Using as

The GNU Assembler

(GNU Binutils)

Version 2.34

Compiled by Ben Shushu 2020.7
Wechat: runninglinuxkernel

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

Table of Contents

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 [Command-Line Options], page 25.

```
as [-a[cdghlns][=file]] [–alternate] [-D]
  [–compress-debug-sections]  [–nocompress-debug-sections]
  [–debug-prefix-map old=new]
  [–defsym sym=val] [-f] [-g] [–gstabs]
  [–gstabs+] [–gdwarf-2] [–gdwarf-sections]
  [–gdwarf-cie-version=VERSION]
  [–help] [-I dir] [-J]
  [-K] [-L] [–listing-lhs-width=NUM]
  [–listing-lhs-width2=NUM] [–listing-rhs-width=NUM]
  [–listing-cont-lines=NUM] [–keep-locals]
  [–no-pad-sections]
  [-o objfile] [-R]
  [–hash-size=NUM] [–reduce-memory-overheads]
  [–statistics]
  [-v] [-version] [–version]
  [-W] [–warn] [–fatal-warnings] [-w] [-x]
  [-Z] [@FILE]
  [–sectname-subst] [–size-check=[error|warning]]
  [–elf-stt-common=[no|yes]]
  [–generate-missing-build-notes=[no|yes]]
  [–target-help] [target-options]
  [–|files ...]

Target AArch64 options:
   [-EB|-EL]
   [-mabi=ABI]

Target Alpha options:
   [-mcpu]
   [-mdebug | -no-mdebug]
   [-replace | -noreplace]
   [-relax] [-g] [-Gsize]
   [-F] [-32addr]

Target ARC options:
   [-mcpu=cpu]
   [-mA6|-mARC600|-mARC601|-mA7|-mARC700|-mEM|-mHS]
   [-mcode-density]
   [-mrelax]
   [-EB|-EL]

Target ARM options:
   [-mcpu=processor[+extension...]]
   [-march=architecture[+extension...]]
   [-mfpu=floating-point-format]
   [-mfloat-abi=abi]
   [-meabi=ver]
   [-mthumb]
   [-EB|-EL]
   [-mapcs-32|-mapcs-26|-mapcs-float|
    -mapcs-reentrant]
   [-mthumb-interwork] [-k]
```

Target Blackfin options:
[**-mcpu=***processor*[*-sirevision*]]
[**-mfdpic**]
[**-mno-fdpic**]
[**-mnopic**]

Target BPF options:
[**-EL**] [**-EB**]

Target CRIS options:
[**–underscore** | **–no-underscore**]
[**–pic**] [**-N**]
[**–emulation=criself** | **–emulation=crisaout**]
[**–march=v0_v10** | **–march=v10** | **–march=v32** | **–march=common_v10_v32**]

Target C-SKY options:
[**-march=***arch*] [**-mcpu=***cpu*]
[**-EL**] [**-mlittle-endian**] [**-EB**] [**-mbig-endian**]
[**-fpic**] [**-pic**]
[**-mljump**] [**-mno-ljump**]
[**-force2bsr**] [**-mforce2bsr**] [**-no-force2bsr**] [**-mno-force2bsr**]
[**-jsri2bsr**] [**-mjsri2bsr**] [**-no-jsri2bsr**] [**-mno-jsri2bsr**]
[**-mnolrw**] [**-mno-lrw**]
[**-melrw**] [**-mno-elrw**]
[**-mlaf**] [**-mliterals-after-func**]
[**-mno-laf**] [**-mno-literals-after-func**]
[**-mlabr**] [**-mliterals-after-br**]
[**-mno-labr**] [**-mnoliterals-after-br**]
[**-mistack**] [**-mno-istack**]
[**-mhard-float**] [**-mmp**] [**-mcp**] [**-mcache**]
[**-msecurity**] [**-mtrust**]
[**-mdsp**] [**-medsp**] [**-mvdsp**]

Target D10V options:
[**-O**]

Target D30V options:
[**-O**|**-n**|**-N**]

Target EPIPHANY options:
[**-mepiphany**|**-mepiphany16**]

Target i386 options:
[**–32**|**–x32**|**–64**] [**-n**]
[**-march=***CPU*[**+***EXTENSION*...]] [**-mtune=***CPU*]

Target IA-64 options:
[**-mconstant-gp**|**-mauto-pic**]
[**-milp32**|**-milp64**|**-mlp64**|**-mp64**]
[**-mle**|**mbe**]
[**-mtune=itanium1**|**-mtune=itanium2**]
[**-munwind-check=warning**|**-munwind-check=error**]
[**-mhint.b=ok**|**-mhint.b=warning**|**-mhint.b=error**]
[**-x**|**-xexplicit**] [**-xauto**] [**-xdebug**]

Target IP2K options:
[**-mip2022**|**-mip2022ext**]

Target M32C options:
[**-m32c**|**-m16c**] [**-relax**] [**-h-tick-hex**]

Target M32R options:
[–**m32rx**|–[**no-**]**warn-explicit-parallel-conflicts**|
–**W**[**n**]**p**]

Target M680X0 options:
[**-l**] [**-m68000**|**-m68010**|**-m68020**|...]

Target M68HC11 options:
[**-m68hc11**|**-m68hc12**|**-m68hcs12**|**-mm9s12x**|**-mm9s12xg**]
[**-mshort**|**-mlong**]
[**-mshort-double**|**-mlong-double**]
[–**force-long-branches**] [–**short-branches**]
[–**strict-direct-mode**] [–**print-insn-syntax**]
[–**print-opcodes**] [–**generate-example**]

Target MCORE options:
[**-jsri2bsr**] [**-sifilter**] [**-relax**]
[**-mcpu=**[**210**|**340**]]

Target Meta options:
[**-mcpu=***cpu*] [**-mfpu=***cpu*] [**-mdsp=***cpu*]
Target MICROBLAZE options:

Target MIPS options:
[**-nocpp**] [**-EL**] [**-EB**] [**-O**[*optimization level*]]
[**-g**[*debug level*]] [**-G** *num*] [**-KPIC**] [**-call_shared**]
[**-non_shared**] [**-xgot** [**-mvxworks-pic**]
[**-mabi=***ABI*] [**-32**] [**-n32**] [**-64**] [**-mfp32**] [**-mgp32**]
[**-mfp64**] [**-mgp64**] [**-mfpxx**]
[**-modd-spreg**] [**-mno-odd-spreg**]
[**-march=***CPU*] [**-mtune=***CPU*] [**-mips1**] [**-mips2**]
[**-mips3**] [**-mips4**] [**-mips5**] [**-mips32**] [**-mips32r2**]
[**-mips32r3**] [**-mips32r5**] [**-mips32r6**] [**-mips64**] [**-mips64r2**]
[**-mips64r3**] [**-mips64r5**] [**-mips64r6**]
[**-construct-floats**] [**-no-construct-floats**]
[**-mignore-branch-isa**] [**-mno-ignore-branch-isa**]
[**-mnan=***encoding*]
[**-trap**] [**-no-break**] [**-break**] [**-no-trap**]
[**-mips16**] [**-no-mips16**]
[**-mmips16e2**] [**-mno-mips16e2**]
[**-mmicromips**] [**-mno-micromips**]
[**-msmartmips**] [**-mno-smartmips**]
[**-mips3d**] [**-no-mips3d**]
[**-mdmx**] [**-no-mdmx**]
[**-mdsp**] [**-mno-dsp**]
[**-mdspr2**] [**-mno-dspr2**]
[**-mdspr3**] [**-mno-dspr3**]
[**-mmsa**] [**-mno-msa**]
[**-mxpa**] [**-mno-xpa**]
[**-mmt**] [**-mno-mt**]
[**-mmcu**] [**-mno-mcu**]
[**-mcrc**] [**-mno-crc**]
[**-mginv**] [**-mno-ginv**]
[**-mloongson-mmi**] [**-mno-loongson-mmi**]

[**-mloongson-cam**] [**-mno-loongson-cam**]
[**-mloongson-ext**] [**-mno-loongson-ext**]
[**-mloongson-ext2**] [**-mno-loongson-ext2**]
[**-minsn32**] [**-mno-insn32**]
[**-mfix7000**] [**-mno-fix7000**]
[**-mfix-rm7000**] [**-mno-fix-rm7000**]
[**-mfix-vr4120**] [**-mno-fix-vr4120**]
[**-mfix-vr4130**] [**-mno-fix-vr4130**]
[**-mfix-r5900**] [**-mno-fix-r5900**]
[**-mdebug**] [**-no-mdebug**]
[**-mpdr**] [**-mno-pdr**]

Target MMIX options:
[–**fixed-special-register-names**] [–**globalize-symbols**]
[–**gnu-syntax**] [–**relax**] [–**no-predefined-symbols**]
[–**no-expand**] [–**no-merge-gregs**] [**-x**]
[–**linker-allocated-gregs**]

Target Nios II options:
[**-relax-all**] [**-relax-section**] [**-no-relax**]
[**-EB**] [**-EL**]

Target NDS32 options:
[**-EL**] [**-EB**] [**-O**] [**-Os**] [**-mcpu=***cpu*]
[**-misa=***isa*] [**-mabi=***abi*] [**-mall-ext**]
[**-m**[**no-**]**16-bit**] [**-m**[**no-**]**perf-ext**] [**-m**[**no-**]**perf2-ext**]
[**-m**[**no-**]**string-ext**] [**-m**[**no-**]**dsp-ext**] [**-m**[**no-**]**mac**] [**-m**[**no-**]**div**]
[**-m**[**no-**]**audio-isa-ext**] [**-m**[**no-**]**fpu-sp-ext**] [**-m**[**no-**]**fpu-dp-ext**]
[**-m**[**no-**]**fpu-fma**] [**-mfpu-freg=***FREG*] [**-mreduced-regs**]
[**-mfull-regs**] [**-m**[**no-**]**dx-regs**] [**-mpic**] [**-mno-relax**]
[**-mb2bb**]

Target PDP11 options:
[**-mpic**|**-mno-pic**] [**-mall**] [**-mno-extensions**]
[**-m***extension*|**-mno-***extension*]
[**-m***cpu*] [**-m***machine*]

Target picoJava options:
[**-mb**|**-me**]

Target PowerPC options:
[**-a32**|**-a64**]
[**-mpwrx**|**-mpwr2**|**-mpwr**|**-m601**|**-mppc**|**-mppc32**|**-m603**|**-m604**|**-m403**|**-m405**|
-m440|**-m464**|**-m476**|**-m7400**|**-m7410**|**-m7450**|**-m7455**|**-m750cl**|**-mgekko**|
-mbroadway|**-mppc64**|**-m620**|**-me500**|**-e500x2**|**-me500mc**|**-me500mc64**|**-me5500**|
-me6500|**-mppc64bridge**|**-mbooke**|**-mpower4**|**-mpwr4**|**-mpower5**|**-mpwr5**|**-mpwr5x**|
-mpower6|**-mpwr6**|**-mpower7**|**-mpwr7**|**-mpower8**|**-mpwr8**|**-mpower9**|**-mpwr9-ma2**|
-mcell|**-mspe**|**-mspe2**|**-mtitan**|**-me300**|**-mcom**]
[**-many**] [**-maltivec**|**-mvsx**|**-mhtm**|**-mvle**]
[**-mregnames**|**-mno-regnames**]
[**-mrelocatable**|**-mrelocatable-lib**|**-K PIC**] [**-memb**]
[**-mlittle**|**-mlittle-endian**|**-le**|**-mbig**|**-mbig-endian**|**-be**]
[**-msolaris**|**-mno-solaris**]
[**-nops=***count*]

Target PRU options:
[**-link-relax**]
[**-mnolink-relax**]

[**-mno-warn-regname-label**]

Target RISC-V options:
[**-fpic**|**-fPIC**|**-fno-pic**]
[**-march=***ISA*]
[**-mabi=***ABI*]

Target RL78 options:
[**-mg10**]
[**-m32bit-doubles**|**-m64bit-doubles**]

Target RX options:
[**-mlittle-endian**|**-mbig-endian**]
[**-m32bit-doubles**|**-m64bit-doubles**]
[**-muse-conventional-section-names**]
[**-msmall-data-limit**]
[**-mpid**]
[**-mrelax**]
[**-mint-register=***number*]
[**-mgcc-abi**|**-mrx-abi**]

Target s390 options:
[**-m31**|**-m64**] [**-mesa**|**-mzarch**] [**-march=***CPU*]
[**-mregnames**|**-mno-regnames**]
[**-mwarn-areg-zero**]

Target SCORE options:
[**-EB**] [**-EL**] [**-FIXDD**] [**-NWARN**]
[**-SCORE5**] [**-SCORE5U**] [**-SCORE7**] [**-SCORE3**]
[**-march=score7**] [**-march=score3**]
[**-USE_R1**] [**-KPIC**] [**-O0**] [**-G** *num*] [**-V**]

Target SPARC options:
[**-Av6**|**-Av7**|**-Av8**|**-Aleon**|**-Asparclet**|**-Asparclite**
-Av8plus|**-Av8plusa**|**-Av8plusb**|**-Av8plusc**|**-Av8plusd**
-Av8plusv|**-Av8plusm**|**-Av9**|**-Av9a**|**-Av9b**|**-Av9c**
-Av9d|**-Av9e**|**-Av9v**|**-Av9m**|**-Asparc**|**-Asparcvis**
-Asparcvis2|**-Asparcfmaf**|**-Asparcima**|**-Asparcvis3**
-Asparcvisr|**-Asparc5**]
[**-xarch=v8plus**|**-xarch=v8plusa**] |**-xarch=v8plusb**|**-xarch=v8plusc**
-xarch=v8plusd|**-xarch=v8plusv**|**-xarch=v8plusm**|**-xarch=v9**
-xarch=v9a|**-xarch=v9b**|**-xarch=v9c**|**-xarch=v9d**|**-xarch=v9e**
-xarch=v9v|**-xarch=v9m**|**-xarch=sparc**|**-xarch=sparcvis**
-xarch=sparcvis2|**-xarch=sparcfmaf**|**-xarch=sparcima**
-xarch=sparcvis3|**-xarch=sparcvisr**|**-xarch=sparc5**
-bump]
[**-32**|**-64**]
[**–enforce-aligned-data**] [**–dcti-couples-detect**]

Target TIC54X options:
[**-mcpu=54**[**123589**]|**-mcpu=54**[**56**]**lp**] [**-mfar-mode**|**-mf**]
[**-merrors-to-file** <*filename*>|**-me** <*filename*>]

Target TIC6X options:
[**-march=***arch*] [**-mbig-endian**|**-mlittle-endian**]
[**-mdsbt**|**-mno-dsbt**] [**-mpid=no**|**-mpid=near**|**-mpid=far**]
[**-mpic**|**-mno-pic**]

Target TILE-Gx options:
[**-m32**|**-m64**] [**-EB**] [**-EL**]

Target Visium options:
[**-mtune**=*arch*]

Target Xtensa options:
[–[**no-**]**text-section-literals**] [–[**no-**]**auto-litpools**]
[–[**no-**]**absolute-literals**]
[–[**no-**]**target-align**] [–[**no-**]**longcalls**]
[–[**no-**]**transform**]
[–**rename-section** *oldname*=*newname*]
[–[**no-**]**trampolines**]

Target Z80 options:
[**-z80**] | [**-z180**] | [**-r800**] | [**-ez80**] | [**-ez80-adl**]
[**-local-prefix**=*PREFIX*]
[**-colonless**]
[**-sdcc**]
[**-fp-s**=*FORMAT*]
[**-fp-d**=*FORMAT*]
[**-strict**] | [**-full**]
[**-with-inst**=*INST*[**,...**]] [**-Wnins** *INST*[**,...**]]
[**-without-inst**=*INST*[**,...**]] [**-Fins** *INST*[**,...**]]
[**-ignore-undocumented-instructions**] [**-Wnud**]
[**-ignore-unportable-instructions**] [**-Wnup**]
[**-warn-undocumented-instructions**] [**-Wud**]
[**-warn-unportable-instructions**] [**-Wup**]
[**-forbid-undocumented-instructions**] [**-Fud**]
[**-forbid-unportable-instructions**] [**-Fup**]

`@`*file*
: Read command-line options from *file*. The options read are inserted in place of the original @*file* option. If *file* does not exist, or cannot be read, then the option will be treated literally, and not removed.

  Options in *file* are separated by whitespace. A whitespace character may be included in an option by surrounding the entire option in either single or double quotes. Any character (including a backslash) may be included by prefixing the character to be included with a backslash. The *file* may itself contain additional @*file* options; any such options will be processed recursively.

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

  `-ac`
  : omit false conditionals

  `-ad`
  : omit debugging directives

  `-ag`
  : include general information, like as version and options passed

  `-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`'.

`--alternate`
Begin in alternate macro mode. See Section 7.4 [`.altmacro`], page 52.

`--compress-debug-sections`
Compress DWARF debug sections using zlib with SHF_COMPRESSED from the ELF ABI. The resulting object file may not be compatible with older linkers and object file utilities. Note if compression would make a given section *larger* then it is not compressed.

`--compress-debug-sections=none`
`--compress-debug-sections=zlib`
`--compress-debug-sections=zlib-gnu`
`--compress-debug-sections=zlib-gabi`
These options control how DWARF debug sections are compressed. `--compress-debug-sections=none` is equivalent to `--nocompress-debug-sections`. `--compress-debug-sections=zlib` and `--compress-debug-sections=zlib-gabi` are equivalent to `--compress-debug-sections`. `--compress-debug-sections=zlib-gnu` compresses DWARF debug sections using zlib. The debug sections are renamed to begin with '`.zdebug`'. Note if compression would make a given section *larger* then it is not compressed nor renamed.

`--nocompress-debug-sections`
Do not compress DWARF debug sections. This is usually the default for all targets except the x86/x86_64, but a configure time option can be used to override this.

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

`--debug-prefix-map` *`old=new`*
When assembling files in directory *`old`*, record debugging information describing them as in *`new`* instead.

`--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. The value of the symbol can be overridden inside a source file via the use of a `.set` pseudo-op.

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

`-g`
`--gen-debug`
: Generate debugging information for each assembler source line using whichever debug format is preferred by the target. This currently means either STABS, ECOFF or DWARF2.

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

`--gstabs+`
: Generate stabs debugging information for each assembler line, with GNU extensions that probably only gdb can handle, and that could make other debuggers crash or refuse to read your program. This may help debugging assembler code. Currently the only GNU extension is the location of the current working directory at assembling time.

`--gdwarf-2`
: Generate DWARF2 debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it. Note—this option is only supported by some targets, not all of them.

`--gdwarf-sections`
: Instead of creating a .debug_line section, create a series of .debug_line.*foo* sections where *foo* is the name of the corresponding code section. For example a code section called *.text.func* will have its dwarf line number information placed into a section called *.debug_line.text.func*. If the code section is just called *.text* then debug line section will still be called just *.debug_line* without any suffix.

`--gdwarf-cie-version=`*version*
: Control which version of DWARF Common Information Entries (CIEs) are produced. When this flag is not specificed the default is version 1, though some targets can modify this default. Other possible values for *version* are 3 or 4.

`--size-check=error`
`--size-check=warning`
: Issue an error or warning for invalid ELF .size directive.

`--elf-stt-common=no`
`--elf-stt-common=yes`
: These options control whether the ELF assembler should generate common symbols with the `STT_COMMON` type. The default can be controlled by a configure option `--enable-elf-stt-common`.

`--generate-missing-build-notes=yes`
`--generate-missing-build-notes=no`
: These options control whether the ELF assembler should generate GNU Build attribute notes if none are present in the input sources. The default can be controlled by the `--enable-generate-build-notes` configure option.

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

`--target-help`
: Print a summary of all target specific 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. These symbols start with system-specific local label prefixes, typically '`.L`' for ELF systems or '`L`' for traditional a.out systems. See Section 5.3 [Symbol Names], page 43.

`--listing-lhs-width=`*number*
Set the maximum width, in words, of the output data column for an assembler listing to *number*.

`--listing-lhs-width2=`*number*
Set the maximum width, in words, of the output data column for continuation lines in an assembler listing to *number*.

`--listing-rhs-width=`*number*
Set the maximum width of an input source line, as displayed in a listing, to *number* bytes.

`--listing-cont-lines=`*number*
Set the maximum number of lines printed in a listing for a single line of input to *number* + 1.

`--no-pad-sections`
Stop the assembler for padding the ends of output sections to the alignment of that section. The default is to pad the sections, but this can waste space which might be needed on targets which have tight memory constraints.

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

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

`--hash-size=`*number*
Set the default size of GAS's hash tables to a prime number close to *number*. Increasing this value can reduce the length of time it takes the assembler to perform its tasks, at the expense of increasing the assembler's memory requirements. Similarly reducing this value can reduce the memory requirements at the expense of speed.

`--reduce-memory-overheads`
This option reduces GAS's memory requirements, at the expense of making the assembly processes slower. Currently this switch is a synonym for '`--hash-size=4051`', but in the future it may have other effects as well.

`--sectname-subst`
Honor substitution sequences in section names. See [`.section` *name*], page 77.

`--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.

See Section 9.1.1 [AArch64 Options], page 94, for the options available when as is configured for the 64-bit mode of the ARM Architecture (AArch64).

See ⟨undefined⟩ [Alpha Options], page ⟨undefined⟩, for the options available when as is configured for an Alpha processor.

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

`-mcpu=`*`cpu`*
: This option selects the core processor variant.

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

`-mcode-density`
: Enable Code Density extenssion instructions.

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

`-mcpu=`*`processor`*`[+`*`extension...`*`]`
: Specify which ARM processor variant is the target.

`-march=`*`architecture`*`[+`*`extension...`*`]`
: Specify which ARM architecture variant is used by the target.

`-mfpu=`*`floating-point-format`*
: Select which Floating Point architecture is the target.

`-mfloat-abi=`*`abi`*
: Select which floating point ABI is in use.

`-mthumb` Enable Thumb only instruction decoding.

`-mapcs-32 | -mapcs-26 | -mapcs-float | -mapcs-reentrant`
: 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.

`-mccs` Turns on CodeComposer Studio assembly syntax compatibility mode.

`-k` Specify that PIC code has been generated.

See ⟨undefined⟩ [Blackfin Options], page ⟨undefined⟩, for the options available when as is configured for the Blackfin processor family.

See ⟨undefined⟩ [BPF Options], page ⟨undefined⟩, for the options available when as is configured for the Linux kernel BPF processor family.

See the info pages for documentation of the CRIS-specific options.

See ⟨undefined⟩ [C-SKY Options], page ⟨undefined⟩, for the options available when as is configured for the C-SKY processor family.

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 Adapteva EPIPHANY series.

See ⟨undefined⟩ [Epiphany Options], page ⟨undefined⟩, for the options available when as is configured for an Epiphany processor.

See ⟨undefined⟩ [i386-Options], page ⟨undefined⟩, for the options available when as is configured for an i386 processor.

The following options are available when as is configured for the Ubicom IP2K series.

`-mip2022ext`
: Specifies that the extended IP2022 instructions are allowed.

`-mip2022` Restores the default behaviour, which restricts the permitted instructions to just the basic IP2022 ones.

The following options are available when as is configured for the Renesas M32C and M16C processors.

`-m32c` Assemble M32C instructions.

`-m16c` Assemble M16C instructions (the default).

`-relax` Enable support for link-time relaxations.

`-h-tick-hex`
: Support H'00 style hex constants in addition to 0x00 style.

The following options are available when as is configured for the Renesas M32R (formerly 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.

See ⟨undefined⟩ [Nios II Options], page ⟨undefined⟩, for the options available when as is configured for an Altera Nios II processor.

For details about the PDP-11 machine dependent features options, see ⟨undefined⟩ [PDP-11-Options], page ⟨undefined⟩.

`-mpic | -mno-pic`
: Generate position-independent (or position-dependent) code. The default is `-mpic`.

`-mall`
`-mall-extensions`
: Enable all instruction set extensions. This is the default.

`-mno-extensions`
: Disable all instruction set extensions.

`-m`*extension* `| -mno-`*extension*
: Enable (or disable) a particular instruction set extension.

`-m`*cpu*
: Enable the instruction set extensions supported by a particular CPU, and disable all other extensions.

`-m`*machine*
: Enable the instruction set extensions supported by a particular machine model, and disable all other extensions.

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.

See ⟨undefined⟩ [PRU Options], page ⟨undefined⟩, for the options available when as is configured for a PRU processor.

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

`-m68hc11 | -m68hc12 | -m68hcs12 | -mm9s12x | -mm9s12xg`
: Specify what processor is the target. The default is defined by the configuration option when building the assembler.

`--xgate-ramoffset`
: Instruct the linker to offset RAM addresses from S12X address space into XGATE address space.

`-mshort`
: Specify to use the 16-bit integer ABI.

`-mlong`
: Specify to use the 32-bit integer ABI.

`-mshort-double`
: Specify to use the 32-bit double ABI.

`-mlong-double`
: Specify to use the 64-bit double ABI.

`--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 UltraSPARC extensions.

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

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

The following options are available when as is configured for the 'c54x architecture.

`-mfar-mode`
: Enable extended addressing mode. All addresses and relocations will assume extended addressing (usually 23 bits).

`-mcpu=`*`CPU_VERSION`*
: Sets the CPU version being compiled for.

`-merrors-to-file` *`FILENAME`*
: Redirect error output to a file, for broken systems which don't support such behaviour in the shell.

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`
`-mips4`
`-mips5`
`-mips32`
`-mips32r2`
`-mips32r3`
`-mips32r5`
`-mips32r6`
`-mips64`
`-mips64r2`
`-mips64r3`
`-mips64r5`
`-mips64r6`

Generate code for a particular MIPS Instruction Set Architecture level. '`-mips1`' is an alias for '`-march=r3000`', '`-mips2`' is an alias for '`-march=r6000`', '`-mips3`' is an alias for '`-march=r4000`' and '`-mips4`' is an alias for '`-march=r8000`'. '`-mips5`', '`-mips32`', '`-mips32r2`', '`-mips32r3`', '`-mips32r5`', '`-mips32r6`', '`-mips64`', '`-mips64r2`', '`-mips64r3`', '`-mips64r5`', and '`-mips64r6`' correspond to generic MIPS V, MIPS32, MIPS32 Release 2, MIPS32 Release 3, MIPS32 Release 5, MIPS32 Release 6, MIPS64, MIPS64 Release 2, MIPS64 Release 3, MIPS64 Release 5, and MIPS64 Release 6 ISA processors, respectively.

`-march=`*cpu*

Generate code for a particular MIPS CPU.

`-mtune=`*cpu*

Schedule and tune for a particular MIPS CPU.

`-mfix7000`
`-mno-fix7000`

Cause nops to be inserted if the read of the destination register of an mfhi or mflo instruction occurs in the following two instructions.

`-mfix-rm7000`
`-mno-fix-rm7000`

Cause nops to be inserted if a dmult or dmultu instruction is followed by a load instruction.

`-mfix-r5900`
`-mno-fix-r5900`

Do not attempt to schedule the preceding instruction into the delay slot of a branch instruction placed at the end of a short loop of six instructions or fewer and always schedule a `nop` instruction there instead. The short loop bug under certain conditions causes loops to execute only once or twice, due to a hardware bug in the R5900 chip.

`-mdebug`
`-no-mdebug`
: Cause stabs-style debugging output to go into an ECOFF-style .mdebug section instead of the standard ELF .stabs sections.

`-mpdr`
`-mno-pdr`
: Control generation of `.pdr` sections.

`-mgp32`
`-mfp32`
: The register sizes are normally inferred from the ISA and ABI, but these flags force a certain group of registers to be treated as 32 bits wide at all times. '`-mgp32`' controls the size of general-purpose registers and '`-mfp32`' controls the size of floating-point registers.

`-mgp64`
`-mfp64`
: The register sizes are normally inferred from the ISA and ABI, but these flags force a certain group of registers to be treated as 64 bits wide at all times. '`-mgp64`' controls the size of general-purpose registers and '`-mfp64`' controls the size of floating-point registers.

`-mfpxx`
: The register sizes are normally inferred from the ISA and ABI, but using this flag in combination with '`-mabi=32`' enables an ABI variant which will operate correctly with floating-point registers which are 32 or 64 bits wide.

`-modd-spreg`
`-mno-odd-spreg`
: Enable use of floating-point operations on odd-numbered single-precision registers when supported by the ISA. '`-mfpxx`' implies '`-mno-odd-spreg`', otherwise the default is '`-modd-spreg`'.

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

`-mmips16e2`
`-mno-mips16e2`
: Enable the use of MIPS16e2 instructions in MIPS16 mode. This is equivalent to putting `.module mips16e2` at the start of the assembly file. '`-mno-mips16e2`' turns off this option.

`-mmicromips`
`-mno-micromips`
: Generate code for the microMIPS processor. This is equivalent to putting `.module micromips` at the start of the assembly file. '`-mno-micromips`' turns off this option. This is equivalent to putting `.module nomicromips` at the start of the assembly file.

`-msmartmips`
`-mno-smartmips`
: Enables the SmartMIPS extension to the MIPS32 instruction set. This is equivalent to putting `.module smartmips` at the start of the assembly file. '`-mno-smartmips`' turns off this option.

`-mips3d`
`-no-mips3d`
: Generate code for the MIPS-3D Application Specific Extension. This tells the assembler to accept MIPS-3D instructions. '`-no-mips3d`' turns off this option.

`-mdmx`
`-no-mdmx` Generate code for the MDMX Application Specific Extension. This tells the assembler to accept MDMX instructions. '`-no-mdmx`' turns off this option.

`-mdsp`
`-mno-dsp` Generate code for the DSP Release 1 Application Specific Extension. This tells the assembler to accept DSP Release 1 instructions. '`-mno-dsp`' turns off this option.

`-mdspr2`
`-mno-dspr2`
: Generate code for the DSP Release 2 Application Specific Extension. This option implies '`-mdsp`'. This tells the assembler to accept DSP Release 2 instructions. '`-mno-dspr2`' turns off this option.

`-mdspr3`
`-mno-dspr3`
: Generate code for the DSP Release 3 Application Specific Extension. This option implies '`-mdsp`' and '`-mdspr2`'. This tells the assembler to accept DSP Release 3 instructions. '`-mno-dspr3`' turns off this option.

`-mmsa`
`-mno-msa` Generate code for the MIPS SIMD Architecture Extension. This tells the assembler to accept MSA instructions. '`-mno-msa`' turns off this option.

`-mxpa`
`-mno-xpa` Generate code for the MIPS eXtended Physical Address (XPA) Extension. This tells the assembler to accept XPA instructions. '`-mno-xpa`' turns off this option.

`-mmt`
`-mno-mt` Generate code for the MT Application Specific Extension. This tells the assembler to accept MT instructions. '`-mno-mt`' turns off this option.

`-mmcu`
`-mno-mcu` Generate code for the MCU Application Specific Extension. This tells the assembler to accept MCU instructions. '`-mno-mcu`' turns off this option.

`-mcrc`
`-mno-crc` Generate code for the MIPS cyclic redundancy check (CRC) Application Specific Extension. This tells the assembler to accept CRC instructions. '`-mno-crc`' turns off this option.

`-mginv`
`-mno-ginv`
: Generate code for the Global INValidate (GINV) Application Specific Extension. This tells the assembler to accept GINV instructions. '`-mno-ginv`' turns off this option.

`-mloongson-mmi`
`-mno-loongson-mmi`
: Generate code for the Loongson MultiMedia extensions Instructions (MMI) Application Specific Extension. This tells the assembler to accept MMI instructions. '`-mno-loongson-mmi`' turns off this option.

`-mloongson-cam`
`-mno-loongson-cam`
: Generate code for the Loongson Content Address Memory (CAM) instructions. This tells the assembler to accept Loongson CAM instructions. '`-mno-loongson-cam`' turns off this option.

`-mloongson-ext`
`-mno-loongson-ext`
: Generate code for the Loongson EXTensions (EXT) instructions. This tells the assembler to accept Loongson EXT instructions. '`-mno-loongson-ext`' turns off this option.

`-mloongson-ext2`
`-mno-loongson-ext2`
: Generate code for the Loongson EXTensions R2 (EXT2) instructions. This option implies '`-mloongson-ext`'. This tells the assembler to accept Loongson EXT2 instructions. '`-mno-loongson-ext2`' turns off this option.

`-minsn32`
`-mno-insn32`
: Only use 32-bit instruction encodings when generating code for the microMIPS processor. This option inhibits the use of any 16-bit instructions. This is equivalent to putting `.set insn32` at the start of the assembly file. '`-mno-insn32`' turns off this option. This is equivalent to putting `.set noinsn32` at the start of the assembly file. By default '`-mno-insn32`' is selected, allowing all instructions to be used.

`--construct-floats`
`--no-construct-floats`
: The '`--no-construct-floats`' option disables the construction of double width floating point constants by loading the two halves of the value into the two single width floating point registers that make up the double width register. By default '`--construct-floats`' is selected, allowing construction of these floating point constants.

`--relax-branch`
`--no-relax-branch`
: The '`--relax-branch`' option enables the relaxation of out-of-range branches. By default '`--no-relax-branch`' is selected, causing any out-of-range branches to produce an error.

`-mignore-branch-isa`
`-mno-ignore-branch-isa`
: Ignore branch checks for invalid transitions between ISA modes. The semantics of branches does not provide for an ISA mode switch, so in most cases the ISA

mode a branch has been encoded for has to be the same as the ISA mode of the branch's target label. Therefore GAS has checks implemented that verify in branch assembly that the two ISA modes match. '`-mignore-branch-isa`' disables these checks. By default '`-mno-ignore-branch-isa`' is selected, causing any invalid branch requiring a transition between ISA modes to produce an error.

`-mnan=`*encoding*
: Select between the IEEE 754-2008 (`-mnan=2008`) or the legacy (`-mnan=legacy`) NaN encoding format. The latter is the default.

`--emulation=`*name*
: This option was formerly used to switch between ELF and ECOFF output on targets like IRIX 5 that supported both. MIPS ECOFF support was removed in GAS 2.24, so the option now serves little purpose. It is retained for backwards compatibility.

  The available configuration names are: '`mipself`', '`mipslelf`' and '`mipsbelf`'. Choosing '`mipself`' now has no effect, since the output is always ELF. '`mipslelf`' and '`mipsbelf`' select little- and big-endian output respectively, but '`-EL`' and '`-EB`' are now the preferred options instead.

`-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.

`-n`
: When this option is used, `as` will issue a warning every time it generates a nop instruction from a macro.

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.

See ⟨undefined⟩ [Meta Options], page ⟨undefined⟩, for the options available when as is configured for a Meta processor.

See the info pages for documentation of the MMIX-specific options.

See ⟨undefined⟩ [NDS32 Options], page ⟨undefined⟩, for the options available when as is configured for a NDS32 processor.

See ⟨undefined⟩ [PowerPC-Opts], page ⟨undefined⟩, for the options available when as is configured for a PowerPC processor.

See Section 9.3.1 [RISC-V-Options], page 117, for the options available when as is configured for a RISC-V processor.

See the info pages for documentation of the RX-specific options.

The following options are available when as is configured for the s390 processor family.

`-m31`
`-m64` Select the word size, either 31/32 bits or 64 bits.

`-mesa`

`-mzarch` Select the architecture mode, either the Enterprise System Architecture (esa) or the z/Architecture mode (zarch).

`-march=`*processor*
Specify which s390 processor variant is the target, '`g5`' (or '`arch3`'), '`g6`', '`z900`' (or '`arch5`'), '`z990`' (or '`arch6`'), '`z9-109`', '`z9-ec`' (or '`arch7`'), '`z10`' (or '`arch8`'), '`z196`' (or '`arch9`'), '`zEC12`' (or '`arch10`'), '`z13`' (or '`arch11`'), '`z14`' (or '`arch12`'), or '`z15`' (or '`arch13`').

`-mregnames`
`-mno-regnames`
Allow or disallow symbolic names for registers.

`-mwarn-areg-zero`
Warn whenever the operand for a base or index register has been specified but evaluates to zero.

See ⟨undefined⟩ [TIC6X Options], page ⟨undefined⟩, for the options available when as is configured for a TMS320C6000 processor.

See ⟨undefined⟩ [TILE-Gx Options], page ⟨undefined⟩, for the options available when as is configured for a TILE-Gx processor.

See ⟨undefined⟩ [Visium Options], page ⟨undefined⟩, for the options available when as is configured for a Visium processor.

See ⟨undefined⟩ [Xtensa Options], page ⟨undefined⟩, for the options available when as is configured for an Xtensa processor.

See ⟨undefined⟩ [Z80 Options], page ⟨undefined⟩, for the options available when as is configured for an Z80 processor.

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 9 [Machine Dependencies], page 93). 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 7.68 [`.org`], page 72).

1.3 Object File Formats

The GNU assembler can be configured to produce several alternative object file formats. For the most part, this does not affect how you write assembly language programs; but directives for debugging symbols are typically different in different file formats. See Section 5.5 [Symbol Attributes], page 45.

1.4 Command Line

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

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

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

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

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

1.5 Input Files

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

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

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

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

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

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

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

Filenames and Line-numbers

There are two ways of locating a line in the input file (or files) and either may be used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a "logical" file. See Section 1.7 [Error and Warning Messages], page 23.

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 7.35 [`.file`], page 62.

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`. 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 both a logical file name (see Section 7.35 [`.file`], page 62) and a logical line number (see Section 7.53 [`.line`], page 67) have been given then they will be used, otherwise the file name and line number in the current assembler source file will be used. The message text is intended to be self explanatory (in the grand Unix tradition).

Note the file name must be set via the logical version of the `.file` directive, not the DWARF2 version of the `.file` directive. For example:

```
.file 2 "bar.c"
   error_assembler_source
.file "foo.c"
.line 30
    error_c_source
```

produces this output:

```
Assembler messages:
asm.s:2: Error: no such instruction: ‘error_assembler_source’
foo.c:31: Error: no such instruction: ‘error_c_source’
```

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 9 [Machine Dependencies], page 93, for options specific to particular machine architectures.

If you are invoking `as` via the GNU C compiler, 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[cdghlns]`

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 '`-ag`' option to print a first section with general assembly information, like as version, switches passed, or time stamp.

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`'.

Note if the assembler source is coming from the standard input (e.g., because it is being created by `gcc` and the '`-pipe`' command-line switch is being used) then the listing will not contain any comments or preprocessor directives. This is because the listing code buffers input source lines from stdin only after they have been preprocessed by the assembler. This reduces memory usage and makes the code more efficient.

2.2 `--alternate`

Begin in alternate macro mode, see Section 7.4 [`.altmacro`], page 52.

2.3 `-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`.

2.4 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 31.

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

2.5 `.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 7.46 [`.include`], page 65). 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.

2.6 Difference Tables: `-K`

`as` sometimes alters the code emitted for directives of the form '`.word` *sym1-sym2*'. See Section 7.107 [`.word`], page 86. You can use the '`-K`' option if you want a warning issued when this is done.

2.7 Include Local Symbols: `-L`

Symbols beginning with system-specific local label prefixes, typically '`.L`' for ELF systems or '`L`' for traditional a.out systems, are called *local symbols*. See Section 5.3 [Symbol Names], page 43. Normally you do not see such symbols 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 symbols, so you do not normally debug with them.

This option tells `as` to retain those local symbols in the object file. Usually if you do this you also tell the linker `ld` to preserve those symbols.

2.8 Configuring listing output: `--listing`

The listing feature of the assembler can be enabled via the command-line switch '`-a`' (see Section 2.1 [a], page 25). This feature combines the input source file(s) with a hex dump of the corresponding locations in the output object file, and displays them as a listing file. The format of this listing can be controlled by directives inside the assembler source (i.e., `.list` (see Section 7.55 [List], page 67), `.title` (see Section 7.97 [Title], page 84), `.sbttl` (see Section 7.80 [Sbttl], page 76), `.psize` (see Section 7.74 [Psize], page 75), and `.eject` (see Section 7.20 [Eject], page 60) and also by the following switches:

`--listing-lhs-width='number'`
: Sets the maximum width, in words, of the first line of the hex byte dump. This dump appears on the left hand side of the listing output.

`--listing-lhs-width2='number'`
: Sets the maximum width, in words, of any further lines of the hex byte dump for a given input source line. If this value is not specified, it defaults to being the same as the value specified for '`--listing-lhs-width`'. If neither switch is used the default is to one.

`--listing-rhs-width='number'`
: Sets the maximum width, in characters, of the source line that is displayed alongside the hex dump. The default value for this parameter is 100. The source line is displayed on the right hand side of the listing output.

`--listing-cont-lines='number'`
: Sets the maximum number of continuation lines of hex dump that will be displayed for a given single line of source input. The default value is 4.

2.9 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` assembler from Microtec Research. The exact nature of the MRI syntax will not be documented here; see the MRI manuals for more information. Note in particular that the handling of macros and macro arguments is somewhat different. The purpose of this option is to permit assembling existing MRI assembler code using `as`.

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

- global symbols in common section

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

- complex relocations

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

- `END` pseudo-op specifying start address

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

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

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

- `ORG` pseudo-op

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

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

- EBCDIC strings

  EBCDIC strings are not supported.

- packed binary coded decimal

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

- `FEQU` pseudo-op

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

- `NOOBJ` pseudo-op

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

- `OPT` branch control options

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

- `OPT` list control options

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

- other `OPT` options

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

- `OPT D` option is default

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

- `XREF` pseudo-op.

  The m68k `XREF` pseudo-op is ignored.

2.10 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.

2.11 Output Section Padding

Normally the assembler will pad the end of each output section up to its alignment boundary. But this can waste space, which can be significant on memory constrained targets. So the `--no-pad-sections` option will disable this behaviour.

2.12 Name the Object File: `-o`

There is always one object file output when you run `as`. By default it has the name `a.out`. You use this option (which takes exactly one filename) to give the object file a different name.

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

2.13 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 37.)

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 or ELF 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`.

2.14 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).

2.15 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.

2.16 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.

2.17 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.

2.18 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 7.46 [`.include`], page 65). 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 33), 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 a *line comment* character up to the next newline is considered a comment and is ignored. The line comment character is target specific, and some targets multiple comment characters. Some targets also have line comment characters that only work if they are the first character on a line. Some targets use a sequence of two characters to introduce a line comment. Some targets can also change their line comment characters depending upon command-line options that have been used. For more details see the *Syntax* section in the documentation for individual targets.

If the line comment character is the hash sign (‘`#`’) then it still has the special ability to enable and disable preprocessing (see Section 3.1 [Preprocessing], page 31) and to specify logical line numbers:

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 47): the logical line number of the *next* line. Then a string (see Section 3.6.1.1 [Strings], page 33) 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 9 [Machine Dependencies], page 93. No symbol may begin with a digit. Case is significant. There is no length limit; all characters are significant. Multibyte characters are supported. 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 43.

Symbol names may also be enclosed in double quote `"` characters. In such cases any characters are allowed, except for the NUL character. If a double quote character is to be included in the symbol name it must be preceeded by a backslash `\` character.

3.5 Statements

A *statement* ends at a newline character (‘`\n`’) or a *line separator character*. The line separator character is target specific and described in the *Syntax* section of each target’s documentation. Not all targets support a line separator character. The newline or line separator character is considered to be 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 43.

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

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

3.6 Constants

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

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

3.6.1 Character Constants

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

3.6.1.1 Strings

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

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

`backslash-f`
Mnemonic for FormFeed; for ASCII this is octal code 014.

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

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

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

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

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

`\\` Represents one '\' character.

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

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

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

3.6.1.2 Characters

A single character may be written as a single quote immediately followed by that character. Some backslash escapes apply to characters, `\b`, `\f`, `\n`, `\r`, `\t`, and `\"` with the same meaning as for strings, plus `\'` for a single quote. 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 47).

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 and Renesas / SuperH SH 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 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.

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 or ELF output, `as` can also generate whatever other named sections you specify using the '`.section`' directive (see Section 7.82 [`.section`], page 76). If you do not use any directives that place output in the '`.text`' or '`.data`' sections, these sections still exist, but are empty.

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

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

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

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

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

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

  (*address*) − (*start-address of section*)?
- Is the reference to an address "Program-Counter relative"?

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

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

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

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

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

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

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

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

4.2 Linker Sections

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

named sections
text section
data section

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

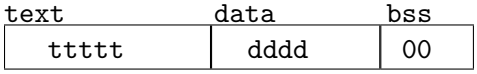
Address 0 of this section is always "relocated" to runtime address 0. This is useful if you want to refer to an address that `ld` must not change when

relocating. In this sense we speak of absolute addresses being "unrelocatable": they do not change during relocation.

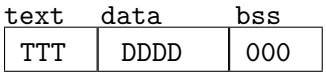
undefined section
This "section" is a catch-all for address references to objects not in the preceding sections.

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.

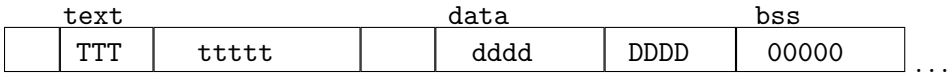
Partial program #1:



Partial program #2:



linked program:



...

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*'. When generating ELF output, you can also use the `.subsection` directive (see Section 7.93 [SubSection], page 83) to specify a subsection: '`.subsection` *expression*'. *Expression* should be an absolute expression (see Chapter 6 [Expressions], page 47). 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 7.51 [`.lcomm`], page 66.

The `.comm` pseudo-op may be used to declare a common symbol, which is another form of uninitialized symbol; see Section 7.11 [`.comm`], page 57.

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 7.82

[`.section`], page 76. You may only assemble zero values into the section. Typically the section will only contain symbol definitions and `.skip` directives (see Section 7.87 [`.skip`], page 81).

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 47). This is equivalent to using the `.set` directive. See Section 7.83 [`.set`], page 80. In the same way, using a double equals sign '`=`''`=`' here represents an equivalent of the `.eqv` directive. See Section 7.29 [`.eqv`], page 61.

Blackfin does not support symbol assignment with '`=`'.

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 9 [Machine Dependencies], page 93. That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted for a particular target machine), and underscores.

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

Symbol names do not start with a digit. An exception to this rule is made for Local Labels. See below.

Multibyte characters are supported. To generate a symbol name containing multibyte characters enclose it within double quotes and use escape codes. cf See Section 3.6.1.1 [Strings], page 33. Generating a multibyte symbol name from a label is not currently supported.

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

A local symbol is any symbol beginning with certain local label prefixes. By default, the local label prefix is '`.L`' for ELF systems or '`L`' for traditional a.out systems, but each target may have its own set of local label prefixes. On the HPPA local symbols begin with '`L$`'.

Local symbols are defined and used within the assembler, but they are normally not saved in object files. Thus, they are not visible when debugging. You may use the '`-L`' option (see Section 2.7 [Include Local Symbols], page 26) to retain the local symbols in the object files.

Local Labels

Local labels are different from local symbols. Local labels help compilers and programmers use names temporarily. They create symbols which are guaranteed to be unique over the entire scope of the input source code and which can be referred to by a simple notation. To define a local label, write a label of the form '**N:**' (where **N** represents any non-negative integer). To refer to the most recent previous definition of that label write '**N**`b`', using the same number as when you defined the label. To refer to the next definition of a local label, write '**N**`f`'. The '`b`' stands for "backwards" and the '`f`' stands for "forwards".

There is no restriction on how you can use these labels, and you can reuse them too. So that it is possible to repeatedly define the same local label (using the same number '**N**'), although you can only refer to the most recently defined local label of that number (for a backwards reference) or the next definition of a specific local label for a forward reference. It is also worth noting that the first 10 local labels ('**0:**'. . . '**9:**') are implemented in a slightly more efficient manner than the others.

Here is an example:

```
1:        branch 1f
2:        branch 1b
1:        branch 2f
2:        branch 1b
```

Which is the equivalent of:

```
label_1:  branch label_3
label_2:  branch label_1
label_3:  branch label_4
label_4:  branch label_3
```

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

`local label prefix`
: All local symbols begin with the system-specific local label prefix. Normally both `as` and `ld` forget symbols that start with the local label prefix. 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.

`number`
: This is the number that was used in the local label definition. So if the label is written '`55:`' then the number is '`55`'.

`C-B`
: This unusual character is included so you do not accidentally invent a symbol of the same name. The character has ASCII value of '`\002`' (control-B).

`ordinal number`
: This is a serial number to keep the labels distinct. The first definition of '`0:`' gets the number '`1`'. The 15th definition of '`0:`' gets the number '`15`', and so on.

Likewise the first definition of '`1:`' gets the number '`1`' and its 15th definition gets '`15`' as well.

So for example, the first `1:` may be named `.L1C-B1`, and the 44th `3:` may be named `.L3C-B44`.

Dollar Local Labels

On some targets `as` also supports an even more local form of local labels called dollar labels. These labels go out of scope (i.e., they become undefined) as soon as a non-local label is defined. Thus they remain valid for only a small region of the input source code. Normal local labels, by contrast, remain in scope for the entire file, or until they are redefined by another occurrence of the same local label.

Dollar labels are defined in exactly the same way as ordinary local labels, except that they have a dollar sign suffix to their numeric value, e.g., '**`55$:`**'.

They can also be distinguished from ordinary local labels by their transformed names which use ASCII character '`\001`' (control-A) as the magic character to distinguish them from ordinary labels. For example, the fifth definition of '`6$`' may be named '`.L6C-A5`'.

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 7.17 [`.desc`], page 59). 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`, `.tag`, and `.weak` 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. Low 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.

   `==` *Is Equal To*

   `<>`
   `!=` *Is Not Equal To*

   `<` *Is Less Than*

   `>` *Is Greater Than*

   `>=` *Is Greater Than Or Equal To*

   `<=` *Is Less Than Or Equal To*

   The comparison operators can be used as infix operators. A true results has a value of -1 whereas a false result has a value of 0. Note, these operators perform signed comparisons.

4. Lowest Precedence

   `&&` *Logical And.*

| | *Logical Or.*

These two logical operations can be used to combine the results of sub expressions. Note, unlike the comparison operators a true result returns a value of 1 but a false results does still return 0. Also note that the logical or operator has a slightly lower precedence than logical and.

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 names are case insensitive for most targets, and usually written 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 9 [Machine Dependencies], page 93.

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 (COFF)

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

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. If this expression is omitted then a default value of 0 is used, effectively disabling alignment requirements.

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 most 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 arc, hppa, i386 using ELF, iq2000, m68k, or1k, s390, sparc, tic4x and xtensa, 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 the tic54x, the first expression is the alignment request in words.

For other systems, including ppc, i386 using a.out format, 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 is 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 .altmacro

Enable alternate macro mode, enabling:

`LOCAL` *name* `[ , ... ]`
: One additional directive, `LOCAL`, is available. It is used to generate a string replacement for each of the *name* arguments, and replace any instances of *name* in each macro expansion. The replacement string is unique in the assembly, and different for each separate macro expansion. `LOCAL` allows you to write macros that define symbols, without fear of conflict between separate macro expansions.

`String delimiters`
: You can write strings delimited in these other ways besides `"`*`string`*`"`:

  `'`*`string`*`'` You can delimit strings with single-quote characters.

  `<`*`string`*`>` You can delimit strings with matching angle brackets.

`single-character string escape`
: To include any single character literally in a string (even if the character would otherwise have some special meaning), you can prefix the character with '`!`' (an exclamation mark). For example, you can write '`<4.3 !> 5.4!!>`' to get the literal text '`4.3 > 5.4!`'.

`Expression results as strings`
: You can write '`%`*`expr`*' to evaluate the expression *expr* and use the result as a string.

7.5 .ascii "*string*"...

`.ascii` expects zero or more string literals (see Section 3.6.1.1 [Strings], page 33) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

7.6 .asciz "*string*"...

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

7.7 .balign[wl] [*abs-expr*[, *abs-expr*[, *abs-expr*]]]

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment request in bytes. For example '`.balign 8`' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed. If the expression is omitted then a default value of 0 is used, effectively disabling alignment requirements.

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 most systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

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

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

7.8 Bundle directives

7.8.1 `.bundle_align_mode` *abs-expr*

`.bundle_align_mode` enables or disables *aligned instruction bundle* mode. In this mode, sequences of adjacent instructions are grouped into fixed-sized *bundles*. If the argument is zero, this mode is disabled (which is the default state). If the argument it not zero, it gives the size of an instruction bundle as a power of two (as for the `.p2align` directive, see Section 7.69 [P2align], page 73).

For some targets, it's an ABI requirement that no instruction may span a certain aligned boundary. A *bundle* is simply a sequence of instructions that starts on an aligned boundary. For example, if *abs-expr* is `5` then the bundle size is 32, so each aligned chunk of 32 bytes is a bundle. When aligned instruction bundle mode is in effect, no single instruction may span a boundary between bundles. If an instruction would start too close to the end of a bundle for the length of that particular instruction to fit within the bundle, then the space at the end of that bundle is filled with no-op instructions so the instruction starts in the next bundle. As a corollary, it's an error if any single instruction's encoding is longer than the bundle size.

7.8.2 `.bundle_lock` and `.bundle_unlock`

The `.bundle_lock` and directive `.bundle_unlock` directives allow explicit control over instruction bundle padding. These directives are only valid when `.bundle_align_mode` has been used to enable aligned instruction bundle mode. It's an error if they appear when `.bundle_align_mode` has not been used at all, or when the last directive was `.bundle_align_mode 0`.

For some targets, it's an ABI requirement that certain instructions may appear only as part of specified permissible sequences of multiple instructions, all within the same bundle. A pair of `.bundle_lock` and `.bundle_unlock` directives define a *bundle-locked* instruction sequence. For purposes of aligned instruction bundle mode, a sequence starting with `.bundle_lock` and ending with `.bundle_unlock` is treated as a single instruction. That is, the entire sequence must fit into a single bundle and may not span a bundle boundary. If

necessary, no-op instructions will be inserted before the first instruction of the sequence so that the whole sequence starts on an aligned bundle boundary. It's an error if the sequence is longer than the bundle size.

For convenience when using `.bundle_lock` and `.bundle_unlock` inside assembler macros (see Section 7.61 [Macro], page 69), bundle-locked sequences may be nested. That is, a second `.bundle_lock` directive before the next `.bundle_unlock` directive has no effect except that it must be matched by another closing `.bundle_unlock` so that there is the same number of `.bundle_lock` and `.bundle_unlock` directives.

7.9 `.byte` *expressions*

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

7.10 CFI directives

7.10.1 `.cfi_sections` *section_list*

`.cfi_sections` may be used to specify whether CFI directives should emit `.eh_frame` section and/or `.debug_frame` section. If *section_list* is `.eh_frame`, `.eh_frame` is emitted, if *section_list* is `.debug_frame`, `.debug_frame` is emitted. To emit both use `.eh_frame, .debug_frame`. The default if this directive is not used is `.cfi_sections .eh_frame`.

On targets that support compact unwinding tables these can be generated by specifying `.eh_frame_entry` instead of `.eh_frame`.

Some targets may support an additional name, such as `.c6xabi.exidx` which is used by the target.

The `.cfi_sections` directive can be repeated, with the same or different arguments, provided that CFI generation has not yet started. Once CFI generation has started however the section list is fixed and any attempts to redefine it will result in an error.

7.10.2 `.cfi_startproc [simple]`

`.cfi_startproc` is used at the beginning of each function that should have an entry in `.eh_frame`. It initializes some internal data structures. Don't forget to close the function by `.cfi_endproc`.

Unless `.cfi_startproc` is used along with parameter `simple` it also emits some architecture dependent initial CFI instructions.

7.10.3 `.cfi_endproc`

`.cfi_endproc` is used at the end of a function where it closes its unwind entry previously opened by `.cfi_startproc`, and emits it to `.eh_frame`.

7.10.4 `.cfi_personality` *encoding* `[,` *exp*`]`

`.cfi_personality` defines personality routine and its encoding. *encoding* must be a constant determining how the personality should be encoded. If it is 255 (`DW_EH_PE_omit`), second argument is not present, otherwise second argument should be a constant or a symbol name. When using indirect encodings, the symbol provided should be the location

where personality can be loaded from, not the personality routine itself. The default after `.cfi_startproc` is `.cfi_personality 0xff`, no personality routine.

7.10.5 `.cfi_personality_id` *id*

`cfi_personality_id` defines a personality routine by its index as defined in a compact unwinding format. Only valid when generating compact EH frames (i.e. with `.cfi_sections eh_frame_entry`.

7.10.6 `.cfi_fde_data [`*opcode1* `[, ...]]`

`cfi_fde_data` is used to describe the compact unwind opcodes to be used for the current function. These are emitted inline in the `.eh_frame_entry` section if small enough and there is no LSDA, or in the `.gnu.extab` section otherwise. Only valid when generating compact EH frames (i.e. with `.cfi_sections eh_frame_entry`.

7.10.7 `.cfi_lsda` *encoding* `[,` *exp*`]`

`.cfi_lsda` defines LSDA and its encoding. *encoding* must be a constant determining how the LSDA should be encoded. If it is 255 (`DW_EH_PE_omit`), the second argument is not present, otherwise the second argument should be a constant or a symbol name. The default after `.cfi_startproc` is `.cfi_lsda 0xff`, meaning that no LSDA is present.

7.10.8 `.cfi_inline_lsda [`*align*`]`

`.cfi_inline_lsda` marks the start of a LSDA data section and switches to the corresponding `.gnu.extab` section. Must be preceded by a CFI block containing a `.cfi_lsda` directive. Only valid when generating compact EH frames (i.e. with `.cfi_sections eh_frame_entry`.

The table header and unwinding opcodes will be generated at this point, so that they are immediately followed by the LSDA data. The symbol referenced by the `.cfi_lsda` directive should still be defined in case a fallback FDE based encoding is used. The LSDA data is terminated by a section directive.

The optional *align* argument specifies the alignment required. The alignment is specified as a power of two, as with the `.p2align` directive.

7.10.9 `.cfi_def_cfa` *register*`,` *offset*

`.cfi_def_cfa` defines a rule for computing CFA as: *take address from register and add offset to it.*

7.10.10 `.cfi_def_cfa_register` *register*

`.cfi_def_cfa_register` modifies a rule for computing CFA. From now on *register* will be used instead of the old one. Offset remains the same.

7.10.11 `.cfi_def_cfa_offset` *offset*

`.cfi_def_cfa_offset` modifies a rule for computing CFA. Register remains the same, but *offset* is new. Note that it is the absolute offset that will be added to a defined register to compute CFA address.

7.10.12 `.cfi_adjust_cfa_offset` *offset*

Same as `.cfi_def_cfa_offset` but *offset* is a relative value that is added/subtracted from the previous offset.

7.10.13 `.cfi_offset` *register*, *offset*

Previous value of *register* is saved at offset *offset* from CFA.

7.10.14 `.cfi_val_offset` *register*, *offset*

Previous value of *register* is CFA + *offset*.

7.10.15 `.cfi_rel_offset` *register*, *offset*

Previous value of *register* is saved at offset *offset* from the current CFA register. This is transformed to `.cfi_offset` using the known displacement of the CFA register from the CFA. This is often easier to use, because the number will match the code it's annotating.

7.10.16 `.cfi_register` *register1*, *register2*

Previous value of *register1* is saved in register *register2*.

7.10.17 `.cfi_restore` *register*

`.cfi_restore` says that the rule for *register* is now the same as it was at the beginning of the function, after all initial instruction added by `.cfi_startproc` were executed.

7.10.18 `.cfi_undefined` *register*

From now on the previous value of *register* can't be restored anymore.

7.10.19 `.cfi_same_value` *register*

Current value of *register* is the same like in the previous frame, i.e. no restoration needed.

7.10.20 `.cfi_remember_state` and `.cfi_restore_state`

`.cfi_remember_state` pushes the set of rules for every register onto an implicit stack, while `.cfi_restore_state` pops them off the stack and places them in the current row. This is useful for situations where you have multiple `.cfi_*` directives that need to be undone due to the control flow of the program. For example, we could have something like this (assuming the CFA is the value of `rbp`):

```
        je label
        popq %rbx
        .cfi_restore %rbx
        popq %r12
        .cfi_restore %r12
        popq %rbp
        .cfi_restore %rbp
        .cfi_def_cfa %rsp, 8
        ret
label:
        /* Do something else */
```

Here, we want the `.cfi` directives to affect only the rows corresponding to the instructions before `label`. This means we'd have to add multiple `.cfi` directives after `label` to

recreate the original save locations of the registers, as well as setting the CFA back to the value of `rbp`. This would be clumsy, and result in a larger binary size. Instead, we can write:

```
            je label
            popq %rbx
            .cfi_remember_state
            .cfi_restore %rbx
            popq %r12
            .cfi_restore %r12
            popq %rbp
            .cfi_restore %rbp
            .cfi_def_cfa %rsp, 8
            ret
    label:
            .cfi_restore_state
            /* Do something else */
```

That way, the rules for the instructions after `label` will be the same as before the first `.cfi_restore` without having to use multiple `.cfi` directives.

7.10.21 `.cfi_return_column` *register*

Change return column *register*, i.e. the return address is either directly in *register* or can be accessed by rules for *register*.

7.10.22 `.cfi_signal_frame`

Mark current function as signal trampoline.

7.10.23 `.cfi_window_save`

SPARC register window has been saved.

7.10.24 `.cfi_escape` ***expression***[, ...]

Allows the user to add arbitrary bytes to the unwind info. One might use this to add OS-specific CFI opcodes, or generic CFI opcodes that GAS does not yet support.

7.10.25 `.cfi_val_encoded_addr` *register*, *encoding*, *label*

The current value of *register* is *label*. The value of *label* will be encoded in the output file according to *encoding*; see the description of `.cfi_personality` for details on this encoding.

The usefulness of equating a register to a fixed label is probably limited to the return address register. Here, it can be useful to mark a code segment that has only one return address which is reached by a direct branch and no copy of the return address exists in memory or another register.

7.11 `.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 or (as a GNU extension) PE, the `.comm` directive takes an optional third argument. This is the desired alignment of the symbol, specified for ELF as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address should be zero), and for PE as a power of two (for example, an alignment of 5 means aligned to a 32-byte boundary). 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 on ELF, or the default section alignment of 4 on PE[1].

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

7.12 `.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.

7.13 `.dc[`*size*`]` *expressions*

The `.dc` directive expects zero or more *expressions* separated by commas. These expressions are evaluated and their values inserted into the current section. The size of the emitted value depends upon the suffix to the `.dc` directive:

'`.a`' Emits N-bit values, where N is the size of an address on the target system.

'`.b`' Emits 8-bit values.

'`.d`' Emits double precision floating-point values.

'`.l`' Emits 32-bit values.

'`.s`' Emits single precision floating-point values.

'`.w`' Emits 16-bit values. Note - this is true even on targets where the `.word` directive would emit 32-bit values.

'`.x`' Emits long double precision floating-point values.

If no suffix is used then '`.w`' is assumed.

The byte ordering is target dependent, as is the size and format of floating point values.

[1] This is not the same as the executable image file alignment controlled by `ld`'s '`--section-alignment`' option; image file sections in PE are aligned to multiples of 4096, which is far too large an alignment for ordinary variables. It is rather the default alignment for (non-debug) sections within object ('`*.o`') files, which are less strictly aligned.

7.14 .dcb[*size*] *number* [,*fill*]

This directive emits *number* copies of *fill*, each of *size* bytes. Both *number* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero. The *size* suffix, if present, must be one of:

‘.b’ Emits single byte values.

‘.d’ Emits double-precision floating point values.

‘.l’ Emits 4-byte values.

‘.s’ Emits single-precision floating point values.

‘.w’ Emits 2-byte values.

‘.x’ Emits long double-precision floating point values.

If the *size* suffix is omitted then ‘.w’ is assumed.

The byte ordering is target dependent, as is the size and format of floating point values.

7.15 .ds[*size*] *number* [,*fill*]

This directive emits *number* copies of *fill*, each of *size* bytes. Both *number* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero. The *size* suffix, if present, must be one of:

‘.b’ Emits single byte values.

‘.d’ Emits 8-byte values.

‘.l’ Emits 4-byte values.

‘.p’ Emits 12-byte values.

‘.s’ Emits 4-byte values.

‘.w’ Emits 2-byte values.

‘.x’ Emits 12-byte values.

Note - unlike the .dcb directive the ‘.d’, ‘.s’ and ‘.x’ suffixes do not indicate that floating-point values are to be inserted.

If the *size* suffix is omitted then ‘.w’ is assumed.

The byte ordering is target dependent.

7.16 .def *name*

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

7.17 .desc *symbol*, *abs-expression*

This directive sets the descriptor of the symbol (see Section 5.5 [Symbol Attributes], page 45) 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.

7.18 .dim

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

7.19 .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 9 [Machine Dependencies], page 93.

7.20 .eject

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

7.21 .else

`.else` is part of the `as` support for conditional assembly; see Section 7.44 [`.if`], page 64. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

7.22 .elseif

`.elseif` is part of the `as` support for conditional assembly; see Section 7.44 [`.if`], page 64. It is shorthand for beginning a new `.if` block that would otherwise fill the entire `.else` section.

7.23 .end

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

7.24 .endef

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

7.25 .endfunc

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

7.26 .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 7.44 [`.if`], page 64.

7.27 .equ *symbol, expression*

This directive sets the value of *symbol* to *expression*. It is synonymous with '`.set`'; see Section 7.83 [`.set`], page 80.

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

The syntax for `equ` on the Z80 is '*`symbol`* `equ` *`expression`*'. On the Z80 it is an error if *symbol* is already defined, but the symbol is not protected from later redefinition. Compare Section 7.28 [Equiv], page 61.

7.28 `.equiv` *`symbol, expression`*

The `.equiv` directive is like `.equ` and `.set`, except that the assembler will signal an error if *symbol* is already defined. Note a symbol which has been referenced but not actually defined is considered to be undefined.

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

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

plus it protects the symbol from later redefinition.

7.29 `.eqv` *`symbol, expression`*

The `.eqv` directive is like `.equiv`, but no attempt is made to evaluate the expression or any part of it immediately. Instead each time the resulting symbol is used in an expression, a snapshot of its current value is taken.

7.30 `.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 an error in conditionally compiled code.

7.31 `.error "`*`string`*`"`

Similarly to `.err`, this directive emits an error, but you can specify a string that will be emitted as the error message. If you don't specify the message, it defaults to `".error directive invoked in source file"`. See Section 1.7 [Error and Warning Messages], page 23.

```
.error "This code has not been assembled and tested."
```

7.32 `.exitm`

Exit early from the current macro definition. See Section 7.61 [Macro], page 69.

7.33 `.extern`

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

7.34 `.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.

7.35 .file

There are two different versions of the `.file` directive. Targets that support DWARF2 line number information use the DWARF2 version of `.file`. Other targets use the default version.

Default Version

This version of the `.file` directive tells `as` that we are about to start a new logical file. The syntax is:

```
.file string
```

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.

DWARF2 Version

When emitting DWARF2 line number information, `.file` assigns filenames to the `.debug_line` file name table. The syntax is:

```
.file fileno filename
```

The *fileno* operand should be a unique positive integer to use as the index of the entry in the table. The *filename* operand is a C string literal.

The detail of filename indices is exposed to the user because the filename table is shared with the `.debug_info` section of the DWARF2 debugging information, and thus the user must know the exact indices that table entries will have.

7.36 .fill *repeat , size , value*

repeat, *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.

7.37 .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 9 [Machine Dependencies], page 93.

7.38 .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`.

7.39 .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 ⟨undefined⟩ [HPPA Assembler Directives], page ⟨undefined⟩.

7.40 .gnu_attribute *tag*,*value*

Record a GNU object attribute for this file. See Chapter 8 [Object Attributes], page 89.

7.41 .hidden *names*

This is one of the ELF visibility directives. The other two are `.internal` (see Section 7.48 [`.internal`], page 65) and `.protected` (see Section 7.73 [`.protected`], page 74).

This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to `hidden` which means that the symbols are not visible to other components. Such symbols are always considered to be `protected` as well.

7.42 .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`'.

7.43 .ident

This directive is used by some assemblers to place tags in object files. The behavior of this directive varies depending on the target. When using the a.out object file format, `as` simply accepts the directive for source-file compatibility with existing assemblers, but does not emit anything for it. When using COFF, comments are emitted to the `.comment` or `.rdata` section, depending on the target. When using ELF, comments are emitted to the `.comment` section.

7.44 `.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 7.26 [`.endif`], page 60); optionally, you may include code for the alternative condition, flagged by `.else` (see Section 7.21 [`.else`], page 60). 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. Note a symbol which has been referenced but not yet defined is considered to be undefined.

`.ifb` *text* Assembles the following section of code if the operand is blank (empty).

`.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.

`.ifnb` *text*
: Like `.ifb`, but the sense of the test is reversed: this assembles the following section of code if the operand is non-blank (non-empty).

`.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. Note a symbol which has been referenced but not yet defined is considered to be undefined.

`.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.

7.45 `.incbin "`*file*`"[,`*skip*`[,`*count*`]]`

The `incbin` directive includes *file* verbatim at the current location. You can control the search paths used with the '`-I`' command-line option (see Chapter 2 [Command-Line Options], page 25). Quotation marks are required around *file*.

The *skip* argument skips a number of bytes from the start of the *file*. The *count* argument indicates the maximum number of bytes to read. Note that the data is not aligned in any way, so it is the user's responsibility to make sure that proper alignment is provided both before and after the `incbin` directive.

7.46 `.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 25). Quotation marks are required around *file*.

7.47 `.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.

7.48 `.internal` *names*

This is one of the ELF visibility directives. The other two are `.hidden` (see Section 7.41 [`.hidden`], page 63) and `.protected` (see Section 7.73 [`.protected`], page 74).

This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to `internal` which means that the symbols are considered to be `hidden` (i.e., not visible to other components), and that some extra, processor specific processing must also be performed upon the symbols as well.

7.49 .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@-
```

For some caveats with the spelling of *symbol*, see also Section 7.61 [Macro], page 69.

7.50 .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@-
```

For some caveats with the spelling of *symbol*, see also the discussion at See Section 7.61 [Macro], page 69.

7.51 .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 7.39 [`.global`], page 63), 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.

7.52 .lflags

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

7.53 .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.

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.

7.54 .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.

7.55 .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 25), the initial value of the listing counter is one.

7.56 `.ln` *line-number*

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

7.57 `.loc` *fileno lineno* [*column*] [*options*]

When emitting DWARF2 line number information, the `.loc` directive will add a row to the `.debug_line` line number matrix corresponding to the immediately following assembly instruction. The *fileno*, *lineno*, and optional *column* arguments will be applied to the `.debug_line` state machine before the row is added.

The *options* are a sequence of the following tokens in any order:

`basic_block`
: This option will set the `basic_block` register in the `.debug_line` state machine to `true`.

`prologue_end`
: This option will set the `prologue_end` register in the `.debug_line` state machine to `true`.

`epilogue_begin`
: This option will set the `epilogue_begin` register in the `.debug_line` state machine to `true`.

`is_stmt` *value*
: This option will set the `is_stmt` register in the `.debug_line` state machine to `value`, which must be either 0 or 1.

`isa` *value* This directive will set the `isa` register in the `.debug_line` state machine to *value*, which must be an unsigned integer.

`discriminator` *value*
: This directive will set the `discriminator` register in the `.debug_line` state machine to *value*, which must be an unsigned integer.

`view` *value*
: This option causes a row to be added to `.debug_line` in reference to the current address (which might not be the same as that of the following assembly instruction), and to associate *value* with the `view` register in the `.debug_line` state machine. If *value* is a label, both the `view` register and the label are set to the number of prior `.loc` directives at the same program location. If *value* is the literal `0`, the `view` register is set to zero, and the assembler asserts that there aren't any prior `.loc` directives at the same program location. If *value* is the literal `-0`, the assembler arrange for the `view` register to be reset in this row, even if there are prior `.loc` directives at the same program location.

7.58 .loc_mark_labels *enable*

When emitting DWARF2 line number information, the `.loc_mark_labels` directive makes the assembler emit an entry to the `.debug_line` line number matrix with the `basic_block` register in the state machine set whenever a code label is seen. The *enable* argument should be either 1 or 0, to enable or disable this function respectively.

7.59 .local *names*

This directive, which is available for ELF targets, marks each symbol in the comma-separated list of `names` as a local symbol so that it will not be externally visible. If the symbols do not already exist, they will be created.

For targets where the `.lcomm` directive (see Section 7.51 [Lcomm], page 66) does not accept an alignment argument, which is the case for most ELF targets, the `.local` directive can be used in combination with `.comm` (see Section 7.11 [Comm], page 57) to define aligned local common data.

7.60 .long *expressions*

`.long` is the same as '`.int`'. See Section 7.47 [`.int`], page 65.

7.61 .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 qualify the macro argument to indicate whether all invocations must specify a non-blank value (through '`:req`'), or whether it takes all of the remaining arguments (through '`:vararg`'). You can supply a default value for any macro argument by following the name with '`=deflt`'. You

cannot define two macros with the same *macname* unless it has been subject to the `.purgem` directive (see Section 7.75 [Purgem], page 75) between the two definitions. 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*).

`.macro m p1:req, p2=0, p3:vararg`
: Begin the definition of a macro called `m`, with at least three arguments. The first argument must always have a value specified, but not the second, which instead has a default value. The third formal will get assigned all remaining arguments specified at invocation time.

  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`'.

Note that since each of the *macargs* can be an identifier exactly as any other one permitted by the target architecture, there may be occasional problems if the target hand-crafts special meanings to certain characters when they occur in a special position. For example, if the colon (`:`) is generally permitted to be part of a symbol name, but the architecture specific code special-cases it when occurring as the final character of a symbol (to denote a label), then the macro parameter replacement code will have no way of knowing that and consider the whole construct (including the colon) an identifier, and check only this identifier for being the subject to parameter substitution. So for example this macro definition:

```
.macro label l
\l:
.endm
```

might not work as expected. Invoking '`label foo`' might not create a label called '`foo`' but instead just insert the text '`\l:`' into the assembler source, probably generating an error about an unrecognised identifier.

Similarly problems might occur with the period character (‘`.`’) which is often allowed inside opcode names (and hence identifier names). So for example constructing a macro to build an opcode from a base name and a length specifier like this:

```
.macro opcode base length
        \base.\length
.endm
```

and invoking it as ‘`opcode store l`’ will not create a ‘`store.l`’ instruction but instead generate some kind of error as the assembler tries to interpret the text ‘`\base.\length`’.

There are several possible ways around this problem:

`Insert white space`
: If it is possible to use white space characters then this is the simplest solution. eg:

```
.macro label l
\l :
.endm
```

`Use '\()'`
: The string ‘`\()`’ can be used to separate the end of a macro argument from the following text. eg:

```
.macro opcode base length
        \base\().\length
.endm
```

`Use the alternate macro syntax mode`
: In the alternative macro syntax mode the ampersand character (‘`&`’) can be used as a separator. eg:

```
.altmacro
.macro label l
l&:
.endm
```

Note: this problem of correctly identifying string parameters to pseudo ops also applies to the identifiers used in `.irp` (see Section 7.49 [Irp], page 66) and `.irpc` (see Section 7.50 [Irpc], page 66) as well.

`.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.*

`LOCAL` *name* `[ , ... ]`
: *Warning:* `LOCAL` *is only available if you select "alternate macro syntax" with* ‘`--alternate`’ *or* `.altmacro`. See Section 7.4 [`.altmacro`], page 52.

7.62 .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 2.9 [MRI mode], page 27.

7.63 .noaltmacro

Disable alternate macro mode. See Section 7.4 [Altmacro], page 52.

7.64 .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.

7.65 .nops *size*[, *control*]

This directive emits *size* bytes filled with no-op instructions. *size* is absolute expression, which must be a positve value. *control* controls how no-op instructions should be generated. If the comma and *control* are omitted, *control* is assumed to be zero.

Note: For Intel 80386 and AMD x86-64 targets, *control* specifies the size limit of a no-op instruction. The valid values of *control* are between 0 and 4 in 16-bit mode, between 0 and 7 when tuning for older processors in 32-bit mode, between 0 and 11 in 64-bit mode or when tuning for newer processors in 32-bit mode. When 0 is used, the no-op instruction size limit is set to the maximum supported size.

7.66 .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.

7.67 .offset *loc*

Set the location counter to *loc* in the absolute section. *loc* must be an absolute expression. This directive may be useful for defining symbols with absolute values. Do not confuse it with the `.org` directive.

7.68 .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.

7.69 .p2align[wl] [*abs-expr*[, *abs-expr*[, *abs-expr*]]]

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the number of low-order zero bits the location counter must have after advancement. For example '`.p2align 3`' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed. If the expression is omitted then a default value of 0 is used, effectively disabling alignment requirements.

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 most systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

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

The `.p2alignw` and `.p2alignl` directives are variants of the `.p2align` directive. The `.p2alignw` directive treats the fill pattern as a two byte word value. The `.p2alignl` directives treats the fill pattern as a four byte longword value. For example, `.p2alignw 2,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

7.70 .popsection

This is one of the ELF section stack manipulation directives. The others are `.section` (see Section 7.82 [Section], page 76), `.subsection` (see Section 7.93 [SubSection], page 83), `.pushsection` (see Section 7.76 [PushSection], page 75), and `.previous` (see Section 7.71 [Previous], page 74).

This directive replaces the current section (and subsection) with the top section (and subsection) on the section stack. This section is popped off the stack.

7.71 .previous

This is one of the ELF section stack manipulation directives. The others are `.section` (see Section 7.82 [Section], page 76), `.subsection` (see Section 7.93 [SubSection], page 83), `.pushsection` (see Section 7.76 [PushSection], page 75), and `.popsection` (see Section 7.70 [PopSection], page 73).

This directive swaps the current section (and subsection) with most recently referenced section/subsection pair prior to this one. Multiple `.previous` directives in a row will flip between two sections (and their subsections). For example:

```
.section A
 .subsection 1
  .word 0x1234
 .subsection 2
  .word 0x5678
.previous
 .word 0x9abc
```

Will place 0x1234 and 0x9abc into subsection 1 and 0x5678 into subsection 2 of section A. Whilst:

```
.section A
.subsection 1
  # Now in section A subsection 1
  .word 0x1234
.section B
.subsection 0
  # Now in section B subsection 0
  .word 0x5678
.subsection 1
  # Now in section B subsection 1
  .word 0x9abc
.previous
  # Now in section B subsection 0
  .word 0xdef0
```

Will place 0x1234 into section A, 0x5678 and 0xdef0 into subsection 0 of section B and 0x9abc into subsection 1 of section B.

In terms of the section stack, this directive swaps the current section with the top section on the section stack.

7.72 .print *string*

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

7.73 .protected *names*

This is one of the ELF visibility directives. The other two are `.hidden` (see Section 7.41 [Hidden], page 63) and `.internal` (see Section 7.48 [Internal], page 65).

This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to `protected` which means that any references to the symbols from within the components that defines them must be resolved to the definition in that component, even if a definition in another component would normally preempt this.

7.74 .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`.

7.75 .purgem *name*

Undefine the macro *name*, so that later uses of the string will not be expanded. See Section 7.61 [Macro], page 69.

7.76 .pushsection *name* [, *subsection*] [, "*flags*"[, @*type*[,*arguments*]]]

This is one of the ELF section stack manipulation directives. The others are `.section` (see Section 7.82 [Section], page 76), `.subsection` (see Section 7.93 [SubSection], page 83), `.popsection` (see Section 7.70 [PopSection], page 73), and `.previous` (see Section 7.71 [Previous], page 74).

This directive pushes the current section (and subsection) onto the top of the section stack, and then replaces the current section and subsection with `name` and `subsection`. The optional `flags`, `type` and `arguments` are treated the same as in the `.section` (see Section 7.82 [Section], page 76) directive.

7.77 .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.

7.78 .reloc *offset*, *reloc_name*[, *expression*]

Generate a relocation at *offset* of type *reloc_name* with value *expression*. If *offset* is a number, the relocation is generated in the current section. If *offset* is an expression that resolves to a symbol plus offset, the relocation is generated in the given symbol's section. *expression*, if present, must resolve to a symbol plus addend or to an absolute value, but note that not all targets support an addend. e.g. ELF REL targets such as i386 store an addend in the section contents rather than in the relocation. This low level interface does not support addends stored in the section.

7.79 .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
```

A count of zero is allowed, but nothing is generated. Negative counts are not allowed and if encountered will be treated as if they were zero.

7.80 .sbttl "*subheading*"

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

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

7.81 .scl *class*

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

7.82 .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.

COFF Version

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

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

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

`b` bss section (uninitialized data)

`n` section is not loaded

`w` writable section

`d` data section

`e`	exclude section from linking
`r`	read-only section
`x`	executable section
`s`	shared section (meaningful for PE targets)
`a`	ignored. (For compatibility with the ELF version)
`y`	section is not readable (meaningful for PE targets)
`0-9`	single-digit power-of-two section alignment (GNU extension)

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. Note the `n` and `w` flags remove attributes from the section, rather than adding them, so if they are used on their own it will be as if no flags had been specified at all.

If the optional argument to the `.section` directive is not quoted, it is taken as a subsection number (see Section 4.4 [Sub-Sections], page 39).

ELF Version

This is one of the ELF section stack manipulation directives. The others are `.subsection` (see Section 7.93 [SubSection], page 83), `.pushsection` (see Section 7.76 [PushSection], page 75), `.popsection` (see Section 7.70 [PopSection], page 73), and `.previous` (see Section 7.71 [Previous], page 74).

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

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

If the '`--sectname-subst`' command-line option is provided, the *name* argument may contain a substitution sequence. Only `%S` is supported at the moment, and substitutes the current section name. For example:

```
.macro exception_code
.section %S.exception
[exception code here]
.previous
.endm

.text
[code]
exception_code
[...]

.section .init
[init code]
exception_code
[...]
```

The two `exception_code` invocations above would create the `.text.exception` and `.init.exception` sections respectively. This is useful e.g. to discriminate between ancillary sections that are tied to setup code to be discarded after use from ancillary sections that need to stay resident without having to define multiple `exception_code` macros just for that purpose.

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

`a` section is allocatable

`d` section is a GNU_MBIND section

`e` section is excluded from executable and shared library.

`w` section is writable

`x` section is executable

`M` section is mergeable

`S` section contains zero terminated strings

`G` section is a member of a section group

`T` section is used for thread-local-storage

`?` section is a member of the previously-current section's group, if any

`<number>` a numeric value indicating the bits to be set in the ELF section header's flags field. Note - if one or more of the alphabetic characters described above is also included in the flags field, their bit values will be ORed into the resulting value.

`<target specific>`
some targets extend this list with their own flag characters

Note - once a section's flags have been set they cannot be changed. There are a few exceptions to this rule however. Processor and application specific flags can be added to an already defined section. The `.interp`, `.strtab` and `.symtab` sections can have the allocate flag (`a`) set after they are initially defined, and the `.note-GNU-stack` section may have the executable (`x`) flag added.

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)

`@note` section contains data which is used by things other than the program

`@init_array`
section contains an array of pointers to init functions

`@fini_array`
section contains an array of pointers to finish functions

`@preinit_array`
section contains an array of pointers to pre-init functions

`@<number>`
a numeric value to be set as the ELF section header's type field.

`@<target specific>`
some targets extend this list with their own types

Many targets only support the first three section types. The type may be enclosed in double quotes if necessary.

Note on targets where the `@` character is the start of a comment (eg ARM) then another character is used instead. For example the ARM port uses the `%` character.

Note - some sections, eg `.text` and `.data` are considered to be special and have fixed types. Any attempt to declare them with a different type will generate an error from the assembler.

If *flags* contains the `M` symbol then the *type* argument must be specified as well as an extra argument—*entsize*—like this:

```
.section name , "flags"M, @type, entsize
```

Sections with the `M` flag but not `S` flag must contain fixed size constants, each *entsize* octets long. Sections with both `M` and `S` must contain zero terminated strings where each character is *entsize* bytes long. The linker may remove duplicates within sections with the same name, same entity size and same flags. *entsize* must be an absolute expression. For sections with both `M` and `S`, a string which is a suffix of a larger string is considered a duplicate. Thus `"def"` will be merged with `"abcdef"`; A reference to the first `"def"` will be changed to a reference to `"abcdef"+3`.

If *flags* contains the `G` symbol then the *type* argument must be present along with an additional field like this:

```
.section name , "flags"G, @type, GroupName[, linkage]
```

The *GroupName* field specifies the name of the section group to which this particular section belongs. The optional linkage field can contain:

`comdat` indicates that only one copy of this section should be retained

`.gnu.linkonce`
: an alias for comdat

Note: if both the *M* and *G* flags are present then the fields for the Merge flag should come first, like this:

```
.section name , "flags"MG, @type, entsize, GroupName[, linkage]
```

If *flags* contains the `?` symbol then it may not also contain the `G` symbol and the *GroupName* or *linkage* fields should not be present. Instead, `?` says to consider the section that's current before this directive. If that section used `G`, then the new section will use `G` with those same *GroupName* and *linkage* fields implicitly. If not, then the `?` symbol has no effect.

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

`#exclude` section is excluded from executable and shared library.

`#tls` section is used for thread local storage

This directive replaces the current section and subsection. See the contents of the gas testsuite directory `gas/testsuite/gas/elf` for some examples of how this directive and the other section stack directives work.

7.83 `.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 45).

You may `.set` a symbol many times in the same assembly provided that the values given to the symbol are constants. Values that are based on expressions involving other symbols are allowed, but some targets may restrict this to only being done once per assembly. This is because those targets do not set the addresses of symbols at assembly time, but rather delay the assignment until a final link is performed. This allows the linker a chance to change the code in the files, changing the location of, and the relative distance between, various different symbols.

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

On Z80 `set` is a real instruction, use `.set` or '*symbol* `defl` *expression*' instead.

7.84 `.short` *expressions*

`.short` is normally the same as '`.word`'. See Section 7.107 [`.word`], page 86.

In some configurations, however, `.short` and `.word` generate numbers of different lengths. See Chapter 9 [Machine Dependencies], page 93.

7.85 `.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 9 [Machine Dependencies], page 93.

7.86 `.size`

This directive is used to set the size associated with a symbol.

COFF Version

For COFF targets, the `.size` directive is only permitted inside `.def`/`.endef` pairs. It is used like this:

```
.size expression
```

ELF Version

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

```
.size name , expression
```

This directive sets the size associated with a symbol *name*. The size in bytes is computed from *expression* which can make use of label arithmetic. This directive is typically used to set the size of function symbols.

7.87 `.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`'.

7.88 `.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 7.99 [`.uleb128`], page 85.

7.89 `.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 ⟨undefined⟩ [HPPA Assembler Directives], page ⟨undefined⟩, for a summary.

7.90 `.stabd, .stabn, .stabs`

There are three directives that begin '`.stab`'. All emit symbols (see Chapter 5 [Symbols], page 43), for use by symbolic debuggers. The symbols are not entered in the `as` hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required:

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

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

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

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

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

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

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

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

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

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

7.91 `.string` "*str*", `.string8` "*str*", `.string16`

"*str*", `.string32` "*str*", `.string64` "*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 33.

The variants `string16`, `string32` and `string64` differ from the `string` pseudo opcode in that each 8-bit character from *str* is copied and expanded to 16, 32 or 64 bits respectively. The expanded characters are stored in target endianness byte order.

Example:

```
.string32 "BYE"
expands to:
.string   "B\0\0\0Y\0\0\0E\0\0\0"  /* On little endian targets.  */
.string   "\0\0\0B\0\0\0Y\0\0\0E"  /* On big endian targets.  */
```

7.92 `.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.

7.93 `.subsection` *name*

This is one of the ELF section stack manipulation directives. The others are `.section` (see Section 7.82 [Section], page 76), `.pushsection` (see Section 7.76 [PushSection], page 75), `.popsection` (see Section 7.70 [PopSection], page 73), and `.previous` (see Section 7.71 [Previous], page 74).

This directive replaces the current subsection with `name`. The current section is not changed. The replaced subsection is put onto the section stack in place of the then current top of stack subsection.

7.94 `.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 can be used like this:

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

If the symbol *name* is defined within the file being assembled, the `.symver` 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.

If the symbol *name* is not defined within the file being assembled, all references to *name* will be changed to *name2@nodename*. If no reference to *name* is made, *name2@nodename* will be removed from the symbol table.

Another usage of the `.symver` directive is:

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

In this case, the symbol *name* must exist and be defined within the file being assembled. It is similar to *name2@nodename*. The difference is *name2@@nodename* will also be used to resolve references to *name2* by the linker.

The third usage of the `.symver` directive is:

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

When *name* is not defined within the file being assembled, it is treated as *name2@nodename*. When *name* is defined within the file being assembled, the symbol name, *name*, will be changed to *name2@@nodename*.

7.95 .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.

7.96 .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.

7.97 .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.

7.98 .type

This directive is used to set the type of a symbol.

COFF Version

For COFF targets, this directive is permitted only within `.def`/`.endef` pairs. It is used like this:

```
.type int
```

This records the integer *int* as the type attribute of a symbol table entry.

ELF Version

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

```
.type name , type description
```

This sets the type of symbol *name* to be either a function symbol or an object symbol. There are five different syntaxes supported for the *type description* field, in order to provide compatibility with various other assemblers.

Because some of the characters used in these syntaxes (such as '`@`' and '`#`') are comment characters for some architectures, some of the syntaxes below do not work on all architectures. The first variant will be accepted by the GNU assembler on all architectures so that variant should be used for maximum portability, if you do not need to assemble your code with other assemblers.

The syntaxes supported are:

```
.type <name> STT_<TYPE_IN_UPPER_CASE>
.type <name>,#<type>
.type <name>,@<type>
.type <name>,%<type>
.type <name>,"<type>"
```

The types supported are:

`STT_FUNC`
`function` Mark the symbol as being a function name.

`STT_GNU_IFUNC`
`gnu_indirect_function`
Mark the symbol as an indirect function when evaluated during reloc processing. (This is only supported on assemblers targeting GNU systems).

`STT_OBJECT`
`object` Mark the symbol as being a data object.

`STT_TLS`
`tls_object`
Mark the symbol as being a thread-local data object.

`STT_COMMON`
`common` Mark the symbol as being a common data object.

`STT_NOTYPE`
`notype` Does not mark the symbol in any way. It is supported just for completeness.

`gnu_unique_object`
Marks the symbol as being a globally unique data object. The dynamic linker will make sure that in the entire process there is just one symbol with this name and type in use. (This is only supported on assemblers targeting GNU systems).

Changing between incompatible types other than from/to STT_NOTYPE will result in a diagnostic. An intermediate change to STT_NOTYPE will silence this.

Note: Some targets support extra types in addition to those listed above.

7.99 `.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 7.88 [`.sleb128`], page 81.

7.100 `.val` *addr*

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

7.101 `.version "`*string*`"`

This directive creates a `.note` section and places into it an ELF formatted note of type NT_VERSION. The note's name is set to `string`.

7.102 `.vtable_entry` *table*`,` *offset*

This directive finds or creates a symbol `table` and creates a `VTABLE_ENTRY` relocation for it with an addend of `offset`.

7.103 .vtable_inherit *child*, *parent*

This directive finds the symbol `child` and finds or creates the symbol `parent` and then creates a `VTABLE_INHERIT` relocation for the parent whose addend is the value of the child symbol. As a special case the parent name of `0` is treated as referring to the `*ABS*` section.

7.104 .warning "*string*"

Similar to the directive `.error` (see Section 7.31 [`.error "`*string*`"`], page 61), but just emits a warning.

7.105 .weak *names*

This directive sets the weak attribute on the comma separated list of symbol `names`. If the symbols do not already exist, they will be created.

On COFF targets other than PE, weak symbols are a GNU extension. This directive sets the weak attribute on the comma separated list of symbol `names`. If the symbols do not already exist, they will be created.

On the PE target, weak symbols are supported natively as weak aliases. When a weak symbol is created that is not an alias, GAS creates an alternate symbol to hold the default value.

7.106 .weakref *alias*, *target*

This directive creates an alias to the target symbol that enables the symbol to be referenced with weak-symbol semantics, but without actually making it weak. If direct references or definitions of the symbol are present, then the symbol will not be weak, but if all references to it are through weak references, the symbol will be marked as weak in the symbol table.

The effect is equivalent to moving all references to the alias to a separate assembly source file, renaming the alias to the symbol in it, declaring the symbol as weak there, and running a reloadable link to merge the object files resulting from the assembly of the new source file and the old source file that had the references to the alias removed.

The alias itself never makes to the symbol table, and is entirely handled within the assembler.

7.107 .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 9 [Machine Dependencies], page 93), 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.

7.108 `.zero` *size*

This directive emits *size* 0-valued bytes. *size* must be an absolute expression. This directive is actually an alias for the '`.skip`' directive so it can take an optional second argument of the value to store in the bytes instead of zero. Using '`.zero`' in this way would be confusing however.

7.109 `.2byte` *expression* `[,` *expression*`]*`

This directive expects zero or more expressions, separated by commas. If there are no expressions then the directive does nothing. Otherwise each expression is evaluated in turn and placed in the next two bytes of the current output section, using the endian model of the target. If an expression will not fit in two bytes, a warning message is displayed and the least significant two bytes of the expression's value are used. If an expression cannot be evaluated at assembly time then relocations will be generated in order to compute the value at link time.

This directive does not apply any alignment before or after inserting the values. As a result of this, if relocations are generated, they may be different from those used for inserting values with a guaranteed alignment.

This directive is only available for ELF targets,

7.110 `.4byte` *expression* `[,` *expression*`]*`

Like the `.2byte` directive, except that it inserts unaligned, four byte long values into the output.

7.111 `.8byte` *expression* `[,` *expression*`]*`

Like the `.2byte` directive, except that it inserts unaligned, eight byte long bignum values into the output.

7.112 Deprecated Directives

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

```
.abort
.line
```

8 Object Attributes

`as` assembles source files written for a specific architecture into object files for that architecture. But not all object files are alike. Many architectures support incompatible variations. For instance, floating point arguments might be passed in floating point registers if the object file requires hardware floating point support—or floating point arguments might be passed in integer registers if the object file supports processors with no hardware floating point unit. Or, if two objects are built for different generations of the same architecture, the combination may require the newer generation at run-time.

This information is useful during and after linking. At link time, `ld` can warn about incompatible object files. After link time, tools like `gdb` can use it to process the linked file correctly.

Compatibility information is recorded as a series of object attributes. Each attribute has a *vendor*, *tag*, and *value*. The vendor is a string, and indicates who sets the meaning of the tag. The tag is an integer, and indicates what property the attribute describes. The value may be a string or an integer, and indicates how the property affects this object. Missing attributes are the same as attributes with a zero value or empty string value.

Object attributes were developed as part of the ABI for the ARM Architecture. The file format is documented in *ELF for the ARM Architecture*.

8.1 GNU Object Attributes

The `.gnu_attribute` directive records an object attribute with vendor '`gnu`'.

Except for '`Tag_compatibility`', which has both an integer and a string for its value, GNU attributes have a string value if the tag number is odd and an integer value if the tag number is even. The second bit (`tag & 2` is set for architecture-independent attributes and clear for architecture-dependent ones.

8.1.1 Common GNU attributes

These attributes are valid on all architectures.

Tag_compatibility (32)

> The compatibility attribute takes an integer flag value and a vendor name. If the flag value is 0, the file is compatible with other toolchains. If it is 1, then the file is only compatible with the named toolchain. If it is greater than 1, the file can only be processed by other toolchains under some private arrangement indicated by the flag value and the vendor name.

8.1.2 MIPS Attributes

Tag_GNU_MIPS_ABI_FP (4)

> The floating-point ABI used by this object file. The value will be:
>
> - 0 for files not affected by the floating-point ABI.
> - 1 for files using the hardware floating-point ABI with a standard double-precision FPU.
> - 2 for files using the hardware floating-point ABI with a single-precision FPU.

- 3 for files using the software floating-point ABI.
- 4 for files using the deprecated hardware floating-point ABI which used 64-bit floating-point registers, 32-bit general-purpose registers and increased the number of callee-saved floating-point registers.
- 5 for files using the hardware floating-point ABI with a double-precision FPU with either 32-bit or 64-bit floating-point registers and 32-bit general-purpose registers.
- 6 for files using the hardware floating-point ABI with 64-bit floating-point registers and 32-bit general-purpose registers.
- 7 for files using the hardware floating-point ABI with 64-bit floating-point registers, 32-bit general-purpose registers and a rule that forbids the direct use of odd-numbered single-precision floating-point registers.

8.1.3 PowerPC Attributes

Tag_GNU_Power_ABI_FP (4)

The floating-point ABI used by this object file. The value will be:

- 0 for files not affected by the floating-point ABI.
- 1 for files using double-precision hardware floating-point ABI.
- 2 for files using the software floating-point ABI.
- 3 for files using single-precision hardware floating-point ABI.

Tag_GNU_Power_ABI_Vector (8)

The vector ABI used by this object file. The value will be:

- 0 for files not affected by the vector ABI.
- 1 for files using general purpose registers to pass vectors.
- 2 for files using AltiVec registers to pass vectors.
- 3 for files using SPE registers to pass vectors.

8.1.4 IBM z Systems Attributes

Tag_GNU_S390_ABI_Vector (8)

The vector ABI used by this object file. The value will be:

- 0 for files not affected by the vector ABI.
- 1 for files using software vector ABI.
- 2 for files using hardware vector ABI.

8.1.5 MSP430 Attributes

Tag_GNU_MSP430_Data_Region (4)

The data region used by this object file. The value will be:

- 0 for files not using the large memory model.
- 1 for files which have been compiled with the condition that all data is in the lower memory region, i.e. below address 0x10000.
- 2 for files which allow data to be placed in the full 20-bit memory range.

8.2 Defining New Object Attributes

If you want to define a new GNU object attribute, here are the places you will need to modify. New attributes should be discussed on the '`binutils`' mailing list.

- This manual, which is the official register of attributes.
- The header for your architecture `include/elf`, to define the tag.
- The `bfd` support file for your architecture, to merge the attribute and issue any appropriate link warnings.
- Test cases in `ld/testsuite` for merging and link warnings.
- `binutils/readelf.c` to display your attribute.
- GCC, if you want the compiler to mark the attribute automatically.

9 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.

9.1 AArch64 Dependent Features

9.1.1 Options

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

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

`-mabi=`*abi*

Specify which ABI the source code uses. The recognized arguments are: `ilp32` and `lp64`, which decides the generated object file in ELF32 and ELF64 format respectively. The default is `lp64`.

`-mcpu=`*processor*`[+`*extension*`...]`

This option specifies the target processor. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target processor. The following processor names are recognized: `cortex-a34`, `cortex-a35`, `cortex-a53`, `cortex-a55`, `cortex-a57`, `cortex-a65`, `cortex-a65ae`, `cortex-a72`, `cortex-a73`, `cortex-a75`, `cortex-a76`, `cortex-a76ae`, `cortex-a77`, `ares`, `exynos-m1`, `falkor`, `neoverse-n1`, `neoverse-e1`, `qdf24xx`, `saphira`, `thunderx`, `vulcan`, `xgene1` and `xgene2`. The special name `all` may be used to allow the assembler to accept instructions valid for any supported processor, including all optional extensions.

In addition to the basic instruction set, the assembler can be told to accept, or restrict, various extension mnemonics that extend the processor. See Section 9.1.2 [AArch64 Extensions], page 95.

If some implementations of a particular processor can have an extension, then then those extensions are automatically enabled. Consequently, you will not normally have to specify any additional extensions.

`-march=`*architecture*`[+`*extension*`...]`

This option specifies the target architecture. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target architecture. The following architecture names are recognized: `armv8-a`, `armv8.1-a`, `armv8.2-a`, `armv8.3-a`, `armv8.4-a` `armv8.5-a`, and `armv8.6-a`.

If both `-mcpu` and `-march` are specified, the assembler will use the setting for `-mcpu`. If neither are specified, the assembler will default to `-mcpu=all`.

The architecture option can be extended with the same instruction set extension options as the `-mcpu` option. Unlike `-mcpu`, extensions are not always enabled by default, See Section 9.1.2 [AArch64 Extensions], page 95.

`-mverbose-error`

This option enables verbose error messages for AArch64 gas. This option is enabled by default.

`-mno-verbose-error`

This option disables verbose error messages in AArch64 gas.

9.1.2 Architecture Extensions

The table below lists the permitted architecture extensions that are supported by the assembler and the conditions under which they are automatically enabled.

Multiple extensions may be specified, separated by a `+`. Extension mnemonics may also be removed from those the assembler accepts. This is done by prepending `no` to the option that adds the extension. Extensions that are removed must be listed after all extensions that have been added.

Enabling an extension that requires other extensions will automatically cause those extensions to be enabled. Similarly, disabling an extension that is required by other extensions will automatically cause those extensions to be disabled.

Extension	**Minimum Architecture**	**Enabled by default**	**Description**
`i8mm`	ARMv8.2-A	ARMv8.6-A or later	Enable Int8 Matrix Multiply extension.
`f32mm`	ARMv8.2-A	No	Enable F32 Matrix Multiply extension.
`f64mm`	ARMv8.2-A	No	Enable F64 Matrix Multiply extension.
`bf16`	ARMv8.2-A	ARMv8.6-A or later	Enable BFloat16 extension.
`compnum`	ARMv8.2-A	ARMv8.3-A or later	Enable the complex number SIMD extensions. This implies `fp16` and `simd`.
`crc`	ARMv8-A	ARMv8.1-A or later	Enable CRC instructions.
`crypto`	ARMv8-A	No	Enable cryptographic extensions. This implies `fp`, `simd`, `aes` and `sha2`.
`aes`	ARMv8-A	No	Enable the AES cryptographic extensions. This implies `fp` and `simd`.
`sha2`	ARMv8-A	No	Enable the SHA2 cryptographic extensions. This implies `fp` and `simd`.
`sha3`	ARMv8.2-A	No	Enable the ARMv8.2-A SHA2 and SHA3 cryptographic extensions. This implies `fp`, `simd` and `sha2`.
`sm4`	ARMv8.2-A	No	Enable the ARMv8.2-A SM3 and SM4 cryptographic extensions. This implies `fp` and `simd`.
`fp`	ARMv8-A	ARMv8-A or later	Enable floating-point extensions.
`fp16`	ARMv8.2-A	ARMv8.2-A or later	Enable ARMv8.2 16-bit floating-point support. This implies `fp`.
`lor`	ARMv8-A	ARMv8.1-A or later	Enable Limited Ordering Regions extensions.
`lse`	ARMv8-A	ARMv8.1-A or later	Enable Large System extensions.
`pan`	ARMv8-A	ARMv8.1-A or later	Enable Privileged Access Never support.

`profile`	ARMv8.2-A	No	Enable statistical profiling extensions.
`ras`	ARMv8-A	ARMv8.2-A or later	Enable the Reliability, Availability and Serviceability extension.
`rcpc`	ARMv8.2-A	ARMv8.3-A or later	Enable the weak release consistency extension.
`rdma`	ARMv8-A	ARMv8.1-A or later	Enable ARMv8.1 Advanced SIMD extensions. This implies `simd`.
`simd`	ARMv8-A	ARMv8-A or later	Enable Advanced SIMD extensions. This implies `fp`.
`sve`	ARMv8.2-A	No	Enable the Scalable Vector Extensions. This implies `fp16`, `simd` and `compnum`.
`dotprod`	ARMv8.2-A	ARMv8.4-A or later	Enable the Dot Product extension. This implies `simd`.
`fp16fml`	ARMv8.2-A	ARMv8.4-A or later	Enable ARMv8.2 16-bit floating-point multiplication variant support. This implies `fp16`.
`sb`	ARMv8-A	ARMv8.5-A or later	Enable the speculation barrier instruction sb.
`predres`	ARMv8-A	ARMv8.5-A or later	Enable the Execution and Data and Prediction instructions.
`rng`	ARMv8.5-A	No	Enable ARMv8.5-A random number instructions.
`ssbs`	ARMv8-A	ARMv8.5-A or later	Enable Speculative Store Bypassing Safe state read and write.
`memtag`	ARMv8.5-A	No	Enable ARMv8.5-A Memory Tagging Extensions.
`tme`	ARMv8-A	No	Enable Transactional Memory Extensions.
`sve2`	ARMv8-A	No	Enable the SVE2 Extension.
`sve2-bitperm`	ARMv8-A	No	Enable SVE2 BITPERM Extension.
`sve2-sm4`	ARMv8-A	No	Enable SVE2 SM4 Extension.
`sve2-aes`	ARMv8-A	No	Enable SVE2 AES Extension. This also enables the .Q->.B form of the `pmullt` and `pmullb` instructions.
`sve2-sha3`	ARMv8-A	No	Enable SVE2 SHA3 Extension.

9.1.3 Syntax

9.1.3.1 Special Characters

The presence of a '`//`' on a line indicates the start of a comment that extends to the end of the current line. If a '`#`' appears as the first character of a line, the whole line is treated as a comment.

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

The '`#`' can be optionally used to indicate immediate operands.

9.1.3.2 Register Names

Please refer to the section '4.4 Register Names' of 'ARMv8 Instruction Set Overview', which is available at http://infocenter.arm.com.

9.1.3.3 Relocations

Relocations for 'MOVZ' and 'MOVK' instructions can be generated by prefixing the label with '#:abs_g2:' etc. For example to load the 48-bit absolute address of *foo* into x0:

```
movz x0, #:abs_g2:foo // bits 32-47, overflow check
movk x0, #:abs_g1_nc:foo // bits 16-31, no overflow check
movk x0, #:abs_g0_nc:foo // bits  0-15, no overflow check
```

Relocations for 'ADRP', and 'ADD', 'LDR' or 'STR' instructions can be generated by prefixing the label with ':pg_hi21:' and '#:lo12:' respectively.

For example to use 33-bit (+/-4GB) pc-relative addressing to load the address of *foo* into x0:

```
adrp x0, :pg_hi21:foo
add  x0, x0, #:lo12:foo
```

Or to load the value of *foo* into x0:

```
adrp x0, :pg_hi21:foo
ldr  x0, [x0, #:lo12:foo]
```

Note that ':pg_hi21:' is optional.

```
adrp x0, foo
```

is equivalent to

```
adrp x0, :pg_hi21:foo
```

9.1.4 Floating Point

The AArch64 architecture uses IEEE floating-point numbers.

9.1.5 AArch64 Machine Directives

`.arch` *name*

Select the target architecture. Valid values for *name* are the same as for the `-march` command-line option.

Specifying `.arch` clears any previously selected architecture extensions.

`.arch_extension` *name*

Add or remove an architecture extension to the target architecture. Valid values for *name* are the same as those accepted as architectural extensions by the `-mcpu` command-line option.

`.arch_extension` may be used multiple times to add or remove extensions incrementally to the architecture being compiled for.

`.bss` This directive switches to the `.bss` section.

`.cpu` *name* Set the target processor. Valid values for *name* are the same as those accepted by the `-mcpu=` command-line option.

`.dword` *expressions*

The `.dword` directive produces 64 bit values.

`.even` The `.even` directive aligns the output on the next even byte boundary.

`.float16` *value [,...,value_n]*
Place the half precision floating point representation of one or more floating-point values into the current section. The format used to encode the floating point values is always the IEEE 754-2008 half precision floating point format.

`.inst` *expressions*
Inserts the expressions into the output as if they were instructions, rather than data.

`.ltorg` This directive causes the current contents of the literal pool to be dumped into the current section (which is assumed to be the .text section) at the current location (aligned to a word boundary). GAS maintains a separate literal pool for each section and each sub-section. The `.ltorg` directive will only affect the literal pool of the current section and sub-section. At the end of assembly all remaining, un-empty literal pools will automatically be dumped.

Note - older versions of GAS would dump the current literal pool any time a section change occurred. This is no longer done, since it prevents accurate control of the placement of literal pools.

`.pool` This is a synonym for .ltorg.

name `.req` *register name*
This creates an alias for *register name* called *name*. For example:

```
foo .req w0
```

ip0, ip1, lr and fp are automatically defined to alias to X16, X17, X30 and X29 respectively.

`.tlsdescadd`
Emits a TLSDESC_ADD reloc on the next instruction.

`.tlsdesccall`
Emits a TLSDESC_CALL reloc on the next instruction.

`.tlsdescldr`
Emits a TLSDESC_LDR reloc on the next instruction.

`.unreq` *alias-name*
This undefines a register alias which was previously defined using the `req` directive. For example:

```
foo .req w0
.unreq foo
```

An error occurs if the name is undefined. Note - this pseudo op can be used to delete builtin in register name aliases (eg 'w0'). This should only be done if it is really necessary.

`.variant_pcs` *symbol*
This directive marks *symbol* referencing a function that may follow a variant procedure call standard with different register usage convention from the base procedure call standard.

`.xword` *expressions*
: The `.xword` directive produces 64 bit values. This is the same as the `.dword` directive.

`.cfi_b_key_frame`
: The `.cfi_b_key_frame` directive inserts a 'B' character into the CIE corresponding to the current frame's FDE, meaning that its return address has been signed with the B-key. If two frames are signed with differing keys then they will not share the same CIE. This information is intended to be used by the stack unwinder in order to properly authenticate return addresses.

9.1.6 Opcodes

GAS implements all the standard AArch64 opcodes. It also implements several pseudo opcodes, including several synthetic load instructions.

`LDR =`
: `ldr <register> , =<expression>`

  The constant expression will be placed into the nearest literal pool (if it not already there) and a PC-relative LDR instruction will be generated.

For more information on the AArch64 instruction set and assembly language notation, see '`ARMv8 Instruction Set Overview`' available at `http://infocenter.arm.com`.

9.1.7 Mapping Symbols

The AArch64 ELF specification requires that special symbols be inserted into object files to mark certain features:

`$x` At the start of a region of code containing AArch64 instructions.

`$d` At the start of a region of data.

9.2 ARM Dependent Features

9.2.1 Options

`-mcpu=`*`processor`*`[+`*`extension...`*`]`

This option specifies the target processor. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target processor. The following processor names are recognized: `arm1`, `arm2`, `arm250`, `arm3`, `arm6`, `arm60`, `arm600`, `arm610`, `arm620`, `arm7`, `arm7m`, `arm7d`, `arm7dm`, `arm7di`, `arm7dmi`, `arm70`, `arm700`, `arm700i`, `arm710`, `arm710t`, `arm720`, `arm720t`, `arm740t`, `arm710c`, `arm7100`, `arm7500`, `arm7500fe`, `arm7t`, `arm7tdmi`, `arm7tdmi-s`, `arm8`, `arm810`, `strongarm`, `strongarm1`, `strongarm110`, `strongarm1100`, `strongarm1110`, `arm9`, `arm920`, `arm920t`, `arm922t`, `arm940t`, `arm9tdmi`, `fa526` (Faraday FA526 processor), `fa626` (Faraday FA626 processor), `arm9e`, `arm926e`, `arm926ej-s`, `arm946e-r0`, `arm946e`, `arm946e-s`, `arm966e-r0`, `arm966e`, `arm966e-s`, `arm968e-s`, `arm10t`, `arm10tdmi`, `arm10e`, `arm1020`, `arm1020t`, `arm1020e`, `arm1022e`, `arm1026ej-s`, `fa606te` (Faraday FA606TE processor), `fa616te` (Faraday FA616TE processor), `fa626te` (Faraday FA626TE processor), `fmp626` (Faraday FMP626 processor), `fa726te` (Faraday FA726TE processor), `arm1136j-s`, `arm1136jf-s`, `arm1156t2-s`, `arm1156t2f-s`, `arm1176jz-s`, `arm1176jzf-s`, `mpcore`, `mpcorenovfp`, `cortex-a5`, `cortex-a7`, `cortex-a8`, `cortex-a9`, `cortex-a15`, `cortex-a17`, `cortex-a32`, `cortex-a35`, `cortex-a53`, `cortex-a55`, `cortex-a57`, `cortex-a72`, `cortex-a73`, `cortex-a75`, `cortex-a76`, `cortex-a76ae`, `cortex-a77`, `ares`, `cortex-r4`, `cortex-r4f`, `cortex-r5`, `cortex-r7`, `cortex-r8`, `cortex-r52`, `cortex-m35p`, `cortex-m33`, `cortex-m23`, `cortex-m7`, `cortex-m4`, `cortex-m3`, `cortex-m1`, `cortex-m0`, `cortex-m0plus`, `exynos-m1`, `marvell-pj4`, `marvell-whitney`, `neoverse-n1`, `xgene1`, `xgene2`, `ep9312` (ARM920 with Cirrus Maverick coprocessor), `i80200` (Intel XScale processor) `iwmmxt` (Intel XScale processor with Wireless MMX technology coprocessor) and `xscale`. The special name `all` may be used to allow the assembler to accept instructions valid for any ARM processor.

In addition to the basic instruction set, the assembler can be told to accept various extension mnemonics that extend the processor using the co-processor instruction space. For example, `-mcpu=arm920+maverick` is equivalent to specifying `-mcpu=ep9312`.

Multiple extensions may be specified, separated by a `+`. The extensions should be specified in ascending alphabetical order.

Some extensions may be restricted to particular architectures; this is documented in the list of extensions below.

Extension mnemonics may also be removed from those the assembler accepts. This is done be prepending `no` to the option that adds the extension. Extensions that are removed should be listed after all extensions which have been added, again in ascending alphabetical order. For example, `-mcpu=ep9312+nomaverick` is equivalent to specifying `-mcpu=arm920`.

The following extensions are currently supported: `bf16` (BFloat16 extensions for v8.6-A architecture), `i8mm` (Int8 Matrix Multiply extensions for v8.6-A architecture), `crc` `crypto` (Cryptography Extensions for v8-A architecture, implies `fp+simd`), `dotprod` (Dot Product Extensions for v8.2-A architecture, implies `fp+simd`), `fp` (Floating Point Extensions for v8-A architecture), `fp16` (FP16 Extensions for v8.2-A architecture, implies `fp`), `fp16fml` (FP16 Floating Point Multiplication Variant Extensions for v8.2-A architecture, implies `fp16`), `idiv` (Integer Divide Extensions for v7-A and v7-R architectures), `iwmmxt`, `iwmmxt2`, `xscale`, `maverick`, `mp` (Multiprocessing Extensions for v7-A and v7-R architectures), `os` (Operating System for v6M architecture), `predres` (Execution and Data Prediction Restriction Instruction for v8-A architectures, added by default from v8.5-A), `sb` (Speculation Barrier Instruction for v8-A architectures, added by default from v8.5-A), `sec` (Security Extensions for v6K and v7-A architectures), `simd` (Advanced SIMD Extensions for v8-A architecture, implies `fp`), `virt` (Virtualization Extensions for v7-A architecture, implies `idiv`), `pan` (Privileged Access Never Extensions for v8-A architecture), `ras` (Reliability, Availability and Serviceability extensions for v8-A architecture), `rdma` (ARMv8.1 Advanced SIMD extensions for v8-A architecture, implies `simd`) and `xscale`.

`-march=`*`architecture`*`[+`*`extension...`*`]`

This option specifies the target architecture. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target architecture. The following architecture names are recognized: `armv1`, `armv2`, `armv2a`, `armv2s`, `armv3`, `armv3m`, `armv4`, `armv4xm`, `armv4t`, `armv4txm`, `armv5`, `armv5t`, `armv5txm`, `armv5te`, `armv5texp`, `armv6`, `armv6j`, `armv6k`, `armv6z`, `armv6kz`, `armv6-m`, `armv6s-m`, `armv7`, `armv7-a`, `armv7ve`, `armv7-r`, `armv7-m`, `armv7e-m`, `armv8-a`, `armv8.1-a`, `armv8.2-a`, `armv8.3-a`, `armv8-r`, `armv8.4-a`, `armv8.5-a`, `armv8-m.base`, `armv8-m.main`, `armv8.1-m.main`, `armv8.6-a`, `iwmmxt`, `iwmmxt2` and `xscale`. If both `-mcpu` and `-march` are specified, the assembler will use the setting for `-mcpu`.

The architecture option can be extended with a set extension options. These extensions are context sensitive, i.e. the same extension may mean different things when used with different architectures. When used together with a `-mfpu` option, the union of both feature enablement is taken. See their availability and meaning below:

For `armv5te`, `armv5texp`, `armv5tej`, `armv6`, `armv6j`, `armv6k`, `armv6z`, `armv6kz`, `armv6zk`, `armv6t2`, `armv6kt2` and `armv6zt2`:

`+fp`: Enables VFPv2 instructions. `+nofp`: Disables all FPU instrunctions.

For `armv7`:

`+fp`: Enables VFPv3 instructions with 16 double-word registers. `+nofp`: Disables all FPU instructions.

For `armv7-a`:

`+fp`: Enables VFPv3 instructions with 16 double-word registers. `+vfpv3-d16`: Alias for `+fp`. `+vfpv3`: Enables VFPv3 instructions with 32 double-word registers. `+vfpv3-d16-fp16`: Enables VFPv3 with half precision floating-point

conversion instructions and 16 double-word registers. `+vfpv3-fp16`: Enables VFPv3 with half precision floating-point conversion instructions and 32 double-word registers. `+vfpv4-d16`: Enables VFPv4 instructions with 16 double-word registers. `+vfpv4`: Enables VFPv4 instructions with 32 double-word registers. `+simd`: Enables VFPv3 and NEONv1 instructions with 32 double-word registers. `+neon`: Alias for `+simd`. `+neon-vfpv3`: Alias for `+simd`. `+neon-fp16`: Enables VFPv3, half precision floating-point conversion and NEONv1 instructions with 32 double-word registers. `+neon-vfpv4`: Enables VFPv4 and NEONv1 with Fused-MAC instructions and 32 double-word registers. `+mp`: Enables Multiprocessing Extensions. `+sec`: Enables Security Extensions. `+nofp`: Disables all FPU and NEON instructions. `+nosimd`: Disables all NEON instructions.

For `armv7ve`:

`+fp`: Enables VFPv4 instructions with 16 double-word registers. `+vfpv4-d16`: Alias for `+fp`. `+vfpv3-d16`: Enables VFPv3 instructions with 16 double-word registers. `+vfpv3`: Enables VFPv3 instructions with 32 double-word registers. `+vfpv3-d16-fp16`: Enables VFPv3 with half precision floating-point conversion instructions and 16 double-word registers. `+vfpv3-fp16`: Enables VFPv3 with half precision floating-point conversion instructions and 32 double-word registers. `+vfpv4`: Enables VFPv4 instructions with 32 double-word registers. `+simd`: Enables VFPv4 and NEONv1 with Fused-MAC instructions and 32 double-word registers. `+neon-vfpv4`: Alias for `+simd`. `+neon`: Enables VFPv3 and NEONv1 instructions with 32 double-word registers. `+neon-vfpv3`: Alias for `+neon`. `+neon-fp16`: Enables VFPv3, half precision floating-point conversion and NEONv1 instructions with 32 double-word registers. double-word registers. `+nofp`: Disables all FPU and NEON instructions. `+nosimd`: Disables all NEON instructions.

For `armv7-r`:

`+fp.sp`: Enables single-precision only VFPv3 instructions with 16 double-word registers. `+vfpv3xd`: Alias for `+fp.sp`. `+fp`: Enables VFPv3 instructions with 16 double-word registers. `+vfpv3-d16`: Alias for `+fp`. `+vfpv3xd-fp16`: Enables single-precision only VFPv3 and half floating-point conversion instructions with 16 double-word registers. `+vfpv3-d16-fp16`: Enables VFPv3 and half precision floating-point conversion instructions with 16 double-word registers. `+idiv`: Enables integer division instructions in ARM mode. `+nofp`: Disables all FPU instructions.

For `armv7e-m`:

`+fp`: Enables single-precision only VFPv4 instructions with 16 double-word registers. `+vfpvf4-sp-d16`: Alias for `+fp`. `+fpv5`: Enables single-precision only VFPv5 instructions with 16 double-word registers. `+fp.dp`: Enables VFPv5 instructions with 16 double-word registers. `+fpv5-d16"`: Alias for `+fp.dp`. `+nofp`: Disables all FPU instructions.

For `armv8-m.main`:

`+dsp`: Enables DSP Extension. `+fp`: Enables single-precision only VFPv5 instructions with 16 double-word registers. `+fp.dp`: Enables VFPv5 instructions

with 16 double-word registers. `+nofp`: Disables all FPU instructions. `+nodsp`: Disables DSP Extension.

For `armv8.1-m.main`:

`+dsp`: Enables DSP Extension. `+fp`: Enables single and half precision scalar Floating Point Extensions for Armv8.1-M Mainline with 16 double-word registers. `+fp.dp`: Enables double precision scalar Floating Point Extensions for Armv8.1-M Mainline, implies `+fp`. `+mve`: Enables integer only M-profile Vector Extension for Armv8.1-M Mainline, implies `+dsp`. `+mve.fp`: Enables Floating Point M-profile Vector Extension for Armv8.1-M Mainline, implies `+mve` and `+fp`. `+nofp`: Disables all FPU instructions. `+nodsp`: Disables DSP Extension. `+nomve`: Disables all M-profile Vector Extensions.

For `armv8-a`:

`+crc`: Enables CRC32 Extension. `+simd`: Enables VFP and NEON for Armv8-A. `+crypto`: Enables Cryptography Extensions for Armv8-A, implies `+simd`. `+sb`: Enables Speculation Barrier Instruction for Armv8-A. `+predres`: Enables Execution and Data Prediction Restriction Instruction for Armv8-A. `+nofp`: Disables all FPU, NEON and Cryptography Extensions. `+nocrypto`: Disables Cryptography Extensions.

For `armv8.1-a`:

`+simd`: Enables VFP and NEON for Armv8.1-A. `+crypto`: Enables Cryptography Extensions for Armv8-A, implies `+simd`. `+sb`: Enables Speculation Barrier Instruction for Armv8-A. `+predres`: Enables Execution and Data Prediction Restriction Instruction for Armv8-A. `+nofp`: Disables all FPU, NEON and Cryptography Extensions. `+nocrypto`: Disables Cryptography Extensions.

For `armv8.2-a` and `armv8.3-a`:

`+simd`: Enables VFP and NEON for Armv8.1-A. `+fp16`: Enables FP16 Extension for Armv8.2-A, implies `+simd`. `+fp16fml`: Enables FP16 Floating Point Multiplication Variant Extensions for Armv8.2-A, implies `+fp16`. `+crypto`: Enables Cryptography Extensions for Armv8-A, implies `+simd`. `+dotprod`: Enables Dot Product Extensions for Armv8.2-A, implies `+simd`. `+sb`: Enables Speculation Barrier Instruction for Armv8-A. `+predres`: Enables Execution and Data Prediction Restriction Instruction for Armv8-A. `+nofp`: Disables all FPU, NEON, Cryptography and Dot Product Extensions. `+nocrypto`: Disables Cryptography Extensions.

For `armv8.4-a`:

`+simd`: Enables VFP and NEON for Armv8.1-A and Dot Product Extensions for Armv8.2-A. `+fp16`: Enables FP16 Floating Point and Floating Point Multiplication Variant Extensions for Armv8.2-A, implies `+simd`. `+crypto`: Enables Cryptography Extensions for Armv8-A, implies `+simd`. `+sb`: Enables Speculation Barrier Instruction for Armv8-A. `+predres`: Enables Execution and Data Prediction Restriction Instruction for Armv8-A. `+nofp`: Disables all FPU, NEON, Cryptography and Dot Product Extensions. `+nocryptp`: Disables Cryptography Extensions.

For `armv8.5-a`:

`+simd`: Enables VFP and NEON for Armv8.1-A and Dot Product Extensions for Armv8.2-A. `+fp16`: Enables FP16 Floating Point and Floating Point Multiplication Variant Extensions for Armv8.2-A, implies `+simd`. `+crypto`: Enables Cryptography Extensions for Armv8-A, implies `+simd`. `+nofp`: Disables all FPU, NEON, Cryptography and Dot Product Extensions. `+nocryptp`: Disables Cryptography Extensions.

`-mfpu=`*`floating-point-format`*

This option specifies the floating point format to assemble for. The assembler will issue an error message if an attempt is made to assemble an instruction which will not execute on the target floating point unit. The following format options are recognized: `softfpa`, `fpe`, `fpe2`, `fpe3`, `fpa`, `fpa10`, `fpa11`, `arm7500fe`, `softvfp`, `softvfp+vfp`, `vfp`, `vfp10`, `vfp10-r0`, `vfp9`, `vfpxd`, `vfpv2`, `vfpv3`, `vfpv3-fp16`, `vfpv3-d16`, `vfpv3-d16-fp16`, `vfpv3xd`, `vfpv3xd-d16`, `vfpv4`, `vfpv4-d16`, `fpv4-sp-d16`, `fpv5-sp-d16`, `fpv5-d16`, `fp-armv8`, `arm1020t`, `arm1020e`, `arm1136jf-s`, `maverick`, `neon`, `neon-vfpv3`, `neon-fp16`, `neon-vfpv4`, `neon-fp-armv8`, `crypto-neon-fp-armv8`, `neon-fp-armv8.1` and `crypto-neon-fp-armv8.1`.

In addition to determining which instructions are assembled, this option also affects the way in which the `.double` assembler directive behaves when assembling little-endian code.

The default is dependent on the processor selected. For Architecture 5 or later, the default is to assemble for VFP instructions; for earlier architectures the default is to assemble for FPA instructions.

`-mfp16-format=`*`format`*

This option specifies the half-precision floating point format to use when assembling floating point numbers emitted by the `.float16` directive. The following format options are recognized: `ieee`, `alternative`. If `ieee` is specified then the IEEE 754-2008 half-precision floating point format is used, if `alternative` is specified then the Arm alternative half-precision format is used. If this option is set on the command line then the format is fixed and cannot be changed with the `float16_format` directive. If this value is not set then the IEEE 754-2008 format is used until the format is explicitly set with the `float16_format` directive.

`-mthumb` This option specifies that the assembler should start assembling Thumb instructions; that is, it should behave as though the file starts with a `.code 16` directive.

`-mthumb-interwork`

This option specifies that the output generated by the assembler should be marked as supporting interworking. It also affects the behaviour of the `ADR` and `ADRL` pseudo opcodes.

`-mimplicit-it=never`
`-mimplicit-it=always`
`-mimplicit-it=arm`
`-mimplicit-it=thumb`

The `-mimplicit-it` option controls the behavior of the assembler when conditional instructions are not enclosed in IT blocks. There are four possible behaviors. If `never` is specified, such constructs cause a warning in ARM code and an error in Thumb-2 code. If `always` is specified, such constructs are accepted in both ARM and Thumb-2 code, where the IT instruction is added implicitly. If `arm` is specified, such constructs are accepted in ARM code and cause an error in Thumb-2 code. If `thumb` is specified, such constructs cause a warning in ARM code and are accepted in Thumb-2 code. If you omit this option, the behavior is equivalent to `-mimplicit-it=arm`.

`-mapcs-26`
`-mapcs-32`

These options specify that the output generated by the assembler should be marked as supporting the indicated version of the Arm Procedure. Calling Standard.

`-matpcs` This option specifies that the output generated by the assembler should be marked as supporting the Arm/Thumb Procedure Calling Standard. If enabled this option will cause the assembler to create an empty debugging section in the object file called .arm.atpcs. Debuggers can use this to determine the ABI being used by.

`-mapcs-float`

This indicates the floating point variant of the APCS should be used. In this variant floating point arguments are passed in FP registers rather than integer registers.

`-mapcs-reentrant`

This indicates that the reentrant variant of the APCS should be used. This variant supports position independent code.

`-mfloat-abi=`*abi*

This option specifies that the output generated by the assembler should be marked as using specified floating point ABI. The following values are recognized: `soft`, `softfp` and `hard`.

`-meabi=`*ver*

This option specifies which EABI version the produced object files should conform to. The following values are recognized: `gnu`, `4` and `5`.

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

Note: If a program is being built for a system with big-endian data and little-endian instructions then it should be assembled with the `-EB` option, (all of it, code and data) and then linked with the `--be8` option. This will reverse the endianness of the instructions back to little-endian, but leave the data as big-endian.

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

`-k` This option specifies that the output of the assembler should be marked as position-independent code (PIC).

`--fix-v4bx`
Allow `BX` instructions in ARMv4 code. This is intended for use with the linker option of the same name.

`-mwarn-deprecated`
`-mno-warn-deprecated`
Enable or disable warnings about using deprecated options or features. The default is to warn.

`-mccs` Turns on CodeComposer Studio assembly syntax compatibility mode.

`-mwarn-syms`
`-mno-warn-syms`
Enable or disable warnings about symbols that match the names of ARM instructions. The default is to warn.

9.2.2 Syntax

9.2.2.1 Instruction Set Syntax

Two slightly different syntaxes are support for ARM and THUMB instructions. The default, `divided`, uses the old style where ARM and THUMB instructions had their own, separate syntaxes. The new, `unified` syntax, which can be selected via the `.syntax` directive, and has the following main features:

- Immediate operands do not require a `#` prefix.
- The `IT` instruction may appear, and if it does it is validated against subsequent conditional affixes. In ARM mode it does not generate machine code, in THUMB mode it does.
- For ARM instructions the conditional affixes always appear at the end of the instruction. For THUMB instructions conditional affixes can be used, but only inside the scope of an `IT` instruction.
- All of the instructions new to the V6T2 architecture (and later) are available. (Only a few such instructions can be written in the `divided` syntax).
- The `.N` and `.W` suffixes are recognized and honored.
- All instructions set the flags if and only if they have an `s` affix.

9.2.2.2 Special Characters

The presence of a '`@`' anywhere on a line indicates the start of a comment that extends to the end of that line.

If a '`#`' appears as the first character of a line then the whole line is treated as a comment, but in this case the line could also be a logical line number directive (see Section 3.3 [Comments], page 31) or a preprocessor control command (see Section 3.1 [Preprocessing], page 31).

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

Either '#' or '$' can be used to indicate immediate operands.

TODO Explain about /data modifier on symbols.

9.2.2.3 Register Names

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

9.2.2.4 ARM relocation generation

Specific data relocations can be generated by putting the relocation name in parentheses after the symbol name. For example:

```
.word foo(TARGET1)
```

This will generate an 'R_ARM_TARGET1' relocation against the symbol *foo*. The following relocations are supported: GOT, GOTOFF, TARGET1, TARGET2, SBREL, TLSGD, TLSLDM, TLSLDO, TLSDESC, TLSCALL, GOTTPOFF, GOT_PREL and TPOFF.

For compatibility with older toolchains the assembler also accepts (PLT) after branch targets. On legacy targets this will generate the deprecated 'R_ARM_PLT32' relocation. On EABI targets it will encode either the 'R_ARM_CALL' or 'R_ARM_JUMP24' relocation, as appropriate.

Relocations for 'MOVW' and 'MOVT' instructions can be generated by prefixing the value with '#:lower16:' and '#:upper16' respectively. For example to load the 32-bit address of foo into r0:

```
MOVW r0, #:lower16:foo
MOVT r0, #:upper16:foo
```

Relocations 'R_ARM_THM_ALU_ABS_G0_NC', 'R_ARM_THM_ALU_ABS_G1_NC', 'R_ARM_THM_ALU_ABS_G2_NC' and 'R_ARM_THM_ALU_ABS_G3_NC' can be generated by prefixing the value with '#:lower0_7:#', '#:lower8_15:#', '#:upper0_7:#' and '#:upper8_15:#' respectively. For example to load the 32-bit address of foo into r0:

```
MOVS r0, #:upper8_15:#foo
LSLS r0, r0, #8
ADDS r0, #:upper0_7:#foo
LSLS r0, r0, #8
ADDS r0, #:lower8_15:#foo
LSLS r0, r0, #8
ADDS r0, #:lower0_7:#foo
```

9.2.2.5 NEON Alignment Specifiers

Some NEON load/store instructions allow an optional address alignment qualifier. The ARM documentation specifies that this is indicated by '@ *align*'. However GAS already interprets the '@' character as a "line comment" start, so ': *align*' is used instead. For example:

```
vld1.8 {q0}, [r0, :128]
```

9.2.3 Floating Point

The ARM family uses IEEE floating-point numbers.

9.2.4 ARM Machine Directives

`.align` *expression* [, *expression*]
: This is the generic *.align* directive. For the ARM however if the first argument is zero (ie no alignment is needed) the assembler will behave as if the argument had been 2 (ie pad to the next four byte boundary). This is for compatibility with ARM's own assembler.

`.arch` *name*
: Select the target architecture. Valid values for *name* are the same as for the `-march` command-line option without the instruction set extension.

  Specifying `.arch` clears any previously selected architecture extensions.

`.arch_extension` *name*
: Add or remove an architecture extension to the target architecture. Valid values for *name* are the same as those accepted as architectural extensions by the `-mcpu` and `-march` command-line options.

  `.arch_extension` may be used multiple times to add or remove extensions incrementally to the architecture being compiled for.

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

`.bss`
: This directive switches to the `.bss` section.

`.cantunwind`
: Prevents unwinding through the current function. No personality routine or exception table data is required or permitted.

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

`.cpu` *name*
: Select the target processor. Valid values for *name* are the same as for the `-mcpu` command-line option without the instruction set extension.

  Specifying `.cpu` clears any previously selected architecture extensions.

name `.dn` *register* *name* [`.`*type*] [[*index*]]
name `.qn` *register* *name* [`.`*type*] [[*index*]]
: The `dn` and `qn` directives are used to create typed and/or indexed register aliases for use in Advanced SIMD Extension (Neon) instructions. The former should be used to create aliases of double-precision registers, and the latter to create aliases of quad-precision registers.

  If these directives are used to create typed aliases, those aliases can be used in Neon instructions instead of writing types after the mnemonic or after each operand. For example:

```
x .dn d2.f32
y .dn d3.f32
z .dn d4.f32[1]
vmul x,y,z
```

  This is equivalent to writing the following:

```
vmul.f32 d2,d3,d4[1]
```

  Aliases created using `dn` or `qn` can be destroyed using `unreq`.

`.eabi_attribute` *tag, value*

Set the EABI object attribute *tag* to *value*.

The *tag* is either an attribute number, or one of the following: `Tag_CPU_raw_name`, `Tag_CPU_name`, `Tag_CPU_arch`, `Tag_CPU_arch_profile`, `Tag_ARM_ISA_use`, `Tag_THUMB_ISA_use`, `Tag_FP_arch`, `Tag_WMMX_arch`, `Tag_Advanced_SIMD_arch`, `Tag_MVE_arch`, `Tag_PCS_config`, `Tag_ABI_PCS_R9_use`, `Tag_ABI_PCS_RW_data`, `Tag_ABI_PCS_RO_data`, `Tag_ABI_PCS_GOT_use`, `Tag_ABI_PCS_wchar_t`, `Tag_ABI_FP_rounding`, `Tag_ABI_FP_denormal`, `Tag_ABI_FP_exceptions`, `Tag_ABI_FP_user_exceptions`, `Tag_ABI_FP_number_model`, `Tag_ABI_align_needed`, `Tag_ABI_align_preserved`, `Tag_ABI_enum_size`, `Tag_ABI_HardFP_use`, `Tag_ABI_VFP_args`, `Tag_ABI_WMMX_args`, `Tag_ABI_optimization_goals`, `Tag_ABI_FP_optimization_goals`, `Tag_compatibility`, `Tag_CPU_unaligned_access`, `Tag_FP_HP_extension`, `Tag_ABI_FP_16bit_format`, `Tag_MPextension_use`, `Tag_DIV_use`, `Tag_nodefaults`, `Tag_also_compatible_with`, `Tag_conformance`, `Tag_T2EE_use`, `Tag_Virtualization_use`

The *value* is either a `number`, `"string"`, or `number, "string"` depending on the tag.

Note - the following legacy values are also accepted by *tag*: `Tag_VFP_arch`, `Tag_ABI_align8_needed`, `Tag_ABI_align8_preserved`, `Tag_VFP_HP_extension`,

`.even` This directive aligns to an even-numbered address.

`.extend` *expression* `[,` *expression*`]*`

`.ldouble` *expression* `[,` *expression*`]*`

These directives write 12byte long double floating-point values to the output section. These are not compatible with current ARM processors or ABIs.

`.float16` *value [,...,value_n]*

Place the half precision floating point representation of one or more floating-point values into the current section. The exact format of the encoding is specified by `.float16_format`. If the format has not been explicitly set yet (either via the `.float16_format` directive or the command line option) then the IEEE 754-2008 format is used.

`.float16_format` *format*

Set the format to use when encoding float16 values emitted by the `.float16` directive. Once the format has been set it cannot be changed. `format` should be one of the following: `ieee` (encode in the IEEE 754-2008 half precision format) or `alternative` (encode in the Arm alternative half precision format).

`.fnend` Marks the end of a function with an unwind table entry. The unwind index table entry is created when this directive is processed.

If no personality routine has been specified then standard personality routine 0 or 1 will be used, depending on the number of unwind opcodes required.

`.fnstart` Marks the start of a function with an unwind table entry.

`.force_thumb`

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

`.fpu` *name* Select the floating-point unit to assemble for. Valid values for *name* are the same as for the `-mfpu` command-line option.

`.handlerdata`
Marks the end of the current function, and the start of the exception table entry for that function. Anything between this directive and the `.fnend` directive will be added to the exception table entry.

Must be preceded by a `.personality` or `.personalityindex` directive.

`.inst` *opcode* `[ , ... ]`
`.inst.n` *opcode* `[ , ... ]`
`.inst.w` *opcode* `[ , ... ]`
Generates the instruction corresponding to the numerical value *opcode*. `.inst.n` and `.inst.w` allow the Thumb instruction size to be specified explicitly, overriding the normal encoding rules.

`.ldouble` *expression* `[,` *expression*`]*`
See `.extend`.

`.ltorg` This directive causes the current contents of the literal pool to be dumped into the current section (which is assumed to be the .text section) at the current location (aligned to a word boundary). `GAS` maintains a separate literal pool for each section and each sub-section. The `.ltorg` directive will only affect the literal pool of the current section and sub-section. At the end of assembly all remaining, un-empty literal pools will automatically be dumped.

Note - older versions of `GAS` would dump the current literal pool any time a section change occurred. This is no longer done, since it prevents accurate control of the placement of literal pools.

`.movsp` *reg* `[, #`*offset*`]`
Tell the unwinder that *reg* contains an offset from the current stack pointer. If *offset* is not specified then it is assumed to be zero.

`.object_arch` *name*
Override the architecture recorded in the EABI object attribute section. Valid values for *name* are the same as for the `.arch` directive. Typically this is useful when code uses runtime detection of CPU features.

`.packed` *expression* `[,` *expression*`]*`
This directive writes 12-byte packed floating-point values to the output section. These are not compatible with current ARM processors or ABIs.

`.pad #`*count*
Generate unwinder annotations for a stack adjustment of *count* bytes. A positive value indicates the function prologue allocated stack space by decrementing the stack pointer.

`.personality` *name*
Sets the personality routine for the current function to *name*.

`.personalityindex` *index*
Sets the personality routine for the current function to the EABI standard routine number *index*

`.pool` This is a synonym for .ltorg.

name `.req` *register name*
This creates an alias for *register name* called *name*. For example:

```
foo .req r0
```

`.save` *reglist*
Generate unwinder annotations to restore the registers in *reglist*. The format of *reglist* is the same as the corresponding store-multiple instruction.

core registers

```
.save {r4, r5, r6, lr}
stmfd sp!, {r4, r5, r6, lr}
```

FPA registers

```
.save f4, 2
sfmfd f4, 2, [sp]!
```

VFP registers

```
.save {d8, d9, d10}
fstmdx sp!, {d8, d9, d10}
```

iWMMXt registers

```
.save {wr10, wr11}
wstrd wr11, [sp, #-8]!
wstrd wr10, [sp, #-8]!
or
.save wr11
wstrd wr11, [sp, #-8]!
.save wr10
wstrd wr10, [sp, #-8]!
```

`.setfp` *fpreg*`,` *spreg* `[,` `#`*offset*`]`
Make all unwinder annotations relative to a frame pointer. Without this the unwinder will use offsets from the stack pointer.

The syntax of this directive is the same as the `add` or `mov` instruction used to set the frame pointer. *spreg* must be either `sp` or mentioned in a previous `.movsp` directive.

```
.movsp ip
mov ip, sp
...
.setfp fp, ip, #4
add fp, ip, #4
```

`.secrel32` *expression* `[,` *expression*`]*`
This directive emits relocations that evaluate to the section-relative offset of each expression's symbol. This directive is only supported for PE targets.

`.syntax [unified | divided]`
This directive sets the Instruction Set Syntax as described in the Section 9.2.2.1 [ARM-Instruction-Set], page 106, section.

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

`.thumb_func`
This directive specifies that the following symbol is the name of a Thumb encoded function. This information is necessary in order to allow the assembler and linker to generate correct code for interworking between Arm and Thumb

instructions and should be used even if interworking is not going to be performed. The presence of this directive also implies `.thumb`

This directive is not necessary when generating EABI objects. On these targets the encoding is implicit when generating Thumb code.

`.thumb_set`
: This performs the equivalent of a `.set` directive in that it creates a symbol which is an alias for another symbol (possibly not yet defined). This directive also has the added property in that it marks the aliased symbol as being a thumb function entry point, in the same way that the `.thumb_func` directive does.

`.tlsdescseq` *tls-variable*
: This directive is used to annotate parts of an inlined TLS descriptor trampoline. Normally the trampoline is provided by the linker, and this directive is not needed.

`.unreq` *alias-name*
: This undefines a register alias which was previously defined using the `req`, `dn` or `qn` directives. For example:

```
foo .req r0
.unreq foo
```

An error occurs if the name is undefined. Note - this pseudo op can be used to delete builtin in register name aliases (eg 'r0'). This should only be done if it is really necessary.

`.unwind_raw` *offset*`,` *byte1*`, ...`
: Insert one of more arbitrary unwind opcode bytes, which are known to adjust the stack pointer by *offset* bytes.

For example `.unwind_raw 4, 0xb1, 0x01` is equivalent to `.save {r0}`

`.vsave` *vfp-reglist*
: Generate unwinder annotations to restore the VFP registers in *vfp-reglist* using FLDMD. Also works for VFPv3 registers that are to be restored using VLDM. The format of *vfp-reglist* is the same as the corresponding store-multiple instruction.

VFP registers

```
.vsave {d8, d9, d10}
fstmdd sp!, {d8, d9, d10}
```

VFPv3 registers

```
.vsave {d15, d16, d17}
vstm sp!, {d15, d16, d17}
```

Since FLDMX and FSTMX are now deprecated, this directive should be used in favour of `.save` for saving VFP registers for ARMv6 and above.

9.2.5 Opcodes

`as` implements all the standard ARM opcodes. It also implements several pseudo opcodes, including several synthetic load instructions.

`NOP`

```
nop
```

This pseudo op will always evaluate to a legal ARM instruction that does nothing. Currently it will evaluate to MOV r0, r0.

`LDR`

```
ldr <register> , = <expression>
```

If expression evaluates to a numeric constant then a MOV or MVN instruction will be used in place of the LDR instruction, if the constant can be generated by either of these instructions. Otherwise the constant will be placed into the nearest literal pool (if it not already there) and a PC relative LDR instruction will be generated.

`ADR`

```
adr <register> <label>
```

This instruction will load the address of *label* into the indicated register. The instruction will evaluate to a PC relative ADD or SUB instruction depending upon where the label is located. If the label is out of range, or if it is not defined in the same file (and section) as the ADR instruction, then an error will be generated. This instruction will not make use of the literal pool.

If *label* is a thumb function symbol, and thumb interworking has been enabled via the `-mthumb-interwork` option then the bottom bit of the value stored into *register* will be set. This allows the following sequence to work as expected:

```
adr     r0, thumb_function
blx     r0
```

`ADRL`

```
adrl <register> <label>
```

This instruction will load the address of *label* into the indicated register. The instruction will evaluate to one or two PC relative ADD or SUB instructions depending upon where the label is located. If a second instruction is not needed a NOP instruction will be generated in its place, so that this instruction is always 8 bytes long.

If the label is out of range, or if it is not defined in the same file (and section) as the ADRL instruction, then an error will be generated. This instruction will not make use of the literal pool.

If *label* is a thumb function symbol, and thumb interworking has been enabled via the `-mthumb-interwork` option then the bottom bit of the value stored into *register* will be set.

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

9.2.6 Mapping Symbols

The ARM ELF specification requires that special symbols be inserted into object files to mark certain features:

`$a` At the start of a region of code containing ARM instructions.

`$t` At the start of a region of code containing THUMB instructions.

`$d` At the start of a region of data.

The assembler will automatically insert these symbols for you - there is no need to code them yourself. Support for tagging symbols ($b, $f, $p and $m) which is also mentioned in the current ARM ELF specification is not implemented. This is because they have been dropped from the new EABI and so tools cannot rely upon their presence.

9.2.7 Unwinding

The ABI for the ARM Architecture specifies a standard format for exception unwind information. This information is used when an exception is thrown to determine where control should be transferred. In particular, the unwind information is used to determine which function called the function that threw the exception, and which function called that one, and so forth. This information is also used to restore the values of callee-saved registers in the function catching the exception.

If you are writing functions in assembly code, and those functions call other functions that throw exceptions, you must use assembly pseudo ops to ensure that appropriate exception unwind information is generated. Otherwise, if one of the functions called by your assembly code throws an exception, the run-time library will be unable to unwind the stack through your assembly code and your program will not behave correctly.

To illustrate the use of these pseudo ops, we will examine the code that G++ generates for the following C++ input:

```
void callee (int *);

int
caller ()
{
  int i;
  callee (&i);
  return i;
}
```

This example does not show how to throw or catch an exception from assembly code. That is a much more complex operation and should always be done in a high-level language, such as C++, that directly supports exceptions.

The code generated by one particular version of G++ when compiling the example above is:

```
_Z6callerv:
        .fnstart
.LFB2:
        @ Function supports interworking.
        @ args = 0, pretend = 0, frame = 8
        @ frame_needed = 1, uses_anonymous_args = 0
        stmfd   sp!, {fp, lr}
        .save {fp, lr}
.LCFI0:
        .setfp fp, sp, #4
        add     fp, sp, #4
.LCFI1:
```

```
        .pad #8
        sub     sp, sp, #8
.LCFI2:
        sub     r3, fp, #8
        mov     r0, r3
        bl      _Z6calleePi
        ldr     r3, [fp, #-8]
        mov     r0, r3
        sub     sp, fp, #4
        ldmfd   sp!, {fp, lr}
        bx      lr
.LFE2:
        .fnend
```

Of course, the sequence of instructions varies based on the options you pass to GCC and on the version of GCC in use. The exact instructions are not important since we are focusing on the pseudo ops that are used to generate unwind information.

An important assumption made by the unwinder is that the stack frame does not change during the body of the function. In particular, since we assume that the assembly code does not itself throw an exception, the only point where an exception can be thrown is from a call, such as the `bl` instruction above. At each call site, the same saved registers (including `lr`, which indicates the return address) must be located in the same locations relative to the frame pointer.

The `.fnstart` (see [.fnstart pseudo op], page 109) pseudo op appears immediately before the first instruction of the function while the `.fnend` (see [.fnend pseudo op], page 109) pseudo op appears immediately after the last instruction of the function. These pseudo ops specify the range of the function.

Only the order of the other pseudos ops (e.g., `.setfp` or `.pad`) matters; their exact locations are irrelevant. In the example above, the compiler emits the pseudo ops with particular instructions. That makes it easier to understand the code, but it is not required for correctness. It would work just as well to emit all of the pseudo ops other than `.fnend` in the same order, but immediately after `.fnstart`.

The `.save` (see [.save pseudo op], page 111) pseudo op indicates registers that have been saved to the stack so that they can be restored before the function returns. The argument to the `.save` pseudo op is a list of registers to save. If a register is "callee-saved" (as specified by the ABI) and is modified by the function you are writing, then your code must save the value before it is modified and restore the original value before the function returns. If an exception is thrown, the run-time library restores the values of these registers from their locations on the stack before returning control to the exception handler. (Of course, if an exception is not thrown, the function that contains the `.save` pseudo op restores these registers in the function epilogue, as is done with the `ldmfd` instruction above.)

You do not have to save callee-saved registers at the very beginning of the function and you do not need to use the `.save` pseudo op immediately following the point at which the registers are saved. However, if you modify a callee-saved register, you must save it on the stack before modifying it and before calling any functions which might throw an exception. And, you must use the `.save` pseudo op to indicate that you have done so.

The `.pad` (see [.pad], page 110) pseudo op indicates a modification of the stack pointer that does not save any registers. The argument is the number of bytes (in decimal) that are subtracted from the stack pointer. (On ARM CPUs, the stack grows downwards, so subtracting from the stack pointer increases the size of the stack.)

The `.setfp` (see [.setfp pseudo op], page 111) pseudo op indicates the register that contains the frame pointer. The first argument is the register that is set, which is typically `fp`. The second argument indicates the register from which the frame pointer takes its value. The third argument, if present, is the value (in decimal) added to the register specified by the second argument to compute the value of the frame pointer. You should not modify the frame pointer in the body of the function.

If you do not use a frame pointer, then you should not use the `.setfp` pseudo op. If you do not use a frame pointer, then you should avoid modifying the stack pointer outside of the function prologue. Otherwise, the run-time library will be unable to find saved registers when it is unwinding the stack.

The pseudo ops described above are sufficient for writing assembly code that calls functions which may throw exceptions. If you need to know more about the object-file format used to represent unwind information, you may consult the *Exception Handling ABI for the ARM Architecture* available from `http://infocenter.arm.com`.

9.3 RISC-V Dependent Features

9.3.1 RISC-V Options

The following table lists all available RISC-V specific options.

`-fpic`
`-fPIC` Generate position-independent code

`-fno-pic` Don't generate position-independent code (default)

`-march=ISA`
: Select the base isa, as specified by ISA. For example -march=rv32ima.

`-mabi=ABI`
: Selects the ABI, which is either "ilp32" or "lp64", optionally followed by "f", "d", or "q" to indicate single-precision, double-precision, or quad-precision floating-point calling convention, or none to indicate the soft-float calling convention. Also, "ilp32" can optionally be followed by "e" to indicate the RVE ABI, which is always soft-float.

`-mrelax` Take advantage of linker relaxations to reduce the number of instructions required to materialize symbol addresses. (default)

`-mno-relax`
: Don't do linker relaxations.

9.3.2 RISC-V Directives

The following table lists all available RISC-V specific directives.

`.align` *size-log-2*
: Align to the given boundary, with the size given as log2 the number of bytes to align to.

`.half` *value*
`.word` *value*
`.dword` *value*
: Emits a half-word, word, or double-word value at the current position.

`.dtprelword` *value*
`.dtpreldword` *value*
: Emits a DTP-relative word (or double-word) at the current position. This is meant to be used by the compiler in shared libraries for DWARF debug info for thread local variables.

`.bss` Sets the current section to the BSS section.

`.uleb128` *value*
`.sleb128` *value*
: Emits a signed or unsigned LEB128 value at the current position. This only accepts constant expressions, because symbol addresses can change with relaxation, and we don't support relocations to modify LEB128 values at link time.

`.option` *argument*

Modifies RISC-V specific assembler options inline with the assembly code. This is used when particular instruction sequences must be assembled with a specific set of options. For example, since we relax addressing sequences to shorter GP-relative sequences when possible the initial load of GP must not be relaxed and should be emitted as something like

```
.option push
.option norelax
la gp, __global_pointer$
.option pop
```

in order to produce after linker relaxation the expected

```
auipc gp, %pcrel_hi(__global_pointer$)
addi gp, gp, %pcrel_lo(__global_pointer$)
```

instead of just

```
addi gp, gp, 0
```

It's not expected that options are changed in this manner during regular use, but there are a handful of esoteric cases like the one above where users need to disable particular features of the assembler for particular code sequences. The complete list of option arguments is shown below:

`push`
`pop` Pushes or pops the current option stack. These should be used whenever changing an option in line with assembly code in order to ensure the user's command-line options are respected for the bulk of the file being assembled.

`rvc`
`norvc` Enables or disables the generation of compressed instructions. Instructions are opportunistically compressed by the RISC-V assembler when possible, but sometimes this behavior is not desirable.

`pic`
`nopic` Enables or disables position-independent code generation. Unless you really know what you're doing, this should only be at the top of a file.

`relax`
`norelax` Enables or disables relaxation. The RISC-V assembler and linker opportunistically relax some code sequences, but sometimes this behavior is not desirable.

`.insn` *value*
`.insn` *value*

This directive permits the numeric representation of an instructions and makes the assembler insert the operands according to one of the instruction formats for '`.insn`' (Section 9.3.3 [RISC-V-Formats], page 119). For example, the instruction '`add a0, a1, a2`' could be written as '`.insn r 0x33, 0, 0, a0, a1, a2`'.

`.attribute` *tag*, *value*

Set the object attribute *tag* to *value*.

The *tag* is either an attribute number, or one of the following: `Tag_RISCV_arch`, `Tag_RISCV_stack_align`, `Tag_RISCV_unaligned_access`, `Tag_RISCV_priv_spec`, `Tag_RISCV_priv_spec_minor`, `Tag_RISCV_priv_spec_revision`.

9.3.3 Instruction Formats

The RISC-V Instruction Set Manual Volume I: User-Level ISA lists 12 instruction formats where some of the formats have multiple variants. For the '`.insn`' pseudo directive the assembler recognizes some of the formats. Typically, the most general variant of the instruction format is used by the '`.insn`' directive.

The following table lists the abbreviations used in the table of instruction formats:

opcode — Unsigned immediate or opcode name for 7-bits opcode.

opcode2 — Unsigned immediate or opcode name for 2-bits opcode.

func7 — Unsigned immediate for 7-bits function code.

func6 — Unsigned immediate for 6-bits function code.

func4 — Unsigned immediate for 4-bits function code.

func3 — Unsigned immediate for 3-bits function code.

func2 — Unsigned immediate for 2-bits function code.

rd — Destination register number for operand x, can be GPR or FPR.

rd' — Destination register number for operand x, only accept s0-s1, a0-a5, fs0-fs1 and fa0-fa5.

rs1 — First source register number for operand x, can be GPR or FPR.

rs1' — First source register number for operand x, only accept s0-s1, a0-a5, fs0-fs1 and fa0-fa5.

rs2 — Second source register number for operand x, can be GPR or FPR.

rs2'	Second source register number for operand x, only accept s0-s1, a0-a5, fs0-fs1 and fa0-fa5.
simm12	Sign-extended 12-bit immediate for operand x.
simm20	Sign-extended 20-bit immediate for operand x.
simm6	Sign-extended 6-bit immediate for operand x.
uimm8	Unsigned 8-bit immediate for operand x.
symbol	Symbol or lable reference for operand x.

The following table lists all available opcode name:

`C0`	
`C1`	
`C2`	Opcode space for compressed instructions.
`LOAD`	Opcode space for load instructions.
`LOAD_FP`	Opcode space for floating-point load instructions.
`STORE`	Opcode space for store instructions.
`STORE_FP`	Opcode space for floating-point store instructions.
`AUIPC`	Opcode space for auipc instruction.
`LUI`	Opcode space for lui instruction.
`BRANCH`	Opcode space for branch instructions.
`JAL`	Opcode space for jal instruction.
`JALR`	Opcode space for jalr instruction.
`OP`	Opcode space for ALU instructions.
`OP_32`	Opcode space for 32-bits ALU instructions.
`OP_IMM`	Opcode space for ALU with immediate instructions.
`OP_IMM_32`	Opcode space for 32-bits ALU with immediate instructions.
`OP_FP`	Opcode space for floating-point operation instructions.
`MADD`	Opcode space for madd instruction.
`MSUB`	Opcode space for msub instruction.

`NMADD` Opcode space for nmadd instruction.

`NMSUB` Opcode space for msub instruction.

`AMO` Opcode space for atomic memory operation instructions.

`MISC_MEM` Opcode space for misc instructions.

`SYSTEM` Opcode space for system instructions.

`CUSTOM_0`

`CUSTOM_1`

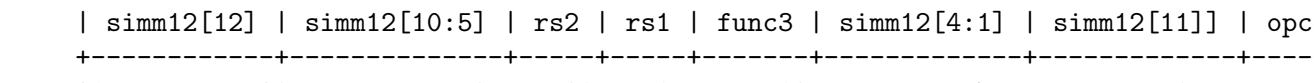
`CUSTOM_2`

`CUSTOM_3` Opcode space for customize instructions.

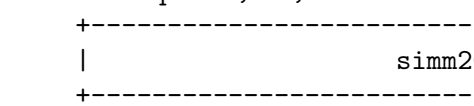
An instruction is two or four bytes in length and must be aligned on a 2 byte boundary. The first two bits of the instruction specify the length of the instruction, 00, 01 and 10 indicates a two byte instruction, 11 indicates a four byte instruction.

The following table lists the RISC-V instruction formats that are available with the '`.insn`' pseudo directive:

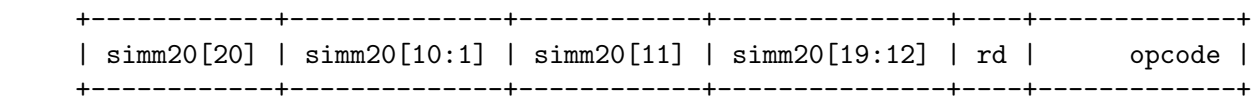
```
R type: .insn r opcode, func3, func7, rd, rs1, rs2
          +-------+-----+-----+-------+----+-------------+
          | func7 | rs2 | rs1 | func3 | rd |      opcode |
          +-------+-----+-----+-------+----+-------------+
          31      25    20    15      12   7             0

R type with 4 register operands: .insn r opcode, func3, func2, rd, rs1, rs2, rs3
R4 type: .insn r4 opcode, func3, func2, rd, rs1, rs2, rs3
          +-----+-------+-----+-----+-------+----+-------------+
          | rs3 | func2 | rs2 | rs1 | func3 | rd |      opcode |
          +-----+-------+-----+-----+-------+----+-------------+
          31    27      25    20    15      12   7             0

I type: .insn i opcode, func3, rd, rs1, simm12
          +-------------+-----+-------+----+-------------+
          |      simm12 | rs1 | func3 | rd |      opcode |
          +-------------+-----+-------+----+-------------+
          31            20    15      12   7             0

S type: .insn s opcode, func3, rd, rs1, simm12
          +--------------+-----+-----+-------+-------------+-------------+
          | simm12[11:5] | rs2 | rs1 | func3 | simm12[4:0] |      opcode |
          +--------------+-----+-----+-------+-------------+-------------+
          31             25    20    15      12            7             0

SB type: .insn sb opcode, func3, rd, rs1, symbol
SB type: .insn sb opcode, func3, rd, simm12(rs1)
B type: .insn s opcode, func3, rd, rs1, symbol
B type: .insn s opcode, func3, rd, simm12(rs1)
          +-----------+-------------+-----+-----+-------+-------------+-------------+----
```



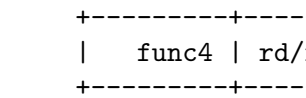
U type: .insn u opcode, rd, simm20



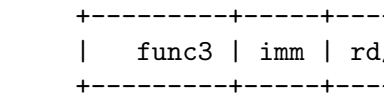
UJ type: .insn uj opcode, rd, symbol
J type: .insn j opcode, rd, symbol



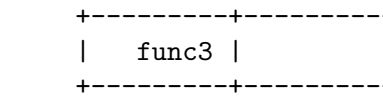
CR type: .insn cr opcode2, func4, rd, rs2



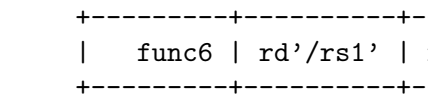
CI type: .insn ci opcode2, func3, rd, simm6



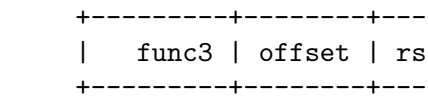
CIW type: .insn ciw opcode2, func3, rd, uimm8



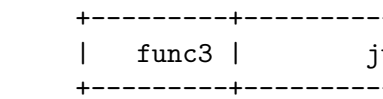
CA type: .insn ca opcode2, func6, func2, rd, rs2



CB type: .insn cb opcode2, func3, rs1, symbol



CJ type: .insn cj opcode2, symbol



For the complete list of all instruction format variants see The RISC-V Instruction Set Manual Volume I: User-Level ISA.

9.3.4 RISC-V Object Attribute

RISC-V attributes have a string value if the tag number is odd and an integer value if the tag number is even.

Tag_RISCV_stack_align (4)
: Tag_RISCV_strict_align records the N-byte stack alignment for this object. The default value is 16 for RV32I or RV64I, and 4 for RV32E.

  The smallest value will be used if object files with different Tag_RISCV_stack_align values are merged.

Tag_RISCV_arch (5)
: Tag_RISCV_arch contains a string for the target architecture taken from the option `-march`. Different architectures will be integrated into a superset when object files are merged.

  Note that the version information of the target architecture must be presented explicitly in the attribute and abbreviations must be expanded. The version information, if not given by `-march`, must be in accordance with the default specified by the tool. For example, the architecture `RV32I` has to be recorded in the attribute as `RV32I2P0` in which `2P0` stands for the default version of its base ISA. On the other hand, the architecture `RV32G` has to be presented as `RV32I2P0_M2P0_A2P0_F2P0_D2P0` in which the abbreviation `G` is expanded to the `IMAFD` combination with default versions of the standard extensions.

Tag_RISCV_unaligned_access (6)
: Tag_RISCV_unaligned_access is 0 for files that do not allow any unaligned memory accesses, and 1 for files that do allow unaligned memory accesses.

Tag_RISCV_priv_spec (8)
Tag_RISCV_priv_spec_minor (10)
Tag_RISCV_priv_spec_revision (12)
: Tag_RISCV_priv_spec contains the major/minor/revision version information of the privileged specification. It will report errors if object files of different privileged specification versions are merged.

10 Reporting Bugs

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

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

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

10.1 Have You Found a Bug?

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

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

10.2 How to Report Bugs

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

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

In any event, we also recommend that you send bug reports for `as` to `http://www.sourceware.org/bugzilla/`.

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

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

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

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" This cannot help us fix a bug, so it is basically useless. We respond by asking for enough details to

enable us to investigate. You might as well expedite matters by sending them to begin with.

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

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

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

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

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

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

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

Here are some things that are not necessary:

- A description of the envelope of the bug.

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

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

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

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

- A patch for the bug.

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

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

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

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

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

11 Acknowledgements

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

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

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

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

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

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

The Intel 80386 machine description was written by Eliot Dresselhaus.

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

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

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

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

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

Ian Lance Taylor of Cygnus Support merged the Motorola and MIT syntax for the 68k, completed support for some COFF targets (68k, i386 SVR3, and SCO Unix), added support for MIPS ECOFF and ELF targets, wrote the initial RS/6000 and PowerPC assembler, and made a few other minor patches.

[1] Any more details?

Steve Chamberlain made GAS able to generate listings.

Hewlett-Packard contributed support for the HP9000/300.

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

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

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

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

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

David Heine, Sterling Augustine, Bob Wilson and John Ruttenberg from Tensilica, Inc. added support for Xtensa processors.

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

Jon Beniston added support for the Lattice Mico32 architecture.

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

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

AS Index

#

$

—

•

:

\

2

4

8

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z