Using sde-as

The GNU Assembler
for the MIPS family

Version 2.9-mipssde-030910

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 `sde-as`
Edited by Cygnus Support

1 Overview

This manual is a user guide to the GNU assembler `sde-as`. This version of the manual describes `sde-as` configured to generate code for MIPS architectures.

Here is a brief summary of how to invoke `sde-as`. For details, see Chapter 2 [Comand-Line Options], page 9.

```
sde-as [ -a[cdhlns][=file] ] [ -D ]  [ --defsym sym=val ]
 [ -f ] [ --gstabs ] [ --gdwarf2 ] [ --help ] [ -I dir ] [ -J ] [ -K ] [ -L ]
 [ --keep-locals ] [ -o objfile ] [ -R ] [ --statistics ] [ -v ]
 [ -version ] [ --version ] [ -W ] [ --warn ] [ --fatal-warnings ]
 [ -w ] [ -X ] [ -x ] [ -Z ]
 [ -EL ] [ -EB ] [ -G num ] [ -O[num] ]
 [ -mcpu=cpu ] [ -mabi=abi ]
 [ -mips1 ] [ -mips2 ] [ -mips3 ] [ -mips4 ] [ -mips5 ]
 [ -mips32 ] [ -mips32r2 ] [ -mips64 ] [ -mips64r2 ]
 [ -mips16 ] [ -mips16e ]
 [ -msmartmips ] [ -mips3D ]
 [ -mgp32 ] [ -mgp64 ] [ -mfp32 ] [ -mfp64 ]
 [ -mhard-float ]  [ -msingle-float ]
 [ -msoft-float ]  [ -mno-float ]
 [ -mno-fix-cw4010 ] [ -mno-fix-vr4300 ] [ -mno-fix-r4000 ]
 [ -mdiv-checks ] [ -mno-div-checks ]
 [ -membedded-data ] [ -mno-gpconst ]
 [ --trap ] [ --no-break ] [ --break ] [ --no-trap ]
 [ -KPIC ] [ -call_shared ] [ -non_shared ] [ -xgot ]
 [ -membedded-pic ]
 [ -- | 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.
`--gdwarf2`	Generate DWARF2 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`	This option is accepted but has no effect on the MIPS family.
`-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 `sde-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` `--no-warn`	Suppress warning messages.
`-X` `--fatal-warnings`	Treat warnings as errors.
`--warn`	Don’t suppress warning messages or treat them as errors.
`-w`	Ignored.
`-x`	Ignored.
`-X`	Treat warnings as errors.
`-Z`	Generate an object file even after errors.

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

The following options are available when sde-as is configured for a MIPS processor. For a full description of these arguments, see Section 8.74 [Assembler options], page 49.

`-G num` This option sets the largest size of an object that can be referenced implicitly with the `gp` register. Set to zero to disable.

`-O`
`-Onum` Selects the assembler optimization level.

`-EB`
`-EL` Use '`-EB`' to select big-endian output, and '`-EL`' for little-endian.

`-mcpu=cpu`
Generate code for a particular MIPS CPU.

`-mips1, -mips2, -mips3, -mips4, -mips5`
`-mips32, -mips32r2, -mips64, -mips64r2`
Generate code for a particular MIPS ISA (Instruction Set Architecture) level.

`-mips16` Enables the MIPS16 compressed instruction set extension. This will not generate MIPS16 code automatically, you must include '`.set mips16`' and '`.set nomips16`' directives around sections of assembler code which are written for MIPS16. Most users will never, and should never, write MIPS16 assembler code. MIPS16 is meant as an intermediate code generated by the compiler to reduce code size – possibly at the cost of some speed. MIPS16 CPUs always run the normal 32-bit MIPS instruction set as well, which is usually a better choice for assembler modules.

`-mips16e` Enables the enhanced MIPS16e compressed instruction set extension. See '`-mips16`' above for usage.

`-msmartmips`
Enables the SmartMIPS extension to the MIPS32 instruction set.

`-mips3D` Enables the MIPS-3D extension to the MIPS64 instruction set.

`-mabi=32|o64|n32|64|eabi|meabi`
Generate code for the indicated ABI.

`-mgp32` Assume that the 32 general purpose registers are 32 bits wide.

`-mgp64` Assume that the 32 general purpose registers are 64 bits wide.

`-mfp32` Assume that 32 32-bit floating point registers are available (equivalent to 16 64-bit registers, when used in pairs).

`-mfp64` Assume that 32 64-bit floating point registers are available.

`-mhard-float`
Enable use of the floating-point coprocessor instructions. This is the default.

`-msingle-float`
Enable use of the floating-point coprocessor instructions, but only for single-precision arithmetic.

`-msoft-float`
`-mno-float`

Generate an error message if any floating-point instructions are used.

`-mno-div-checks`
`-mdiv-checks`

Disable (or enable) the automatic generation of code to check for division by zero, or divide overflow.

`--trap`
`--no-break`

Generate code which takes a trap exception rather than a break exception when multiply or divide overflow error is detected.

`--break`
`--no-trap`

Generate code to take a break exception rather than a trap exception when an divide or multiply overflow is detected. This is the default.

`-KPIC`
`-call_shared`

Enable the generation of MIPS/abi position-independent code.

`-xgot` Generate assume a “large” global offset table references for MIPS/abi code.

`-non_shared`

Disable position-independent code. This is the default.

`-membedded-pic`

Generate PIC code suitable for some embedded systems. Not supported on MIPS SDE.

`-membedded-data`
`-mno-gpconst`

Place floating-point immediates in read-only data section.

`-mno-fix-cw4010`

Disables assembler workaround for early versions of the LSI CW4010 CPU.

`-mno-fix-vr4300`

Disables assembler workaround for early versions of the Vr4300 CPU.

`-mno-fix-r4000`

Disables assembler workaround for early versions of the R4000.

1.1 Structure of this Manual

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

We also cover special features in the MIPS configuration of `sde-as`, including assembler directives.

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.

1.2 The GNU Assembler

GNU `as` is really a family of assemblers. This manual describes `sde-as`, a member of that family which is configured for the MIPS architectures. 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.

`sde-as` is primarily intended to assemble the output of the GNU C compiler `sde-gcc` for use by the linker `sde-ld`. Nevertheless, we've tried to make `sde-as` assemble correctly everything that other assemblers for the same machine would assemble.

Unlike older assemblers, `sde-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 41).

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 26. On the MIPS, `sde-as` is configured to produce ELF format object files.

1.4 Command Line

After the program name `sde-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 `sde-as` to assemble.

Except for '`--`' any command line argument that begins with a hyphen ('`-`') is an option. Each option changes the behavior of `sde-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:

```
sde-as -o my-object-file.o mumble.s
sde-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 `sde-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 `sde-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 `sde-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 `sde-as` no file names it attempts to read one input file from the `sde-as` standard input, which is normally your terminal. You may have to type ⟨ctl-D⟩ to tell `sde-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, `sde-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 `sde-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 `sde-as` source is itself synthesized from other files. `sde-as` understands the '`#`' directives emitted by the `sde-gcc` preprocessor. See also Section 7.22 [`.file`], page 35.

1.6 Output (Object) File

Every time you run `sde-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`. 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 `sde-ld`. It contains assembled program code, information to help `sde-ld` integrate the assembled program into a runnable file, and (optionally) symbolic information for the debugger.

1.7 Error and Warning Messages

`sde-as` may write warnings and error messages to the standard error file (usually your terminal). This should not happen when a compiler runs `sde-as` automatically. Warnings report an assumption made so that `sde-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 7.22 [`.file`], page 35) 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 38) 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 49, for options specific to the MIPS.

If you are invoking `sde-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
```

This passes two options to the assembler: '`-alh`' (emit a listing to standard output with with high-level and assembly source) and '`-L`' (retain local symbols in the symbol table).

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 `sde-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), `sde-as` does not work correctly.

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

Use this option to add a *path* to the list of directories `sde-as` searches for files specified in `.include` directives (see Section 7.30 [`.include`], page 37). 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, `sde-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`

On the MIPS family, this option is allowed, but has no effect. It is permitted for compatibility with the GNU assembler on other platforms, where it can be used to warn when the assembler alters the machine code generated for '`.word`' directives in difference tables. The MIPS family does not have the addressing limitations that sometimes lead to this alteration on other platforms.

2.6 Include Local Labels: `-L`

Labels beginning with '`L`' (upper case only) are called *local labels*. See Section 5.3 [Symbol Names], page 25. 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 `sde-as` and `sde-ld` discard such labels, so you do not normally debug with them.

This option tells `sde-as` to retain those '`L...`' symbols in the object file. Usually if you do this you also tell the linker `sde-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.

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 `sde-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 `sde-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. `sde-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 `sde-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 `sde-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. `sde-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`

`sde-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 `sde-as`. By default it has the name '`a.out`'. You use this option (which takes exactly one filename) to give the object file a different name.

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

2.10 Join Data and Text Sections: `-R`

`-R` tells `sde-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 `sde-as`. In future, `-R` may work this way.

When `sde-as` is generating an object format which supports named sections, this option is only useful if you use sections named '`.text`' and '`.data`'.

2.11 Display Assembly Statistics: `--statistics`

Use '`--statistics`' to display two statistics about the resources used by `sde-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 `sde-as` is different in some ways from the output of some existing assembler. This switch requests `sde-as` to use the traditional format instead.

For example, it disables the exception frame optimizations which `sde-as` normally does by default on `sde-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 Control Warnings: `-W`, `--warn`, `--no-warn`

`sde-as` should never give a warning or error message when assembling compiler output. But programs written by people often cause `sde-as` to give a warning that a particular assumption was made. All such warnings are directed to the standard error file.

If you use the '`-W`' and '`--no-warn`' options, no warnings are issued. This only affects the warning messages: it does not change any particular of how `sde-as` assembles your file. Errors, which stop the assembly, are still reported.

You can switch these options off again by specifying '`--warn`', which causes warnings to be output as usual.

2.15 Make Warnings Fatal: `-X`, `--fatal-warnings`

If you use the '`-X`' or '`--fatal-warnings`' option, `sde-as` considers files that generate warnings to be in error.

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

After an error message, `sde-as` normally produces no output. If for some reason you are interested in object file output even after `sde-as` gives an error message on your program, use the '`-Z`' option. If there are any errors, `sde-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. `sde-as` syntax is similar to what many other assemblers use; it is inspired by the BSD 4.2 assembler.

3.1 Preprocessing

The `sde-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 37). 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 `sde-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 see Chapter 8 [Machine Dependencies], page 49.

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 29): 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 `sde-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 '`_.$`'. 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 25.

3.5 Statements

A *statement* ends at a newline character ('`\n`') or at a semicolon ('`;`'). The newline or semicolon is considered part of the preceding statement. Newlines and semicolons 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.

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.

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 25.

```
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 `sde-as` to interpret the second character literally as a backslash (which prevents `sde-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 `sde-as` has no other interpretation, so `sde-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 (or semicolon '`;`') 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. `sde-as` assumes your character code is ASCII: *`'A`* means 65, *`'B`* means 66, and so on.

3.6.2 Number Constants

`sde-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 30).

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 `sde-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 `sde-as` specialized to that computer.

A flonum is written by writing (in order)

- The digit '`0`'.
- A letter, to tell `sde-as` the rest of the number is a flonum.
- 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.

`sde-as` does all processing using integers. Flonums are computed independently of any floating point hardware in the computer running `sde-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 `sde-ld` reads many object files (partial programs) and combines their contents to form a runnable program. When `sde-as` emits an object file, the partial program is assumed to start at address 0. `sde-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 `sde-as` uses sections.

`sde-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.

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

`sde-as` can also generate whatever other named sections you specify using the ‘`.section`’ directive (see Section 7.56 [`.section`], page 44). If you do not use any directives that place output in the ‘`.text`’ or ‘`.data`’ sections, these sections still exist, but are empty.

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

To let `sde-ld` know which data changes when the sections are relocated, and how to change that data, `sde-as` also writes to the object file details of the relocation needed. To perform relocation `sde-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 `sde-as` ever uses is expressed as

(*section*) + (*offset into section*)

Further, most expressions `sde-as` computes have this section-relative nature.

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 `sde-ld` mixes partial programs, addresses in the absolute section remain unchanged. For example, address `{absolute 0}` is “relocated” to run-time address 0 by `sde-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. `sde-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 `sde-ld`; others are invented for use of `sde-as` and have no meaning except during assembly.

4.2 Linker Sections

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

named sections

These sections hold your program. `sde-as` and `sde-ld` treat them as separate but equal sections. Anything you can say of one section is true another.

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.

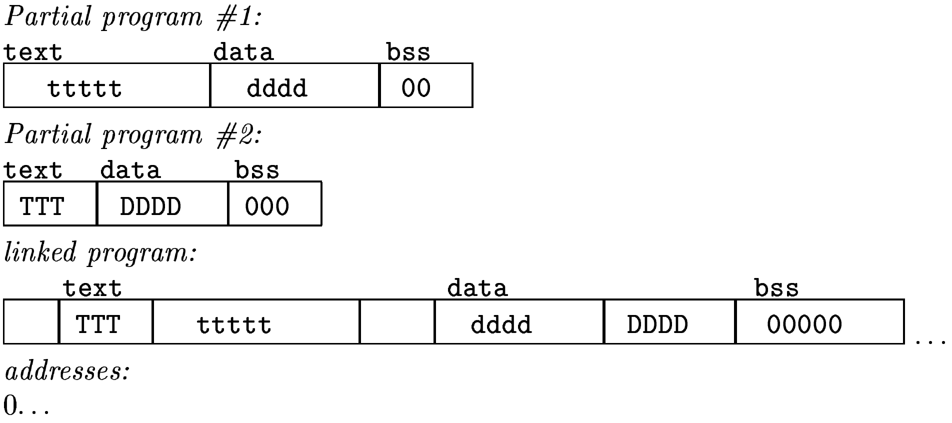
absolute section

Address 0 of this section is always "relocated" to runtime address 0. This is useful if you want to refer to an address that `sde-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.



4.3 Assembler Internal Sections

These sections are meant only for the internal use of `sde-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 `sde-as` warning messages, so it might be helpful to have an idea of their meanings to `sde-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

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. `sde-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.

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; `sde-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. 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 29.) 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 `sde-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 38.

The `.comm` pseudo-op may be used to declare a common symbol, which is another form of uninitialized symbol; see See Section 7.7 [`.comm`], page 32.

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 7.56 [`.section`], page 44. You may only assemble zero values into the section. Typically the section will only contain symbol definitions and `.skip` directives (see Section 7.62 [`.skip`], page 45).

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:* `sde-as` does not place symbols in the object file in the same order they were declared. This may break some debuggers.

5.1 Labels

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

5.2 Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign '`=`', followed by an expression (see Chapter 6 [Expressions], page 29). This is equivalent to using the `.set` directive. See Section 7.57 [`.set`], page 44.

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 49. That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted in Chapter 8 [Machine Dependencies], page 49), and underscores.

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. You can define and use as many local symbol names as you require, and they can be re-used throughout the program. You may refer to them using a positive decimal number. To define a local symbol, write a label of the form '`N:`' (where `N` represents any positive number, or zero). To refer to the most recent previous definition of that symbol write '`Nb`', using the same value of `N` as when you defined the label. To refer to the next definition of a local label, write '`Nf`'. The '`b`' stands for "backwards" and the '`f`' stands for "forwards".

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

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`', or in the case of ELF format '`.L`'. Normally both `sde-as` and `sde-ld` don't generate symbol table entries for local labels. These labels are used for symbols you are never intended to see. If you use the '`-L`' option then `sde-as` retains these symbols in the object file. If you also instruct `sde-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.

`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 `sde-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, `sde-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 `sde-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 `sde-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.

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 `sde-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. `sde-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 `sde-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 `sde-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

`sde-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.

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 `sde-as` to quit also. One day `.abort` will not be supported.

7.2 `.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, 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, mips, and the arm and strongarm, 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).

7.3 `.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.

7.4 `.asciz "string"...`

`.asciz` is just like `.ascii`, but each string is followed by a zero byte. The “z” in ‘`.asciz`’ stands for “zero”.

7.5 `.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.

7.6 `.byte expressions`

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

7.7 `.comm symbol , length [, align]`

`.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 `sde-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 `sde-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 `sde-ld` allocates uninitialized memory for the common symbol, it will use the alignment when placing the symbol. If no alignment is specified, `sde-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.

7.8 `.data` *`subsection`*

`.data` tells `sde-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.

7.9 `.double` *`flonums`*

`.double` expects zero or more flonums, separated by commas. It assembles floating point numbers. On the MIPS family '`.double`' emits 64-bit floating-point numbers in IEEE format.

7.10 `.eject`

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

7.11 `.else`

`.else` is part of the `sde-as` support for conditional assembly; see Section 7.29 [`.if`], page 36. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

7.12 `.elseif`

`.elseif` is part of the `sde-as` support for conditional assembly; see Section 7.29 [`.if`], page 36. It is shorthand for beginning a new `.if` block that would otherwise fill the entire `.else` section.

7.13 `.end`

`.end` marks the end of the assembly file. `sde-as` does not process anything in the file past the `.end` directive.

7.14 `.endfunc`

`.endfunc` marks the end of a function specified with `.func`.

7.15 `.endif`

`.endif` is part of the `sde-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 36.

7.16 `.equ` *symbol, expression*

This directive sets the value of *symbol* to *expression*. It is synonymous with '`.set`'; see Section 7.57 [`.set`], page 44.

7.17 `.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
```

7.18 `.err`

If `sde-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.

7.19 `.exitm`

Exit early from the current macro definition. See Section 7.42 [Macro], page 40.

7.20 `.extern`

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

7.21 `.fail` *expression*

Generates an error or a warning. If the value of the *expression* is 500 or more, `sde-as` will print a warning message. If the value is less than 500, `sde-as` will print an error message. The message will include the value of *expression*. This can occasionally be useful inside complex nested macros or conditional assembly.

7.22 .file *string*

`.file` tells `sde-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 `sde-as` programs.

7.23 .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 `sde-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.

7.24 .float *flonums*

This directive assembles zero or more flonums, separated by commas. It has the same effect as `.single`. On the MIPS family, `.float` emits 32-bit floating point numbers in IEEE format.

7.25 .func *name*[,*label*]

`.func` emits debugging information to denote function *name*, and is ignored unless the file is assembled with debugging enabled. Only '`--gstabs`' is currently supported. *label* is the entry point of the function and if omitted *name* prepended with the '`leading char`' is used. '`leading char`' is usually `_` or nothing, depending on the target. All functions are currently defined to have `void` return type. The function must be terminated with `.endfunc`.

7.26 .global *symbol*, .globl *symbol*

`.global` makes the symbol visible to `sde-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.

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`'.

7.28 `.ident`

This directive is used by some assemblers to place tags in object files. `sde-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 7.15 [`.endif`], page 34); optionally, you may include code for the alternative condition, flagged by `.else` (see Section 7.11 [`.else`], page 33). If you have several conditions to check, `.elseif` may be used to avoid nesting blocks if/else within each subsequent `.else` block.

The following variants of `.if` are also supported:

`.ifdef` *`symbol`*
: Assembles the following section of code if the specified *symbol* has been defined.

`.ifc` *`string1,string2`*
: Assembles the following section of code if the two strings are the same. The strings may be optionally quoted with single quotes. If they are not quoted, the first string stops at the first comma, and the second string stops at the end of the line. Strings which contain whitespace should be quoted. The string comparison is case sensitive.

`.ifeq` *`absolute expression`*
: Assembles the following section of code if the argument is zero.

`.ifeqs` *`string1,string2`*
: Another form of `.ifc`. The strings must be quoted using double quotes.

`.ifge` *`absolute expression`*
: Assembles the following section of code if the argument is greater than or equal to zero.

`.ifgt` *`absolute expression`*
: Assembles the following section of code if the argument is greater than zero.

`.ifle` *`absolute expression`*
: Assembles the following section of code if the argument is less than or equal to zero.

`.iflt` *`absolute expression`*
: Assembles the following section of code if the argument is less than zero.

`.ifnc` *`string1,string2`*.
: Like `.ifc`, but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.

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

`.ifne` *`absolute expression`*
: Assembles the following section of code if the argument is not equal to zero (in other words, this is equivalent to `.if`).

`.ifnes` *`string1,string2`*
: Like `.ifeqs`, but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.

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`* `[,` *`align`*`]`

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 35), so is normally not visible to `sde-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, 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 no alignment is specified, `sde-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.

7.35 `.lflags`

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

7.36 `.line` *`line-number`*

Even though this is a directive associated with the `a.out` or `b.out` object-code formats, `sde-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 `sde-as` to enter MRI mode. If *val* is zero, this tells `sde-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 37.

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.
`\@`	`sde-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, `sde-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 `sde-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 `.popsection`

The `.popsection` directive can be used on ELF targets to redirect assembler output to the section which was saved on the stack by the matching `.pushsection` directive.

7.48 `.previous`

The `.previous` directive can be used on ELF targets. It redirects assembler output back to the section which was selected before the last `.section` or `.struct` directive. It implements a one-deep stack only, which will be overwritten by the next section change directive. This directive is portable to other ELF assemblers, but see `.pushsection` below for a more powerful GNU-specific alternative.

7.49 `.pushsection` *name*

The `.pushsection` directive can be used on ELF targets to switch to a new section, after first pushing the current section onto a stack; this is useful if you want to be able to change sections safely inside a macro. It has exactly the same syntax as `.section` directive (see Section 7.56 [`.section`], page 44), and you return to the previous section using the matching `.popsection` directive. This directive is not portable to other ELF assemblers.

7.50 `.print` *string*

`sde-as` will print *string* on the standard output during assembly. You must put *string* in double quotes.

7.51 `.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.

`sde-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`.

7.52 `.purgem` *name*

Undefine the macro *name*, so that later uses of the string will not be expanded. See Section 7.42 [Macro], page 40.

7.53 `.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.

7.54 `.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
```

7.55 `.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.

7.56 `.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 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

7.57 `.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 26).

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.

7.58 `.short` *expressions*

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

7.59 `.single` *flonums*

This directive assembles zero or more flonums, separated by commas. It has the same effect as `.float`. On the MIPS family, `.single` emits 32-bit floating point numbers in IEEE format.

7.60 `.size`

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

7.61 `.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 7.70 [Uleb128], page 47.

7.62 `.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`'.

7.63 `.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`'.

7.64 `.stabd, .stabn, .stabs`

There are three directives that begin '`.stab`'. All emit symbols (see Chapter 5 [Symbols], page 25), for use by symbolic debuggers. The symbols are not entered in the `sde-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 `sde-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.

7.65 `.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.

7.66 `.struct` *expression*

Switch to the absolute section, and set the section offset to *expression*, which must be an absolute expression. You might use this as follows:

```
        .struct 0
field1:
        .space 4
field2:
        .space 4
field3:
```

This would define the symbol `field1` to have the value 0, the symbol `field2` to have the value 4, and the symbol `field3` to have the value 8. Assembly would be left in the absolute section, and you would need to use a `.section` directive of some sort to change to some other section before further assembly. For example with ELF targets you could use the `.previous` directive to return to the previous section.

7.67 `.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.

7.68 `.text` *subsection*

Tells `sde-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.

7.69 `.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.

7.70 `.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 7.61 [Sleb128], page 45.

7.71 `.internal, .hidden, .protected`

These directives can be used to set the visibility of a specified symbol. By default a symbol's visibility is set by its binding (local, global or weak), but these directives can be used to override that.

A visibility of `protected` means that any references to the symbol from within the component that defines the symbol must be resolved to the definition in that component, even if a definition in another component would normally preempt this.

A visibility of `hidden` means that the symbol is not visible to other components. Such a symbol is always considered to be protected as well.

A visibility of `internal` is the same as a visibility of `hidden`, except that some extra, processor specific processing must also be performed upon the symbol.

For ELF targets, the directives are used like this:

```
.internal name
.hidden name
.protected name
```

7.72 `.word` *expressions*

This directive expects zero or more *expressions*, of any section, separated by commas. For each expression, `sde-as` emits a 32-bit number.

7.73 Deprecated Directives

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

`.abort`

`.line`

8 MIPS Dependent Features

GNU `sde-as` for MIPS architectures supports several different MIPS processors, and a range of MIPS ISA levels. For information about the MIPS instruction set, and an overview of MIPS assembly conventions, view the list of publications in the development tools section of MIPS Technologies' website (`http://www.mips.com/devTools/bookstore.html`).

8.74 Assembler options

The MIPS configurations of GNU `sde-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. Set to zero to disable this optimization. The default value is 8, unless generating position-independent code, in which case it is 0, because the `gp` register is then used for other purposes.

`-O`
`-O`*`num`*

Selects the assembler optimization level. By default the assembler will try to avoid inserting `nop` instructions, and will fill a branch delay slot with the instruction immediately before the branch, if that is safe to do. You can disable the branch delay slot filling only by specifying '`-O0`'.

`-EB`
`-EL`

Use '`-EB`' to select big-endian output, and '`-EL`' for little-endian.

`-mcpu=`*`cpu`*

Generate code for a particular MIPS CPU. This option enables use of any extra instructions specific to each CPU, and enables or disables insertion of `nop` instructions as required for different CPUs.

The list of recognised CPU is as follows, grouped with aliases on the same line:

```
r2000, r2k,
r3000, r3k,
r6000, r6k,
r4000, r4400, r4200,
r4310, r4300,
r4111, r4100,
orion, r4600, r4700,
r4640, r4650,
r8000, r8k,
rc32364, rc3236x, cronus,
r5000, r5k,
r5400, r5432, r5464, r54xx,
r5500, r55xx
rc64574, rc64575, rc6457x,
r10000, r12000, r10k, r12k,
4kc, 4kp, 4km, 4kec, 4kep, 4kem, 4ksc, 4ksd. m4k,
5kc, 5kf,
24kc, 24kf,
20kc, 25kf,
rm5230, rm5231, rm5260, rm5261,
rm5270, rm5271, rm52xx,
rm7000, rm7k,
r1900, pr1900, r3900, pr3900 tx39,
r4900, tx49,
lr33000, lr33k,
4001, tr4101, tr4102, tr410x,
cw4001, cw4002, cw400x
4010, cw4010, cw401x
atm2, atmizer2, apu,
4020, cw4020
```

`-mips1`
`-mips2`
`-mips3`
`-mips4`
`-mips5`
`-mips32`
`-mips32r2`
`-mips64`
`-mips64r2`

Generate code for a particular MIPS Instruction Set Architecture level. You can also switch instruction sets during the assembly; see Section 8.77 [Directives to override the ISA level], page 53.

`-mips16` MIPS16 is an instruction set extension which provides a subset of true MIPS instructions, with a restricted set of registers, and coded as 16-bit instructions. It can make for much smaller program binaries. CPUs supporting MIPS16 switch from interpreting conventional MIPS to MIPS16 instructions when they are asked to fetch an instruction from an odd address.

This flag is actually ignored by the assembler, which always starts off using the conventional MIPS instruction set. It is difficult and usually pointless to try to write assembler code for MIPS16—if you really need to then you can mix MIPS16 and conventional MIPS code using explicit '`.set mips16`' and '`.set nomips16`' directives.

MIPS16 CPUs can be either 32-bit or 64-bit implementations. To compile for a 64-bit MIPS16 CPU you should first specify a 64-bit base ISA, e.g. '`-mips3`'.

`-mips16e` MIPS16e is an enhancement of the MIPS16 instruction set, which provides even greater code size reductions. It is only ever available when combined with a MIPS32 or MIPS64 compliant.

Like '`-mips16`', the '`-mips16e`' command line option does not cause MIPS16e code to be generated—that still needs a '`.set mips16`' directive in the code. But specifying '`-mips16e`' with no other ISA flag will cause MIPS32 to be selected as the 32-bit ISA.

To generate 64-bit MIPS16e code, first specify a 64-bit base ISA, e.g. '`-mips64`'.

`-msmartmips`

Enables the SmartMIPS extensions to the MIPS32 instruction set, which provides a number of new instructions which target smartcard and cryptographic applications.

`-mips3D` MIPS-3D is an extension to the MIPS64 instruction set, which provides a paired single floating-point vector type, and a number of new floating-point instructions which target geometry processing.

`-mabi=32`
`-mabi=o64`
`-mabi=n32`
`-mabi=64`
`-mabi=eabi`
`-mabi=meabi`

Generate code for the indicated ABI. At the moment, the only effect of this option is that '`-mabi=n64`' will select 64-bit ELF object code format and address computations; the other ABI modes have no effect on the assembler.

`-mgp32` Assume that the 32 general purpose registers are 32 bits wide. This is the default when a 32-bit ISA is specified or implied. Such code will run correctly on 64-bit MIPS CPUs so long as every function which might call your code is built the same way. This option affects how some macro instructions are expanded, but you may also want to generate 32-bit code while other enhanced features of the 64-bit ISAs. Also, some 32-bit OSes only save the 32-bit registers on a context switch, so it is essential never to use the 64-bit registers.

`-mgp64` Assume that the 32 general purpose registers are 64 bits wide. This is the default when a 64-bit ISA is specified or implied, and illegal unless your CPU implements a 64-bit instruction set; so it's hard to see when you'd use this option explicitly, but is provided in the interests of symmetry with '`-mgp32`'.

`-mfp32` Assume that 32 32-bit floating point registers are available, but only the even-numbered 16 are used for arithmetic (the odd-numbered registers are used quietly by the assembler for loading/storing the high-order bits of double-precision values). This is the default when a 32-bit ISA is specified or implied.

`-mfp64` Assume that 32 64-bit floating point registers are available. This is the default when a 64-bit ISA is specified or implied, and indeed it is usually usually illegal with 32-bit ISAs. The exception is that it can be used with '`-mips32r2`', since MIPS32 release 2 adds new instructions which allow a 32-bit CPU to be combined with a 64-bit FPU.

`-mhard-float`

Enable use of the floating-point coprocessor instructions. This is the default.

`-msingle-float`

Enable use of the floating-point coprocessor instructions, but only for single-precision arithmetic (as implemented on the r4640 and r4650). Any use of double-precision arithmetic will cause the assembler to generate an error message.

`-msoft-float`
`-mno-float`

These two options are equivalent, and will cause the assembler to generate an error message if any floating-point instructions are used.

`-mno-div-checks`
`-mdiv-checks`
Disable (or enable) the automatic generation of code to check for division by zero, or divide overflow.

`--trap`
`--no-break`
`sde-as` automatically macro expands certain division and multiplication instructions to check for overflow and division by zero. This option causes `sde-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 divide or multiply overflow is detected. This is the default.

`-KPIC`
`-call_shared`
These options both enable the generation of MIPS/abi position-independent code, as used on many modern Unix systems. This can also be enabled by using the '`.abicalls`' pseudo-op.

`-xgot` When generating MIPS/abi code, assume a "large" global offset table (more than 32K global symbols). This generates a longer sequence of instructions for each GOT reference.

`-non_shared`
Disable generation of position-independent code. This is the default.

`-membedded-pic`
Generate PIC code suitable for some embedded systems. All calls are made using PC relative address, and all data is addressed using the *gp* register. No more than 65536 bytes of global data may be used. This currently only works on targets which use ECOFF; it does not work with ELF. Since MIPS SDE uses ELF as its object format, this feature is not supported.

`-membedded-data`
`-mno-gpconst`
These equivalent options cause any floating-point immediate values to be placed in the read-only data section, rather than the `.lit4` or `.lit8` data sections.

`-mno-fix-cw4010`
Disables an assembler workaround for early versions of the LSI CW4010 CPU, which inserts a `nop` before any branch which is itself a branch target. This workaround is enabled automatically if either no CPU is selected and the ISA is MIPS II or lower, or if the r3000, cw4010 or atm2 CPU type is selected.

`-mno-fix-vr4300`
Disables an assembler workaround for early versions of the Vr4300 CPU, which inserts a `nop` between a floating-point multiply and an immediately following

integer or floating-point multiply. This workaround is enabled automatically if either no CPU is selected and the ISA is MIPS III or lower, or if the r3000, r4000 or Vr4300 CPU type is selected.

`-mno-fix-r4000`
Disables an assembler workaround for early versions of the R4000, which inserts a `nop` between a variable shift instruction and a multiply or divide. This workaround is enabled automatically if either no CPU is selected and the ISA is MIPS III or lower, or if the r3000 or r4000 CPU type is selected.

8.75 MIPS object code

Assembling for a MIPS ECOFF or ELF 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. In the case of ELF the read-only data section is called '`.rodata`'.

When assembling for ECOFF or ELF, the assembler uses the `$gp` (`$28`) register to form the address of "small data. Any symbol in the `.sdata` or `.sbss` sections is considered "small" in this sense. For external objects, or for objects in the `.bss` section, you can use the `sde-gcc` '`-G`' option to control the size of objects addressed via `$gp`; the default value is 8, meaning that a reference to any object eight bytes or smaller uses `$gp`. Passing '`-G 0`' to `sde-as` prevents it from using the `$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, while leaving `sym` otherwise undefined.

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

8.76 Directives for debugging information

MIPS ELF `sde-as` supports several directives used for generating debugging information which are not supported by traditional MIPS assemblers. These are `.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!

8.77 Directives to override the ISA level

GNU `sde-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 5, 32 or 64. A non-zero value 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. The '`.set mips0`' directive 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 64 bit instructions while assembling with a 32 bit ISA. Use this directive with care!

The '`.set gp64`' directive affects how immediate values and certain MIPS macro instructions are expanded, so that 64-bit values will be held in a single register. This is only valid if a 64 bit ISA has already been selected. The '`.set gp32`' directive has the opposite effect, and the '`.set nogp64`' or '`.set nogp32`' directives return to the level selected by the command line options.

The '`.set fp64`' directive tells the assembler to assume the existence of 32 64-bit floating-point registers. This is only valid if a 64 bit ISA has already been selected. The '`.set fp32`' directive has the opposite effect, and the '`.set nofp64`' or '`.set nofp32`' directives return to the level selected by the command line options.

The directive '`.set mips16`' puts the assembler into MIPS16 mode, in which it will generate the compressed instruction set. Use '`.set nomips16`' to return to normal 32 bit mode. The directive '`.set mips16e`' is similar, but enables the extended MIPS16e instruction set.

The '`.set smartmips`' directive enables use of the SmartMIPS extensions to the MIPS32 ISA; the '`.set nosmartmips`' directive reverses that.

Traditional MIPS assemblers do not support these directives.

8.78 Directives for extending MIPS16 instructions

By default, MIPS16 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 MIPS16 mode. Traditional MIPS assemblers do not support this directive.

8.79 Directive to mark data as an instruction

The `.insn` directive tells `sde-as` that the following data is actually instructions. This makes a difference in MIPS16 mode: when loading the address of a label which precedes instructions, `sde-as` automatically adds 1 to the value, so that jumping to the loaded address will do the right thing.

8.80 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.

9 Reporting Bugs

Your bug reports play an essential role in making `sde-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 `sde-as` work better. Bug reports are your contribution to the maintenance of `sde-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 `sde-as` bug. Reliable assemblers never crash.
- If `sde-as` produces an error message for valid input, that is a bug.
- If `sde-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 `sde-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 `sde-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 `sde-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 `sde-as`. `sde-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 `sde-as`.
- Any patches you may have applied to the `sde-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 `sde-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 `sde-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 `sde-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 `sde-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 `sde-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 `sde-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 `sde-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 `sde-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 `sde-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 for MIPS ECOFF and ELF targets, wrote the initial RS/6000 and PowerPC assembler, and made a few other minor patches.

---

[1] Any more details?

Steve Chamberlain made `sde-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).

Linas Vepstas added GAS support for the ESA/390 "IBM 370" architecture.

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

Timothy Wall, Michael Hayes, and Greg Smart contributed to the various tic* flavors. 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

Table of Contents

8 MIPS Dependent Features 49

9 Reporting Bugs 57

10 Acknowledgements 61

Index 63