generally these mnemonics are made by substituting j for b at the start of a Motorola mnemonic.

The following table summarizes the pseudo-operations. A * flags cases that are more fully described after the table:

		Displacement			
		68020	68000/10		
	Pseudo-Op	BYTE	WORD	LONG	LONG	non-PC relative
	jbsr	bsrs	bsr	bsrl	jsr	jsr
	jra	bras	bra	bral	jmp	jmp
*	jXX	bXXs	bXX	bXXl	bNXs;jmpl	bNXs;jmp
*	dbXX	dbXX	dbXX	dbXX; bra; jmpl		
*	fjXX	fbXXw	fbXXw	fbXXl		fbNXw;jmp

```
     XX: condition
     NX: negative of condition XX

                 *--see full description below

jbsr
jra
     These are the simplest jump pseudo-operations; they always map to
     one particular machine instruction, depending on the displacement
     to the branch target.

 jXX
     Here,  jXX stands for an entire family of pseudo-operations, where
     XX is a conditional branch or condition-code test.  The full list
     of pseudo-ops in this family is:
           jhi   jls   jcc   jcs   jne   jeq   jvc
           jvs   jpl   jmi   jge   jlt   jgt   jle

     For the cases of non-PC relative displacements and long
     displacements on the 68000 or 68010, as will issue a longer code
     fragment in terms of NX, the opposite condition to XX.  For
     example, for the non-PC relative case:
              jXX foo
     gives
               bNXs oof
               jmp foo
           oof:

dbXX
     The full family of pseudo-operations covered here is
           dbhi   dbls   dbcc   dbcs   dbne   dbeq   dbvc
           dbvs   dbpl   dbmi   dbge   dblt   dbgt   dble
           dbf    dbra   dbt

     Other than for word and byte displacements, when the source reads
     dbXX foo, as will emit
               dbXX oo1
               bra oo2
           oo1:jmpl foo
           oo2:

fjXX
     This family includes
           fjne   fjeq   fjge   fjlt   fjgt   fjle   fjf
           fjt    fjgl   fjgle  fjnge  fjngl  fjngle fjngt
           fjnle  fjnlt  fjoge  fjogl  fjogt  fjole  fjolt
           fjor   fjseq  fjsf   fjsne  fjst   fjueq  fjuge
           fjugt  fjule  fjult  fjun

     For branch targets that are not PC relative, as emits
               fbNX oof
               jmp foo
           oof:
     when it encounters fjXX foo.
```

1.175 as.guide/M68K-Chars

```
Special Characters
..................
```

   The immediate character is # for Sun compatibility. The line-comment character is |. If a # appears at the beginning of a line, it is treated as a comment unless it looks like # line file, in which case it is treated normally.

1.176 as.guide/Sparc-Dependent

```
                SPARC Dependent Features
========================
```

```
                Sparc-Opts
                                  Options

                Sparc-Float
                                Floating Point

                Sparc-Directives
                            Sparc Machine Directives
```

1.177 as.guide/Sparc-Opts

```
Options
-------
```

   The SPARC chip family includes several successive levels (or other variants) of chip, using the same core instruction set, but including a few additional instructions at each level.

   By default, as assumes the core instruction set (SPARC v6), but "bumps" the architecture level as needed: it switches to successively higher architectures as it encounters instructions that only exist in the higher levels.

```
-Av6 | -Av7 | -Av8 | -Asparclite
     Use one of the -A options to select one of the SPARC architectures
     explicitly.  If you select an architecture explicitly, as reports
     a fatal error if it encounters an instruction or feature requiring
     a higher level.

-bump
     Permit the assembler to "bump" the architecture level as required,
     but warn whenever it is necessary to switch to another level.
```

1.178 as.guide/Sparc-Float

```
Floating Point
--------------

   The Sparc uses IEEE floating-point numbers.
```

1.179 as.guide/Sparc-Directives

```
Sparc Machine Directives
------------------------

   The Sparc version of as supports the following additional machine
directives:

.common
     This must be followed by a symbol name, a positive number, and
     "bss".  This behaves somewhat like .comm, but the syntax is
     different.

.half
     This is functionally identical to .short.

.proc
     This directive is ignored.  Any text following it on the same line
     is also ignored.

.reserve
     This must be followed by a symbol name, a positive number, and
     "bss".  This behaves somewhat like .lcomm, but the syntax is
     different.

.seg
     This must be followed by "text", "data", or "data1".  It behaves
     like .text, .data, or .data 1.

.skip
     This is functionally identical to the .space directive.

.word
     On the Sparc, the .word directive produces 32 bit values, instead
     of the 16 bit values it produces on many other machines.
```

1.180 as.guide/i386-Dependent

```
                80386 Dependent Features
=========================


                i386-Options
                                Options

                i386-Syntax
                                 AT&T Syntax versus Intel Syntax

                i386-Opcodes
                                Opcode Naming

                i386-Regs
                                   Register Naming

                i386-prefixes
                               Opcode Prefixes

                i386-Memory
                                 Memory References

                i386-jumps
                                  Handling of Jump Instructions

                i386-Float
                                  Floating Point

                i386-Notes
                                  Notes
```

1.181 as.guide/i386-Options

```
Options
-------

   The 80386 has no machine dependent options.
```

1.182 as.guide/i386-Syntax

```
AT&T Syntax versus Intel Syntax
-------------------------------

   In order to maintain compatibility with the output of gcc, as
supports AT&T System V/386 assembler syntax.  This is quite different
from Intel syntax.  We mention these differences because almost all
80386 documents used only Intel syntax.  Notable differences between
```

the two syntaxes are:

* AT&T immediate operands are preceded by $; Intel immediate operands are undelimited (Intel push 4 is AT&T pushl $4). AT&T register operands are preceded by %; Intel register operands are undelimited. AT&T absolute (as opposed to PC relative) jump/call operands are prefixed by *; they are undelimited in Intel syntax.

* AT&T and Intel syntax use the opposite order for source and destination operands. Intel add eax, 4 is addl $4, %eax. The source, dest convention is maintained for compatibility with previous Unix assemblers.

* In AT&T syntax the size of memory operands is determined from the last character of the opcode name. Opcode suffixes of b, w, and l specify byte (8-bit), word (16-bit), and long (32-bit) memory references. Intel syntax accomplishes this by prefixes memory operands (not the opcodes themselves) with byte ptr, word ptr, and dword ptr. Thus, Intel mov al, byte ptr foo is movb foo, %al in AT&T syntax.

* Immediate form long jumps and calls are lcall/ljmp $section, $offset in AT&T syntax; the Intel syntax is call/jmp far section:offset. Also, the far return instruction is lret $stack-adjust in AT&T syntax; Intel syntax is ret far stack-adjust.

* The AT&T assembler does not provide support for multiple section programs. Unix style systems expect all programs to be single sections.

1.183 as.guide/i386-Opcodes

Opcode Naming

Opcode names are suffixed with one character modifiers which specify the size of operands. The letters b, w, and l specify byte, word, and long operands. If no suffix is specified by an instruction and it contains no memory operands then as tries to fill in the missing suffix based on the destination register operand (the last one by convention). Thus, mov %ax, %bx is equivalent to movw %ax, %bx; also, mov $1, %bx is equivalent to movw $1, %bx. Note that this is incompatible with the AT&T Unix assembler which assumes that a missing opcode suffix implies long operand size. (This incompatibility does not affect compiler output since compilers always explicitly specify the opcode suffix.)

Almost all opcodes have the same names in AT&T and Intel format. There are a few exceptions. The sign extend and zero extend instructions need two sizes to specify them. They need a size to sign/zero extend from and a size to zero extend to. This is accomplished by using two opcode suffixes in AT&T syntax. Base names for sign extend and zero extend are movs... and movz... in AT&T syntax (movsx and movzx in Intel syntax). The opcode suffixes are tacked

on to this base name, the from suffix before the to suffix. Thus, movsbl %al, %edx is AT&T syntax for "move sign extend from %al to %edx." Possible suffixes, thus, are bl (from byte to long), bw (from byte to word), and wl (from word to long).

The Intel-syntax conversion instructions

* cbw -- sign-extend byte in %al to word in %ax,

* cwde -- sign-extend word in %ax to long in %eax,

* cwd -- sign-extend word in %ax to long in %dx:%ax,

* cdq -- sign-extend dword in %eax to quad in %edx:%eax,

are called cbtw, cwtl, cwtd, and cltd in AT&T naming. as accepts either naming for these instructions.

Far call/jump instructions are lcall and ljmp in AT&T syntax, but are call far and jump far in Intel convention.

1.184 as.guide/i386-Regs

Register Naming

Register operands are always prefixes with %. The 80386 registers consist of

* the 8 32-bit registers %eax (the accumulator), %ebx, %ecx, %edx, %edi, %esi, %ebp (the frame pointer), and %esp (the stack pointer).

* the 8 16-bit low-ends of these: %ax, %bx, %cx, %dx, %di, %si, %bp, and %sp.

* the 8 8-bit registers: %ah, %al, %bh, %bl, %ch, %cl, %dh, and %dl (These are the high-bytes and low-bytes of %ax, %bx, %cx, and %dx)

* the 6 section registers %cs (code section), %ds (data section), %ss (stack section), %es, %fs, and %gs.

* the 3 processor control registers %cr0, %cr2, and %cr3.

* the 6 debug registers %db0, %db1, %db2, %db3, %db6, and %db7.

* the 2 test registers %tr6 and %tr7.

* the 8 floating point register stack %st or equivalently %st(0), %st(1), %st(2), %st(3), %st(4), %st(5), %st(6), and %st(7).

1.185 as.guide/i386-prefixes

```
Opcode Prefixes
---------------

   Opcode prefixes are used to modify the following opcode.  They are
used to repeat string instructions, to provide section overrides, to
perform bus lock operations, and to give operand and address size
(16-bit operands are specified in an instruction by prefixing what would
normally be 32-bit operands with a "operand size" opcode prefix).
Opcode prefixes are usually given as single-line instructions with no
operands, and must directly precede the instruction they act upon.  For
example, the scas (scan string) instruction is repeated with:
             repne
             scas

   Here is a list of opcode prefixes:

   * Section override prefixes cs, ds, ss, es, fs, gs.  These are
     automatically added by specifying using the section:memory-operand
     form for memory references.

   * Operand/Address size prefixes data16 and addr16 change 32-bit
     operands/addresses into 16-bit operands/addresses.  Note that
     16-bit addressing modes (i.e. 8086 and 80286 addressing modes) are
     not supported (yet).

   * The bus lock prefix lock inhibits interrupts during execution of
     the instruction it precedes.  (This is only valid with certain
     instructions; see a 80386 manual for details).

   * The wait for coprocessor prefix wait waits for the coprocessor to
     complete the current instruction.  This should never be needed for
     the 80386/80387 combination.

   * The rep, repe, and repne prefixes are added to string instructions
     to make them repeat %ecx times.
```

1.186 as.guide/i386-Memory

```
Memory References
-----------------

   An Intel syntax indirect memory reference of the form

     section:[base + index*scale + disp]

is translated into the AT&T syntax

     section:disp(base, index, scale)

where base and index are the optional 32-bit base and index registers,
disp is the optional displacement, and scale, taking the values 1, 2,
```

4, and 8, multiplies index to calculate the address of the operand. If no scale is specified, scale is taken to be 1. section specifies the optional section register for the memory operand, and may override the default section register (see a 80386 manual for section register defaults). Note that section overrides in AT&T syntax must have be preceded by a %. If you specify a section override which coincides with the default section register, as will not output any section register override prefixes to assemble the given instruction. Thus, section overrides can be specified to emphasize which section register is used for a given memory operand.

Here are some examples of Intel and AT&T style memory references:

AT&T: -4(%ebp), Intel: [ebp - 4]
base is %ebp; disp is -4. section is missing, and the default section is used (%ss for addressing with %ebp as the base register). index, scale are both missing.

AT&T: foo(,%eax,4), Intel: [foo + eax*4]
index is %eax (scaled by a scale 4); disp is foo. All other fields are missing. The section register here defaults to %ds.

AT&T: foo(,1); Intel [foo]
This uses the value pointed to by foo as a memory operand. Note that base and index are both missing, but there is only one ,. This is a syntactic exception.

AT&T: %gs:foo; Intel gs:foo
This selects the contents of the variable foo with section register section being %gs.

Absolute (as opposed to PC relative) call and jump operands must be prefixed with *. If no * is specified, as will always choose PC relative addressing for jump/call labels.

Any instruction that has a memory operand must specify its size (byte, word, or long) with an opcode suffix (b, w, or l, respectively).

1.187 as.guide/i386-jumps

Handling of Jump Instructions

Jump instructions are always optimized to use the smallest possible displacements. This is accomplished by using byte (8-bit) displacement jumps whenever the target is sufficiently close. If a byte displacement is insufficient a long (32-bit) displacement is used. We do not support word (16-bit) displacement jumps (i.e. prefixing the jump instruction with the addr16 opcode prefix), since the 80386 insists upon masking %eip to 16 bits after the word displacement is added.

Note that the jcxz, jecxz, loop, loopz, loope, loopnz and loopne instructions only come in byte displacements, so that it is possible that use of these instructions (gcc does not use them) will cause the

```
assembler to print an error message (and generate incorrect code).  The
AT&T 80386 assembler tries to get around this problem by expanding jcxz
foo to
              jcxz cx_zero
              jmp cx_nonzero
     cx_zero: jmp foo
     cx_nonzero:
```

1.188 as.guide/i386-Float

```
Floating Point
--------------
```

All 80387 floating point types except packed BCD are supported. (BCD support may be added without much difficulty). These data types are 16-, 32-, and 64- bit integers, and single (32-bit), double (64-bit), and extended (80-bit) precision floating point. Each supported type has an opcode suffix and a constructor associated with it. Opcode suffixes specify operand's data types. Constructors build these data types into memory.

* Floating point constructors are .float or .single, .double, and .tfloat for 32-, 64-, and 80-bit formats. These correspond to opcode suffixes s, l, and t. t stands for temporary real, and that the 80387 only supports this format via the fldt (load temporary real to stack top) and fstpt (store temporary real and pop stack) instructions.

* Integer constructors are .word, .long or .int, and .quad for the 16-, 32-, and 64-bit integer formats. The corresponding opcode suffixes are s (single), l (long), and q (quad). As with the temporary real format the 64-bit q format is only present in the fildq (load quad integer to stack top) and fistpq (store quad integer and pop stack) instructions.

Register to register operations do not require opcode suffixes, so that fst %st, %st(1) is equivalent to fstl %st, %st(1).

Since the 80387 automatically synchronizes with the 80386 fwait instructions are almost never needed (this is not the case for the 80286/80287 and 8086/8087 combinations). Therefore, as suppresses the fwait instruction whenever it is implicitly selected by one of the fn... instructions. For example, fsave and fnsave are treated identically. In general, all the fn... instructions are made equivalent to f... instructions. If fwait is desired it must be explicitly coded.

1.189 as.guide/i386-Notes

Notes

There is some trickery concerning the mul and imul instructions that deserves mention. The 16-, 32-, and 64-bit expanding multiplies (base opcode 0xf6; extension 4 for mul and 5 for imul) can be output only in the one operand form. Thus, imul %ebx, %eax does not select the expanding multiply; the expanding multiply would clobber the %edx register, and this would confuse gcc output. Use imul %ebx to get the 64-bit product in %edx:%eax.

We have added a two operand form of imul when the first operand is an immediate mode expression and the second operand is a register. This is just a shorthand, so that, multiplying %eax by 69, for example, can be done with imul $69, %eax rather than imul $69, %eax, %eax.

1.190 as.guide/Z8000-Dependent

Z8000 Dependent Features
=======================

The Z8000 as supports both members of the Z8000 family: the unsegmented Z8002, with 16 bit addresses, and the segmented Z8001 with 24 bit addresses.

When the assembler is in unsegmented mode (specified with the unsegm directive), an address will take up one word (16 bit) sized register. When the assembler is in segmented mode (specified with the segm directive), a 24-bit address takes up a long (32 bit) register. See

Assembler Directives for the Z8000
, for a list of other Z8000 specific assembler directives.

Z8000 Options
No special command-line options for Z8000

Z8000 Syntax
Assembler syntax for the Z8000

Z8000 Directives
Special directives for the Z8000

Z8000 Opcodes
Opcodes

1.191 as.guide/Z8000 Options

```
Options
-------

   as has no additional command-line options for the Zilog Z8000 family.
```

1.192 as.guide/Z8000 Syntax

```
                Syntax
------



                 Z8000-Chars
                                Special Characters

                 Z8000-Regs
                                 Register Names

                 Z8000-Addressing
                            Addressing Modes
```

1.193 as.guide/Z8000-Chars

```
Special Characters
..................

   ! is the line comment character.

   You can use ; instead of a newline to separate statements.
```

1.194 as.guide/Z8000-Regs

```
Register Names
..............

   The Z8000 has sixteen 16 bit registers, numbered 0 to 15.  You can
refer to different sized groups of registers by register number, with
the prefix r for 16 bit registers, rr for 32 bit registers and rq for
64 bit registers.  You can also refer to the contents of the first
eight (of the sixteen 16 bit registers) by bytes.  They are named  rnh
and  rnl.

byte registers
```

```
     r0l r0h r1h r1l r2h r2l r3h r3l
     r4h r4l r5h r5l r6h r6l r7h r7l
word registers
     r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15
long word registers
     rr0 rr2 rr4 rr6 rr8 rr10 rr12 rr14
quad word registers
     rq0 rq4 rq8 rq12
```

1.195 as.guide/Z8000-Addressing

```
Addressing Modes
................

   as understands the following addressing modes for the Z8000:

 rn
     Register direct

 @rn
     Indirect register

addr
     Direct: the 16 bit or 24 bit address (depending on whether the
     assembler is in segmented or unsegmented mode) of the operand is
     in the instruction.

address(rn)
     Indexed: the 16 or 24 bit address is added to the 16 bit register
     to produce the final address in memory of the operand.

 rn(#imm)
     Base Address: the 16 or 24 bit register is added to the 16 bit sign
     extended immediate displacement to produce the final address in
     memory of the operand.

 rn(rm)
     Base Index: the 16 or 24 bit register rn is added to the sign
     extended 16 bit index register rm to produce the final address in
     memory of the operand.

 #xx
     Immediate data xx.
```

1.196 as.guide/Z8000 Directives

```
Assembler Directives for the Z8000
----------------------------------

   The Z8000 port of as includes these additional assembler directives,
```

for compatibility with other Z8000 assemblers. As shown, these do not begin with . (unlike the ordinary as directives).

segm
: Generates code for the segmented Z8001.

unsegm
: Generates code for the unsegmented Z8002.

name
: Synonym for .file

global
: Synonum for .global

wval
: Synonym for .word

lval
: Synonym for .long

bval
: Synonym for .byte

sval
: Assemble a string. sval expects one string literal, delimited by single quotes. It assembles each byte of the string into consecutive addresses. You can use the escape sequence %xx (where xx represents a two-digit hexadecimal number) to represent the character whose ASCII value is xx. Use this feature to describe single quote and other characters that may not appear in string literals as themselves. For example, the C statement char *a = "he said \"it's 50% off\""; is represented in Z8000 assembly language (shown with the assembler output in hex at the left) as

```
68652073    sval    'he said %22it%27s 50%25 off%22%00'
61696420
22697427
73203530
25206F66
662200
```

rsect
: synonym for .section

block
: synonym for .space

even
: synonym for .align 1

1.197 as.guide/Z8000 Opcodes

Opcodes

For detailed information on the Z8000 machine instruction set, see ‘Z8000 Technical Manual’.

The following table summarizes the opcodes and their arguments:

```
rs   16 bit source register
rd   16 bit destination register
rbs   8 bit source register
rbd   8 bit destination register
rrs   32 bit source register
rrd   32 bit destination register
rqs   64 bit source register
rqd   64 bit destination register
addr 16/24 bit address
imm  immediate data
```

```
adc rd,rs
adcb rbd,rbs
add rd,@rs
add rd,addr
add rd,addr(rs)
add rd,imm16
add rd,rs
addb rbd,@rs
addb rbd,addr
addb rbd,addr(rs)
addb rbd,imm8
addb rbd,rbs
addl rrd,@rs
addl rrd,addr
addl rrd,addr(rs)
addl rrd,imm32
addl rrd,rrs
and rd,@rs
and rd,addr
and rd,addr(rs)
and rd,imm16
and rd,rs
andb rbd,@rs
andb rbd,addr
andb rbd,addr(rs)
andb rbd,imm8
andb rbd,rbs
bit @rd,imm4
bit addr(rd),imm4
bit addr,imm4
bit rd,imm4
bit rd,rs
bitb @rd,imm4
bitb addr(rd),imm4
bitb addr,imm4
bitb rbd,imm4
bitb rbd,rs
clrb addr
clrb addr(rd)
clrb rbd
com @rd
com addr
com addr(rd)
com rd
comb @rd
comb addr
comb addr(rd)
comb rbd
comflg flags
cp @rd,imm16
cp addr(rd),imm16
cp addr,imm16
cp rd,@rs
cp rd,addr
cp rd,addr(rs)
cp rd,imm16
cp rd,rs
cpb @rd,imm8
cpb addr(rd),imm8
cpb addr,imm8
cpb rbd,@rs
cpb rbd,addr
cpb rbd,addr(rs)
cpb rbd,imm8
cpb rbd,rbs
cpd rd,@rs,rr,cc
cpdb rbd,@rs,rr,cc
cpdr rd,@rs,rr,cc
cpdrb rbd,@rs,rr,cc
cpi rd,@rs,rr,cc
cpib rbd,@rs,rr,cc
cpir rd,@rs,rr,cc
cpirb rbd,@rs,rr,cc
cpl rrd,@rs
cpsir @rd,@rs,rr,cc
cpsirb @rd,@rs,rr,cc
dab rbd
dbjnz rbd,disp7
dec @rd,imm4m1
dec addr(rd),imm4m1
dec addr,imm4m1
dec rd,imm4m1
decb @rd,imm4m1
decb addr(rd),imm4m1
decb addr,imm4m1
decb rbd,imm4m1
di i2
div rrd,@rs
div rrd,addr
div rrd,addr(rs)
div rrd,imm16
div rrd,rs
divl rqd,@rs
divl rqd,addr
divl rqd,addr(rs)
divl rqd,imm32
divl rqd,rrs
djnz rd,disp7
ei i2
ex rd,@rs
ex rd,addr
ex rd,addr(rs)
ex rd,rs
exb rbd,@rs
exb rbd,addr
exb rbd,addr(rs)
exb rbd,rbs
ext0e imm8
ext0f imm8
ext8e imm8
ext8f imm8
```

```
bpt
call @rd
call addr
call addr(rd)
calr disp12
clr @rd
clr addr
clr addr(rd)
clr rd
clrb @rd
inc addr,imm4m1
inc rd,imm4m1
incb @rd,imm4m1
incb addr(rd),imm4m1
incb addr,imm4m1
incb rbd,imm4m1
ind @rd,@rs,ra
indb @rd,@rs,rba
inib @rd,@rs,ra
inibr @rd,@rs,ra
iret
jp cc,@rd
jp cc,addr
jp cc,addr(rd)
jr cc,disp8
ld @rd,imm16
ld @rd,rs
ld addr(rd),imm16
ld addr(rd),rs
ld addr,imm16
ld addr,rs
ld rd(imm16),rs
ld rd(rx),rs
ld rd,@rs
ld rd,addr
ld rd,addr(rs)
ld rd,imm16
ld rd,rs
ld rd,rs(imm16)
ld rd,rs(rx)
lda rd,addr
lda rd,addr(rs)
lda rd,rs(imm16)
lda rd,rs(rx)
ldar rd,disp16
ldb @rd,imm8
ldb @rd,rbs
ldb addr(rd),imm8
ldb addr(rd),rbs
ldb addr,imm8
ldb addr,rbs
ldb rbd,@rs
ldb rbd,addr
ldb rbd,addr(rs)
ldb rbd,imm8
ldb rbd,rbs
ldb rbd,rs(imm16)
cpl rrd,addr
cpl rrd,addr(rs)
cpl rrd,imm32
cpl rrd,rrs
cpsd @rd,@rs,rr,cc
cpsdb @rd,@rs,rr,cc
cpsdr @rd,@rs,rr,cc
cpsdrb @rd,@rs,rr,cc
cpsi @rd,@rs,rr,cc
cpsib @rd,@rs,rr,cc
ldb rbd,rs(rx)
ldb rd(imm16),rbs
ldb rd(rx),rbs
ldctl ctrl,rs
ldctl rd,ctrl
ldd @rs,@rd,rr
lddb @rs,@rd,rr
lddr @rs,@rd,rr
lddrb @rs,@rd,rr
ldi @rd,@rs,rr
ldib @rd,@rs,rr
ldir @rd,@rs,rr
ldirb @rd,@rs,rr
ldk rd,imm4
ldl @rd,rrs
ldl addr(rd),rrs
ldl addr,rrs
ldl rd(imm16),rrs
ldl rd(rx),rrs
ldl rrd,@rs
ldl rrd,addr
ldl rrd,addr(rs)
ldl rrd,imm32
ldl rrd,rrs
ldl rrd,rs(imm16)
ldl rrd,rs(rx)
ldm @rd,rs,n
ldm addr(rd),rs,n
ldm addr,rs,n
ldm rd,@rs,n
ldm rd,addr(rs),n
ldm rd,addr,n
ldps @rs
ldps addr
ldps addr(rs)
ldr disp16,rs
ldr rd,disp16
ldrb disp16,rbs
ldrb rbd,disp16
ldrl disp16,rrs
ldrl rrd,disp16
mbit
mreq rd
mres
mset
mult rrd,@rs
mult rrd,addr
exts rrd
extsb rd
extsl rqd
halt
in rd,@rs
in rd,imm16
inb rbd,@rs
inb rbd,imm16
inc @rd,imm4m1
inc addr(rd),imm4m1
mult rrd,addr(rs)
mult rrd,imm16
mult rrd,rs
multl rqd,@rs
multl rqd,addr
multl rqd,addr(rs)
multl rqd,imm32
multl rqd,rrs
neg @rd
neg addr
neg addr(rd)
neg rd
negb @rd
negb addr
negb addr(rd)
negb rbd
nop
or rd,@rs
or rd,addr
or rd,addr(rs)
or rd,imm16
or rd,rs
orb rbd,@rs
orb rbd,addr
orb rbd,addr(rs)
orb rbd,imm8
orb rbd,rbs
out @rd,rs
out imm16,rs
outb @rd,rbs
outb imm16,rbs
outd @rd,@rs,ra
outdb @rd,@rs,rba
outib @rd,@rs,ra
outibr @rd,@rs,ra
pop @rd,@rs
pop addr(rd),@rs
pop addr,@rs
pop rd,@rs
popl @rd,@rs
popl addr(rd),@rs
popl addr,@rs
popl rrd,@rs
push @rd,@rs
push @rd,addr
push @rd,addr(rs)
push @rd,imm16
```

```
push @rd,rs
pushl @rd,@rs
pushl @rd,addr
pushl @rd,addr(rs)
pushl @rd,rrs
res @rd,imm4
res addr(rd),imm4
res addr,imm4
res rd,imm4
res rd,rs
resb @rd,imm4
resb addr(rd),imm4
resb addr,imm4
resb rbd,imm4
resb rbd,rs
resflg imm4
ret cc
rl rd,imm1or2
rlb rbd,imm1or2
rlc rd,imm1or2
rlcb rbd,imm1or2
rldb rbb,rba
rr rd,imm1or2
rrb rbd,imm1or2
rrc rd,imm1or2
rrcb rbd,imm1or2
rrdb rbb,rba
rsvd36
rsvd38
rsvd78
rsvd7e
rsvd9d
rsvd9f
rsvdb9
rsvdbf
sbc rd,rs
sbcb rbd,rbs
sc imm8
sda rd,rs
sdab rbd,rs
sdal rrd,rs
sdl rd,rs
sdlb rbd,rs
sdll rrd,rs
set @rd,imm4
set addr(rd),imm4
set addr,imm4
set rd,imm4
set rd,rs
setb @rd,imm4
setb addr(rd),imm4
setb addr,imm4
setb rbd,imm4
setb rbd,rs
setflg imm4
sinb rbd,imm16
sinb rd,imm16
sind @rd,@rs,ra
sindb @rd,@rs,rba
sinib @rd,@rs,ra
sinibr @rd,@rs,ra
sla rd,imm8
slab rbd,imm8
slal rrd,imm8
sll rd,imm8
sllb rbd,imm8
slll rrd,imm8
sout imm16,rs
soutb imm16,rbs
soutd @rd,@rs,ra
soutdb @rd,@rs,rba
soutib @rd,@rs,ra
soutibr @rd,@rs,ra
sra rd,imm8
srab rbd,imm8
sral rrd,imm8
srl rd,imm8
srlb rbd,imm8
srll rrd,imm8
sub rd,@rs
sub rd,addr
sub rd,addr(rs)
sub rd,imm16
sub rd,rs
subb rbd,@rs
subb rbd,addr
subb rbd,addr(rs)
subb rbd,imm8
subb rbd,rbs
subl rrd,@rs
subl rrd,addr
subl rrd,addr(rs)
subl rrd,imm32
subl rrd,rrs
tcc cc,rd
tccb cc,rbd
test @rd
test addr
test addr(rd)
test rd
testb @rd
testb addr
testb addr(rd)
testb rbd
testl @rd
testl addr
testl addr(rd)
testl rrd
trdb @rd,@rs,rba
trdrb @rd,@rs,rba
trib @rd,@rs,rbr
trirb @rd,@rs,rbr
trtdrb @ra,@rb,rbr
trtib @ra,@rb,rr
trtirb @ra,@rb,rbr
trtrb @ra,@rb,rbr
tset @rd
tset addr
tset addr(rd)
tset rd
tsetb @rd
tsetb addr
tsetb addr(rd)
tsetb rbd
xor rd,@rs
xor rd,addr
xor rd,addr(rs)
xor rd,imm16
xor rd,rs
xorb rbd,@rs
xorb rbd,addr
xorb rbd,addr(rs)
xorb rbd,imm8
xorb rbd,rbs
xorb rbd,rbs
```

1.198 as.guide/Acknowledgements

```
Acknowledgements
****************
```

   If you've contributed to as and your name isn't listed here, it is not meant as a slight. We just don't know about it. Send mail to the

maintainer, and we'll correct the situation. Currently (June 1993), the maintainer is Ken Raeburn (email address raeburn@cygnus.com).

Dean Elsner wrote the original GNU assembler for the VAX.(1)

Jay Fenlason maintained GAS for a while, adding support for gdb-specific debug information and the 68k series machines, most of the preprocessing pass, and extensive changes in messages.c, input-file.c, write.c.

K. Richard Pixley maintained GAS for a while, adding various enhancements and many bug fixes, including merging support for several processors, breaking GAS up to handle multiple object file format backends (including heavy rewrite, testing, an integration of the coff and b.out backends), adding configuration including heavy testing and verification of cross assemblers and file splits and renaming, converted GAS to strictly ansi C including full prototypes, added support for m680[34]0 & cpu32, considerable work on i960 including a COFF port (including considerable amounts of reverse engineering), a SPARC opcode file rewrite, DECstation, rs6000, and hp300hpux host ports, updated "know" assertions and made them work, much other reorganization, cleanup, and lint.

Ken Raeburn wrote the high-level BFD interface code to replace most of the code in format-specific I/O modules.

The original VMS support was contributed by David L. Kashtan. Eric Youngdale has done much work with it since.

The Intel 80386 machine description was written by Eliot Dresselhaus.

Minh Tran-Le at IntelliCorp contributed some AIX 386 support.

The Motorola 88k machine description was contributed by Devon Bowen of Buffalo University and Torbjorn Granlund of the Swedish Institute of Computer Science.

Keith Knowles at the Open Software Foundation wrote the original MIPS back end (tc-mips.c, tc-mips.h), and contributed Rose format support (which hasn't been merged in yet). Ralph Campbell worked with the MIPS code to support a.out format.

Support for the Zilog Z8k and Hitachi H8/300 and H8/500 processors (tc-z8k, tc-h8300, tc-h8500), and IEEE 695 object file format (obj-ieee), was written by Steve Chamberlain of Cygnus Support. Steve also modified the COFF back end to use BFD for some low-level operations, for use with the H8/300 and AMD 29k targets.

John Gilmore built the AMD 29000 support, added .include support, and simplified the configuration of which versions accept which pseudo-ops. He updated the 68k machine description so that Motorola's opcodes always produced fixed-size instructions (e.g. jsr), while synthetic instructions remained shrinkable (jbsr). John fixed many bugs, including true tested cross-compilation support, and one bug in relaxation that took a week and required the apocryphal one-bit fix.

Ian Lance Taylor of Cygnus Support merged the Motorola and MIT

syntaxes for the 68k, completed support for some COFF targets (68k, i386 SVR3, and SCO Unix), and made a few other minor patches.

Steve Chamberlain made as able to generate listings.

Support for the HP9000/300 was contributed by Hewlett-Packard.

Support for ELF format files has been worked on by Mark Eichin of Cygnus Support (original, incomplete implementation for SPARC), Pete Hoogenboom and Jeff Law at the University of Utah (HPPA mainly), Michael Meissner of the Open Software Foundation (i386 mainly), and Ken Raeburn of Cygnus Support (sparc, and some initial 64-bit support).

Several engineers at Cygnus Support have also provided many small bug fixes and configuration enhancements.

Many others have contributed large or small bugfixes and enhancements. If you've contributed significant work and are not mentioned on this list, and want to be, let us know. Some of the history has been lost; we aren't intentionally leaving anyone out.

---------- Footnotes ----------

(1) Any more details?

1.199 as.guide/Copying

1.200 as.guide/Index

Index

*****

```
-al
                                    a

-an
                                    a

-as
                                    a

-Asparclite
                            Sparc-Opts

-Av6
                                Sparc-Opts

-Av8
                                Sparc-Opts

-D
                                  D

-f
                                  f

-I path
                                I

-K
                                  K

-L
                                  L

-o
                                  o

-R
                                  R

-v
                                  v

-version
                                v

-W
                                  W

.o
                                  Object

29K support
                            AMD29K-Dependent

$ in symbol names
                    SH-Chars
```