Using as

The GNU Assembler

January 1994

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

Using `as`
Edited by 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 9.

```
as [ -a[cdhlns][=file] ] [ -D ]  [ --defsym sym=val ]
  [ -f ] [ --gstabs ] [ --help ] [ -I dir ] [ -J ] [ -K ] [ -L ]
  [ --keep-locals ] [ -o objfile ] [ -R ] [ --statistics ] [ -v ]
  [ -version ] [ --version ] [ -W ] [ -w ] [ -x ] [ -Z ]
  [ -mbig-endian | -mlittle-endian ]
  [ -m[arm]1 | -m[arm]2 | -m[arm]250 | -m[arm]3 | -m[arm]6 | -m[arm]7[t][[d]m[i]] ]
  [ -m[arm]v2 | -m[arm]v2a | -m[arm]v3 | -m[arm]v3m | -m[arm]v4 | -m[arm]v4t ]
  [ -mthumb | -mall ]
  [ -mfpa10 | -mfpa11 | -mfpe-old | -mno-fpu ]
  [ -EB | -EL ]
  [ -mapcs-32 | -mapcs-26 ]
  [ -O ]
  [ -Av6 | -Av7 | -Av8 | -Asparclet | -Asparclite
    -Av8plus | -Av8plusa | -Av9 | -Av9a ]
  [ -xarch=v8plus | -xarch=v8plusa ] [ -bump ] [ -32 | -64 ]
  [ -ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC ]
  [ -b ] [ -no-relax ]
  [ -l ] [ -m68000 | -m68010 | -m68020 | ... ]
  [ -nocpp ] [ -EL ] [ -EB ] [ -G num ] [ -mcpu=CPU ]
  [ -mips1 ] [ -mips2 ] [ -mips3 ] [ -m4650 ] [ -no-m4650 ]
  [ --trap ] [ --break ]
  [ --emulation=name ]
  [ -- | files ... ]
```

`-a[cdhlmns]`
: Turn on listings, in any of a variety of ways:

  `-ac` omit false conditionals

  `-ad` omit debugging directives

  `-ah` include high-level source

  `-al` include assembly

  `-am` include macro expansions

  `-an` omit forms processing

  `-as` include symbols

  `=file` set the name of the listing file

  You may combine these options; for example, use '`-aln`' for assembly listing without forms processing. The '`=file`' option, if used, must be the last one. By itself, '`-a`' defaults to '`-ahls`'.

`-D`
: Ignored. This option is accepted for script compatibility with calls to other assemblers.

`--defsym` *sym*=*value*
: Define the symbol *sym* to be *value* before assembling the input file. *value* must be an integer constant. As in C, a leading '`0x`' indicates a hexadecimal value, and a leading '`0`' indicates an octal value.

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

`--gstabs`
: Generate stabs debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it.

`--help`
: Print a summary of the command line options and exit.

`-I` *dir*
: Add directory *dir* to the search list for `.include` directives.

`-J`
: Don't warn about signed overflow.

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

`-L`
`--keep-locals`
: Keep (in the symbol table) local symbols. On traditional a.out systems these start with '`L`', but different systems have different local label prefixes.

`-o` *objfile*
: Name the object-file output from `as` *objfile*.

`-R`
: Fold the data section into the text section.

`--statistics`
: Print the maximum space (in bytes) and total time (in seconds) used by assembly.

`--strip-local-absolute`
: Remove local absolute symbols from the outgoing symbol table.

`-v`
`-version`
: Print the `as` version.

`--version`
: Print the `as` version and exit.

`-W`
: Suppress warning messages.

`-w`
: Ignored.

`-x`
: Ignored.

`-Z`
: Generate an object file even after errors.

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

The following options are available when as is configured for an ARC processor.

`-mbig-endian`
: Generate "big endian" format output.

`-mlittle-endian`
: Generate "little endian" format output.

The following options are available when as is configured for the ARM processor family.

`-m[arm]1 | -m[arm]2 | -m[arm]250 | -m[arm]3 | -m[arm]6 | -m[arm]7[t][[d]m] |`
`-m[arm]v2 | -m[arm]v2a | -m[arm]v3 | -m[arm]v3m | -m[arm]v4 | -m[arm]v4t`
: Specify which variant of the ARM architecture is the target.

`-mthumb | -mall`
: Enable or disable Thumb only instruction decoding.

`-mfpa10 | -mfpa11 | -mfpe-old | -mno-fpu`
: Select which Floating Point architcture is the target.

`-mapcs-32 | -mapcs-26`
: Select which procedure calling convention is in use.

`-EB | -EL`
: Select either big-endian (-EB) or little-endian (-EL) output.

The following options are available when as is configured for a D10V processor.

`-O`
: Optimize output by parallelizing instructions.

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

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

`-b`
: Add code to collect statistics about branches taken.

`-no-relax`
: Do not alter compare-and-branch instructions for long displacements; error if necessary.

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

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

`-m68000 | -m68008 | -m68010 | -m68020 | -m68030 | -m68040 | -m68060`
`| -m68302 | -m68331 | -m68332 | -m68333 | -m68340 | -mcpu32 | -m5200`
: Specify what processor in the 68000 family is the target. The default is normally the 68020, but this can be changed at configuration time.

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

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

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

`-Av6 | -Av7 | -Av8 | -Asparclet | -Asparclite`
`-Av8plus | -Av8plusa | -Av9 | -Av9a`
: Explicitly select a variant of the SPARC architecture.

  '`-Av8plus`' and '`-Av8plusa`' select a 32 bit environment. '`-Av9`' and '`-Av9a`' select a 64 bit environment.

  '`-Av8plusa`' and '`-Av9a`' enable the SPARC V9 instruction set with UltraSPARC extensions.

`-xarch=v8plus | -xarch=v8plusa`
: For compatibility with the Solaris v9 assembler. These options are equivalent to -Av8plus and -Av8plusa, respectively.

`-bump` Warn when the assembler switches to another architecture.

The following options are available when as is configured for a MIPS processor.

`-G` *num* This option sets the largest size of an object that can be referenced implicitly with the `gp` register. It is only accepted for targets that use ECOFF format, such as a DECstation running Ultrix. The default value is 8.

`-EB` Generate "big endian" format output.

`-EL` Generate "little endian" format output.

`-mips1`
`-mips2`
`-mips3` Generate code for a particular MIPS Instruction Set Architecture level. '`-mips1`' corresponds to the R2000 and R3000 processors, '`-mips2`' to the R6000 processor, and '`-mips3`' to the R4000 processor.

`-m4650`
`-no-m4650`
: Generate code for the MIPS R4650 chip. This tells the assembler to accept the '`mad`' and '`madu`' instruction, and to not schedule '`nop`' instructions around accesses to the '`HI`' and '`LO`' registers. '`-no-m4650`' turns off this option.

`-mcpu=`*CPU*
: Generate code for a particular MIPS cpu. This has little effect on the assembler, but it is passed by `gcc`.

`--emulation=`*name*
: This option causes `as` to emulate `as` configured for some other target, in all respects, including output format (choosing between ELF and ECOFF only), handling of pseudo-opcodes which may generate debugging information or store symbol table information, and default endianness. The available configuration names are: '`mipsecoff`', '`mipself`', '`mipslecoff`', '`mipsbecoff`', '`mipslelf`', '`mipsbelf`'. The first two do not alter the default endianness from that of the primary target for which the assembler was configured; the others change the default to little- or big-endian as indicated by the '`b`' or '`l`' in the name. Using '`-EB`' or '`-EL`' will override the endianness selection in any case.

  This option is currently supported only when the primary target `as` is configured for is a MIPS ELF or ECOFF target. Furthermore, the primary target or others specified with '--enable-targets=...' at configuration time must include support for the other format, if both are to be available. For example, the Irix 5 configuration includes support for both.

Eventually, this option will support more configurations, with more fine-grained control over the assembler's behavior, and will be supported for more processors.

-nocpp
: as ignores this option. It is accepted for compatibility with the native tools.

--trap
--no-trap
--break
--no-break
: Control how to deal with multiplication overflow and division by zero. '--trap' or '--no-break' (which are synonyms) take a trap exception (and only work for Instruction Set Architecture level 2 and higher); '--break' or '--no-trap' (also synonyms, and the default) take a break exception.

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 The GNU Assembler

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

as is primarily intended to assemble the output of the GNU C compiler gcc for use by the linker ld. Nevertheless, we've tried to make as assemble correctly everything that other assemblers for the same machine would assemble. Any exceptions are documented explicitly (see Chapter 8 [Machine Dependencies], page 51). 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 7.45 [.org], page 43).

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 28.

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` produces 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 7.

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 34.

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. Its default name is `a.out`, or `b.out` when `as` is configured for the Intel 80960. You can give it another name by using the `-o` option. Conventionally, object file names end with ‘`.o`’. The default name is used for historical reasons: older assemblers were capable of assembling self-contained programs directly into a runnable program. (For some formats, this isn’t currently possible, but it can be done for the `a.out` format.)

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

1.7 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 34) 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 7.36 [`.line`], page 40) 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 Dependencies], page 51, for options specific to particular machine architectures.

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

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

emits a listing to standard output with high-level and assembly source.

Usually you do not need to use this '`-Wa`' mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. (You can call the GNU compiler driver with the '`-v`' option to see precisely what options it passes to each compilation pass, including the assembler.)

2.1 Enable Listings: `-a[cdhlns]`

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

Use the '`-ac`' option to omit false conditionals from a listing. Any lines which are not assembled because of a false `.if` (or `.ifdef`, or any other conditional), or a true `.if` followed by an `.else`, will be omitted from the listing.

Use the '`-ad`' option to omit debugging directives from the listing.

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

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

2.2 `-D`

This option has no effect whatsoever, but it is accepted to make it more likely that scripts written for other assemblers 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 doing whitespace and comment preprocessing on the input file(s) before assembling them. See Section 3.1 [Preprocessing], page 15.

> *Warning:* if you use '`-f`' when the files actually need to be preprocessed (if they contain comments, for example), `as` does not work correctly.

2.4 `.include` search path: `-I` *path*

Use this option to add a *path* to the list of directories `as` searches for files specified in `.include` directives (see Section 7.30 [`.include`], page 39). 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 7.69 [`.word`], page 50. 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 27. Normally you do not 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 do not 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`'.

By default, a local label is any label beginning with '`L`', but each target is allowed to redefine the local label prefix. On the HPPA local labels begin with '`L$`'. '`;`' for the ARM family;

2.7 Assemble in MRI Compatibility Mode: `-M`

The `-M` or `--mri` option selects MRI compatibility mode. This changes the syntax and pseudo-op handling of `as` to make it compatible with the `ASM68K` or the `ASM960` (depending upon the configured target) assembler from Microtec Research. The exact nature of the MRI syntax will not be documented here; see the MRI manuals for more information. Note in particular that the handling of macros and macro arguments is somewhat different. The purpose of this option is to permit assembling existing MRI assembler code using `as`.

The MRI compatibility is not complete. Certain operations of the MRI assembler depend upon its object file format, and can not be supported using other object file formats. Supporting these would require enhancing each object file format individually. These are:

- global symbols in common section

  The m68k MRI assembler supports common sections which are merged by the linker. Other object file formats do not support this. `as` handles common sections by treating them as a single common symbol. It permits local symbols to be defined within a common section, but it can not support global symbols, since it has no way to describe them.

- complex relocations

  The MRI assemblers support relocations against a negated section address, and relocations which combine the start addresses of two or more sections. These are not support by other object file formats.

- `END` pseudo-op specifying start address

  The MRI `END` pseudo-op permits the specification of a start address. This is not supported by other object file formats. The start address may instead be specified using the `-e` option to the linker, or in a linker script.

- `IDNT`, `.ident` and `NAME` pseudo-ops

  The MRI `IDNT`, `.ident` and `NAME` pseudo-ops assign a module name to the output file. This is not supported by other object file formats.

- `ORG` pseudo-op

  The m68k MRI `ORG` pseudo-op begins an absolute section at a given address. This differs from the usual `as` `.org` pseudo-op, which changes the location within the current section. Absolute sections are not supported by other object file formats. The address of a section may be assigned within a linker script.

There are some other features of the MRI assembler which are not supported by `as`, typically either because they are difficult or because they seem of little consequence. Some of these may be supported in future releases.

- EBCDIC strings

  EBCDIC strings are not supported.

- packed binary coded decimal

  Packed binary coded decimal is not supported. This means that the `DC.P` and `DCB.P` pseudo-ops are not supported.

- `FEQU` pseudo-op

  The m68k `FEQU` pseudo-op is not supported.

- `NOOBJ` pseudo-op

  The m68k `NOOBJ` pseudo-op is not supported.

- `OPT` branch control options

  The m68k `OPT` branch control options—`B`, `BRS`, `BRB`, `BRL`, and `BRW`—are ignored. `as` automatically relaxes all branches, whether forward or backward, to an appropriate size, so these options serve no purpose.

- `OPT` list control options

  The following m68k `OPT` list control options are ignored: `C`, `CEX`, `CL`, `CRE`, `E`, `G`, `I`, `M`, `MEX`, `MC`, `MD`, `X`.

- other `OPT` options

  The following m68k `OPT` options are ignored: `NEST`, `O`, `OLD`, `OP`, `P`, `PCO`, `PCR`, `PCS`, `R`.

- `OPT D` option is default

  The m68k `OPT D` option is the default, unlike the MRI assembler. `OPT NOD` may be used to turn it off.

- `XREF` pseudo-op.

  The m68k `XREF` pseudo-op is ignored.
- `.debug` pseudo-op

  The i960 `.debug` pseudo-op is not supported.
- `.extended` pseudo-op

  The i960 `.extended` pseudo-op is not supported.
- `.list` pseudo-op.

  The various options of the i960 `.list` pseudo-op are not supported.
- `.optimize` pseudo-op

  The i960 `.optimize` pseudo-op is not supported.
- `.output` pseudo-op

  The i960 `.output` pseudo-op is not supported.
- `.setreal` pseudo-op

  The i960 `.setreal` pseudo-op is not supported.

2.8 Dependency tracking: `--MD`

`as` can generate a dependency file for the file it creates. This file consists of a single rule suitable for `make` describing the dependencies of the main source file.

The rule is written to the file named in its argument.

This feature is used in the automatic updating of makefiles.

2.9 Name the Object File: `-o`

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

Whatever the object file is called, `as` overwrites any existing file of the same name.

2.10 Join Data and Text Sections: `-R`

`-R` tells `as` to write the object file as if all data-section data lives in the text section. This is only done at the very last moment: your binary data are the same, but data section parts are relocated differently. The data section part of your object file is zero bytes long because all its bytes are appended to the text section. (See Chapter 4 [Sections and Relocation], page 21.)

When you specify `-R` it would be possible to generate shorter address displacements (because we do not 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`'.

`-R` is not supported for any of the HPPA targets. Using `-R` generates a warning from `as`.

2.11 Display Assembly Statistics: `--statistics`

Use '`--statistics`' to display two statistics about the resources used by `as`: the maximum amount of space allocated during the assembly (in bytes), and the total execution time taken for the assembly (in CPU seconds).

2.12 Compatible output: `--traditional-format`

For some targets, the output of `as` is different in some ways from the output of some existing assembler. This switch requests `as` to use the traditional format instead.

For example, it disables the exception frame optimizations which `as` normally does by default on `gcc` output.

2.13 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.

2.14 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.

2.15 Generate Object File in Spite of Errors: `-Z`

After an error message, `as` normally produces no output. If for some reason you are interested in object file output even after `as` gives an error message on your program, use the '`-Z`' option. If there are any errors, `as` continues anyways, and writes an object file after a final warning message of the form '*n* `errors,` *m* `warnings, generating bad object file.`'

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

3.1 Preprocessing

The `as` internal preprocessor:

- 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.

It does not do macro processing, include file handling, or anything else you may get from your C compiler's preprocessor. You can do include file processing with the `.include` directive (see Section 7.30 [`.include`], page 39). You can use the GNU C compiler driver to get other "CPP" style preprocessing, by giving the input file a '`.S`' suffix. See section "Options Controlling the Kind of Output" in *Using GNU CC*.

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

If the first line of an input file is `#NO_APP` or if you use the '`-f`' option, whitespace and comments are not removed from the input file. Within an input file, you can ask for whitespace and comment removal in specific portions of the by putting a line that says `#APP` before the text that may contain whitespace or comments, and putting a line that says `#NO_APP` after this text. This feature is mainly intend to support `asm` statements in compilers whose output is otherwise free of comments and whitespace.

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 17), 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 '`;`' for the AMD 29K family; '`;`' on the ARC; '`;`' for the H8/300 family; '`!`' for the H8/500 family; '`;`' for the HPPA; '`#`' on the i960; '`!`' for the Hitachi SH; '`!`' on the SPARC; '`#`' on the m32r; '`|`' on the 680x0; '`#`' on the Vax; '`!`' for the Z8000; '`#`' on the V850; see Chapter 8 [Machine Dependencies], page 51.

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

The V850 assembler also supports a double dash as starting a comment that extends to the end of the line.

'`--`';

To be compatible with past assemblers, lines that begin with '`#`' have a special interpretation. Following the '`#`' should be an absolute expression (see Chapter 6 [Expressions], page 31): the logical line number of the *next* line. Then a string (see Section 3.6.1.1 [Strings], page 17) 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 three characters '`_.$`'. On most machines, you can also use `$` in symbol names; exceptions are noted in Chapter 8 [Machine Dependencies], page 51. 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 27.

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 Dependencies], page 51.) The newline or separator character is considered part of the preceding statement. Newlines and separators within character constants are an exception: they do not 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 assembles into a machine language instruction. Different versions of `as` for different computers 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 27.

For HPPA targets, labels need not be immediately followed by a colon, but the definition of a label must begin in column zero. This also implies that only one label may be defined on each line.

```
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.

`\x` *hex-digits...*
: A hex character code. All trailing hex digits are combined. Either upper or lower case `x` works.

`\\` 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 `\` gives a warning, but assembles 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, do not 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 32).

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`'. ('`0`' is optional on the HPPA.)
- A letter, to tell `as` the rest of the number is a flonum. `e` is recommended. Case is not important.

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

  On the ARC, the letter must be one of the letters '`DFRS`' (in upper or lower case).

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

  On the HPPA architecture, the letter must be '`E`' (upper case only).
- 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` assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification, but it suffices to explain how `as` uses sections.

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

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

When it generates COFF output, `as` can also generate whatever other named sections you specify using the '`.section`' directive (see Section 7.52 [`.section`], page 45). If you do not use any directives that place output in the '`.text`' or '`.data`' sections, these sections still exist, but are empty.

When `as` generates SOM or ELF output for the HPPA, `as` can also generate whatever other named sections you specify using the '`.space`' and '`.subspace`' directives. See *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001) for details on the '`.space`' and '`.subspace`' assembler directives.

Additionally, `as` uses different names for the standard text, data, and bss sections when generating SOM output. Program text is placed into the '`$CODE$`' section, data into '`$DATA$`', and BSS into '`$BSS$`'.

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

When generating either SOM or ELF output files on the HPPA, the text section starts at address `0`, the data section at address `0x4000000`, and the bss section follows the data section.

To let `ld` know which data changes 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, most expressions `as` computes have this section-relative nature. (For some object formats, such as SOM for the HPPA, some expressions are symbol-relative instead.)

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 the linker never arranges two partial programs' data sections with overlapping addresses after linking, *by definition* their absolute sections must overlap. Address `{absolute 239}` in one part of a program is always the same address when the program is running as address `{absolute 239}` in any other part of the 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* is 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 programs' 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 Linker 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 contains 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 do not 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:

text	data	bss
ttttt	dddd	00

Partial program #2:

text	data	bss
TTT	DDDD	000

linked program:

	text			data		bss
	TTT	ttttt		dddd	DDDD	00000

...

addresses:

0...

4.3 Assembler Internal Sections

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

ASSEMBLER-INTERNAL-LOGIC-ERROR!

An internal assembler logic error has been found. This means there is a bug in the assembler.

expr section

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

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 go into the object file together with other objects in the same subsection. 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 do not use subsections, everything goes 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 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 31.) 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 changes it, and any label definition captures its current value. The location counter of the section where statements are being assembled 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.

The `.lcomm` pseudo-op defines a symbol in the bss section; see Section 7.34 [`.lcomm`], page 40.

The `.comm` pseudo-op may be used to declare a common symbol, which is another form of uninitialized symbol; see See Section 7.9 [`.comm`], page 35.

When assembling for a target which supports multiple sections, such as ELF or COFF, you may switch into the `.bss` section and define symbols as usual; see Section 7.52 [`.section`], page 45. You may only assemble zero values into the section. Typically the section will only contain symbol definitions and `.skip` directives (see Section 7.58 [`.skip`], page 47).

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.

On the HPPA, the usual form for a label need not be immediately followed by a colon, but instead must start in column zero. Only one label may be defined on a single line. To work around this, the HPPA version of `as` also provides a special directive `.label` for defining labels more flexibly.

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 31). This is equivalent to using the `.set` directive. See Section 7.53 [`.set`], page 46.

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 Dependencies], page 51. That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted in Chapter 8 [Machine Dependencies], page 51), and underscores. For the AMD 29K family, '`?`' is also allowed in the body of a symbol name, though not at its beginning.

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

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

Local Symbol Names

Local symbols help compilers and programmers use names temporarily. There are ten local symbol names, which are re-used throughout the program. You may refer to them using the names '`0`' '`1`' ... '`9`'. To define a local symbol, write a label of the form '**`N`**`:`' (where **N** represents any digit). To refer to the most recent previous definition of that symbol write '**`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 use the '`-L`' option then `as` retains 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 do not 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 .`' defines `melvin` to contain its own address. Assigning a value to `.` is treated the same as a `.org` directive. Thus, the expression '`.=.+4`' is the same as saying '`.space 4`'.

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 file, and `ld` tries to determine its value from other files linked into the same program. 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 7.12 [`.desc`], page 35). 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.

5.5.5 Symbol Attributes for SOM

The SOM format for the HPPA supports a multitude of symbol attributes set with the `.EXPORT` and `.IMPORT` directives.

The attributes are described in *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001) under the `IMPORT` and `EXPORT` assembler directive documentation.

6 Expressions

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

The result of an expression must be an absolute number, or else an offset into a particular section. If an expression is not absolute, and there is not enough information when `as` sees the expression to know its section, a second pass over the source program might be necessary to interpret the expression—but the second pass is currently not implemented. `as` aborts with an error message in this situation.

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` assumes 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. You may not add together arguments from different sections.

   `-` *Subtraction*. If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections.

In short, it's only meaningful to add or subtract the *offsets* in an address; you can only have a defined section in one of the two arguments.

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 that are available regardless of the target machine configuration for the GNU assembler. Some machine configurations provide additional directives. See Chapter 8 [Machine Dependencies], page 51.

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*, *abs-expr*

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment required, as described below.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The way the required alignment is specified varies from system to system. For the a29k, hppa, m68k, m88k, w65, sparc, and Hitachi SH, and i386 using ELF format, the first expression is the alignment request in bytes. For example '`.align 8`' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

For other systems, including the i386 using a.out format, it is the number of low-order zero bits the location counter must have after advancement. For example '`.align 3`' advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

This inconsistency is due to the different behaviors of the various native assemblers for these systems which GAS must emulate. GAS also provides `.balign` and `.p2align` directives, described later, which have a consistent behavior across all architectures (but are specific to GAS).

7.4 `.app-file` *string*

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

7.5 `.ascii "`*string*`"`...

`.ascii` expects zero or more string literals (see Section 3.6.1.1 [Strings], page 17) 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 `.balign[wl]` *abs-expr*, *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 alignment request in bytes. For example '`.balign 8`' advances the location counter until it is 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 fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.balignw` and `.balignl` directives are variants of the `.balign` directive. The `.balignw` directive treats the fill pattern as a two byte word value. The `.balignl` directives treats the fill pattern as a four byte longword value. For example, `.balignw 4,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

7.8 `.byte` *expressions*

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

7.9 `.comm` *symbol* , *length*

`.comm` declares a common symbol named *symbol*. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If `ld` does not see a definition for the symbol–just one or more common symbols–then it will allocate *length* bytes of uninitialized memory. *length* must be an absolute expression. If `ld` sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

When using ELF, the `.comm` directive takes an optional third argument. This is the desired alignment of the symbol, specified as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address should be zero). The alignment must be an absolute expression, and it must be a power of two. If `ld` allocates uninitialized memory for the common symbol, it will use the alignment when placing the symbol. If no alignment is specified, `as` will set the alignment to the largest power of two less than or equal to the size of the symbol, up to a maximum of 16.

The syntax for `.comm` differs slightly on the HPPA. The syntax is '*symbol* `.comm,` *length*'; *symbol* is optional.

7.10 `.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.

7.11 `.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.

7.12 `.desc` *symbol*, *abs-expression*

This directive sets the descriptor of the symbol (see Section 5.5 [Symbol Attributes], page 28) 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` accepts it, but produces no output, when configured for COFF.

7.13 `.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.

7.14 `.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 Dependencies], page 51.

7.15 `.eject`

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

7.16 `.else`

`.else` is part of the `as` support for conditional assembly; see Section 7.29 [`.if`], page 38. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

7.17 `.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.

7.18 `.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 7.29 [`.if`], page 38.

7.19 `.equ` ***symbol, expression***

This directive sets the value of *symbol* to *expression*. It is synonymous with '`.set`'; see Section 7.53 [`.set`], page 46.

The syntax for `equ` on the HPPA is '*symbol* `.equ` *expression*'.

7.20 `.equiv` ***symbol, expression***

The `.equiv` directive is like `.equ` and `.set`, except that the assembler will signal an error if *symbol* is already defined.

Except for the contents of the error message, this is roughly equivalent to

```
.ifdef SYM
.err
.endif
.equ SYM,VAL
```

7.21 `.err`

If `as` assembles a `.err` directive, it will print an error message and, unless the `-Z` option was used, it will not generate an object file. This can be used to signal error an conditionally compiled code.

7.22 `.extern`

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

7.23 `.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 Dependencies], page 51.

7.24 `.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.

7.25 `.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 Dependencies], page 51.

7.26 `.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* takes its attributes from a symbol of the same name from another file linked into the same program.

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

On the HPPA, `.global` is not always enough to make it accessible to other partial programs. You may need the HPPA-only `.EXPORT` directive as well. See Section 8.7.5 [HPPA Assembler Directives], page 67.

7.27 `.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`'.

7.28 `.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.

7.29 `.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 7.18 [`.endif`], page 36); optionally, you may include code for the alternative condition, flagged by `.else` (see Section 7.16 [`.else`], page 36).

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.

7.30 `.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 9). Quotation marks are required around *file*.

7.31 `.int` ***expressions***

Expect zero or more *expressions*, of any section, separated by commas. For each expression, emit a number that, at run time, is the value of that expression. The byte order and bit size of the number depends on what kind of target the assembly is for.

7.32 `.irp` ***symbol, values***...

Evaluate a sequence of statements assigning different values to *symbol*. The sequence of statements starts at the `.irp` directive, and is terminated by an `.endr` directive. For each *value*, *symbol* is set to *value*, and the sequence of statements is assembled. If no *value* is listed, the sequence of statements is assembled once, with *symbol* set to the null string. To refer to *symbol* within the sequence of statements, use *symbol*.

For example, assembling

```
        .irp    param,1,2,3
        move    d\param,sp@-
        .endr
```

is equivalent to assembling

```
        move    d1,sp@-
        move    d2,sp@-
        move    d3,sp@-
```

7.33 `.irpc` ***symbol, values***...

Evaluate a sequence of statements assigning different values to *symbol*. The sequence of statements starts at the `.irpc` directive, and is terminated by an `.endr` directive. For each character in *value*, *symbol* is set to the character, and the sequence of statements is assembled. If no *value* is listed, the sequence of statements is assembled once, with *symbol* set to the null string. To refer to *symbol* within the sequence of statements, use *symbol*.

For example, assembling

```
        .irpc    param,123
        move    d\param,sp@-
        .endr
```

is equivalent to assembling

```
        move    d1,sp@-
        move    d2,sp@-
        move    d3,sp@-
```

7.34 `.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 that at run-time the bytes start off zeroed. *Symbol* is not declared global (see Section 7.26 [`.global`], page 38), so is normally not visible to `ld`.

Some targets permit a third argument to be used with `.lcomm`. This argument specifies the desired alignment of the symbol in the bss section.

The syntax for `.lcomm` differs slightly on the HPPA. The syntax is '*symbol* `.lcomm,` *length*'; *symbol* is optional.

7.35 `.lflags`

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

7.36 `.line` *line-number*

Change the logical line number. *line-number* must be an absolute expression. The next line has that logical line number. Therefore any other statements on the current line (after a statement separator character) are reported as on logical line number *line-number* − 1. One day `as` will no longer support this directive: it is recognized only for compatibility with existing assembler programs.

Warning: In the AMD29K configuration of as, this command is not available; use the synonym `.ln` in that context.

Even though this is a directive associated with the `a.out` or `b.out` object-code formats, `as` still recognizes it when producing COFF output, and treats '`.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.

7.37 `.linkonce` [*type*]

Mark the current section so that the linker only includes a single copy of it. This may be used to include the same section in several different object files, but ensure that the linker will only include it once in the final output file. The `.linkonce` pseudo-op must be used for each instance of the section. Duplicate sections are detected based on the section name, so it should be unique.

This directive is only supported by a few object file formats; as of this writing, the only object file format which supports it is the Portable Executable format used on Windows NT.

The *type* argument is optional. If specified, it must be one of the following strings. For example:

```
.linkonce same_size
```

Not all types may be supported on all object file formats.

`discard` Silently discard duplicate sections. This is the default.

`one_only` Warn if there are duplicate sections, but still keep only one copy.

`same_size`
Warn if any of the duplicates have different sizes.

`same_contents`
Warn if any of the duplicates do not have exactly the same contents.

7.38 `.ln` *line-number*

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

7.39 `.mri` *val*

If *val* is non-zero, this tells `as` to enter MRI mode. If *val* is zero, this tells `as` to exit MRI mode. This change affects code assembled until the next `.mri` directive, or until the end of the file. See Section 2.7 [MRI mode], page 10.

7.40 `.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 9), the initial value of the listing counter is one.

7.41 `.long` *expressions*

`.long` is the same as '`.int`', see Section 7.31 [`.int`], page 39.

7.42 `.macro`

The commands `.macro` and `.endm` allow you to define macros that generate assembly output. For example, this definition specifies a macro `sum` that puts a sequence of numbers into memory:

```
.macro  sum from=0, to=5
.long   \from
.if     \to-\from
sum     "(\from+1)",\to
.endif
.endm
```

With that definition, '`SUM 0,5`' is equivalent to this assembly input:

```
.long   0
.long   1
.long   2
.long   3
.long   4
.long   5
```

`.macro` *macname*
`.macro` *macname macargs* ...
: Begin the definition of a macro called *macname*. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with '=*deflt*'. For example, these are all valid `.macro` statements:

  `.macro comm`
  : Begin the definition of a macro called `comm`, which takes no arguments.

  `.macro plus1 p, p1`
  `.macro plus1 p p1`
  : Either statement begins the definition of a macro called `plus1`, which takes two arguments; within the macro definition, write '`\p`' or '`\p1`' to evaluate the arguments.

  `.macro reserve_str p1=0 p2`
  : Begin the definition of a macro called `reserve_str`, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as '`reserve_str` *a*,*b*' (with '`\p1`' evaluating to *a* and '`\p2`' evaluating to *b*), or as '`reserve_str` ,*b*' (with '`\p1`' evaluating as the default, in this case '`0`', and '`\p2`' evaluating to *b*).

  When you call a macro, you can specify the argument values either by position, or by keyword. For example, '`sum 9,17`' is equivalent to '`sum to=17, from=9`'.

`.endm`
: Mark the end of a macro definition.

`.exitm`
: Exit early from the current macro definition.

`\@`
: `as` maintains a counter of how many macros it has executed in this pseudo-variable; you can copy that number to your output with '`\@`', but *only within a macro definition*.

7.43 `.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.

7.44 `.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.

7.45 `.org` *new-lc* , *fill*

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` issues a warning, then pretends 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.

7.46 `.p2align[wl]` *abs-expr*, *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 must have after advancement. For example '`.p2align 3`' advances 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 fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.p2alignw` and `.p2alignl` directives are variants of the `.p2align` directive. The `.p2alignw` directive treats the fill pattern as a two byte word value. The `.p2alignl` directives treats the fill pattern as a four byte longword value. For example, `.p2alignw`

`2,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

7.47 `.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 do not use `.psize`, listings use a default line-count of 60. You may omit the comma and *columns* specification; the default width is 200 columns.

`as` generates 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`.

7.48 `.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.

7.49 `.rept` *count*

Repeat the sequence of lines between the `.rept` directive and the next `.endr` directive *count* times.

For example, assembling

```
.rept   3
.long   0
.endr
```

is equivalent to assembling

```
.long   0
.long   0
.long   0
```

7.50 `.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.

7.51 `.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` accepts this directive but ignores it.

7.52 `.section` *name*

Use the `.section` directive to assemble the following code into a section named *name*.

This directive is only supported for targets that actually support arbitrarily named sections; on `a.out` targets, for example, it is not accepted, even with a standard `a.out` section name.

For COFF targets, the `.section` directive is used in one of the following ways:

```
.section name[, "flags"]
.section name[, subsegment]
```

If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized:

`b` bss section (uninitialized data)

`n` section is not loaded

`w` writable section

`d` data section

`r` read-only section

`x` executable section

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to be loaded and writable.

If the optional argument to the `.section` directive is not quoted, it is taken as a subsegment number (see Section 4.4 [Sub-Sections], page 23).

For ELF targets, the `.section` directive is used like this:

```
.section name[, "flags"[, @type]]
```

The optional *flags* argument is a quoted string which may contain any combintion of the following characters:

`a` section is allocatable

`w` section is writable

`x` section is executable

The optional *type* argument may contain one of the following constants:

`@progbits`
section contains data

`@nobits` section does not contain data (i.e., section only occupies space)

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to have none of the above flags: it will not be allocated in memory, nor writable, nor executable. The section will contain data.

For ELF targets, the assembler supports another type of `.section` directive for compatibility with the Solaris assembler:

```
.section "name"[, flags...]
```

Note that the section name is quoted. There may be a sequence of comma separated flags:

`#alloc` section is allocatable

`#write` section is writable

`#execinstr`
section is executable

7.53 `.set` *symbol*, *expression*

Set the value of *symbol* to *expression*. This changes *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 28).

You may `.set` a symbol many times in the same assembly.

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

The syntax for `set` on the HPPA is '*symbol* `.set` *expression*'.

7.54 `.short` *expressions*

`.short` is normally the same as '`.word`'. See Section 7.69 [`.word`], page 50.

In some configurations, however, `.short` and `.word` generate numbers of different lengths; see Chapter 8 [Machine Dependencies], page 51.

7.55 `.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 Dependencies], page 51.

7.56 `.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.

7.57 `.sleb128` *expressions*

sleb128 stands for "signed little endian base 128." This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See Section 7.68 [Uleb128], page 49.

7.58 `.skip` *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. This is the same as '`.space`'.

7.59 `.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. This is the same as '`.skip`'.

> *Warning:* `.space` has a completely different meaning for HPPA targets; use `.block` as a substitute. See *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001) for the meaning of the `.space` directive. See Section 8.7.5 [HPPA Assembler Directives], page 67, for a summary.

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

> *Warning:* In most versions of the GNU assembler, the directive `.space` has the effect of `.block` See Chapter 8 [Machine Dependencies], page 51.

7.60 `.stabd, .stabn, .stabs`

There are three directives that begin '`.stab`'. All emit symbols (see Chapter 5 [Symbols], page 27), 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 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; you 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 is the address of the location counter 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.

7.61 `.string "str"`

Copy the characters in *str* to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a 0 byte. You can use any of the escape sequences described in Section 3.6.1.1 [Strings], page 17.

7.62 `.symver`

Use the `.symver` directive to bind symbols to specific version nodes within a source file. This is only supported on ELF platforms, and is typically used when assembling files to be linked into a shared library. There are cases where it may make sense to use this in objects to be bound into an application itself so as to override a versioned symbol from a shared library.

For ELF targets, the `.symver` directive is used like this:

```
.symver name, name2@nodename
```

In this case, the symbol *name* must exist and be defined within the file being assembled. The `.versym` directive effectively creates a symbol alias with the name *name2@nodename*, and in fact the main reason that we just don't try and create a regular alias is that the `@` character isn't permitted in symbol names. The *name2* part of the name is the actual name of the symbol by which it will be externally referenced. The name *name* itself is merely a name of convenience that is used so that it is possible to have definitions for multiple versions of a function within a single source file, and so that the compiler can unambiguously know which version of a function is being mentioned. The *nodename* portion of the alias should be the name of a node specified in the version script supplied to the linker when building a shared library. If you are attempting to override a versioned symbol from a shared library, then *nodename* should correspond to the nodename of the symbol you are trying to override.

7.63 `.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.

7.64 `.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.

7.65 `.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.

7.66 `.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.

7.67 `.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.

7.68 `.uleb128` *expressions*

uleb128 stands for "unsigned little endian base 128." This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See Section 7.57 [Sleb128], page 47.

7.69 `.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, depend on what target computer the assembly is for.

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 Dependencies], page 51), you can ignore this issue.

In order to assemble compiler output into something that works, `as` occasionlly does 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` creates a *secondary jump table*, immediately before the next label. This secondary jump table is 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 is a long-jump to `sym2`. The original '`.word`' contains `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 are adjusted. If there was a '`.word sym3-sym4`', that also did not fit in sixteen bits, a long-jump to `sym4` is included in the secondary jump table, and the `.word` directives are 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.

7.70 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 ARC Dependent Features

8.1.1 Options

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

By default, `as` assumes the core instruction set (ARC base). The `.cpu` pseudo-op is intended to be used to select the variant.

`-mbig-endian`
`-mlittle-endian`
: Any ARC configuration of `as` can select big-endian or little-endian output at run time (unlike most other GNU development tools, which must be configured for one or the other). Use '`-mbig-endian`' to select big-endian output, and '`-mlittle-endian`' for little-endian.

8.1.2 Floating Point

The ARC cpu family currently does not have hardware floating point support. Software floating point support is provided by `GCC` and uses IEEE floating-point numbers.

8.1.3 ARC Machine Directives

The ARC version of `as` supports the following additional machine directives:

`.cpu` This must be followed by the desired cpu. The ARC is intended to be customizable, `.cpu` is used to select the desired variant [though currently there are none].

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 Macros

The macro syntax used on the AMD 29K is like that described in the AMD 29K Family Macro Assembler Specification. Normal `as` macros should still work.

8.2.2.2 Special Characters

'`;`' is the line comment character.

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

8.2.2.3 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.

`.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 ARM Dependent Features

8.3.1 Options

-marm *[2|250|3|6|60|600|610|620|7|7m|7d|7dm|7di|7dmi|70|700|700i|710|710c|7100|7500|7500fe|7tdmi|8|strongarm|strongarm110]*
: This option specifies the target processor. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target processor.

-marmv *[2|2a|3|3m|4|4t]*
: This option specifies the target architecture. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target architecture.

-mthumb
: This option specifies that only Thumb instructions should be assembled.

-mall
: This option specifies that any Arm or Thumb instruction should be assembled.

-mfpa *[10|11]*
: This option specifies the floating point architecture in use on the target processor.

-mfpe-old
: Do not allow the assemble of floating point multiple instructions.

-mno-fpu
: Do not allow the assembly of any floating point instructions.

-mthumb-interwork
: This option specifies that the output generated by the assembler should be marked as supporting interworking.

-mapcs *[26|32]*
: This option specifies that the output generated by the assembler should be marked as supporting the indicated version of the Arm Procedure. Calling Standard.

-EB
: This option specifies that the output generated by the assembler should be marked as being encoded for a big-endian processor.

-EL
: This option specifies that the output generated by the assembler should be marked as being encoded for a little-endian processor.

8.3.2 Syntax

8.3.2.1 Special Characters

'`;`' is the line comment character.

TODO Explain about /data modifier on symbols.

8.3.2.2 Register Names

TODO Explain about ARM register naming, and the predefined names.

8.3.3 Floating Point

The ARM family uses IEEE floating-point numbers.

8.3.4 ARM Machine Directives

`.code` *[16|32]*
: This directive selects the instruction set being generated. The value 16 selects Thumb, with the value 32 selecting ARM.

`.thumb`
: This performs the same action as *.code 16*.

`.arm`
: This performs the same action as *.code 32*.

`.force_thumb`
: This directive forces the selection of Thumb instructions, even if the target processor does not support those instructions

`.thumb_func`
: This directive specifies that the following symbol is the name of a Thumb encoded function. This information is necessary in order to allow the assembler and linker to generate correct code for interworking between Arm and Thumb instructions and should be used even if interworking is not going to be performed.

8.3.5 Opcodes

`as` implements all the standard ARM opcodes.

TODO Document the pseudo-ops (adr, nop)

For information on the ARM or Thumb instruction sets, see *ARM Software Development Toolkit Reference Manual*, Advanced RISC Machines Ltd.

8.4 D10V Dependent Features

8.4.1 D10V Options

The Mitsubishi D10V version of `as` has a few machine dependent options.

'`-O`'
: The D10V can often execute two sub-instructions in parallel. When this option is used, `as` will attempt to optimize its output by detecting when instructions can be executed in parallel.

'`--nowarnswap`'
: To optimize execution performance, `as` will sometimes swap the order of instructions. Normally this generates a warning. When this option is used, no warning will be generated when instructions are swapped.

8.4.2 Syntax

The D10V syntax is based on the syntax in Mitsubishi's D10V architecture manual. The differences are detailed below.

8.4.2.1 Size Modifiers

The D10V version of `as` uses the instruction names in the D10V Architecture Manual. However, the names in the manual are sometimes ambiguous. There are instruction names that can assemble to a short or long form opcode. How does the assembler pick the correct form? `as` will always pick the smallest form if it can. When dealing with a symbol that is not defined yet when a line is being assembled, it will always use the long form. If you need to force the assembler to use either the short or long form of the instruction, you can append either '`.s`' (short) or '`.l`' (long) to it. For example, if you are writing an assembly program and you want to do a branch to a symbol that is defined later in your program, you can write '`bra.s foo`'. Objdump and GDB will always append '`.s`' or '`.l`' to instructions which have both short and long forms.

8.4.2.2 Sub-Instructions

The D10V assembler takes as input a series of instructions, either one-per-line, or in the special two-per-line format described in the next section. Some of these instructions will be short-form or sub-instructions. These sub-instructions can be packed into a single instruction. The assembler will do this automatically. It will also detect when it should not pack instructions. For example, when a label is defined, the next instruction will never be packaged with the previous one. Whenever a branch and link instruction is called, it will not be packaged with the next instruction so the return address will be valid. Nops are automatically inserted when necessary.

If you do not want the assembler automatically making these decisions, you can control the packaging and execution type (parallel or sequential) with the special execution symbols described in the next section.

8.4.2.3 Special Characters

'`;`' and '`#`' are the line comment characters. Sub-instructions may be executed in order, in reverse-order, or in parallel. Instructions listed in the standard one-per-line format will be executed sequentially. To specify the executing order, use the following symbols:

'`->`'
: Sequential with instruction on the left first.

'`<-`'
: Sequential with instruction on the right first.

'`||`'
: Parallel

The D10V syntax allows either one instruction per line, one instruction per line with the execution symbol, or two instructions per line. For example

`abs a1 -> abs r0`
: Execute these sequentially. The instruction on the right is in the right container and is executed second.

`abs r0 <- abs a1`
: Execute these reverse-sequentially. The instruction on the right is in the right container, and is executed first.

`ld2w r2,@r8+ || mac a0,r0,r7`
: Execute these in parallel.

`ld2w r2,@r8+ ||`
`mac a0,r0,r7`
: Two-line format. Execute these in parallel.

`ld2w r2,@r8+`
`mac a0,r0,r7`
: Two-line format. Execute these sequentially. Assembler will put them in the proper containers.

`ld2w r2,@r8+ ->`
`mac a0,r0,r7`
: Two-line format. Execute these sequentially. Same as above but second instruction will always go into right container.

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

8.4.2.4 Register Names

You can use the predefined symbols '`r0`' through '`r15`' to refer to the D10V registers. You can also use '`sp`' as an alias for '`r15`'. The accumulators are '`a0`' and '`a1`'. There are special register-pair names that may optionally be used in opcodes that require even-numbered registers. Register names are not case sensitive.

Register Pairs

`r0-r1`

`r2-r3`

`r4-r5`

`r6-r7`

`r8-r9`

`r10-r11`

`r12-r13`

`r14-r15`

The D10V also has predefined symbols for these control registers and status bits:

`psw` Processor Status Word

`bpsw` Backup Processor Status Word

`pc` Program Counter

`bpc` Backup Program Counter

`rpt_c` Repeat Count

`rpt_s` Repeat Start address

`rpt_e` Repeat End address

`mod_s` Modulo Start address

`mod_e` Modulo End address

`iba` Instruction Break Address

`f0` Flag 0

`f1` Flag 1

`c` Carry flag

8.4.2.5 Addressing Modes

`as` understands the following addressing modes for the D10V. `R`*n* in the following refers to any of the numbered registers, but *not* the control registers.

`R`*n* Register direct

`@R`*n* Register indirect

`@R`*n*`+` Register indirect with post-increment

`@R`*n*`-` Register indirect with post-decrement

`@-SP` Register indirect with pre-decrement

`@(`*disp*`, R`*n*`)`
Register indirect with displacement

addr PC relative address (for branch or rep).

`#`*imm* Immediate data (the '`#`' is optional and ignored)

8.4.2.6 @WORD Modifier

Any symbol followed by `@word` will be replaced by the symbol's value shifted right by 2. This is used in situations such as loading a register with the address of a function (or any other code fragment). For example, if you want to load a register with the location of the function `main` then jump to that function, you could do it as follws:

```
ldi     r2, main@word
jmp     r2
```

8.4.3 Floating Point

The D10V has no hardware floating point, but the `.float` and `.double` directives generates IEEE floating-point numbers for compatibility with other development tools.

8.4.4 Opcodes

For detailed information on the D10V machine instruction set, see *D10V Architecture: A VLIW Microprocessor for Multimedia Applications* (Mitsubishi Electric Corp.). `as` implements all the standard D10V opcodes. The only changes are those described in the section on size modifiers

8.5 H8/300 Dependent Features

8.5.1 Options

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

8.5.2 Syntax

8.5.2.1 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.

8.5.2.2 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).

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

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

8.5.2.3 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*`)`
`@(`*d*`:24, r`*n*`)`
Register indirect: 16-bit or 24-bit displacement *d* from register *n*. (24-bit displacements are only meaningful on the H8/300H.)

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

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

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

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

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

8.5.3 Floating Point

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

8.5.4 H8/300 Machine Directives

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

`.h8300h` Recognize and emit additional instructions for the H8/300H variant, and also make `.int` emit 32-bit numbers rather than the usual (16-bit) for the H8/300 family.

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

8.5.5 Opcodes

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

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

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

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

```
mov    r0,@foo
```

is equivalent to

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

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

8.6 H8/500 Dependent Features

8.6.1 Options

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

8.6.2 Syntax

8.6.2.1 Special Characters

'`!`' is the line comment character.

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

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

8.6.2.2 Register Names

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

The H8/500 also has these control registers:

`cp` code pointer

`dp` data pointer

`bp` base pointer

`tp` stack top pointer

`ep` extra pointer

`sr` status register

`ccr` condition code register

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

8.6.2.3 Addressing Modes

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

`R`*n* Register direct

`@R`*n* Register indirect

`@(d:8, R`*n*`)`
  Register indirect with 8 bit signed displacement

`@(d:16, R`*n*`)`
  Register indirect with 16 bit signed displacement

`@-R`*n* Register indirect with pre-decrement

`@Rn+` Register indirect with post-increment

`@aa:8` 8 bit absolute address

`@aa:16` 16 bit absolute address

`#xx:8` 8 bit immediate

`#xx:16` 16 bit immediate

8.6.3 Floating Point

The H8/500 family has no hardware floating point, but the `.float` directive generates IEEE floating-point numbers for compatibility with other development tools.

8.6.4 H8/500 Machine Directives

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

8.6.5 Opcodes

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

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

8.7 HPPA Dependent Features

8.7.1 Notes

As a back end for GNU CC `as` has been throughly tested and should work extremely well. We have tested it only minimally on hand written assembly code and no one has tested it much on the assembly output from the HP compilers.

The format of the debugging sections has changed since the original `as` port (version 1.3X) was released; therefore, you must rebuild all HPPA objects and libraries with the new assembler so that you can debug the final executable.

The HPPA `as` port generates a small subset of the relocations available in the SOM and ELF object file formats. Additional relocation support will be added as it becomes necessary.

8.7.2 Options

`as` has no machine-dependent command-line options for the HPPA.

8.7.3 Syntax

The assembler syntax closely follows the HPPA instruction set reference manual; assembler directives and general syntax closely follow the HPPA assembly language reference manual, with a few noteworthy differences.

First, a colon may immediately follow a label definition. This is simply for compatibility with how most assembly language programmers write code.

Some obscure expression parsing problems may affect hand written code which uses the `spop` instructions, or code which makes significant use of the `!` line separator.

`as` is much less forgiving about missing arguments and other similar oversights than the HP assembler. `as` notifies you of missing arguments as syntax errors; this is regarded as a feature, not a bug.

Finally, `as` allows you to use an external symbol without explicitly importing the symbol. *Warning:* in the future this will be an error for HPPA targets.

Special characters for HPPA targets include:

'`;`' is the line comment character.

'`!`' can be used instead of a newline to separate statements.

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

8.7.4 Floating Point

The HPPA family uses IEEE floating-point numbers.

8.7.5 HPPA Assembler Directives

`as` for the HPPA supports many additional directives for compatibility with the native assembler. This section describes them only briefly. For detailed information on HPPA-specific assembler directives, see *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001).

`as` does *not* support the following assembler directives described in the HP manual:

```
.endm           .liston
.enter          .locct
.leave          .macro
.listoff
```

Beyond those implemented for compatibility, `as` supports one additional assembler directive for the HPPA: `.param`. It conveys register argument locations for static functions. Its syntax closely follows the `.export` directive.

These are the additional directives in `as` for the HPPA:

`.block` *n*
`.blockz` *n* Reserve *n* bytes of storage, and initialize them to zero.

`.call` Mark the beginning of a procedure call. Only the special case with *no arguments* is allowed.

`.callinfo [` *param*=*value*`, ... ] [` *flag*`, ... ]`
Specify a number of parameters and flags that define the environment for a procedure.

param may be any of '`frame`' (frame size), '`entry_gr`' (end of general register range), '`entry_fr`' (end of float register range), '`entry_sr`' (end of space register range).

The values for *flag* are '`calls`' or '`caller`' (proc has subroutines), '`no_calls`' (proc does not call subroutines), '`save_rp`' (preserve return pointer), '`save_sp`' (proc preserves stack pointer), '`no_unwind`' (do not unwind this proc), '`hpux_int`' (proc is interrupt routine).

`.code` Assemble into the standard section called '`$TEXT$`', subsection '`$CODE$`'.

`.copyright "`*string*`"`
In the SOM object format, insert *string* into the object code, marked as a copyright string.

`.copyright "`*string*`"`
In the ELF object format, insert *string* into the object code, marked as a version string.

`.enter` Not yet supported; the assembler rejects programs containing this directive.

`.entry` Mark the beginning of a procedure.

`.exit` Mark the end of a procedure.

`.export` *name* `[ ,`*typ* `] [ ,`*param*=*r* `]`
: Make a procedure *name* available to callers. *typ*, if present, must be one of '`absolute`', '`code`' (ELF only, not SOM), '`data`', '`entry`', '`data`', '`entry`', '`millicode`', '`plabel`', '`pri_prog`', or '`sec_prog`'.

  param, if present, provides either relocation information for the procedure arguments and result, or a privilege level. *param* may be '`argw`*n*' (where *n* ranges from `0` to `3`, and indicates one of four one-word arguments); '`rtnval`' (the procedure's result); or '`priv_lev`' (privilege level). For arguments or the result, *r* specifies how to relocate, and must be one of '`no`' (not relocatable), '`gr`' (argument is in general register), '`fr`' (in floating point register), or '`fu`' (upper half of float register). For '`priv_lev`', *r* is an integer.

`.half` *n*
: Define a two-byte integer constant *n*; synonym for the portable `as` directive `.short`.

`.import` *name* `[ ,`*typ* `]`
: Converse of `.export`; make a procedure available to call. The arguments use the same conventions as the first two arguments for `.export`.

`.label` *name*
: Define *name* as a label for the current assembly location.

`.leave`
: Not yet supported; the assembler rejects programs containing this directive.

`.origin` *lc*
: Advance location counter to *lc*. Synonym for the `{No value for ``as''}` portable directive `.org`.

`.param` *name* `[ ,`*typ* `] [ ,`*param*=*r* `]`
: Similar to `.export`, but used for static procedures.

`.proc`
: Use preceding the first statement of a procedure.

`.procend`
: Use following the last statement of a procedure.

label `.reg` *expr*
: Synonym for `.equ`; define *label* with the absolute expression *expr* as its value.

`.space` *secname* `[ ,`*params* `]`
: Switch to section *secname*, creating a new section by that name if necessary. You may only use *params* when creating a new section, not when switching to an existing one. *secname* may identify a section by number rather than by name.

  If specified, the list *params* declares attributes of the section, identified by keywords. The keywords recognized are '`spnum=`*exp*' (identify this section by the number *exp*, an absolute expression), '`sort=`*exp*' (order sections according to this sort key when linking; *exp* is an absolute expression), '`unloadable`' (section contains no loadable data), '`notdefined`' (this section defined elsewhere), and '`private`' (data in this section not available to other programs).

`.spnum` *secnam*

Allocate four bytes of storage, and initialize them with the section number of the section named *secnam*. (You can define the section number with the HPPA `.space` directive.)

`.string "`*str*`"`

Copy the characters in the string *str* to the object file. See Section 3.6.1.1 [Strings], page 17, for information on escape sequences you can use in `as` strings.

Warning! The HPPA version of `.string` differs from the usual `as` definition: it does *not* write a zero byte after copying *str*.

`.stringz "`*str*`"`

Like `.string`, but appends a zero byte after copying *str* to object file.

`.subspa` *name* `[ ,`*params* `]`
`.nsubspa` *name* `[ ,`*params* `]`

Similar to `.space`, but selects a subsection *name* within the current section. You may only specify *params* when you create a subsection (in the first instance of `.subspa` for this *name*).

If specified, the list *params* declares attributes of the subsection, identified by keywords. The keywords recognized are '`quad=`*expr*' ("quadrant" for this subsection), '`align=`*expr*' (alignment for beginning of this subsection; a power of two), '`access=`*expr*' (value for "access rights" field), '`sort=`*expr*' (sorting order for this subspace in link), '`code_only`' (subsection contains only code), '`unloadable`' (subsection cannot be loaded into memory), '`common`' (subsection is common block), '`dup_comm`' (initialized data may have duplicate names), or '`zero`' (subsection is all zeros, do not write in object file).

`.nsubspa` always creates a new subspace with the given name, even if one with the same name already exists.

`.version "`*str*`"`

Write *str* as version identifier in object code.

8.7.6 Opcodes

For detailed information on the HPPA machine instruction set, see *PA-RISC Architecture and Instruction Set Reference Manual* (HP 09740-90039).

8.8 80386 Dependent Features

8.8.1 Options

The 80386 has no machine dependent options.

8.8.2 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.

8.8.3 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.

8.8.4 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)`'.

8.8.5 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.

8.8.6 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` does *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` always chooses 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).

8.8.7 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 if you use these instructions (`gcc` does not use them) you may get an error message (and 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:
```

8.8.8 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)'.

8.8.9 Writing 16-bit Code

While GAS normally writes only "pure" 32-bit i386 code, it has limited support for writing code to run in real mode or in 16-bit protected mode code segments. To do this, insert a '`.code16`' directive before the assembly language instructions to be run in 16-bit mode. You can switch GAS back to writing normal 32-bit code with the '`.code32`' directive.

GAS understands exactly the same assembly language syntax in 16-bit mode as in 32-bit mode. The function of any given instruction is exactly the same regardless of mode, as long as the resulting object code is executed in the mode for which GAS wrote it. So, for example, the '`ret`' mnemonic produces a 32-bit return instruction regardless of whether it is to be run in 16-bit or 32-bit mode. (If GAS is in 16-bit mode, it will add an operand size prefix to the instruction to force it to be a 32-bit return.)

This means, for one thing, that you can use GNU CC to write code to be run in real mode or 16-bit protected mode. Just insert the statement '`asm(".code16");`' at the beginning of your C source file, and while GNU CC will still be generating 32-bit code, GAS will automatically add all the necessary size prefixes to make that code run in 16-bit mode. Of course, since GNU CC only writes small-model code (it doesn't know how to attach segment selectors to pointers like native x86 compilers do), any 16-bit code you write with GNU CC will essentially be limited to a 64K address space. Also, there will be a code size and performance penalty due to all the extra address and operand size prefixes GAS has to add to the instructions.

Note that placing GAS in 16-bit mode does not mean that the resulting code will necessarily run on a 16-bit pre-80386 processor. To write code that runs on such a processor, you would have to refrain from using *any* 32-bit constructs which require GAS to output address or operand size prefixes. At the moment this would be rather difficult, because GAS currently supports *only* 32-bit addressing modes: when writing 16-bit code, it *always* outputs address size prefixes for any instruction that uses a non-register addressing mode. So you can write code that runs on 16-bit processors, but only if that code never references memory.

8.8.10 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`'.

8.9 Intel 80960 Dependent Features

8.9.1 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 you do not specify any of these options, `as` generates code for any instruction or feature that is supported by *some* version of the 960 (even if this means mixing architectures!). In principle, `as` attempts 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.

*NEXT	COUNT: N	*BRLAB 1	...	*BRLAB N

__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 generates these calls automatically when you give it a '`-b`' option. For further details, see the documentation of '`gbr960`'.

`-no-relax`

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 '`-no-relax`' 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 '`-no-relax`'.

8.9.2 Floating Point

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

8.9.3 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 7.34 [.lcomm], page 40.

`.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).

8.9.4 i960 Opcodes

All Intel 960 machine instructions are supported; see Section 8.9.1 [i960 Command-line Options], page 75 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.

8.9.4.1 `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` translates the `callj`; if not, it simply emits the `callj`, leaving it for the linker to resolve.

8.9.4.2 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 '`-no-relax`' 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 `-no-relax`—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:

Compare and		
Branch	*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
```

8.10 M680x0 Dependent Features

8.10.1 M680x0 Options

The Motorola 680x0 version of `as` has a few machine dependent options.

You can use the '`-l`' option to shorten the size of references to undefined symbols. If you do not use the '`-l`' option, references to undefined symbols are wide enough for a full `long` (32 bits). (Since `as` cannot know where these symbols end up, `as` can only allocate space for the linker to fill in later. Since `as` does not know how far away these symbols are, it allocates as much space as it can.) If you use this option, the references are only 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 are always less than 17 bits away.

For some configurations, especially those where the compiler normally does not prepend an underscore to the names of user variables, the assembler requires a '`%`' before any use of a register name. This is intended to let the assembler distinguish between C variables and functions named '`a0`' through '`a7`', and so on. The '`%`' is always accepted, but is not required for certain configurations, notably '`sun3`'. The '`--register-prefix-optional`' option may be used to permit omitting the '`%`' even for configurations for which it is normally required. If this is done, it will generally be impossible to refer to C variables and functions with the same names as register names.

Normally the character '`|`' is treated as a comment character, which means that it can not be used in expressions. The '`--bitwise-or`' option turns '`|`' into a normal character. In this mode, you must either use C style comments, or start comments with a '`#`' character at the beginning of a line.

If you use an addressing mode with a base register without specifying the size, `as` will normally use the full 32 bit value. For example, the addressing mode '`%a0@(%d0)`' is equivalent to '`%a0@(%d0:l)`'. You may use the '`--base-size-default-16`' option to tell `as` to default to using the 16 bit value. In this case, '`%a0@(%d0)`' is equivalent to '`%a0@(%d0:w)`'. You may use the '`--base-size-default-32`' option to restore the default behaviour.

If you use an addressing mode with a displacement, and the value of the displacement is not known, `as` will normally assume that the value is 32 bits. For example, if the symbol '`disp`' has not been defined, `as` will assemble the addressing mode '`%a0@(disp,%d0)`' as though '`disp`' is a 32 bit value. You may use the '`--disp-size-default-16`' option to tell `as` to instead assume that the displacement is 16 bits. In this case, `as` will assemble '`%a0@(disp,%d0)`' as though '`disp`' is a 16 bit value. You may use the '`--disp-size-default-32`' option to restore the default behaviour.

`as` can assemble code for several different members of the Motorola 680x0 family. The default depends upon how `as` was configured when it was built; normally, the default is to assemble code for the 68020 microprocessor. The following options may be used to change the default. These options control which instructions and addressing modes are permitted. The members of the 680x0 family are very similar. For detailed information about the differences, see the Motorola manuals.

'-m68000'
'-m68ec000'
'-m68hc000'
'-m68hc001'
'-m68008'
'-m68302'
'-m68306'
'-m68307'
'-m68322'
'-m68356' Assemble for the 68000. '-m68008', '-m68302', and so on are synonyms for '-m68000', since the chips are the same from the point of view of the assembler.

'-m68010' Assemble for the 68010.

'-m68020'
'-m68ec020'
Assemble for the 68020. This is normally the default.

'-m68030'
'-m68ec030'
Assemble for the 68030.

'-m68040'
'-m68ec040'
Assemble for the 68040.

'-m68060'
'-m68ec060'
Assemble for the 68060.

'-mcpu32'
'-m68330'
'-m68331'
'-m68332'
'-m68333'
'-m68334'
'-m68336'
'-m68340'
'-m68341'
'-m68349'
'-m68360' Assemble for the CPU32 family of chips.

'-m5200' Assemble for the ColdFire family of chips.

'-m68881'
'-m68882' Assemble 68881 floating point instructions. This is the default for the 68020, 68030, and the CPU32. The 68040 and 68060 always support floating point instructions.

'-mno-68881'
: Do not assemble 68881 floating point instructions. This is the default for 68000 and the 68010. The 68040 and 68060 always support floating point instructions, even if this option is used.

'-m68851'
: Assemble 68851 MMU instructions. This is the default for the 68020, 68030, and 68060. The 68040 accepts a somewhat different set of MMU instructions; '-m68851' and '-m68040' should not be used together.

'-mno-68851'
: Do not assemble 68851 MMU instructions. This is the default for the 68000, 68010, and the CPU32. The 68040 accepts a somewhat different set of MMU instructions.

8.10.2 Syntax

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

The 680x0 version of `as` uses instructions names and syntax compatible with the Sun assembler. Intervening periods are ignored; for example, '`movl`' is equivalent to '`mov.l`'.

In the following table *apc* stands for any of the address registers ('`%a0`' through '`%a7`'), the program counter ('`%pc`'), the zero-address relative to the program counter ('`%zpc`'), a suppressed address register ('`%za0`' through '`%za7`'), or it may be omitted entirely. The use of *size* means one of '`w`' or '`l`', and it may be omitted, along with the leading colon, unless a scale is also specified. The use of *scale* means one of '`1`', '`2`', '`4`', or '`8`', and it may always be omitted along with the leading colon.

The following addressing modes are understood:

Immediate
: '`#`*number*'

Data Register
: '`%d0`' through '`%d7`'

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

Address Register Indirect
: '`%a0@`' through '`%a7@`'

Address Register Postincrement
: '`%a0@+`' through '`%a7@+`'

Address Register Predecrement
: '`%a0@-`' through '`%a7@-`'

Indirect Plus Offset
: '*apc*`@(`*number*`)`'

Index
: '*apc*`@(`*number*`,`*register*`:`*size*`:`*scale*`)`'
: The *number* may be omitted.

Postindex '*apc*@(*number*)@(*onumber*,*register*:*size*:*scale*)'
: The *onumber* or the *register*, but not both, may be omitted.

Preindex '*apc*@(*number*,*register*:*size*:*scale*)@(*onumber*)'
: The *number* may be omitted. Omitting the *register* produces the Postindex addressing mode.

Absolute '*symbol*', or '*digits*', optionally followed by ':b', ':w', or ':l'.

8.10.3 Motorola Syntax

The standard Motorola syntax for this chip differs from the syntax already discussed (see Section 8.10.2 [Syntax], page 81). `as` can accept Motorola syntax for operands, even if MIT syntax is used for other operands in the same instruction. The two kinds of syntax are fully compatible.

In the following table *apc* stands for any of the address registers ('%a0' through '%a7'), the program counter ('%pc'), the zero-address relative to the program counter ('%zpc'), or a suppressed address register ('%za0' through '%za7'). The use of *size* means one of 'w' or 'l', and it may always be omitted along with the leading dot. The use of *scale* means one of '1', '2', '4', or '8', and it may always be omitted along with the leading asterisk.

The following additional addressing modes are understood:

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

Address Register Postincrement
: '(%a0)+' through '(%a7)+'

Address Register Predecrement
: '-(%a0)' through '-(%a7)'

Indirect Plus Offset
: '*number*(*%a0*)' through '*number*(*%a7*)', or '*number*(*%pc*)'.

The *number* may also appear within the parentheses, as in '(*number*,*%a0*)'. When used with the *pc*, the *number* may be omitted (with an address register, omitting the *number* produces Address Register Indirect mode).

Index '*number*(*apc*,*register*.*size***scale*)'
: The *number* may be omitted, or it may appear within the parentheses. The *apc* may be omitted. The *register* and the *apc* may appear in either order. If both *apc* and *register* are address registers, and the *size* and *scale* are omitted, then the first register is taken as the base register, and the second as the index register.

Postindex '([*number*,*apc*],*register*.*size***scale*,*onumber*)'
: The *onumber*, or the *register*, or both, may be omitted. Either the *number* or the *apc* may be omitted, but not both.

Preindex `([number,apc,register.size*scale],onumber)`

The *number*, or the *apc*, or the *register*, or any two of them, may be omitted. The *onumber* may be omitted. The *register* and the *apc* may appear in either order. If both *apc* and *register* are address registers, and the *size* and *scale* are omitted, then the first register is taken as the base register, and the second as the index register.

8.10.4 Floating Point

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.

`.extend`
`.ldouble` `Extended` precision (`long double`) floating point constants.

8.10.5 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 a special case of the `.align` directive; it aligns the output to an even byte boundary.

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

8.10.6 Opcodes

8.10.6.1 Branch Improvement

Certain pseudo opcodes are permitted for branch instructions. They expand to the shortest branch instruction that 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` issues a longer code fragment in terms of *NX*, the opposite condition to *XX*. For example, for the non-PC relative case:

```
    jXX foo
```

gives

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

`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` emits

```
     dbXX 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`'.

8.10.6.2 Special Characters

The immediate character is '`#`' for Sun compatibility. The line-comment character is '`|`' (unless the '`--bitwise-or`' option is used). 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.

8.11 MIPS Dependent Features

GNU `as` for MIPS architectures supports several different MIPS processors, and MIPS ISA levels I through IV. For information about the MIPS instruction set, see *MIPS RISC Architecture*, by Kane and Heindrich (Prentice-Hall). For an overview of MIPS assembly conventions, see "Appendix D: Assembly Language Programming" in the same work.

8.11.1 Assembler options

The MIPS configurations of GNU `as` support these special options:

`-G` *num*
: This option sets the largest size of an object that can be referenced implicitly with the `gp` register. It is only accepted for targets that use ECOFF format. The default value is 8.

`-EB`
`-EL`
: Any MIPS configuration of `as` can select big-endian or little-endian output at run time (unlike the other GNU development tools, which must be configured for one or the other). Use '`-EB`' to select big-endian output, and '`-EL`' for little-endian.

`-mips1`
`-mips2`
`-mips3`
`-mips4`
: Generate code for a particular MIPS Instruction Set Architecture level. '`-mips1`' corresponds to the R2000 and R3000 processors, '`-mips2`' to the R6000 processor, '`-mips3`' to the R4000 processor, and '`-mips4`' to the R8000 and R10000 processors. You can also switch instruction sets during the assembly; see Section 8.11.4 [Directives to override the ISA level], page 88.

`-mips16`
`-no-mips16`
: Generate code for the MIPS 16 processor. This is equivalent to putting '`.set mips16`' at the start of the assembly file. '`-no-mips16`' turns off this option.

`-m4650`
`-no-m4650`
: Generate code for the MIPS R4650 chip. This tells the assembler to accept the '`mad`' and '`madu`' instruction, and to not schedule '`nop`' instructions around accesses to the '`HI`' and '`LO`' registers. '`-no-m4650`' turns off this option.

`-m4010`
`-no-m4010`
: Generate code for the LSI R4010 chip. This tells the assembler to accept the R4010 specific instructions ('`addciu`', '`ffc`', etc.), and to not schedule '`nop`' instructions around accesses to the '`HI`' and '`LO`' registers. '`-no-m4010`' turns off this option.

`-mcpu=`*CPU*
: Generate code for a particular MIPS cpu. This has little effect on the assembler, but it is passed by `gcc`.

`-nocpp` This option is ignored. It is accepted for command-line compatibility with other assemblers, which use it to turn off C style preprocessing. With GNU `as`, there is no need for '`-nocpp`', because the GNU assembler itself never runs the C preprocessor.

`--trap`
`--no-break`
: `as` automatically macro expands certain division and multiplication instructions to check for overflow and division by zero. This option causes `as` to generate code to take a trap exception rather than a break exception when an error is detected. The trap instructions are only supported at Instruction Set Architecture level 2 and higher.

`--break`
`--no-trap`
: Generate code to take a break exception rather than a trap exception when an error is detected. This is the default.

8.11.2 MIPS ECOFF object code

Assembling for a MIPS ECOFF target supports some additional sections besides the usual `.text`, `.data` and `.bss`. The additional sections are `.rdata`, used for read-only data, `.sdata`, used for small data, and `.sbss`, used for small common objects.

When assembling for ECOFF, the assembler uses the `$gp` (`$28`) register to form the address of a "small object". Any object in the `.sdata` or `.sbss` sections is considered "small" in this sense. For external objects, or for objects in the `.bss` section, you can use the `gcc` '`-G`' option to control the size of objects addressed via `$gp`; the default value is 8, meaning that a reference to any object eight bytes or smaller uses `$gp`. Passing '`-G 0`' to `as` prevents it from using the `$gp` register on the basis of object size (but the assembler uses `$gp` for objects in `.sdata` or `sbss` in any case). The size of an object in the `.bss` section is set by the `.comm` or `.lcomm` directive that defines it. The size of an external object may be set with the `.extern` directive. For example, '`.extern sym,4`' declares that the object at `sym` is 4 bytes in length, whie leaving `sym` otherwise undefined.

Using small ECOFF objects requires linker support, and assumes that the `$gp` register is correctly initialized (normally done automatically by the startup code). MIPS ECOFF assembly code must not modify the `$gp` register.

8.11.3 Directives for debugging information

MIPS ECOFF `as` supports several directives used for generating debugging information which are not support by traditional MIPS assemblers. These are `.def`, `.endef`, `.dim`, `.file`, `.scl`, `.size`, `.tag`, `.type`, `.val`, `.stabd`, `.stabn`, and `.stabs`. The debugging information generated by the three `.stab` directives can only be read by GDB, not by traditional MIPS debuggers (this enhancement is required to fully support C++ debugging). These directives are primarily used by compilers, not assembly language programmers!

8.11.4 Directives to override the ISA level

GNU `as` supports an additional directive to change the MIPS Instruction Set Architecture level on the fly: `.set mips`*n*. *n* should be a number from 0 to 4. A value from 1 to 4 makes the assembler accept instructions for the corresponding ISA level, from that point on in the assembly. `.set mips`*n* affects not only which instructions are permitted, but also how certain macros are expanded. `.set mips0` restores the ISA level to its original level: either the level you selected with command line options, or the default for your configuration. You can use this feature to permit specific R4000 instructions while assembling in 32 bit mode. Use this directive with care!

The directive '`.set mips16`' puts the assembler into MIPS 16 mode, in which it will assemble instructions for the MIPS 16 processor. Use '`.set nomips16`' to return to normal 32 bit mode.

Traditional MIPS assemblers do not support this directive.

8.11.5 Directives for extending MIPS 16 bit instructions

By default, MIPS 16 instructions are automatically extended to 32 bits when necessary. The directive '`.set noautoextend`' will turn this off. When '`.set noautoextend`' is in effect, any 32 bit instruction must be explicitly extended with the '`.e`' modifier (e.g., '`li.e $4,1000`'). The directive '`.set autoextend`' may be used to once again automatically extend instructions when necessary.

This directive is only meaningful when in MIPS 16 mode. Traditional MIPS assemblers do not support this directive.

8.11.6 Directive to mark data as an instruction

The `.insn` directive tells `as` that the following data is actually instructions. This makes a difference in MIPS 16 mode: when loading the address of a label which precedes instructions, `as` automatically adds 1 to the value, so that jumping to the loaded address will do the right thing.

8.11.7 Directives to save and restore options

The directives `.set push` and `.set pop` may be used to save and restore the current settings for all the options which are controlled by `.set`. The `.set push` directive saves the current settings on a stack. The `.set pop` directive pops the stack and restores the settings.

These directives can be useful inside an macro which must change an option such as the ISA level or instruction reordering but does not want to change the state of the code which invoked the macro.

Traditional MIPS assemblers do not support these directives.

8.12 Hitachi SH Dependent Features

8.12.1 Options

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

8.12.2 Syntax

8.12.2.1 Special Characters

'`!`' is the line comment character.

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

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

8.12.2.2 Register Names

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

The SH also has these control registers:

`pr` procedure register (holds return address)

`pc` program counter

`mach`
`macl` high and low multiply accumulator registers

`sr` status register

`gbr` global base register

`vbr` vector base register (for interrupt vectors)

8.12.2.3 Addressing Modes

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

`R`*n* Register direct

`@R`*n* Register indirect

`@-R`*n* Register indirect with pre-decrement

`@R`*n*`+` Register indirect with post-increment

`@(`*disp*`, R`*n*`)`
Register indirect with displacement

`@(R0, R`*n*`)` Register indexed

`@(`*disp*`, GBR)`
`GBR` offset

`@(R0, GBR)`
GBR indexed

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

`#`*imm* Immediate data

8.12.3 Floating Point

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

8.12.4 SH Machine Directives

`uaword`
`ualong` `as` will issue a warning when a misaligned `.word` or `.long` directive is used. You may use `.uaword` or `.ualong` to indicate that the value is intentionally misaligned.

8.12.5 Opcodes

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

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

```
mov.l  bar,r0
```

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

```
mov.l  @(disp, PC)
```

8.13 SPARC Dependent Features

8.13.1 Options

The SPARC chip family includes several successive levels, using the same core instruction set, but including a few additional instructions at each level. There are exceptions to this however. For details on what instructions each variant supports, please see the chip's architecture reference manual.

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

If not configured for SPARC v9 (`sparc64-*-*`) GAS will not bump passed sparclite by default, an option must be passed to enable the v9 instructions.

GAS treats sparclite as being compatible with v8, unless an architecture is explicitly requested. SPARC v9 is always incompatible with sparclite.

`-Av6 | -Av7 | -Av8 | -Asparclet | -Asparclite`
`-Av8plus | -Av8plusa | -Av9 | -Av9a`
: Use one of the '`-A`' options to select one of the SPARC architectures explicitly. If you select an architecture explicitly, `as` reports a fatal error if it encounters an instruction or feature requiring an incompatible or higher level.

  '`-Av8plus`' and '`-Av8plusa`' select a 32 bit environment.

  '`-Av9`' and '`-Av9a`' select a 64 bit environment and are not available unless GAS is explicitly configured with 64 bit environment support.

  '`-Av8plusa`' and '`-Av9a`' enable the SPARC V9 instruction set with UltraSPARC extensions.

`-xarch=v8plus | -xarch=v8plusa`
: For compatibility with the Solaris v9 assembler. These options are equivalent to -Av8plus and -Av8plusa, respectively.

`-bump`
: Warn whenever it is necessary to switch to another level. If an architecture level is explicitly requested, GAS will not issue warnings until that level is reached, and will then bump the level as required (except between incompatible levels).

`-32 | -64`
: Select the word size, either 32 bits or 64 bits. These options are only available with the ELF object file format, and require that the necessary BFD support has been included.

8.13.2 Enforcing aligned data

SPARC GAS normally permits data to be misaligned. For example, it permits the `.long` pseudo-op to be used on a byte boundary. However, the native SunOS and Solaris assemblers issue an error when they see misaligned data.

You can use the `--enforce-aligned-data` option to make SPARC GAS also issue an error about misaligned data, just as the SunOS and Solaris assemblers do.

The `--enforce-aligned-data` option is not the default because gcc issues misaligned data pseudo-ops when it initializes certain packed data structures (structures defined using the `packed` attribute). You may have to assemble with GAS in order to initialize packed data structures in your own code.

8.13.3 Floating Point

The Sparc uses IEEE floating-point numbers.

8.13.4 Sparc Machine Directives

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

`.align` This must be followed by the desired alignment in bytes.

`.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.

`.xword` On the Sparc V9 processor, the `.xword` directive produces 64 bit values.

8.14 Z8000 Dependent Features

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

When the assembler is in unsegmented mode (specified with the `unsegm` directive), an address takes up one word (16 bit) sized register. When the assembler is in segmented mode (specified with the `segm` directive), a 24-bit address takes up a long (32 bit) register. See Section 8.14.3 [Assembler Directives for the Z8000], page 94, for a list of other Z8000 specific assembler directives.

8.14.1 Options

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

8.14.2 Syntax

8.14.2.1 Special Characters

'`!`' is the line comment character.

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

8.14.2.2 Register Names

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

byte registers

```
r0l r0h r1h r1l r2h r2l r3h r3l
r4h r4l r5h r5l r6h r6l r7h r7l
```

word registers

```
r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15
```

long word registers

```
rr0 rr2 rr4 rr6 rr8 rr10 rr12 rr14
```

quad word registers

```
rq0 rq4 rq8 rq12
```

8.14.2.3 Addressing Modes

as understands the following addressing modes for the Z8000:

`r`*n* Register direct

`@r`*n* Indirect register

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

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

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

`r`*n*`(r`*m*`)` Base Index: the 16 or 24 bit register r*n* is added to the sign extended 16 bit index register r*m* to produce the final address in memory of the operand.

`#`*xx* Immediate data *xx*.

8.14.3 Assembler Directives for the Z8000

The Z8000 port of as includes these additional assembler directives, for compatibility with other Z8000 assemblers. As shown, these do not begin with '`.`' (unlike the ordinary as directives).

`segm` Generates code for the segmented Z8001.

`unsegm` Generates code for the unsegmented Z8002.

`name` Synonym for `.file`

`global` Synonym for `.global`

`wval` Synonym for `.word`

`lval` Synonym for `.long`

`bval` Synonym for `.byte`

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

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

`rsect` synonym for `.section`

`block` synonym for `.space`

`even` special case of `.align`; aligns output to even byte boundary.

8.14.4 Opcodes

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

8.15 VAX Dependent Features

8.15.1 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. Since `as` does not use a temporary disk file, this option makes no difference. '`-t`' needs exactly one filename.

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

8.15.2 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.

8.15.3 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.

8.15.4 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.

8.15.5 VAX Branch Improvement

Certain pseudo opcodes are permitted. They are for branch instructions. They expand to the shortest branch instruction that reaches 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 do not need this feature, avoid 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:
```

8.15.6 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`. Upper and lower case letters are equivalent.

For instance

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

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

8.15.7 Not Supported on VAX

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

8.16 v850 Dependent Features

8.16.1 Options

as supports the following additional command-line options for the V850 processor family:

`-wsigned_overflow`
: Causes warnings to be produced when signed immediate values overflow the space available for then within their opcodes. By default this option is disabled as it is possible to receive spurious warnings due to using exact bit patterns as immediate constants.

`-wunsigned_overflow`
: Causes warnings to be produced when unsigned immediate values overflow the space available for then within their opcodes. By default this option is disabled as it is possible to receive spurious warnings due to using exact bit patterns as immediate constants.

`-mv850`
: Specifies that the assembled code should be marked as being targeted at the V850 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

8.16.2 Syntax

8.16.2.1 Special Characters

'#' is the line comment character.

8.16.2.2 Register Names

as supports the following names for registers:

`general register 0`
: r0, zero

`general register 1`
: r1

`general register 2`
: r2, hp

`general register 3`
: r3, sp

`general register 4`
: r4, gp

`general register 5`
: r5, tp

`general register 6`
: r6

`general register 7`
: r7

`general register 8`
: r8

`general register 9`
: r9

`general register 10`
: r10

`general register 11`
: r11

`general register 12`
: r12

`general register 13`
: r13

`general register 14`
: r14

`general register 15`
: r15

`general register 16`
: r16

`general register 17`
: r17

`general register 18`
: r18

`general register 19`
: r19

`general register 20`
: r20

`general register 21`
: r21

`general register 22`
: r22

`general register 23`
: r23

`general register 24`
: r24

`general register 25`
: r25

`general register 26`
: r26

`general register 27`
: r27

`general register 28`
: r28

`general register 29`
: r29

`general register 30`
: r30, ep

`general register 31`
: r31, lp

`system register 0`
: eipc

`system register 1`
: eipsw

`system register 2`
: fepc

`system register 3`
: fepsw

`system register 4`
: ecr

`system register 5`
: psw

8.16.3 Floating Point

The V850 family uses IEEE floating-point numbers.

8.16.4 V850 Machine Directives

`.offset` <*expression*>
: Moves the offset into the current section to the specified amount.

`.section "name", <type>`
: This is an extension to the standard .section directive. It sets the current section to be `<type>` and creates an alias for this section called `"name"`.

`.v850`
: Specifies that the assembled code should be marked as being targeted at the V850 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

8.16.5 Opcodes

`as` implements all the standard V850 opcodes.

`as` also implements the following pseudo ops:

`hi0()`
: Computes the higher 16 bits of the given expression and stores it into the immediate operand field of the given instruction. For example:

  '`mulhi hi0(here - there), r5, r6`'

  computes the difference between the address of labels 'here' and 'there', takes the upper 16 bits of this difference, shifts it down 16 bits and then mutliplies it by the lower 16 bits in register 5, putting the result into register 6.

`lo()`
: Computes the lower 16 bits of the given expression and stores it into the immediate operand field of the given instruction. For example:

  '`addi lo(here - there), r5, r6`'

  computes the difference between the address of labels 'here' and 'there', takes the lower 16 bits of this difference and adds it to register 5, putting the result into register 6.

`hi()`
: Computes the higher 16 bits of the given expression and then adds the value of the most significant bit of the lower 16 bits of the expression and stores the result into the immediate operand field of the given instruction. For example the following code can be used to compute the address of the label 'here' and store it into register 6:

  '`movhi hi(here), r0, r6`' '`movea lo(here), r6, r6`'

  The reason for this special behaviour is that movea performs a sign extention on its immediate operand. So for example if the address of 'here' was 0xFFFFFFFF then without the special behaviour of the hi() pseudo-op the movhi instruction would put 0xFFFF0000 into r6, then the movea instruction would takes its immediate operand, 0xFFFF, sign extend it to 32 bits, 0xFFFFFFFF, and then add it into r6 giving 0xFFFEFFFF which is wrong (the fifth nibble is E). With the hi() pseudo op adding in the top bit of the lo() pseudo op, the movhi instruction actually stores 0 into r6 (0xFFFF + 1 = 0x0000), so that the movea instruction stores 0xFFFFFFFF into r6 - the right value.

`sdaoff()`
: Computes the offset of the named variable from the start of the Small Data Area (whoes address is held in register 4, the GP register) and stores the result as a 16 bit signed value in the immediate operand field of the given instruction. For example:

  '`ld.w sdaoff(_a_variable)[gp],r6`'

  loads the contents of the location pointed to by the label '_a_variable' into register 6, provided that the label is located somewhere within +/- 32K of the address held in the GP register. [Note the linker assumes that the GP register contains a fixed address set to the address of the label called '__gp'. This can either be set up automatically by the linker, or specifically set by using the '`--defsym __gp=<value>`' command line option].

`tdaoff()` Computes the offset of the named variable from the start of the Tiny Data Area (whoes address is held in register 30, the EP register) and stores the result as a 7 or 8 bit unsigned value in the immediate operand field of the given instruction. For example:

‘`sld.w tdaoff(_a_variable)[ep],r6`’

loads the contents of the location pointed to by the label '_a_variable' into register 6, provided that the label is located somewhere within +256 bytes of the address held in the EP register. [Note the linker assumes that the EP register contains a fixed address set to the address of the label called '__ep'. This can either be set up automatically by the linker, or specifically set by using the ‘`--defsym __ep=<value>`’ command line option].

`zdaoff()` Computes the offset of the named variable from address 0 and stores the result as a 16 bit signed value in the immediate operand field of the given instruction. For example:

‘`movea zdaoff(_a_variable),zero,r6`’

puts the address of the label '_a_variable' into register 6, assuming that the label is somewhere within the first 32K of memory. (Strictly speaking it also possible to access the last 32K of memory as well, as the offsets are signed).

For information on the V850 instruction set, see *V850 Family 32-/16-Bit single-Chip Microcontroller Architecture Manual* from NEC. Ltd.

9 Reporting Bugs

Your bug reports play an essential role in making `as` reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. But in any case the principal function of a bug report is to help the entire community by making the next version of `as` work better. Bug reports are your contribution to the maintenance of `as`.

In order for a bug report to serve its purpose, you must include the information that enables us to fix the bug.

9.1 Have you found a bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the assembler gets a fatal signal, for any input whatever, that is a `as` bug. Reliable assemblers never crash.
- If `as` produces an error message for valid input, that is a bug.
- If `as` does not produce an error message for invalid input, that is a bug. However, you should note that your idea of "invalid input" might be our idea of "an extension" or "support for traditional practice".
- If you are an experienced user of assemblers, your suggestions for improvement of `as` are welcome in any case.

9.2 How to report bugs

A number of companies and individuals offer support for GNU products. If you obtained `as` from a support organization, we recommend you contact that organization first.

You can find contact information for many support companies and individuals in the file '`etc/SERVICE`' in the GNU Emacs distribution.

In any event, we also recommend that you send bug reports for `as` to '`bug-gnu-utils@gnu.org`'.

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and assume that some details do not matter. Thus, you might assume that the name of a symbol you use in an example does not matter. Well, probably it does not, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the assembler into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable us to fix the bug if it is new to us. Therefore, always write your bug reports on the assumption that the bug has not been reported previously.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" Those bug reports are useless, and we urge everyone to *refuse to respond to them* except to chide the sender to report bugs properly.

To enable us to fix the bug, you should include all these things:

- The version of `as`. `as` announces it if you start it with the '`--version`' argument.

  Without this, we will not know whether there is any point in looking for the bug in the current version of `as`.
- Any patches you may have applied to the `as` source.
- The type of machine you are using, and the operating system name and version number.
- What compiler (and its version) was used to compile `as`—e.g. "`gcc-2.7`".
- The command arguments you gave the assembler to assemble your example and observe the bug. To guarantee you will not omit something important, list them all. A copy of the Makefile (or the output from make) is sufficient.

  If we were to try to guess the arguments, we would probably guess wrong and then we might not encounter the bug.
- A complete input file that will reproduce the bug. If the bug is observed when the assembler is invoked via a compiler, send the assembler source, not the high level language source. Most compilers will produce the assembler source when run with the '`-S`' option. If you are using `gcc`, use the options '`-v --save-temps`'; this will save the assembler source in a file with an extension of '`.s`', and also show you exactly how `as` is being run.
- A description of what behavior you observe that you believe is incorrect. For example, "It gets a fatal signal."

  Of course, if the bug is that `as` gets a fatal signal, then we will certainly notice it. But if the bug is incorrect output, we might not notice unless it is glaringly wrong. You might as well not give us a chance to make a mistake.

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of `as` is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and ours would not. If you told us to expect a crash, then when ours fails to crash, we would know that the bug was not happening for us. If you had not told us to expect a crash, then we would not be able to draw any conclusion from our observations.
- If you wish to suggest changes to the `as` source, send us context diffs, as generated by `diff` with the '`-u`', '`-c`', or '`-p`' option. Always send diffs from the old file to the new file. If you even discuss something in the `as` source, refer to it by context, not by line number.

  The line numbers in our development sources will not match those in your sources. Your line numbers would convey no useful information to us.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. We recommend that you save your time for something else.

Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience for us. Errors in the output will be easier to spot, running under the debugger will take less time, and so on.

However, simplification is not vital; if you do not want to do this, report the bug anyway and send us the entire test case you used.

- A patch for the bug.

  A patch for the bug does help us if it is a good one. But do not omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

  Sometimes with a program as complicated as `as` it is very hard to construct an example that will make the program follow a certain path through the code. If you do not send us the example, we will not be able to construct one, so we will not be able to verify that the bug is fixed.

  And if we cannot understand what bug you are trying to fix, or why your patch should be an improvement, we will not install it. A test case will help us to understand.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even we cannot guess right about such things without first using the debugger to find the facts.

10 Acknowledgements

If you have contributed to `as` and your name isn't listed here, it is not meant as a slight. We just don't know about it. Send mail to the maintainer, and we'll correct the situation. Currently the maintainer is Ken Raeburn (email address `raeburn@cygnus.com`).

Dean Elsner wrote the original GNU assembler for the VAX.[1]

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

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

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

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

The Intel 80386 machine description was written by Eliot Dresselhaus.

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

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

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

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

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

Ian Lance Taylor of Cygnus Support merged the Motorola and MIT syntax for the 68k, completed support for some COFF targets (68k, i386 SVR3, and SCO Unix), added support

[1] Any more details?

for MIPS ECOFF and ELF targets, wrote the initial RS/6000 and PowerPC assembler, and made a few other minor patches.

Steve Chamberlain made `as` able to generate listings.

Hewlett-Packard contributed support for the HP9000/300.

Jeff Law wrote GAS and BFD support for the native HPPA object format (SOM) along with a fairly extensive HPPA testsuite (for both SOM and ELF object formats). This work was supported by both the Center for Software Science at the University of Utah and Cygnus Support.

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

Richard Henderson rewrote the Alpha assembler. Klaus Kaempf wrote GAS and BFD support for openVMS/Alpha.

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

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

Index

#

:

@

\

1

2

A

B

C

D

T

Table of Contents