

Using `as`

The GNU Assembler

Version 2.10.90

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

Dean Elsner, Jay Fenlason & friends

Using as
Edited by Cygnus Support

Copyright © 1991, 92, 93, 94, 95, 96, 97, 98, 99, 2000 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

1 Overview

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

Here is a brief summary of how to invoke `as`. For details, see Chapter 2 [Comand-Line Options], page 11.

```
as [ -a[cdhlms][=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 ] [ -Z ]
  [ -mbig-endian | -mlittle-endian ]
  [ -m[arm]1 | -m[arm]2 | -m[arm]250 | -m[arm]3 | -m[arm]6 | -m[arm]60 |
    -m[arm]600 | -m[arm]610 | -m[arm]620 | -m[arm]7[t][[d]m[i]][fe] | -m[arm]70 |
    -m[arm]700 | -m[arm]710[c] | -m[arm]7100 | -m[arm]7500 | -m[arm]8 |
    -m[arm]810 | -m[arm]9 | -m[arm]920 | -m[arm]920t | -m[arm]9tdmi |
    -mstrongarm | -mstrongarm110 | -mstrongarm1100 ]
  [ -m[arm]v2 | -m[arm]v2a | -m[arm]v3 | -m[arm]v3m | -m[arm]v4 | -m[arm]v4t |
    -m[arm]v5 | -[arm]v5t ]
  [ -mthumb | -mall ]
  [ -mfpa10 | -mfpa11 | -mfpe-old | -mno-fpu ]
  [ -EB | -EL ]
  [ -mapcs-32 | -mapcs-26 | -mapcs-float | -mapcs-reentrant ]
  [ -mthumb-interwork ]
  [ -moabi ]
  [ -k ]
  [ -O ]
  [ -O | -n | -N ]
  [ -mb | -me ]
  [ -Av6 | -Av7 | -Av8 | -Asparclet | -Asparclite
    -Av8plus | -Av8plusa | -Av9 | -Av9a ]
  [ -xarch=v8plus | -xarch=v8plusa ] [ -bump ] [ -32 | -64 ]
  [ -ACA | -ACA_A | -ACB | -ACC | -AKA | -AKB | -AKC | -AMC ]
  [ -b ] [ -no-relax ]
  [ --m32rx | --[no-]warn-explicit-parallel-conflicts | --W[n]p ]
  [ -l ] [ -m68000 | -m68010 | -m68020 | ... ]
  [ -jsri2bsr ] [ -sifilter ] [ -relax ]
  [ -mcpu=[210|340] ]
  [ -m68hc11 | -m68hc12 ]
  [ --force-long-branches ] [ --short-branches ] [ --strict-direct-mode ]
  [ --print-insn-syntax ] [ --print-opcodes ] [ --generate-example ]
  [ -nocpp ] [ -EL ] [ -EB ] [ -G num ] [ -mcpu=CPU ]
  [ -mips1 ] [ -mips2 ] [ -mips3 ] [ -m4650 ] [ -no-m4650 ]
  [ --trap ] [ --break ]
  [ --emulation=name ]
  [ -- | files ... ]
```

`-a[cdhlms]`

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 Issue warnings when difference tables altered for long displacements.

-L

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

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

-R Fold the data section into the text section.

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

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

```

-v
-version    Print the as version.

--version
            Print the as version and exit.

-W
--no-warn   Suppress warning messages.

--fatal-warnings
            Treat warnings as errors.

--warn      Don't suppress warning messages or treat them as errors.

-w          Ignored.

-x          Ignored.

-Z          Generate an object file even after errors.

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

```

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

```

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

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

```

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

```

-m[arm] [1|2|3|6|7|8|9] [...]
            Specify which ARM processor variant is the target.

-m[arm] v[2|2a|3|3m|4|4t|5|5t]
            Specify which ARM architecture variant is used by the target.

-mthumb | -marm
            Enable or disable Thumb only instruction decoding.

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

-mapcs-32 | -mapcs-26 | -mapcs-float | -mapcs-reentrant | -moabi
            Select which procedure calling convention is in use.

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

-mthumb-interwork
            Specify that the code has been generated with interworking between Thumb
            and ARM code in mind.

-k          Specify that PIC code has been generated.

```

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

-O Optimize output by parallelizing instructions.

The following options are available when as is configured for a D30V processor.

-O Optimize output by parallelizing instructions.

-n Warn when nops are generated.

-N Warn when a nop after a 32-bit multiply instruction is generated.

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

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

-b Add code to collect statistics about branches taken.

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

The following options are available when as is configured for the Mitsubishi M32R series.

--m32rx Specify which processor in the M32R family is the target. The default is normally the M32R, but this option changes it to the M32RX.

--warn-explicit-parallel-conflicts or **--Wp**
Produce warning messages when questionable parallel constructs are encountered.

--no-warn-explicit-parallel-conflicts or **--Wnp**
Do not produce warning messages when questionable parallel constructs are encountered.

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

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

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

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

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

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

-mb Generate "big endian" format output.

-ml Generate “little endian” format output.

The following options are available when `as` is configured for the Motorola 68HC11 or 68HC12 series.

-m68hc11 | -m68hc12

Specify what processor is the target. The default is defined by the configuration option when building the assembler.

--force-long-branches

Relative branches are turned into absolute ones. This concerns conditional branches, unconditional branches and branches to a sub routine.

-S | --short-branches

Do not turn relative branches into absolute ones when the offset is out of range.

--strict-direct-mode

Do not turn the direct addressing mode into extended addressing mode when the instruction does not support direct addressing mode.

--print-insn-syntax

Print the syntax of instruction in case of error.

--print-opcodes

print the list of instructions with syntax and then exit.

--generate-example

print an example of instruction for each possible instruction and then exit. This option is only useful for testing `as`.

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

-Av6 | -Av7 | -Av8 | -Asparclet | -Asparclite

-Av8plus | -Av8plusa | -Av9 | -Av9a

Explicitly select a variant of the SPARC architecture.

‘**-Av8plus**’ and ‘**-Av8plusa**’ select a 32 bit environment. ‘**-Av9**’ and ‘**-Av9a**’ select a 64 bit environment.

‘**-Av8plusa**’ and ‘**-Av9a**’ enable the SPARC V9 instruction set with Ultra-SPARC extensions.

-xarch=v8plus | -xarch=v8plusa

For compatibility with the Solaris v9 assembler. These options are equivalent to **-Av8plus** and **-Av8plusa**, respectively.

-bump Warn when the assembler switches to another architecture.

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

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

-EB Generate “big endian” format output.

-EL Generate “little endian” format output.

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

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

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

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

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

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

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

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

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

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

-jsri2bsr
-nojsri2bsr Enable or disable the JSRI to BSR transformation. By default this is enabled. The command line option ‘-nojsri2bsr’ can be used to disable it.

- `-sifilter`
- `-nosifilter`
 - Enable or disable the silicon filter behaviour. By default this is disabled. The default can be overridden by the ‘`-sifilter`’ command line option.
- `-relax`
 - Alter jump instructions for long displacements.
- `-mcpu=[210|340]`
 - Select the cpu type on the target hardware. This controls which instructions can be assembled.
- `-EB`
 - Assemble for a big endian target.
- `-EL`
 - Assemble for a little endian target.

1.1 Structure of this Manual

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

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

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

1.2 The GNU Assembler

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

`as` is primarily intended to assemble the output of the GNU C compiler `gcc` for use by the linker `ld`. Nevertheless, we’ve tried to make `as` assemble correctly everything that other assemblers for the same machine would assemble. Any exceptions are documented explicitly (see Chapter 8 [Machine Dependencies], page 55). This doesn’t mean `as` always uses the same syntax as another assembler for the same architecture; for example, we know of several incompatible versions of 680x0 assembly language syntax.

Unlike older assemblers, `as` is designed to assemble a source program in one pass of the source file. This has a subtle impact on the `.org` directive (see Section 7.50 [`.org`], page 46).

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 di-

rectives for debugging symbols are typically different in different file formats. See Section 5.5 [Symbol Attributes], page 30.

1.4 Command Line

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

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

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

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

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

1.5 Input Files

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

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

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

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

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

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

If the source is empty, `as` produces a small, empty object file.

Filenames and Line-numbers

There are two ways of locating a line in the input file (or files) and either may be used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a "logical" file. See Section 1.7 [Error and Warning Messages], page 9.

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

Logical files are simply names declared explicitly by assembler directives; they bear no relation to physical files. Logical file names help error messages reflect the original source file, when `as` source is itself synthesized from other files. `as` understands the `#` directives emitted by the `gcc` preprocessor. See also Section 7.27 [`.file`], page 39.

1.6 Output (Object) File

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

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

1.7 Error and Warning Messages

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

Warning messages have the format

```
file_name:NNN:Warning Message Text
```

(where `NNN` is a line number). If a logical file name has been given (see Section 7.27 [`.file`], page 39) 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 7.41 [`.line`], page 43) 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 55, for options specific to particular machine architectures.

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

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

This passes two options to the assembler: ‘`-alh`’ (emit a listing to standard output 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 `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 17.

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

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

Use this option to add a *path* to the list of directories `as` searches for files specified in `.include` directives (see Section 7.35 [`.include`], page 42). You may use `-I` as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, `as` searches any `-I` directories in the same order as they were specified (left to right) on the command line.

2.5 Difference Tables: `-K`

`as` sometimes alters the code emitted for directives of the form `.word sym1-sym2`; see Section 7.78 [`.word`], page 54. You can use the `-K` option if you want a warning issued when this is done.

2.6 Include Local Labels: `-L`

Labels beginning with `'L'` (upper case only) are called *local labels*. See Section 5.3 [Symbol Names], page 29. Normally you do not see such labels when debugging, because they are intended for the use of programs (like compilers) that compose assembler programs, not for your notice. Normally both `as` and `ld` discard such labels, so you do not normally debug with them.

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

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

2.7 Assemble in MRI Compatibility Mode: `-M`

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

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

- global symbols in common section

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

- complex relocations

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

- **END** pseudo-op specifying start address

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

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

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

- **ORG** pseudo-op

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

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

- EBCDIC strings

EBCDIC strings are not supported.

- packed binary coded decimal

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

- **FEQU** pseudo-op

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

- **NOOBJ** pseudo-op

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

- **OPT** branch control options

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

- **OPT** list control options

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

- other **OPT** options

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

- **OPT D** option is default

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

- **XREF** pseudo-op.

The m68k **XREF** pseudo-op is ignored.

- `.debug` pseudo-op
The i960 `.debug` pseudo-op is not supported.
- `.extended` pseudo-op
The i960 `.extended` pseudo-op is not supported.
- `.list` pseudo-op.
The various options of the i960 `.list` pseudo-op are not supported.
- `.optimize` pseudo-op
The i960 `.optimize` pseudo-op is not supported.
- `.output` pseudo-op
The i960 `.output` pseudo-op is not supported.
- `.setreal` pseudo-op
The i960 `.setreal` pseudo-op is not supported.

2.8 Dependency tracking: `--MD`

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

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

This feature is used in the automatic updating of makefiles.

2.9 Name the Object File: `-o`

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

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

2.10 Join Data and Text Sections: `-R`

`-R` tells `as` to write the object file as if all data-section data lives in the text section. This is only done at the very last moment: your binary data are the same, but data section parts are relocated differently. The data section part of your object file is zero bytes long because all its bytes are appended to the text section. (See Chapter 4 [Sections and Relocation], page 23.)

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

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

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

2.11 Display Assembly Statistics: `--statistics`

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

2.12 Compatible output: `--traditional-format`

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

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

2.13 Announce Version: `-v`

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

2.14 Control Warnings: `-W`, `--warn`, `--no-warn`, `--fatal-warnings`

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

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

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

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

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

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

3 Syntax

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

3.1 Preprocessing

The `as` internal preprocessor:

- adjusts and removes extra whitespace. It leaves one space or tab before the keywords on a line, and turns any other whitespace on the line into a single space.
- removes all comments, replacing them with a single space, or an appropriate number of newlines.
- converts character constants into the appropriate numeric values.

It does not do macro processing, include file handling, or anything else you may get from your C compiler's preprocessor. You can do include file processing with the `.include` directive (see Section 7.35 [`.include`], page 42). 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 19), any whitespace means the same as exactly one space.

3.3 Comments

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

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

```
/*
```

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

```

*/

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

```

Anything from the *line comment* character to the next newline is considered a comment and is ignored. The line comment character is ‘;’ for the AMD 29K family; ‘;’ on the ARC; ‘@’ on the ARM; ‘;’ for the H8/300 family; ‘!’ for the H8/500 family; ‘;’ for the HPPA; ‘#’ on the i960; ‘;’ for picoJava; ‘!’ for the Hitachi SH; ‘!’ on the SPARC; ‘#’ on the m32r; ‘|’ on the 680x0; ‘#’ on the 68HC11 and 68HC12; ‘#’ on the Vax; ‘!’ for the Z8000; ‘#’ on the V850; see Chapter 8 [Machine Dependencies], page 55.

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

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

```
‘--’;
```

To be compatible with past assemblers, lines that begin with ‘#’ have a special interpretation. Following the ‘#’ should be an absolute expression (see Chapter 6 [Expressions], page 33): the logical line number of the *next* line. Then a string (see Section 3.6.1.1 [Strings], page 19) is allowed: if present it is a new logical file name. The rest of the line, if any, should be whitespace.

If the first non-whitespace characters on the line are not numeric, the line is ignored. (Just like a comment.)

```

# This is an ordinary comment.
# 42-6 "new_file_name" # New logical file name
# This is logical line # 36.

```

This feature is deprecated, and may disappear from future versions of as.

3.4 Symbols

A *symbol* is one or more characters chosen from the set of all letters (both upper and lower case), digits and the three characters ‘_.\$’. On most machines, you can also use \$ in symbol names; exceptions are noted in Chapter 8 [Machine Dependencies], page 55. 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 29.

3.5 Statements

A *statement* ends at a newline character (‘\n’) or line separator character. (The line separator is usually ‘;’, unless this conflicts with the comment character; see Chapter 8 [Machine Dependencies], page 55.) The newline or separator character is considered part of the preceding statement. Newlines and separators within character constants are an exception: they do not end statements.

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

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

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

A label is a symbol immediately followed by a colon (:). Whitespace before a label or after a colon is permitted, but you may not have whitespace between a label's symbol and its colon. See Section 5.1 [Labels], page 29.

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

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

3.6 Constants

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

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

3.6.1 Character Constants

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

3.6.1.1 Strings

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

<code>\b</code>	Mnemonic for backspace; for ASCII this is octal code 010.
<code>\f</code>	Mnemonic for FormFeed; for ASCII this is octal code 014.
<code>\n</code>	Mnemonic for newline; for ASCII this is octal code 012.
<code>\r</code>	Mnemonic for carriage-Return; for ASCII this is octal code 015.
<code>\t</code>	Mnemonic for horizontal Tab; for ASCII this is octal code 011.
<code>\ digit digit digit</code>	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, <code>\008</code> has the value 010, and <code>\009</code> the value 011.
<code>\x hex-digits...</code>	A hex character code. All trailing hex digits are combined. Either upper or lower case <code>x</code> works.
<code>\\</code>	Represents one <code>'\'</code> character.
<code>\"</code>	Represents one <code>'\"'</code> character. Needed in strings to represent this character, because an unescaped <code>'\"'</code> would end the string.
<code>\ anything-else</code>	Any other character when escaped by <code>\</code> gives a warning, but assembles as if the <code>'\'</code> 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 <code>as</code> has no other interpretation, so <code>as</code> knows it is giving you the wrong code and warns you of the fact.

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

3.6.1.2 Characters

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

3.6.2 Number Constants

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

3.6.2.1 Integers

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

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

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

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

Integers have the usual values. To denote a negative integer, use the prefix operator ‘-’ discussed under expressions (see Section 6.2.3 [Prefix Operators], page 34).

3.6.2.2 Bignums

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

3.6.2.3 Flonums

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

A flonum is written by writing (in order)

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

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

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

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

On the HPPA architecture, the letter must be ‘E’ (upper case only).

- An optional sign: either ‘+’ or ‘-’.
- An optional *integer part*: zero or more decimal digits.
- An optional *fractional part*: ‘.’ followed by zero or more decimal digits.
- An optional exponent, consisting of:
 - An ‘E’ or ‘e’.
 - Optional sign: either ‘+’ or ‘-’.
 - One or more decimal digits.

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

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

4 Sections and Relocation

4.1 Background

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

The linker `ld` reads many object files (partial programs) and combines their contents to form a runnable program. When `as` emits an object file, the partial program is assumed to start at address 0. `ld` assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification, but it suffices to explain how `as` uses sections.

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

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

When it generates COFF output, `as` can also generate whatever other named sections you specify using the `‘.section’` directive (see Section 7.59 [`.section`], page 48). If you do not use any directives that place output in the `‘.text’` or `‘.data’` sections, these sections still exist, but are empty.

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

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

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

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

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

- Where in the object file is the beginning of this reference to an address?
- How long (in bytes) is this reference?
- Which section does the address refer to? What is the numeric value of

$(address) - (start\text{-}address\ of\ section)?$

- Is the reference to an address “Program-Counter relative”?

In fact, every address `as` ever uses is expressed as

$(section) + (offset\ into\ section)$

Further, most expressions `as` computes have this section-relative nature. (For some object formats, such as SOM for the HPPA, some expressions are symbol-relative instead.)

In this manual we use the notation `{secname N}` to mean “offset *N* into section *secname*.”

Apart from text, data and bss sections you need to know about the *absolute* section. When `ld` mixes partial programs, addresses in the absolute section remain unchanged. For example, address `{absolute 0}` is “relocated” to run-time address 0 by `ld`. Although the linker never arranges two partial programs’ data sections with overlapping addresses after linking, *by definition* their absolute sections must overlap. Address `{absolute 239}` in one part of a program is always the same address when the program is running as address `{absolute 239}` in any other part of the program.

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

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

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

4.2 Linker Sections

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

named sections

text section

data section

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

bss section

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

absolute section

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

undefined section

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

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

Partial program #1:

text	data	bss
ttttt	dddd	00

Partial program #2:

text	data	bss
TTT	DDDD	000

linked program:

text		data			bss		
	TTT	ttttt		dddd	DDDD	00000	...

addresses:

0...

4.3 Assembler Internal Sections

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

ASSEMBLER-INTERNAL-LOGIC-ERROR!

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

expr section

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

4.4 Sub-Sections

Assembled bytes conventionally fall into two sections: `text` and `data`. You may have separate groups of data in named sections that you want to end up near to each other in the object file, even though they are not contiguous in the assembler source. `as` allows you to use *subsections* for this purpose. Within each section, there can be numbered subsections

with values from 0 to 8192. Objects assembled into the same subsection go into the object file together with other objects in the same subsection. For example, a compiler might want to store constants in the text section, but might not want to have them interspersed with the program being assembled. In this case, the compiler could issue a `.text 0` before each section of code being output, and a `.text 1` before each group of constants being output.

Subsections are optional. If you do not use subsections, everything goes in subsection number zero.

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

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

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a `.text expression` or a `.data expression` statement. When generating COFF output, you can also use an extra subsection argument with arbitrary named sections: `.section name, expression`. *Expression* should be an absolute expression. (See Chapter 6 [Expressions], page 33.) If you just say `.text` then `.text 0` is assumed. Likewise `.data` means `.data 0`. Assembly begins in `text 0`. For instance:

```
.text 0      # The default subsection is text 0 anyway.
.ascii "This lives in the first text subsection. *"
.text 1
.ascii "But this lives in the second text subsection."
.data 0
.ascii "This lives in the data section,"
.ascii "in the first data subsection."
.text 0
.ascii "This lives in the first text section,"
.ascii "immediately following the asterisk (*)."
```

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

4.5 bss Section

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

The `.lcomm` pseudo-op defines a symbol in the `bss` section; see Section 7.39 [`.lcomm`], page 43.

The `.comm` pseudo-op may be used to declare a common symbol, which is another form of uninitialized symbol; see See Section 7.8 [`.comm`], page 37.

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 7.59 [`.section`], page 48. You may only assemble zero values into the section. Typically the section will only contain symbol definitions and `.skip` directives (see Section 7.65 [`.skip`], page 50).

5 Symbols

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

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

5.1 Labels

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

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

5.2 Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign ‘=’, followed by an expression (see Chapter 6 [Expressions], page 33). This is equivalent to using the `.set` directive. See Section 7.60 [`.set`], page 49.

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 55. That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted in Chapter 8 [Machine Dependencies], page 55), and underscores. For the AMD 29K family, ‘?’ is also allowed in the body of a symbol name, though not at its beginning.

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

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

Local Symbol Names

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

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

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

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

- L** All local labels begin with ‘L’. Normally both `as` and `ld` forget symbols that start with ‘L’. These labels are used for symbols you are never intended to see. If you use the ‘-L’ option then `as` retains these symbols in the object file. If you also instruct `ld` to retain these symbols, you may use them in debugging.
- digit* If the label is written ‘0:’ then the digit is ‘0’. If the label is written ‘1:’ then the digit is ‘1’. And so on up through ‘9:’.
- C-A** This unusual character is included so you do not accidentally invent a symbol of the same name. The character has ASCII value ‘\001’.
- ordinal number* This is a serial number to keep the labels distinct. The first ‘0:’ gets the number ‘1’; The 15th ‘0:’ gets the number ‘15’; *etc.*. Likewise for the other labels ‘1:’ through ‘9:’.

For instance, the first 1: is named `L1C-A1`, the 44th 3: is named `L3C-A44`.

5.4 The Special Dot Symbol

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

5.5 Symbol Attributes

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

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

5.5.1 Value

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

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

5.5.2 Type

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

5.5.3 Symbol Attributes: `a.out`

5.5.3.1 Descriptor

This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a `.desc` statement (see Section 7.11 [`.desc`], page 37). A descriptor value means nothing to `as`.

5.5.3.2 Other

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

5.5.4 Symbol Attributes for COFF

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

5.5.4.1 Primary Attributes

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

5.5.4.2 Auxiliary Attributes

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

5.5.5 Symbol Attributes for SOM

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

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

6 Expressions

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

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

6.1 Empty Expressions

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

6.2 Integer Expressions

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

6.2.1 Arguments

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

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

Numbers are usually integers.

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

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

6.2.2 Operators

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

6.2.3 Prefix Operator

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

- *Negation*. Two's complement negation.
- ~ *Complementation*. Bitwise not.

6.2.4 Infix Operators

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

1. Highest Precedence

- * *Multiplication*.
- / *Division*. Truncation is the same as the C operator '/'
- % *Remainder*.
- <
- << *Shift Left*. Same as the C operator '<<'
- >
- >> *Shift Right*. Same as the C operator '>>'

2. Intermediate precedence

- | *Bitwise Inclusive Or*.
- & *Bitwise And*.
- ^ *Bitwise Exclusive Or*.
- ! *Bitwise Or Not*.

3. Lowest Precedence

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

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

7 Assembler Directives

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

This chapter discusses directives that are available regardless of the target machine configuration for the GNU assembler. Some machine configurations provide additional directives. See Chapter 8 [Machine Dependencies], page 55.

7.1 `.abort`

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

7.2 `.ABORT`

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

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

7.3 `.align abs-expr, abs-expr, abs-expr`

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

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

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

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

For other systems, including the `i386` using `a.out` format, 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).

7.4 `.ascii "string"...`

`.ascii` expects zero or more string literals (see Section 3.6.1.1 [Strings], page 19) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

7.5 `.asciz "string"...`

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

7.6 `.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` directive 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.

7.7 `.byte expressions`

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

7.8 `.comm symbol , length`

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

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

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

7.9 `.data subsection`

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

7.10 `.def name`

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

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

7.11 `.desc symbol, abs-expression`

This directive sets the descriptor of the symbol (see Section 5.5 [Symbol Attributes], page 30) to the low 16 bits of an absolute expression.

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

7.12 `.dim`

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

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

7.13 `.double flonums`

`.double` expects zero or more flonums, separated by commas. It assembles floating point numbers. The exact kind of floating point numbers emitted depends on how `as` is configured. See Chapter 8 [Machine Dependencies], page 55.

7.14 `.eject`

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

7.15 `.else`

`.else` is part of the `as` support for conditional assembly; see Section 7.34 [`.if`], page 41. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

7.16 `.elseif`

`.elseif` is part of the `as` support for conditional assembly; see Section 7.34 [`.if`], page 41. It is shorthand for beginning a new `.if` block that would otherwise fill the entire `.else` section.

7.17 `.end`

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

7.18 `.endef`

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

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

7.19 `.endfunc`

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

7.20 `.endif`

`.endif` is part of the `as` support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See Section 7.34 [`.if`], page 41.

7.21 `.equ symbol, expression`

This directive sets the value of *symbol* to *expression*. It is synonymous with `.set`; see Section 7.60 [`.set`], page 49.

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

7.22 `.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

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

7.23 `.err`

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

7.24 `.exitm`

Exit early from the current macro definition. See Section 7.47 [Macro], page 45.

7.25 `.extern`

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

7.26 `.fail expression`

Generates an error or a warning. If the value of the *expression* is 500 or more, `as` will print a warning message. If the value is less than 500, `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.

7.27 `.file string`

`.file` tells `as` that we are about to start a new logical file. *string* is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes `"`; but if you wish to specify an empty file name, you must give the quotes-`"`. This statement may go away in future: it is only recognized to be compatible with old `as` programs. In some configurations of `as`, `.file` has already been removed to avoid conflicts with other assemblers. See Chapter 8 [Machine Dependencies], page 55.

7.28 `.fill` *repeat* , *size* , *value*

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

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

7.29 `.float` *flonums*

This directive assembles zero or more flonums, separated by commas. It has the same effect as `.single`. The exact kind of floating point numbers emitted depends on how `as` is configured. See Chapter 8 [Machine Dependencies], page 55.

7.30 `.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`.

7.31 `.global` *symbol*, `.globl` *symbol*

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

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

On the HPPA, `.global` is not always enough to make it accessible to other partial programs. You may need the HPPA-only `.EXPORT` directive as well. See Section 8.8.5 [HPPA Assembler Directives], page 77.

7.32 `.hword` *expressions*

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

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

7.33 `.ident`

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

7.34 `.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 7.20 [`.endif`], page 38); optionally, you may include code for the alternative condition, flagged by `.else` (see Section 7.15 [`.else`], page 38). 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.

7.35 `.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 11). Quotation marks are required around *file*.

7.36 `.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.

7.37 `.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@-
```

7.38 `.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@-
```

7.39 `.lcomm` *symbol, length*

Reserve *length* (an absolute expression) bytes for a local common denoted by *symbol*. The section and value of *symbol* are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. *Symbol* is not declared global (see Section 7.31 [`.global`], page 40), so is normally not visible to `ld`.

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

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

7.40 `.lflags`

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

7.41 `.line` *line-number*

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

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

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

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

7.42 `.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.

7.43 `.ln` *line-number*

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

7.44 `.mri` *val*

If *val* is non-zero, this tells `as` to enter MRI mode. If *val* is zero, this tells `as` to exit MRI mode. This change affects code assembled until the next `.mri` directive, or until the end of the file. See Section 2.7 [MRI mode], page 12.

7.45 `.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 11), the initial value of the listing counter is one.

7.46 `.long` *expressions*

`.long` is the same as '`.int`', see Section 7.36 [`.int`], page 42.

7.47 .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 *'=deft'*. For example, these are all valid `.macro` statements:

`.macro comm`

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

`.macro plus1 p, p1`

`.macro plus1 p p1`

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

`.macro reserve_str p1=0 p2`

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

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

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

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

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

7.48 `.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.

7.49 `.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.

7.50 `.org new-lc , fill`

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

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

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

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

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

7.51 `.p2align[w1] 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 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 `.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` directive 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.

7.52 `.print string`

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

7.53 `.psize lines , columns`

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

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

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

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

7.54 `.purgem name`

Undefine the macro *name*, so that later uses of the string will not be expanded. See Section 7.47 [Macro], page 45.

7.55 `.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.

7.56 `.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
```

7.57 `.sbtbl "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.

7.58 `.scl class`

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

The `.scl` directive is primarily associated with COFF output; when configured to generate `b.out` output format, `as` accepts this directive but ignores it.

7.59 `.section name`

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

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

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

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

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

<code>b</code>	bss section (uninitialized data)
<code>n</code>	section is not loaded
<code>w</code>	writable section
<code>d</code>	data section

r	read-only section
x	executable section
s	shared section (meaningful for PE targets)

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

If the optional argument to the `.section` directive is not quoted, it is taken as a sub-segment number (see Section 4.4 [Sub-Sections], page 25).

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 combination 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

7.60 `.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 30).

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

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

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

7.61 `.short` *expressions*

`.short` is normally the same as ‘`.word`’. See Section 7.78 [`.word`], page 54.

In some configurations, however, `.short` and `.word` generate numbers of different lengths; see Chapter 8 [Machine Dependencies], page 55.

7.62 `.single` *flonums*

This directive assembles zero or more flonums, separated by commas. It has the same effect as `.float`. The exact kind of floating point numbers emitted depends on how `as` is configured. See Chapter 8 [Machine Dependencies], page 55.

7.63 `.size`

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

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

7.64 `.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 7.76 [Uleb128], page 53.

7.65 `.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`’.

7.66 `.space` *size* , *fill*

This directive emits *size* bytes, each of value *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero. This is the same as ‘`.skip`’.

Warning: `.space` has a completely different meaning for HPPA targets; use `.block` as a substitute. See *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001) for the meaning of the `.space` directive. See Section 8.8.5 [HPPA Assembler Directives], page 77, for a summary.

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

Warning: In most versions of the GNU assembler, the directive `.space` has the effect of `.block`. See Chapter 8 [Machine Dependencies], page 55.

7.67 `.stabd`, `.stabn`, `.stabs`

There are three directives that begin '`.stab`'. All emit symbols (see Chapter 5 [Symbols], page 29), for use by symbolic debuggers. The symbols are not entered in the `as` hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required:

<i>string</i>	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.
<i>type</i>	An absolute expression. The symbol's type is set to the low 8 bits of this expression. Any bit pattern is permitted, but <code>ld</code> and debuggers choke on silly bit patterns.
<i>other</i>	An absolute expression. The symbol's "other" attribute is set to the low 8 bits of this expression.
<i>desc</i>	An absolute expression. The symbol's descriptor is set to the low 16 bits of this expression.
<i>value</i>	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.

7.68 `.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 19.

7.69 `.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:
        .struct field1 + 4
field2:
        .struct field2 + 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.

7.70 `.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.

7.71 `.tag structname`

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

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

7.72 `.text` *subsection*

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

7.73 `.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.

7.74 `.type` *int*

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

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

7.75 `.val` *addr*

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

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

7.76 `.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 7.64 [Sleb128], page 50.

7.77 `.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
```

7.78 `.word` expressions

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

The size of the number emitted, and its byte order, depend on what target computer the assembly is for.

Warning: Special Treatment to support Compilers

Machines with a 32-bit address space, but that do less than 32-bit addressing, require the following special treatment. If the machine of interest to you does 32-bit addressing (or doesn't require it; see Chapter 8 [Machine Dependencies], page 55), you can ignore this issue.

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

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

7.79 Deprecated Directives

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

```
.abort
.line
```


8 Machine Dependent Features

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

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

8.18 SPARC Dependent Features

8.18.1 Options

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

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

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

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

`-Av6` | `-Av7` | `-Av8` | `-Asparclet` | `-Asparclite`
`-Av8plus` | `-Av8plusa` | `-Av9` | `-Av9a`

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

‘-Av8plus’ and ‘-Av8plusa’ select a 32 bit environment.

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

‘-Av8plusa’ and ‘-Av9a’ enable the SPARC V9 instruction set with Ultra-SPARC extensions.

`-xarch=v8plus` | `-xarch=v8plusa`

For compatibility with the Solaris v9 assembler. These options are equivalent to `-Av8plus` and `-Av8plusa`, respectively.

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

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

8.18.2 Enforcing aligned data

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

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

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

8.18.3 Floating Point

The Sparc uses IEEE floating-point numbers.

8.18.4 Sparc Machine Directives

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

- `.align` This must be followed by the desired alignment in bytes.
- `.common` This must be followed by a symbol name, a positive number, and `"bss"`. This behaves somewhat like `.comm`, but the syntax is different.
- `.half` This is functionally identical to `.short`.
- `.nword` On the Sparc, the `.nword` directive produces native word sized value, ie. if assembling with -32 it is equivalent to `.word`, if assembling with -64 it is equivalent to `.xword`.
- `.proc` This directive is ignored. Any text following it on the same line is also ignored.
- `.register`
This directive declares use of a global application or system register. It must be followed by a register name `%g2`, `%g3`, `%g6` or `%g7`, comma and the symbol name for that register. If symbol name is `#scratch`, it is a scratch register, if it is `#ignore`, it just suppresses any errors about using undeclared global register, but does not emit any information about it into the object file. This can be useful e.g. if you save the register before use and restore it after.
- `.reserve` This must be followed by a symbol name, a positive number, and `"bss"`. This behaves somewhat like `.lcomm`, but the syntax is different.
- `.seg` This must be followed by `"text"`, `"data"`, or `"data1"`. It behaves like `.text`, `.data`, or `.data 1`.
- `.skip` This is functionally identical to the `.space` directive.
- `.word` On the Sparc, the `.word` directive produces 32 bit values, instead of the 16 bit values it produces on many other machines.
- `.xword` On the Sparc V9 processor, the `.xword` directive produces 64 bit values.

Index

#

#	18
#APP	17
#NO_APP	17

\$

\$ in symbol names	64, 68, 74, 110
--------------------	-----------------

-

--	8
'--base-size-default-16'	95
'--base-size-default-32'	95
'--bitwise-or' option, M680x0	95
'--disp-size-default-16'	95
'--disp-size-default-32'	95
--enforce-aligned-data	112
'--fatal-warnings'	15
'--force-long-branches'	102
'--generate-example'	102
--MD	14
'--no-warn'	15
'--print-insn-syntax'	102
'--print-opcodes'	102
'--register-prefix-optional' option, M680x0	95
'--short-branches'	102
--statistics	15
'--strict-direct-mode'	102
--traditional-format	15
'--warn'	15
'+' option, VAX/VMS	117
'-1' option, VAX/VMS	117
-a	11
-A options, i960	89
-ac	11
-ad	11
-ah	11
-al	11
-an	11
-as	11
-Asparclet	112
-Asparclite	112
-Av6	112
-Av8	112
-Av9	112
-Av9a	112

-b option, i960	89
-D	11
-D, ignored on VAX	116
-d, VAX option	116
-EB command line option, ARM	59
-EB option (MIPS)	105
-EL command line option, ARM	59
-EL option (MIPS)	105
-f	11
-G option (MIPS)	105
'-h' option, VAX/VMS	116
'-H' option, VAX/VMS	117
-I <i>path</i>	12
-J, ignored on VAX	116
-K	12
-k command line option, ARM	59
-L	12
'-l' option, M680x0	95
-M	12
'-m32r' option, M32R	93
'-m32rx' option, M32RX	93
'-m68000' and related options	95
'-m68hc11'	102
'-m68hc12'	102
-mall command line option, ARM	59
-mapcs command line option, ARM	59
-mapcs-float command line option, ARM	59
-mapcs-reentrant command line option, ARM	59
-marm command line option, ARM	59
-marmv command line option, ARM	59
-mbig-endian option (ARC)	56
-mfpa command line option, ARM	59
-mfpe-old command line option, ARM	59
-mlittle-endian option (ARC)	56
-mno-fpu command line option, ARM	59
-moabi command line option, ARM	59
-mthumb command line option, ARM	59
-mthumb-interwork command line option, ARM	59
-mv850 command line option, V850	120
-mv850any command line option, V850	120
-mv850e command line option, V850	120
-no-relax option, i960	90
'-no-warn-explicit-parallel-conflicts' option, M32RX	93
-nocpp ignored (MIPS)	106
-o	14

-R	14	\n (newline character)	20
-S, ignored on VAX	116	\r (carriage return character)	20
-t, ignored on VAX	116	\t (tab)	20
-T, ignored on VAX	116	\xd... (hex character code)	20
-v	15		
-V, redundant on VAX	116	1	
-version	15	16-bit code, i386	87
'-W'	15	2	
'-warn-explicit-parallel-conflicts' option, M32RX	93	29K support	57
'-Wnp' option, M32RX	93	3	
'-Wp' option, M32RX	93	3DNow!, i386	87
-wsigned_overflow command line option, V850	120		
-wunsigned_overflow command line option, V850	120	A	
.		a.out	9
. (symbol)	30	a.out symbol attributes	31
.insn	108	abort directive	35
.ltorg directive, ARM	61	ABORT directive	35
.o	9	absolute section	24
.param on HPPA	77	addition, permitted arguments	34
.pool directive, ARM	61	addresses	33
.set autoextend	107	addresses, format of	24
.set mipsn	107	addressing modes, D10V	65
.set noautoextend	107	addressing modes, D30V	70
.set pop	108	addressing modes, H8/300	71
.set push	108	addressing modes, H8/500	74
.v850 directive, V850	123	addressing modes, M680x0	97
.v850e directive, V850	123	addressing modes, M68HC11	103
:		addressing modes, SH	110
: (label)	19	addressing modes, Z8000	114
@		ADR reg,<label> pseudo op, ARM	61
@word modifier, D10V	66	ADRL reg,<label> pseudo op, ARM	61
\		advancing location counter	46
\" (doublequote character)	20	align directive	35
\\ ('\ character)	20	align directive, ARM	60
\b (backspace character)	19	align directive, SPARC	113
\\ddd (octal character code)	20	altered difference tables	54
\f (formfeed character)	20	alternate syntax for the 680x0	98
		AMD 29K floating point (IEEE)	57
		AMD 29K identifiers	57
		AMD 29K line comment character	57
		AMD 29K machine directives	58
		AMD 29K macros	57

AMD 29K opcodes	58	auxiliary attributes, COFF symbols	31
AMD 29K options (none)	57	auxiliary symbol information, COFF	37
AMD 29K protected registers	57	Av7	112
AMD 29K register names	57	B	
AMD 29K special purpose registers	57	backslash (\)	20
AMD 29K support	57	backspace (\b)	19
ARC architectures	56	balgn directive	36
ARC big-endian output	56	balgnl directive	36
ARC endianness	3	balgnw directive	36
ARC floating point (IEEE)	56	big endian output, ARC	3
ARC little-endian output	56	big endian output, MIPS	5
ARC machine directives	56	big endian output, PJ	4
ARC options	56	big-endian output, ARC	56
ARC support	56	big-endian output, MIPS	105
arch directive, i386	88	bignums	21
architecture options, i960	89	binary integers	21
architecture options, M32R	93	bitfields, not supported on VAX	120
architecture options, M32RX	93	block	115
architecture options, M680x0	95	block directive, AMD 29K	58
architectures, ARC	56	branch improvement, M680x0	99
architectures, SPARC	112	branch improvement, M68HC11	103
arguments for addition	34	branch improvement, VAX	118
arguments for subtraction	34	branch recording, i960	89
arguments in expressions	33	branch statistics table, i960	89
arithmetic functions	33	bss directive, i960	90
arithmetic operands	33	bss section	24, 26
arm directive, ARM	60	bug criteria	127
ARM floating point (IEEE)	60	bug reports	127
ARM identifiers	60	bugs in assembler	127
ARM immediate character	60	bus lock prefixes, i386	85
ARM line comment character	60	bval	115
ARM line separator	60	byte directive	36
ARM machine directives	60	C	
ARM opcodes	61	call instructions, i386	84
ARM options (none)	59	callj , i960 pseudo-opcode	91
ARM register names	60	carriage return (\r)	20
ARM support	59	character constants	19
ascii directive	36	character escape codes	19
asciz directive	36	character, single	20
assembler bugs, reporting	127	characters used in symbols	18
assembler crash	127	code directive, ARM	60
assembler internal logic error	25	code16 directive, i386	87
assembler version	15	code16gcc directive, i386	87
assembler, and linker	23	code32 directive, i386	87
assembly listings, enabling	11		
assigning values to symbols	29, 38		
att_syntax pseudo op, i386	83		
attributes, symbol	30		

COFF auxiliary symbol information	37	D10V line comment character	64
COFF structure debugging	52	D10V opcode summary	66
COFF symbol attributes	31	D10V optimization	4
COFF symbol descriptor	37	D10V options	63
COFF symbol storage class	48	D10V registers	64
COFF symbol type	53	D10V size modifiers	63
COFF symbols, debugging	37	D10V sub-instruction ordering	64
COFF value attribute	53	D10V sub-instructions	63
COMDAT	44	D10V support	63
<code>comm</code> directive	37	D10V syntax	63
command line conventions	8	D30V addressing modes	70
command line options, V850	120	D30V floating point	70
command-line options ignored, VAX	116	D30V Guarded Execution	69
comments	17	D30V line comment character	68
comments, M680x0	101	D30V nops	4
comments, removed by preprocessor	17	D30V nops after 32-bit multiply	4
<code>common</code> directive, SPARC	113	D30V opcode summary	70
common sections	44	D30V optimization	4
common variable storage	26	D30V options	67
compare and jump expansions, i960	91	D30V registers	69
compare/branch instructions, i960	91	D30V size modifiers	67
conditional assembly	41	D30V sub-instruction ordering	68
constant, single character	20	D30V sub-instructions	67
constants	19	D30V support	67
constants, bignum	21	D30V syntax	67
constants, character	19	data alignment on SPARC	112
constants, converted by preprocessor	17	data and text sections, joining	14
constants, floating point	21	<code>data</code> directive	37
constants, integer	21	data section	24
constants, number	20	<code>data1</code> directive, M680x0	99
constants, string	19	<code>data2</code> directive, M680x0	99
conversion instructions, i386	84	<code>dbpc</code> register, V850	123
coprocessor wait, i386	85	<code>dbpsw</code> register, V850	123
<code>cpu</code> directive, SPARC	56	debuggers, and symbol order	29
<code>cputype</code> directive, AMD 29K	58	debugging COFF symbols	37
crash of assembler	127	decimal integers	21
<code>ctbp</code> register, V850	123	<code>def</code> directive	37
<code>ctoff</code> pseudo-op, V850	125	dependency tracking	14
<code>ctpc</code> register, V850	123	deprecated directives	54
<code>ctpsw</code> register, V850	123	<code>desc</code> directive	37
current address	30	descriptor, of <code>a.out</code> symbol	31
current address, advancing	46	<code>dfloat</code> directive, VAX	117
		difference tables altered	54
		difference tables, warning	12
		<code>dim</code> directive	37
		directives and instructions	19
		directives, M680x0	99
		directives, machine independent	35
D			
D10V <code>@word</code> modifier	66		
D10V addressing modes	65		
D10V floating point	66		

directives, Z8000 115
 displacement sizing character, VAX 119
 dot (symbol) 30
 double directive 38
 double directive, i386 86
 double directive, M680x0 99
 double directive, M68HC11 103
 double directive, VAX 117
 doublequote (\") 20

E

ECOFF sections 106
 ecr register, V850 122
 eight-byte integer 47
 eipc register, V850 122
 eipsw register, V850 122
 eject directive 38
 else directive 38
 elseif directive 38
 empty expressions 33
 emulation 6
 end directive 38
 undef directive 38
 endfunc directive 38
 endianness, ARC 3
 endianness, MIPS 5
 endianness, PJ 4
 endif directive 38
 endm directive 45
 EOF, newline must precede 18
 ep register, V850 122
 equ directive 38
 equiv directive 39
 err directive 39
 error messages 9
 error on valid input 127
 errors, caused by warnings 15
 errors, continuing after 15
 ESA/390 floating point (IEEE) 81
 ESA/390 support 80
 ESA/390 Syntax 80
 ESA/390-only directives 81
 escape codes, character 19
 even 115
 even directive, M680x0 99
 exitm directive 45
 expr (internal section) 25

expression arguments 33
 expressions 33
 expressions, empty 33
 expressions, integer 33
 extend directive M680x0 99
 extend directive M68HC11 103
 extended directive, i960 90
 extern directive 39

F

fail directive 39
 faster processing (-f) 11
 fatal signal 127
 fepc register, V850 122
 fepsw register, V850 122
 ffloat directive, VAX 117
 file directive 39
 file directive, AMD 29K 58
 file name, logical 39
 files, including 42
 files, input 8
 fill directive 40
 filling memory 50
 float directive 40
 float directive, i386 86
 float directive, M680x0 99
 float directive, M68HC11 103
 float directive, VAX 117
 floating point numbers 21
 floating point numbers (double) 38
 floating point numbers (single) 40, 50
 floating point, AMD 29K (IEEE) 57
 floating point, ARC (IEEE) 56
 floating point, ARM (IEEE) 60
 floating point, D10V 66
 floating point, D30V 70
 floating point, ESA/390 (IEEE) 81
 floating point, H8/300 (IEEE) 72
 floating point, H8/500 (IEEE) 75
 floating point, HPPA (IEEE) 76
 floating point, i386 86
 floating point, i960 (IEEE) 90
 floating point, M680x0 99
 floating point, M68HC11 103
 floating point, SH (IEEE) 111
 floating point, SPARC (IEEE) 113
 floating point, V850 (IEEE) 123

floating point, VAX	117
flonums	21
<code>force_thumb</code> directive, ARM	60
format of error messages	9
format of warning messages	9
formfeed (<code>\f</code>)	20
<code>func</code> directive	40
functions, in expressions	33

G

<code>gbr960</code> , i960 postprocessor	89
<code>gfloat</code> directive, VAX	117
<code>global</code>	115
<code>global</code> directive	40
<code>gp</code> register, MIPS	106
<code>gp</code> register, V850	121
grouping data	25

H

H8/300 addressing modes	71
H8/300 floating point (IEEE)	72
H8/300 line comment character	71
H8/300 line separator	71
H8/300 machine directives (none)	73
H8/300 opcode summary	73
H8/300 options (none)	71
H8/300 registers	71
H8/300 size suffixes	73
H8/300 support	71
H8/300H, assembling for	73
H8/500 addressing modes	74
H8/500 floating point (IEEE)	75
H8/500 line comment character	74
H8/500 line separator	74
H8/500 machine directives (none)	75
H8/500 opcode summary	75
H8/500 options (none)	74
H8/500 registers	74
H8/500 support	74
<code>half</code> directive, SPARC	113
hex character code (<code>\xd...</code>)	20
hexadecimal integers	21
<code>hfloat</code> directive, VAX	117
<code>hi</code> pseudo-op, V850	124
<code>hi0</code> pseudo-op, V850	123
<code>hidden</code> directive	53

<code>hilo</code> pseudo-op, V850	124
HPPA directives not supported	77
HPPA floating point (IEEE)	76
HPPA Syntax	76
HPPA-only directives	77
<code>hword</code> directive	40

I

i370 support	80
i386 16-bit code	87
i386 arch directive	88
i386 <code>att_syntax</code> pseudo op	83
i386 conversion instructions	84
i386 floating point	86
i386 immediate operands	83
i386 instruction naming	83
i386 instruction prefixes	84
i386 <code>intel_syntax</code> pseudo op	83
i386 jump optimization	86
i386 jump, call, return	83
i386 jump/call operands	83
i386 memory references	85
i386 <code>mul</code> , <code>imul</code> instructions	88
i386 options (none)	83
i386 register operands	83
i386 registers	84
i386 sections	83
i386 size suffixes	83
i386 source, destination operands	83
i386 support	83
i386 syntax compatibility	83
i80306 support	83
i960 architecture options	89
i960 branch recording	89
i960 <code>callj</code> pseudo-opcode	91
i960 compare and jump expansions	91
i960 compare/branch instructions	91
i960 floating point (IEEE)	90
i960 machine directives	90
i960 opcodes	91
i960 options	89
i960 support	89
<code>ident</code> directive	41
identifiers, AMD 29K	57
identifiers, ARM	60
<code>if</code> directive	41
<code>ifc</code> directive	41

<code>ifdef</code> directive	41
<code>ifeq</code> directive	41
<code>ifeqs</code> directive	41
<code>ifge</code> directive	41
<code>ifgt</code> directive	41
<code>ifle</code> directive	41
<code>iflt</code> directive	41
<code>ifnc</code> directive	41
<code>ifndef</code> directive	41
<code>ifne</code> directive	42
<code>ifnes</code> directive	42
<code>ifnotdef</code> directive	41
immediate character, ARM	60
immediate character, M680x0	101
immediate character, VAX	119
immediate operands, i386	83
<code>imul</code> instruction, i386	88
<code>include</code> directive	42
<code>include</code> directive search path	12
indirect character, VAX	119
infix operators	34
inhibiting interrupts, i386	85
input	8
input file linenumbers	8
instruction naming, i386	83
instruction prefixes, i386	84
instruction set, M680x0	99
instruction set, M68HC11	103
instruction summary, D10V	66
instruction summary, D30V	70
instruction summary, H8/300	73
instruction summary, H8/500	75
instruction summary, SH	111
instruction summary, Z8000	115
instructions and directives	19
<code>int</code> directive	42
<code>int</code> directive, H8/300	73
<code>int</code> directive, H8/500	75
<code>int</code> directive, i386	86
integer expressions	33
integer, 16-byte	46
integer, 8-byte	47
integers	21
integers, 16-bit	40
integers, 32-bit	42
integers, binary	21
integers, decimal	21
integers, hexadecimal	21
integers, octal	21
integers, one byte	36
<code>intel_syntax</code> pseudo op, i386	83
internal assembler sections	25
<code>internal</code> directive	53
invalid input	127
invocation summary	1
<code>irp</code> directive	42
<code>irpc</code> directive	43
J	
joining text and data sections	14
jump instructions, i386	84
jump optimization, i386	86
jump/call operands, i386	83
L	
label (:)	19
labels	29
<code>lcomm</code> directive	43
<code>ld</code>	9
<code>ldouble</code> directive M680x0	99
<code>ldouble</code> directive M68HC11	103
<code>LDR reg,=<label1></code> pseudo op, ARM	61
<code>leafproc</code> directive, i960	90
length of symbols	18
<code>lflags</code> directive (ignored)	43
line comment character	18
line comment character, AMD 29K	57
line comment character, ARM	60
line comment character, D10V	64
line comment character, D30V	68
line comment character, H8/300	71
line comment character, H8/500	74
line comment character, M680x0	101
line comment character, SH	110
line comment character, V850	120
line comment character, Z8000	114
<code>line</code> directive	43
<code>line</code> directive, AMD 29K	58
line numbers, in input files	8
line numbers, in warnings/errors	9
line separator character	18
line separator, ARM	60
line separator, H8/300	71
line separator, H8/500	74

line separator, SH	110	M680x0 options	95
line separator, Z8000	114	M680x0 pseudo-opcodes	99
lines starting with #	18	M680x0 size modifiers	97
linker	9	M680x0 support	95
linker, and assembler	23	M680x0 syntax	97
<code>linkonce</code> directive	44	M68HC11 addressing modes	103
<code>list</code> directive	44	M68HC11 and M68HC12 support	102
listing control, turning off	46	M68HC11 branch improvement	103
listing control, turning on	44	M68HC11 floating point	103
listing control: new page	38	M68HC11 opcodes	103
listing control: paper size	47	M68HC11 options	102
listing control: subtitle	48	M68HC11 pseudo-opcodes	103
listing control: title line	53	M68HC11 syntax	102
listings, enabling	11	machine dependencies	55
little endian output, ARC	3	machine directives, AMD 29K	58
little endian output, MIPS	5	machine directives, ARC	56
little endian output, PJ	4	machine directives, ARM	60
little-endian output, ARC	56	machine directives, H8/300 (none)	73
little-endian output, MIPS	105	machine directives, H8/500 (none)	75
<code>ln</code> directive	44	machine directives, i960	90
<code>lo</code> pseudo-op, V850	123	machine directives, SH	111
local common symbols	43	machine directives, SPARC	113
local labels, retaining in output	12	machine directives, V850	123
local symbol names	29	machine directives, VAX	117
location counter	30	machine independent directives	35
location counter, advancing	46	machine instructions (not covered)	7
logical file name	39	machine-independent syntax	17
logical line number	43	<code>macro</code> directive	45
logical line numbers	18	macros	45
<code>long</code> directive	44	Macros, AMD 29K	57
<code>long</code> directive, i386	86	macros, count executed	45
<code>lp</code> register, V850	122	make rules	14
<code>lval</code>	115	manual, structure and purpose	7
M		memory references, i386	85
M32R architecture options	93	merging text and data sections	14
M32R options	93	messages from assembler	9
M32R support	93	minus, permitted arguments	34
M32R warnings	93	MIPS architecture options	105
M680x0 addressing modes	97	MIPS big-endian output	105
M680x0 architecture options	95	MIPS debugging directives	107
M680x0 branch improvement	99	MIPS ECOFF sections	106
M680x0 directives	99	MIPS endianness	5
M680x0 floating point	99	MIPS ISA	5
M680x0 immediate character	101	MIPS ISA override	107
M680x0 line comment character	101	MIPS little-endian output	105
M680x0 opcodes	99	MIPS option stack	108
		MIPS processor	105
		MIT	97

- MMX, i386 87
 - mnemonic suffixes, i386 83
 - mnemonics for opcodes, VAX 117
 - mnemonics, D10V 66
 - mnemonics, D30V 70
 - mnemonics, H8/300 73
 - mnemonics, H8/500 75
 - mnemonics, SH 111
 - mnemonics, Z8000 115
 - Motorola syntax for the 680x0 98
 - MRI compatibility mode 12
 - `mri` directive 44
 - MRI mode, temporarily 44
 - `mul` instruction, i386 88
- N**
- `name` 115
 - named section 48
 - named sections 24
 - names, symbol 29
 - naming object file 14
 - new page, in listings 38
 - newline (`\n`) 20
 - newline, required at file end 18
 - `nolist` directive 46
 - NOP pseudo op, ARM 61
 - null-terminated strings 36
 - number constants 20
 - number of macros executed 45
 - numbered subsections 25
 - numbers, 16-bit 40
 - numeric values 33
 - `nword` directive, SPARC 113
- O**
- object file 9
 - object file format 7
 - object file name 14
 - object file, after errors 15
 - obsolescent directives 54
 - `octa` directive 46
 - octal character code (`\ddd`) 20
 - octal integers 21
 - `offset` directive, V850 123
 - opcode mnemonics, VAX 117
 - opcode summary, D10V 66
 - opcode summary, D30V 70
 - opcode summary, H8/300 73
 - opcode summary, H8/500 75
 - opcode summary, SH 111
 - opcode summary, Z8000 115
 - opcodes for AMD 29K 58
 - opcodes for ARM 61
 - opcodes for V850 123
 - opcodes, i960 91
 - opcodes, M680x0 99
 - opcodes, M68HC11 103
 - operand delimiters, i386 83
 - operand notation, VAX 119
 - operands in expressions 33
 - operator precedence 34
 - operators, in expressions 33
 - operators, permitted arguments 34
 - optimization, D10V 4
 - optimization, D30V 4
 - option summary 1
 - options for AMD29K (none) 57
 - options for ARC 56
 - options for ARM (none) 59
 - options for i386 (none) 83
 - options for SPARC 112
 - options for V850 (none) 120
 - options for VAX/VMS 116
 - options, all versions of assembler 11
 - options, command line 8
 - options, D10V 63
 - options, D30V 67
 - options, H8/300 (none) 71
 - options, H8/500 (none) 74
 - options, i960 89
 - options, M32R 93
 - options, M680x0 95
 - options, M68HC11 102
 - options, PJ 109
 - options, SH (none) 110
 - options, Z8000 114
 - `org` directive 46
 - other attribute, of `a.out` symbol 31
 - output file 9

P

<code>p2align</code> directive	46
<code>p2alignl</code> directive	47
<code>p2alignw</code> directive	47
padding the location counter	35
padding the location counter given a power of two	46
padding the location counter given number of bytes	36
page, in listings	38
paper size, for listings	47
paths for <code>.include</code>	12
patterns, writing in memory	40
PIC code generation for ARM	59
PJ endianness	4
PJ options	109
PJ support	109
plus, permitted arguments	34
precedence of operators	34
precision, floating point	21
prefix operators	34
prefixes, i386	84
preprocessing	17
preprocessing, turning on and off	17
primary attributes, COFF symbols	31
<code>print</code> directive	47
<code>proc</code> directive, SPARC	113
<code>protected</code> directive	53
protected registers, AMD 29K	57
pseudo-opcodes, M680x0	99
pseudo-opcodes, M68HC11	103
pseudo-ops for branch, VAX	118
pseudo-ops, machine independent	35
<code>psize</code> directive	47
psw register, V850	122
<code>purgem</code> directive	47
purpose of GNU assembler	7

Q

<code>quad</code> directive	47
<code>quad</code> directive, i386	86

R

real-mode code, i386	87
<code>register</code> directive, SPARC	113
register names, AMD 29K	57

register names, ARM	60
register names, H8/300	71
register names, V850	121
register names, VAX	119
register operands, i386	83
registers, D10V	64
registers, D30V	69
registers, H8/500	74
registers, i386	84
registers, SH	110
registers, Z8000	114
relocation	23
relocation example	25
repeat prefixes, i386	85
reporting bugs in assembler	127
<code>rept</code> directive	48
<code>req</code> directive, ARM	60
<code>reserve</code> directive, SPARC	113
return instructions, i386	83
<code>rsect</code>	115

S

<code>sbtll</code> directive	48
<code>scl</code> directive	48
<code>sdaoff</code> pseudo-op, V850	124
search path for <code>.include</code>	12
<code>sect</code> directive, AMD 29K	58
<code>section</code> directive	48
<code>section</code> directive, V850	123
section override prefixes, i386	85
section-relative addressing	24
sections	23
sections in messages, internal	25
sections, i386	83
sections, named	24
<code>seg</code> directive, SPARC	113
<code>segm</code>	115
<code>set</code> directive	49
SH addressing modes	110
SH floating point (IEEE)	111
SH line comment character	110
SH line separator	110
SH machine directives	111
SH opcode summary	111
SH options (none)	110
SH registers	110
SH support	110

<code>short</code> directive	50	string constants	19
SIMD, i386	87	<code>string</code> directive	51
single character constant	20	<code>string</code> directive on HPPA	78
<code>single</code> directive	50	string literals	36
<code>single</code> directive, i386	86	string, copying to object file	51
sixteen bit integers	40	<code>struct</code> directive	52
sixteen byte integer	46	structure debugging, COFF	52
<code>size</code> directive	50	sub-instruction ordering, D10V	64
size modifiers, D10V	63	sub-instruction ordering, D30V	68
size modifiers, D30V	67	sub-instructions, D10V	63
size modifiers, M680x0	97	sub-instructions, D30V	67
size prefixes, i386	85	subexpressions	33
size suffixes, H8/300	73	subtitles for listings	48
sizes operands, i386	83	subtraction, permitted arguments	34
<code>skip</code> directive	50	summary of options	1
<code>skip</code> directive, M680x0	99	support	76
<code>skip</code> directive, SPARC	113	supporting files, including	42
<code>sleb128</code> directive	50	suppressing warnings	15
small objects, MIPS ECOFF	106	<code>sval</code>	115
SOM symbol attributes	31	symbol attributes	30
source program	8	symbol attributes, <code>a.out</code>	31
source, destination operands; i386	83	symbol attributes, COFF	31
<code>sp</code> register, V850	121	symbol attributes, SOM	31
<code>space</code> directive	50	symbol descriptor, COFF	37
space used, maximum for assembly	15	symbol names	29
SPARC architectures	112	symbol names, '\$' in	64, 68, 74, 110
SPARC data alignment	112	symbol names, local	29
SPARC floating point (IEEE)	113	symbol names, temporary	29
SPARC machine directives	113	symbol storage class (COFF)	48
SPARC options	112	symbol type	31
SPARC support	112	symbol type, COFF	53
special characters, M680x0	101	symbol value	30
special purpose registers, AMD 29K	57	symbol value, setting	49
<code>stabd</code> directive	51	symbol values, assigning	29
<code>stabn</code> directive	51	symbol versioning	52
<code>stabs</code> directive	51	symbol visibility	53
<code>stabx</code> directives	51	symbol, common	37
standard assembler sections	23	symbol, making visible to linker	40
standard input, as input file	8	symbolic debuggers, information for	51
statement separator character	18	symbols	29
statement separator, ARM	60	symbols with uppercase, VAX/VMS	116
statement separator, H8/300	71	symbols, assigning values to	38
statement separator, H8/500	74	symbols, local common	43
statement separator, SH	110	<code>symver</code> directive	52
statement separator, Z8000	114	syntax compatibility, i386	83
statements, structure of	18	syntax, D10V	63
statistics, about assembly	15	syntax, D30V	67
stopping the assembly	35	syntax, M680x0	97

syntax, M68HC11	102
syntax, machine-independent	17
sysproc directive, i960	91

T

tab (\t)	20
tag directive	52
tdaoff pseudo-op, V850	124
temporary symbol names	29
text and data sections, joining	14
text directive	53
text section	24
tfloat directive, i386	86
thumb directive, ARM	60
Thumb support	59
thumb_func directive, ARM	60
thumb_set directive, ARM	61
time, total for assembly	15
title directive	53
tp register, V850	121
trusted compiler	11
turning preprocessing on and off	17
type directive	53
type of a symbol	31

U

ualong directive, SH	111
uaword directive, SH	111
uleb128 directive	53
undefined section	25
unsegm	115
use directive, AMD 29K	58

V

V850 command line options	120
V850 floating point (IEEE)	123
V850 line comment character	120
V850 machine directives	123
V850 opcodes	123
V850 options (none)	120
V850 register names	121
V850 support	120
val directive	53
value attribute, COFF	53
value of a symbol	30

VAX bitfields not supported	120
VAX branch improvement	118
VAX command-line options ignored	116
VAX displacement sizing character	119
VAX floating point	117
VAX immediate character	119
VAX indirect character	119
VAX machine directives	117
VAX opcode mnemonics	117
VAX operand notation	119
VAX register names	119
VAX support	116
Vax-11 C compatibility	116
VAX/VMS options	116
version of assembler	15
versions of symbols	52
VMS (VAX) options	116

W

warning for altered difference tables	12
warning messages	9
warnings, causing error	15
warnings, M32R	93
warnings, suppressing	15
warnings, switching on	15
whitespace	17
whitespace, removed by preprocessor	17
wide floating point directives, VAX	117
word directive	54
word directive, H8/300	73
word directive, H8/500	75
word directive, i386	86
word directive, SPARC	113
writing patterns in memory	40
wval	115

X

xword directive, SPARC	113
------------------------------	-----

Z

Z800 addressing modes	114	Z8000 options	114
Z8000 directives	115	Z8000 registers	114
Z8000 line comment character	114	Z8000 support	114
Z8000 line separator	114	zdaoff pseudo-op, V850	125
Z8000 opcode summary	115	zero register, V850	121
		zero-terminated strings	36

Table of Contents

1	Overview	1
1.1	Structure of this Manual	7
1.2	The GNU Assembler	7
1.3	Object File Formats	7
1.4	Command Line	8
1.5	Input Files	8
1.6	Output (Object) File	9
1.7	Error and Warning Messages	9
2	Command-Line Options	11
2.1	Enable Listings: <code>-a[cdhlms]</code>	11
2.2	<code>-D</code>	11
2.3	Work Faster: <code>-f</code>	11
2.4	.include search path: <code>-I path</code>	12
2.5	Difference Tables: <code>-K</code>	12
2.6	Include Local Labels: <code>-L</code>	12
2.7	Assemble in MRI Compatibility Mode: <code>-M</code>	12
2.8	Dependency tracking: <code>--MD</code>	14
2.9	Name the Object File: <code>-o</code>	14
2.10	Join Data and Text Sections: <code>-R</code>	14
2.11	Display Assembly Statistics: <code>--statistics</code>	15
2.12	Compatible output: <code>--traditional-format</code>	15
2.13	Announce Version: <code>-v</code>	15
2.14	Control Warnings: <code>-W</code> , <code>--warn</code> , <code>--no-warn</code> , <code>--fatal-warnings</code>	15
2.15	Generate Object File in Spite of Errors: <code>-Z</code>	15
3	Syntax	17
3.1	Preprocessing	17
3.2	Whitespace	17
3.3	Comments	17
3.4	Symbols	18
3.5	Statements	18
3.6	Constants	19
3.6.1	Character Constants	19
3.6.1.1	Strings	19
3.6.1.2	Characters	20
3.6.2	Number Constants	20
3.6.2.1	Integers	21
3.6.2.2	Bignums	21
3.6.2.3	Flonums	21

4	Sections and Relocation	23
4.1	Background	23
4.2	Linker Sections	24
4.3	Assembler Internal Sections	25
4.4	Sub-Sections	25
4.5	bss Section	26
5	Symbols	29
5.1	Labels	29
5.2	Giving Symbols Other Values	29
5.3	Symbol Names	29
5.4	The Special Dot Symbol	30
5.5	Symbol Attributes	30
5.5.1	Value	30
5.5.2	Type	31
5.5.3	Symbol Attributes: <code>a.out</code>	31
5.5.3.1	Descriptor	31
5.5.3.2	Other	31
5.5.4	Symbol Attributes for COFF	31
5.5.4.1	Primary Attributes	31
5.5.4.2	Auxiliary Attributes	31
5.5.5	Symbol Attributes for SOM	31
6	Expressions	33
6.1	Empty Expressions	33
6.2	Integer Expressions	33
6.2.1	Arguments	33
6.2.2	Operators	33
6.2.3	Prefix Operator	34
6.2.4	Infix Operators	34
7	Assembler Directives	35
7.1	<code>.abort</code>	35
7.2	<code>.ABORT</code>	35
7.3	<code>.align abs-expr, abs-expr, abs-expr</code>	35
7.4	<code>.ascii "string"</code>	36
7.5	<code>.asciz "string"</code>	36
7.6	<code>.balign[w1] abs-expr, abs-expr, abs-expr</code>	36
7.7	<code>.byte expressions</code>	36
7.8	<code>.comm symbol, length</code>	37
7.9	<code>.data subsection</code>	37
7.10	<code>.def name</code>	37
7.11	<code>.desc symbol, abs-expression</code>	37
7.12	<code>.dim</code>	37
7.13	<code>.double flonums</code>	38
7.14	<code>.eject</code>	38
7.15	<code>.else</code>	38

7.16	<code>.elseif</code>	38
7.17	<code>.end</code>	38
7.18	<code>.endef</code>	38
7.19	<code>.endfunc</code>	38
7.20	<code>.endif</code>	38
7.21	<code>.equ symbol, expression</code>	38
7.22	<code>.equiv symbol, expression</code>	39
7.23	<code>.err</code>	39
7.24	<code>.exitm</code>	39
7.25	<code>.extern</code>	39
7.26	<code>.fail expression</code>	39
7.27	<code>.file string</code>	39
7.28	<code>.fill repeat, size, value</code>	40
7.29	<code>.float flonums</code>	40
7.30	<code>.func name[, label]</code>	40
7.31	<code>.global symbol, .globl symbol</code>	40
7.32	<code>.hword expressions</code>	40
7.33	<code>.ident</code>	41
7.34	<code>.if absolute expression</code>	41
7.35	<code>.include "file"</code>	42
7.36	<code>.int expressions</code>	42
7.37	<code>.irp symbol, values</code>	42
7.38	<code>.irpc symbol, values</code>	43
7.39	<code>.lcomm symbol, length</code>	43
7.40	<code>.lflags</code>	43
7.41	<code>.line line-number</code>	43
7.42	<code>.linkonce [type]</code>	44
7.43	<code>.ln line-number</code>	44
7.44	<code>.mri val</code>	44
7.45	<code>.list</code>	44
7.46	<code>.long expressions</code>	44
7.47	<code>.macro</code>	45
7.48	<code>.nolist</code>	46
7.49	<code>.octa bignums</code>	46
7.50	<code>.org new-lc, fill</code>	46
7.51	<code>.p2align[w1] abs-expr, abs-expr, abs-expr</code>	46
7.52	<code>.print string</code>	47
7.53	<code>.psize lines, columns</code>	47
7.54	<code>.purgem name</code>	47
7.55	<code>.quad bignums</code>	47
7.56	<code>.rept count</code>	48
7.57	<code>.sbttl "subheading"</code>	48
7.58	<code>.scl class</code>	48
7.59	<code>.section name</code>	48
7.60	<code>.set symbol, expression</code>	49
7.61	<code>.short expressions</code>	50
7.62	<code>.single flonums</code>	50
7.63	<code>.size</code>	50

7.64	<code>.sleb128 expressions</code>	50
7.65	<code>.skip size , fill</code>	50
7.66	<code>.space size , fill</code>	50
7.67	<code>.stabd, .stabn, .stabs</code>	51
7.68	<code>.string "str"</code>	51
7.69	<code>.struct expression</code>	52
7.70	<code>.symver</code>	52
7.71	<code>.tag structname</code>	52
7.72	<code>.text subsection</code>	53
7.73	<code>.title "heading"</code>	53
7.74	<code>.type int</code>	53
7.75	<code>.val addr</code>	53
7.76	<code>.uleb128 expressions</code>	53
7.77	<code>.internal, .hidden, .protected</code>	53
7.78	<code>.word expressions</code>	54
7.79	Deprecated Directives	54
8	Machine Dependent Features	55
8.1	ARC Dependent Features	56
8.1.1	Options	56
8.1.2	Floating Point	56
8.1.3	ARC Machine Directives	56
8.2	AMD 29K Dependent Features	57
8.2.1	Options	57
8.2.2	Syntax	57
8.2.2.1	Macros	57
8.2.2.2	Special Characters	57
8.2.2.3	Register Names	57
8.2.3	Floating Point	57
8.2.4	AMD 29K Machine Directives	58
8.2.5	Opcodes	58
8.3	ARM Dependent Features	59
8.3.1	Options	59
8.3.2	Syntax	60
8.3.2.1	Special Characters	60
8.3.2.2	Register Names	60
8.3.3	Floating Point	60
8.3.4	ARM Machine Directives	60
8.3.5	Opcodes	61
8.4	D10V Dependent Features	63
8.4.1	D10V Options	63
8.4.2	Syntax	63
8.4.2.1	Size Modifiers	63
8.4.2.2	Sub-Instructions	63
8.4.2.3	Special Characters	64
8.4.2.4	Register Names	64
8.4.2.5	Addressing Modes	65
8.4.2.6	@WORD Modifier	66

8.4.3	Floating Point	66
8.4.4	Opcodes	66
8.5	D30V Dependent Features	67
8.5.1	D30V Options	67
8.5.2	Syntax	67
8.5.2.1	Size Modifiers	67
8.5.2.2	Sub-Instructions	67
8.5.2.3	Special Characters	68
8.5.2.4	Guarded Execution	69
8.5.2.5	Register Names	69
8.5.2.6	Addressing Modes	70
8.5.3	Floating Point	70
8.5.4	Opcodes	70
8.6	H8/300 Dependent Features	71
8.6.1	Options	71
8.6.2	Syntax	71
8.6.2.1	Special Characters	71
8.6.2.2	Register Names	71
8.6.2.3	Addressing Modes	71
8.6.3	Floating Point	72
8.6.4	H8/300 Machine Directives	73
8.6.5	Opcodes	73
8.7	H8/500 Dependent Features	74
8.7.1	Options	74
8.7.2	Syntax	74
8.7.2.1	Special Characters	74
8.7.2.2	Register Names	74
8.7.2.3	Addressing Modes	74
8.7.3	Floating Point	75
8.7.4	H8/500 Machine Directives	75
8.7.5	Opcodes	75
8.8	HPPA Dependent Features	76
8.8.1	Notes	76
8.8.2	Options	76
8.8.3	Syntax	76
8.8.4	Floating Point	76
8.8.5	HPPA Assembler Directives	77
8.8.6	Opcodes	79
8.9	ESA/390 Dependent Features	80
8.9.1	Notes	80
8.9.2	Options	80
8.9.3	Syntax	80
8.9.4	Floating Point	81
8.9.5	ESA/390 Assembler Directives	81
8.9.6	Opcodes	82
8.10	80386 Dependent Features	83
8.10.1	Options	83
8.10.2	AT&T Syntax versus Intel Syntax	83

8.10.3	Instruction Naming	83
8.10.4	Register Naming	84
8.10.5	Instruction Prefixes	84
8.10.6	Memory References	85
8.10.7	Handling of Jump Instructions	86
8.10.8	Floating Point	86
8.10.9	Intel's MMX and AMD's 3DNow! SIMD Operations	87
8.10.10	Writing 16-bit Code	87
8.10.11	AT&T Syntax bugs	88
8.10.12	Specifying CPU Architecture	88
8.10.13	Notes	88
8.11	Intel 80960 Dependent Features	89
8.11.1	i960 Command-line Options	89
8.11.2	Floating Point	90
8.11.3	i960 Machine Directives	90
8.11.4	i960 Opcodes	91
8.11.4.1	callj	91
8.11.4.2	Compare-and-Branch	91
8.12	M32R Dependent Features	93
8.12.1	M32R Options	93
8.12.2	M32R Warnings	93
8.13	M680x0 Dependent Features	95
8.13.1	M680x0 Options	95
8.13.2	Syntax	97
8.13.3	Motorola Syntax	98
8.13.4	Floating Point	99
8.13.5	680x0 Machine Directives	99
8.13.6	Opcodes	99
8.13.6.1	Branch Improvement	99
8.13.6.2	Special Characters	101
8.14	M68HC11 and M68HC12 Dependent Features	102
8.14.1	M68HC11 and M68HC12 Options	102
8.14.2	Syntax	102
8.14.3	Floating Point	103
8.14.4	Opcodes	103
8.14.4.1	Branch Improvement	103
8.15	MIPS Dependent Features	105
8.15.1	Assembler options	105
8.15.2	MIPS ECOFF object code	106
8.15.3	Directives for debugging information	107
8.15.4	Directives to override the ISA level	107
8.15.5	Directives for extending MIPS 16 bit instructions	107
8.15.6	Directive to mark data as an instruction	108
8.15.7	Directives to save and restore options	108
8.16	picoJava Dependent Features	109
8.16.1	Options	109

8.17	Hitachi SH Dependent Features	110
8.17.1	Options	110
8.17.2	Syntax	110
8.17.2.1	Special Characters	110
8.17.2.2	Register Names	110
8.17.2.3	Addressing Modes	110
8.17.3	Floating Point	111
8.17.4	SH Machine Directives	111
8.17.5	Opcodes	111
8.18	SPARC Dependent Features	112
8.18.1	Options	112
8.18.2	Enforcing aligned data	112
8.18.3	Floating Point	113
8.18.4	Sparc Machine Directives	113
8.19	Z8000 Dependent Features	114
8.19.1	Options	114
8.19.2	Syntax	114
8.19.2.1	Special Characters	114
8.19.2.2	Register Names	114
8.19.2.3	Addressing Modes	114
8.19.3	Assembler Directives for the Z8000	115
8.19.4	Opcodes	115
8.20	VAX Dependent Features	116
8.20.1	VAX Command-Line Options	116
8.20.2	VAX Floating Point	117
8.20.3	Vax Machine Directives	117
8.20.4	VAX Opcodes	117
8.20.5	VAX Branch Improvement	118
8.20.6	VAX Operands	119
8.20.7	Not Supported on VAX	120
8.21	v850 Dependent Features	120
8.21.1	Options	120
8.21.2	Syntax	120
8.21.2.1	Special Characters	120
8.21.2.2	Register Names	121
8.21.3	Floating Point	123
8.21.4	V850 Machine Directives	123
8.21.5	Opcodes	123
9	Reporting Bugs	127
9.1	Have you found a bug?	127
9.2	How to report bugs	127
10	Acknowledgements	131
	Index	133

