Using as

The GNU Assembler

January 1992

The Free Software Foundation Inc. thanks The Nice Computer Company of Australia for loaning Dean Elsner to write the first (Vax) version of `as` for Project GNU. The proprietors, management and staff of TNCCA thank FSF for distracting the boss while they got some work done.

Dean Elsner, Jay Fenlason & friends

Revision: 1.1.6.1
TEXinfo 2.196

Edited by Roland Pesch for Cygnus Support.

1 Overview

This manual is a user guide to the GNU assembler `as`.

Here is a brief summary of how to invoke `as`. For details, see Chapter 2 [Comand-Line Options], page 5.

```
as [ -a | -al | -as ] [ -D ] [ -f ]
  [ -I path ] [ -k ] [ -L ]
  [ -o objfile ] [ -R ] [ -v ] [ -w ]
  [ -ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC ]
  [ -b ] [ -norelax ]
  [ -l ] [ -mc68000 | -mc68010 | -mc68020 ]
  [ -- | files ... ]
```

`-a | -al | -as`
: Turn on assembly listings; '`-al`', listing only, '`-as`', symbols only, '`-a`', everything.

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

`-f`
: "fast"—skip 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

`-ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC`
: (When configured for Intel 960). Specify which variant of the 960 architecture is the target.

`-b`
: (When configured for Intel 960). Add code to collect statistics about branches taken.

`-norelax`
: (When configured for Intel 960). Do not alter compare-and-branch instructions for long displacements; error if necessary.

`-l`
: (When configured for Motorola 68000). Shorten references to undefined symbols, to one word instead of two

`-mc68000 | -mc68010 | -mc68020`
: (When configured for Motorola 68000). Specify what processor in the 68000 family is the target (default 68020)

`--` | *files* ...
: Standard input, or source files to assemble

1.1 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.

1.2 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 the native assembler would. Any exceptions are documented explicitly (see Chapter 8 [Machine Dependent], page 37). 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 Section 7.38 [`.org`], page 31).

1.3 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 Section 5.5 [Symbol Attributes], page 20.

1.4 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
```

1.5 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 Section 1.7 [Error and Warning Messages], page 4.

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 Section 7.4 [`.app-file`], page 25.

1.6 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 `a.out` 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.

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.

1.7 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 Section 7.4 [`.app-file`], page 25) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (see Section 7.32 [`.ln`], page 30) 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.

2 Command-Line Options

This chapter describes command-line options available in *all* versions of the GNU assembler; see Chapter 8 [Machine Dependent], page 37, for options specific to particular machine architectures.

2.1 Enable Listings: `-a`, `-al`, `-as`

These options enable listing output from the assembler. '`-a`' by itself requests all listing output; '`-al`' requests only the output-program listing, and '`-as`' requests only a symbol table 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`.

If you do not request listing output with one of the '`-a`' options, the listing-control directives have no effect.

2.2 `-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`.

2.3 Work Faster: `-f`

'`-f`' should only be used when assembling programs written by a (trusted) compiler. '`-f`' stops the assembler from pre-processing the input file(s) before assembling them. See Section 3.1 [Pre-processing], page 7.

> *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.

2.4 `.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 Section 7.27 [`.include`], page 29). 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.

2.5 Difference Tables: `-k`

`as` sometimes alters the code emitted for directives of the form '`.word` *sym1-sym2*'; see Section 7.56 [`.word`], page 35. You can use the '`-k`' option if you want a warning issued when this is done.

2.6 Include Local Labels: `-L`

Labels beginning with '`L`' (upper case only) are called *local labels*. See Section 5.3 [Symbol Names], page 19. 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`'.

2.7 Name the Object File: `-o`

There is always one object file output when you run `as`. By default it has the name '`a.out`'. 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.

2.8 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 it bytes are appended to the text section. (See Chapter 4 [Sections and Relocation], page 13.)

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`'.

2.9 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.

2.10 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.

3 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 in BSD 4.2 assembler, except that `as` does not assemble Vax bit-fields.

3.1 Pre-Processing

The 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.

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.

3.2 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 Section 3.6.1 [Character Constants], page 9), any whitespace means the same as exactly one space.

3.3 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

680x0; ';' for the AMD 29K family; ';' for the machine specific family; see Chapter 8 [Machine Dependent], page 37.

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 Chapter 6 [Expressions], page 23) is expected: this will be the logical line number of the **next** line. Then a string (See Section 3.6.1.1 [Strings], page 9.) 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`.

3.4 Symbols

A *symbol* is one or more characters chosen from the set of all letters (both upper and lower case), digits and the two characters '`_.`' On most machines, you can also use `$` in symbol names; exceptions are noted in Chapter 8 [Machine Dependent], page 37. 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 Chapter 5 [Symbols], page 19.

3.5 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 Chapter 8 [Machine Dependent], page 37.) 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 Section 5.1 [Labels], page 19.

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

3.6 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.
```

3.6.1 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.

3.6.1.1 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.

3.6.1.2 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.

3.6.2 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.

3.6.2.1 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 Section 6.2.3 [Prefix Operators], page 23).

3.6.2.2 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.

3.6.2.3 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 AMD 29K and H8/300 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`.

4 Sections and Relocation

4.1 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, `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 7.43 [`.section`], page 32). 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 Section 4.2 [ld Sections], page 14). A *pass1 expression* means an expression with section "pass1" (see Section 4.3 [as Internal Sections], page 15). 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.

4.2 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.

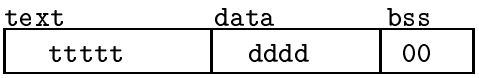
absolute section

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.

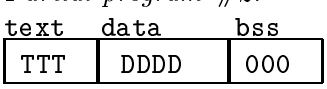
undefined section
: This "section" is a catch-all for address references to objects not in the preceding sections.

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.

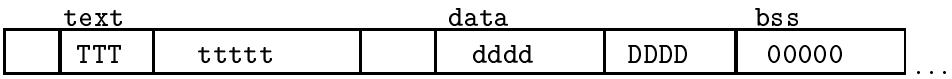
Partial program #1:



Partial program #2:



linked program:



...

addresses:

0...

4.3 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.

absent An expression was expected and none was found.

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

bignum/flonum
: If a number can't be written as a C `int` constant (a bignum or a flonum, but not an integer), it is recorded as belonging to this "section". `as` has to remember that a flonum or a bignum does not fit into 32 bits, and cannot be an argument (see Section 6.2.1 [Arguments], page 23) in an expression: this is done by making a flonum or bignum be in a separate internal section. This is purely for internal `as` convenience; bignum/flonum section behaves similarly to absolute section.

pass1 section
: The expression was impossible to evaluate in the first pass. The assembler will attempt a second pass (second reading of the source) to evaluate the expression. Your expression mentioned an undefined symbol in a way that defies the one-pass (section `+` offset in section) assembly process. No compiler need emit such an expression.

Warning: the second pass is currently not implemented. `as` will abort with an error message if one is required.

difference section

As an assist to the C compiler, expressions of the forms

(*undefined symbol*) − (*expression*)
something − (*undefined symbol*)
(*undefined symbol*) − (*undefined symbol*)

are permitted, and belong to the difference section. `as` re-evaluates such expressions after the source file has been read and the symbol table built. If by that time there are no undefined symbols in the expression then the expression assumes a new section. The intention is to permit statements like '`.word label - base_of_table`' to be assembled in one pass where both `label` and `base_of_table` are undefined. This is useful for compiling C and Algol switch statements, Pascal case statements, FORTRAN computed goto statements and the like.

4.4 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 Chapter 6 [Expressions], page 23.) 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.

4.5 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 Section 7.8 [`.comm`], page 26, see Section 7.29 [`.lcomm`], page 29.

5 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.

5.1 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.

5.2 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 Chapter 6 [Expressions], page 23). This is equivalent to using the `.set` directive. See Section 7.44 [`.set`], page 32.

5.3 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 Chapter 8 [Machine Dependent], page 37.) That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted in Chapter 8 [Machine Dependent], page 37), and underscores. Case of letters is significant: `foo` is a different symbol name than `Foo`.

For the AMD 29K family, '`?`' is also allowed in the body of a symbol name, though not at its beginning.

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 '**N**`b`', using the same digit as when you defined the label. To refer to the next definition of a local label, write '**N**`f`'—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:`'.

`C-A` 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 `L1`*`C-A`*`1`, the 44th `3:` is named `L3`*`C-A`*`44`.

5.4 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 '`.block 4`'.

5.5 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.

5.5.1 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.

5.5.2 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.

5.5.3 Symbol Attributes: `a.out`

5.5.3.1 Descriptor

This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a `.desc` statement (see Section 7.11 [`.desc`], page 26). A descriptor value means nothing to `as`.

5.5.3.2 Other

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

5.5.4 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.

5.5.4.1 Primary Attributes

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

5.5.4.2 Auxiliary Attributes

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

6 Expressions

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

6.1 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.

6.2 Integer Expressions

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

6.2.1 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.

6.2.2 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.

6.2.3 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.

6.2.4 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.

7 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 present regardless of the target machine configuration for the GNU assembler.

7.1 .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.

7.2 .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.

7.3 .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.

7.4 .app-file *string*

`.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 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.

7.5 .ascii "*string*"...

`.ascii` expects zero or more string literals (see Section 3.6.1.1 [Strings], page 9) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

7.6 .asciz "*string*"...

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

7.7 .byte *expressions*

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

7.8 .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.

7.9 .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.

7.10 .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.

7.11 .desc *symbol, abs-expression*

This directive sets the descriptor of the symbol (see Section 5.5 [Symbol Attributes], page 20) 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.

7.12 .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.

7.13 .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 Chapter 8 [Machine Dependent], page 37.

7.14 .eject

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

7.15 .else

`.else` is part of the `as` support for conditional assembly; see Section 7.26 [`.if`], page 28. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

7.16 .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.

7.17 .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 Section 7.26 [`.if`], page 28.

7.18 .equ *symbol, expression*

This directive sets the value of *symbol* to *expression*. It is synonymous with '`.set`'; see Section 7.44 [`.set`], page 32.

7.19 .extern

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

7.20 .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 Chapter 8 [Machine Dependent], page 37.

7.21 .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.

7.22 .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 Chapter 8 [Machine Dependent], page 37.

7.23 .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.

7.24 .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`'.

7.25 .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.

7.26 .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 Section 7.17 [.endif], page 27); optionally, you may include code for the alternative condition, flagged by `.else` (see Section 7.15 [.else], page 27.

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.

7.27 `.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 Chapter 2 [Command-Line Options], page 5). Quotation marks are required around *file*.

7.28 `.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.

7.29 `.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 Section 7.23 [`.global`], page 28), so is normally not visible to `ld`.

7.30 `.lflags`

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

7.31 `.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.

7.32 `.ln` *line-number*

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

7.33 `.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 Chapter 2 [Command-Line Options], page 5), the initial value of the listing counter is one.

7.34 `.long` *expressions*

`.long` is the same as '`.int`', see Section 7.28 [`.int`], page 29.

7.35 `.lsym` *symbol*, *expression*

`.lsym` creates a new symbol named *symbol*, but does not put it in the hash table, ensuring it cannot be referenced by name during the rest of the assembly. This sets the attributes of the symbol to be the same as the expression value:

other `=` *descriptor* `=` `0`
type `=` (section of *expression*)
value `=` *expression*

The new symbol is not flagged as external.

7.36 `.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.

7.37 `.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.

7.38 .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.

7.39 .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`.

7.40 .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.

7.41 .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.

7.42 .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.

7.43 .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.

7.44 .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 Section 5.5 [Symbol Attributes], page 20.)

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.

7.45 .short *expressions*

`.short` is the same as '`.word`'. See Section 7.56 [`.word`], page 35.

7.46 .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 Chapter 8 [Machine Dependent], page 37.

7.47 .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.

7.48 .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.

7.49 .space

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

> *Warning:* In other versions of the GNU assembler, the directive `.space` has the effect of `.block` See Chapter 8 [Machine Dependent], page 37.

7.50 .stabd, .stabn, .stabs

There are three directives that begin '`.stab`'. All emit symbols (see Chapter 5 [Symbols], page 19), 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.

7.51 .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.

7.52 .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.

7.53 .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.

7.54 .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.

7.55 .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.

7.56 .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 Chapter 8 [Machine Dependent], page 37), 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.

7.57 Deprecated Directives

One day these directives won't work. They are included for compatibility with older assemblers.

`.abort`

`.app-file`

`.line`

8 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.

8.1 VAX Dependent Features

8.1.1 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. `as` does not use a temporary disk file, so 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".

8.1.2 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.

8.1.3 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.

8.1.4 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.

8.1.5 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 ...

`j`*COND* *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:

`jacb`*X* *X* may be one of `b` `d` `f` `g` `h` `l` `w`.

(word displacement)
OPCODE ...

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

`jaob`*YYY* *YYY* may be one of `lss` `leq`.

`jsob`*ZZZ* *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:`

8.1.6 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.

8.1.7 Not Supported on VAX

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

8.2 AMD 29K Dependent Features

8.2.1 Options

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

8.2.2 Syntax

8.2.2.1 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).

8.2.2.2 Register Names

General-purpose registers are represented by predefined symbols of the form '`GR`*nnn*' (for global registers) or '`LR`*nnn*' (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
```

8.2.3 Floating Point

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

8.2.4 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.

`.reg` *symbol*, *expression*
`.reg` has the same effect as `.lsym`; see Section 7.35 [`.lsym`], page 30.

`.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`'.

8.2.5 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.

8.3 H8/300 Dependent Features

8.3.1 Options

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

8.3.2 Syntax

8.3.2.1 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.

8.3.2.2 Register Names

You can use predefined symbols of the form '`r`*n*`h`' and '`r`*n*`l`' 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 '`r`*n*' to refer to the H8/300 registers as 16-bit registers (you must use this form for addressing).

The two control registers are called `pc` (program counter; a 16-bit register) and `ccr` (condition code register; an 8-bit register). `r7` is used as the stack pointer, and can also be called `sp`.

8.3.2.3 Addressing Modes

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

`r`*n*
: Register direct

`@r`*n*
: Register indirect

`@(`*d*`, r`*n*`)`
`@(`*d*`:16, r`*n*`)`
: Register indirect: 16-bit displacement *d* from register *n*. (You may specify the '`:16`' for clarity if you wish, but it is not required and has no effect.)

`@r`*n*`+`
: Register indirect with post-increment

`@-r`*n*
: Register indirect with pre-decrement

`@`*aa*
`@`*aa*`:8`
`@`*aa*`:16`
: Absolute address `aa`. You may specify the '`:8`' or '`:16`' for clarity, if you wish; but `as` neither requires this nor uses it—the address size required is taken from context.

`#`*xx*
`#`*xx*`:8`
`#`*xx*`:16`
: Immediate data *xx*. You may specify the '`:8`' or '`:16`' 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.

8.3.3 Floating Point

The H8/300 family uses IEEE floating-point numbers.

8.3.4 H8/300 Machine Directives

`as` has no machine-dependent directives for the H8/300. However, on this platform the '`.int`' and '`.word`' directives generate 16-bit numbers.

8.3.5 Opcodes

For detailed information on the H8/300 machine instruction set, see *H8/300 Series Programming Manual* (Hitachi ADE–602–025).

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

The following table summarizes the opcodes and their arguments:

`Rs`	source register
`Rd`	destination register
`imm`	immediate data
`x:3`	a bit (as a number between 0 and 7)
`d:8`	eight bit displacement from `pc`
`d:16`	sixteen bit displacement from `Rs`

```
add.b   Rs,Rd                biand   #x:3,Rd
add.b   #imm:8,Rd            biand   #x:3,@Rd
add.w   Rs,Rd                biand   #x:3,@aa:8
adds    #1,Rd                bild    #x:3,Rd
adds    #2,Rd                bild    #x:3,@Rd
addx    #imm:8,Rd            bild    #x:3,@aa:8
addx    Rs,Rd                bior    #x:3,Rd
and     #imm:8,Rd            bior    #x:3,@Rd
and     Rs,Rd                bior    #x:3,@aa:8
andc    #imm:8,ccr           bist    #x:3,Rd
band    #x:3,Rd              bist    #x:3,@Rd
band    #x:3,@Rd             bist    #x:3,@aa:8
bra     d:8                  bixor   #x:3,Rd
bt      d:8                  bixor   #x:3,@Rd
brn     d:8                  bixor   #x:3,@aa:8
bf      d:8                  bld     #x:3,Rd
bhi     d:8                  bld     #x:3,@Rd
bls     d:8                  bld     #x:3,@aa:8
bcc     d:8                  bnot    #x:3,Rd
bhs     d:8                  bnot    #x:3,@Rd
bcs     d:8                  bnot    #x:3,@aa:8
blo     d:8                  bnot    Rs,Rd
bne     d:8                  bnot    Rs,@Rd
beq     d:8                  bnot    Rs,@aa:8
bvc     d:8                  bor     #x:3,Rd
bvs     d:8                  bor     #x:3,@Rd
bpl     d:8                  bor     #x:3,@aa:8
bmi     d:8                  bset    #x:3,@Rd
bge     d:8                  bset    #x:3,@aa:8
blt     d:8                  bset    Rs,Rd
bgt     d:8                  bset    Rs,@Rd
ble     d:8                  bset    Rs,@aa:8
bclr    #x:3,Rd              bsr     d:8
bclr    #x:3,@Rd             bst     #x:3,Rd
bclr    #x:3,@aa:8           bst     #x:3,@Rd
bclr    Rs,Rd                bst     #x:3,@aa:8
bclr    Rs,@Rd               btst    #x:3,Rd
```

```
btst    #x:3,@Rd            mov.w   @(d:16, Rs),Rd
btst    #x:3,@aa:8          mov.w   @Rs+,Rd
btst    Rs,Rd               mov.w   @aa:16,Rd
btst    Rs,@Rd              mov.w   Rs,@Rd
btst    Rs,@aa:8            mov.w   Rs,@(d:16, Rd)
bxor    #x:3,Rd             mov.w   Rs,@-Rd
bxor    #x:3,@Rd            mov.w   Rs,@aa:16
bxor    #x:3,@aa:8          movfpe  @aa:16,Rd
cmp.b   #imm:8,Rd           movtpe  Rs,@aa:16
cmp.b   Rs,Rd               mulxu   Rs,Rd
cmp.w   Rs,Rd               neg     Rs
daa     Rs                  nop
das     Rs                  not     Rs
dec     Rs                  or      #imm:8,Rd
divxu   Rs,Rd               or      Rs,Rd
eepmov                      orc     #imm:8,ccr
inc     Rs                  pop     Rs
jmp     @Rs                 push    Rs
jmp     @aa:16              rotl    Rs
jmp     @@aa                rotr    Rs
jsr     @Rs                 rotxl   Rs
jsr     @aa:16              rotxr   Rs
jsr     @@aa:8              rte
ldc     #imm:8,ccr          rts
ldc     Rs,ccr              shal    Rs
mov.b   Rs,Rd               shar    Rs
mov.b   #imm:8,Rd           shll    Rs
mov.b   @Rs,Rd              shlr    Rs
mov.b   @(d:16, Rs),Rd      sleep
mov.b   @Rs+,Rd             stc     ccr,Rd
mov.b   @aa:16,Rd           sub.b   Rs,Rd
mov.b   @aa:8,Rd            sub.w   Rs,Rd
mov.b   Rs,@Rd              subs    #1,Rd
mov.b   Rs,@(d:16, Rd)      subs    #2,Rd
mov.b   Rs,@-Rd             subx    #imm:8,Rd
mov.b   Rs,@aa:16           subx    Rs,Rd
mov.b   Rs,@aa:8            xor     #imm:8,Rd
mov.w   Rs,Rd               xor     Rs,Rd
mov.w   #imm:16,Rd          xorc    #imm:8,ccr
mov.w   @Rs,Rd
```

Four H8/300 instructions (`add`, `cmp`, `mov`, `sub`) are defined with variants using the suffixes '`.b`' and '`.w`' 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` will issue a warning if there's a mismatch between the suffix and the register size.

8.4 Intel 80960 Dependent Features

8.4.1 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`'.

8.4.2 Floating Point

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

8.4.3 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 Section 7.29 [`.lcomm`], page 29.

`.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).

8.4.4 i960 Opcodes

All Intel 960 machine instructions are supported; see Section 8.4.1 [i960 Command-line Options], page 46 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.

8.4.4.1 `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.

8.4.4.2 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
```

8.5 M680x0 Dependent Features

8.5.1 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.

8.5.2 Syntax

The 680x0 version of `as` uses syntax similar to the Sun assembler. Size modifiers are appended directly to the end of the opcode without an intervening period. For example, write '`movl`' rather than '`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@`'

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*'

8.5.3 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.

8.5.4 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.

8.5.5 Opcodes

8.5.5.1 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.

`j`*XX* Here, 'j*XX*' 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*:

```
        jXX foo
```

gives

```
         bNXs oof
         jmp foo
oof:
```

`db`*XX* 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 '`db`*XX* `foo`', `as` will emit

```
        db XX oo1
        bra oo2
oo1:    jmpl foo
oo2:
```

`fj`*XX* 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 '`fj`*XX* `foo`'.

8.5.5.2 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.

8.6 SPARC Dependent Features

8.6.1 Options

The Sparc has no machine dependent options.

8.6.2 Floating Point

The Sparc uses IEEE floating-point numbers.

8.6.3 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.

8.7 80386 Dependent Features

8.7.1 Options

The 80386 has no machine dependent options.

8.7.2 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.

8.7.3 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.

8.7.4 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)`'.

8.7.5 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.

8.7.6 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).

8.7.7 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:
```

8.7.8 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.

8.7.9 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`'.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions

for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore,

by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a

version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Applying These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the program's name and an idea of what it does.*
> Copyright (C) 19*yy* *name of author*
>
> This program is free software; you can redistribute it and/or
> modify it under the terms of the GNU General Public License
> as published by the Free Software Foundation; either version 2
> of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful,
> but WITHOUT ANY WARRANTY; without even the implied warranty of
> MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
> GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License
> along with this program; if not, write to the
> Free Software Foundation, Inc., 675 Mass Ave,
> Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) 19*yy* *name of author*
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
> type 'show w'. This is free software, and you are welcome
> to redistribute it under certain conditions; type 'show c'
> for details.

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in
> the program 'Gnomovision' (which makes passes at compilers)
> written by James Hacker.
>
> *signature of Ty Coon*, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index

(Index is nonexistent)

Short Contents

Table of Contents