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


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, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.
# Table of Contents

1 Overview ............................................. 1
  1.1 Structure of this Manual ............................ 14
  1.2 The GNU Assembler ................................. 14
  1.3 Object File Formats ................................. 15
  1.4 Command Line ....................................... 15
  1.5 Input Files ........................................ 15
  1.6 Output (Object) File ............................... 16
  1.7 Error and Warning Messages ....................... 16

2 Command-Line Options ................................. 17
  2.1 Enable Listings: ‘-a[cdghlns]’ ...................... 17
  2.2 ‘--alternate’ ...................................... 17
  2.3 ‘-D’ .............................................. 18
  2.4 Work Faster: ‘-f’ .................................. 18
  2.5 .include Search Path: ‘-I path’ ..................... 18
  2.6 Difference Tables: ‘-K’ ............................. 18
  2.7 Include Local Symbols: ‘-L’ ......................... 18
  2.8 Configuring listing output: ‘--listing’ ............ 18
  2.9 Assemble in MRI Compatibility Mode: ‘-M’ ........... 19
  2.10 Dependency Tracking: ‘--MD’ ....................... 21
  2.11 Name the Object File: ‘-o’ ...................... 21
  2.12 Join Data and Text Sections: ‘-R’ .................. 21
  2.13 Display Assembly Statistics: ‘--statistics’ ....... 21
  2.14 Compatible Output: ‘--traditional-format’ ..... 21
  2.15 Announce Version: ‘-v’ .......................... 21
  2.16 Control Warnings: ‘-w’, ‘--warn’, ‘--no-warn’, ‘--fatal-warnings’ .......................... 22
  2.17 Generate Object File in Spite of Errors: ‘-Z’ ........ 22

3 Syntax ................................................. 23
  3.1 Preprocessing ...................................... 23
  3.2 Whitespace ........................................ 23
  3.3 Comments ......................................... 23
  3.4 Symbols ........................................... 24
  3.5 Statements ........................................ 24
  3.6 Constants ......................................... 25
    3.6.1 Character Constants ............................ 25
      3.6.1.1 Strings .................................. 25
      3.6.1.2 Characters ............................... 26
    3.6.2 Number Constants .............................. 26
      3.6.2.1 Integers ................................ 26
      3.6.2.2 Bignums ................................. 27
      3.6.2.3 Flonums ................................. 27
4 Sections and Relocation ........................................ 29
  4.1 Background .................................................. 29
  4.2 Linker Sections ............................................... 30
  4.3 Assembler Internal Sections ................................. 31
  4.4 Sub-Sections ................................................. 31
  4.5 bss Section .................................................. 32

5 Symbols .......................................................... 35
  5.1 Labels ......................................................... 35
  5.2 Giving Symbols Other Values ............................... 35
  5.3 Symbol Names ................................................ 35
  5.4 The Special Dot Symbol .................................... 37
  5.5 Symbol Attributes .......................................... 37
    5.5.1 Value .................................................... 37
    5.5.2 Type .................................................... 37
    5.5.3 Symbol Attributes: a.out ............................. 37
      5.5.3.1 Descriptor ........................................ 38
      5.5.3.2 Other .............................................. 38
    5.5.4 Symbol Attributes for COFF .......................... 38
      5.5.4.1 Primary Attributes ............................... 38
      5.5.4.2 Auxiliary Attributes ............................. 38
    5.5.5 Symbol Attributes for SOM ............................ 38

6 Expressions ..................................................... 39
  6.1 Empty Expressions ......................................... 39
  6.2 Integer Expressions ....................................... 39
    6.2.1 Arguments .............................................. 39
    6.2.2 Operators .............................................. 39
    6.2.3 Prefix Operator ...................................... 39
    6.2.4 Infix Operators ...................................... 40

7 Assembler Directives ........................................... 43
  7.1 .abort ....................................................... 43
  7.2 .ABORT (COFF) ............................................. 43
  7.3 .align abs-expr, abs-expr, abs-expr ..................... 43
  7.4 .altmacro .................................................. 44
  7.5 .ascii "string" ............................................ 44
  7.6 .asciz "string" ............................................ 44
  7.7 .balign[wl] abs-expr, abs-expr, abs-expr ............... 44
  7.8 .byte expressions ......................................... 45
  7.9 .cfi_startproc [simple] .................................. 45
  7.10 .cfi_sections section_list ................................ 45
  7.11 .cfi_endproc .............................................. 45
  7.12 .cfi_personality encoding [, exp] ...................... 45
  7.13 .cfi_lcsa encoding [, exp] ............................... 45
  7.14 .cfi_def_cfa register, offset ........................... 46
  7.15 .cfi_def_cfa_register register ........................... 46
.cfi_def_cfa_offset offset ........................................ 46
.cfi_adjust_cfa_offset offset .................................... 46
.cfi_offset register, offset ...................................... 46
.cfi_rel_offset register, offset ................................ 46
.cfi_register register1, register2 ................................ 46
.cfi_restore register ............................................... 46
.cfi_undefined register ............................................ 46
.cfi_same_value register ........................................... 46
.cfi_reference_state ................................................ 46
.cfi_return_column register ....................................... 47
.cfi_signal_frame ................................................... 47
.cfi_window_save .................................................... 47
.cfi_escape expression[...] ........................................ 47
.cfi_val_encoded_addr register, encoding, label ............ 47
.comm symbol, length .............................................. 47
.data subsection .................................................... 48
.def name ........................................................... 48
.desc symbol, abs-expression ..................................... 48
.dim ................................................................. 48
.double flonums ..................................................... 48
.eject ............................................................... 48
.else ............................................................... 48
.elseif ............................................................ 48
.end ................................................................. 49
.endif ............................................................. 49
.endfunc .......................................................... 49
endif .............................................................. 49
.equ symbol, expression .......................................... 49
.equiv symbol, expression ........................................ 49
.eqv symbol, expression ......................................... 49
.err ............................................................... 49
.error "string" ..................................................... 50
.exitm ............................................................. 50
.extern ............................................................ 50
.fail expression .................................................... 50
.file ............................................................... 50
.fill repeat, size, value ........................................ 51
.float flonums ...................................................... 51
.func name[,label] .................................................. 51
.global symbol, .globl symbol .................................. 51
.gnu_attribute tag, value ......................................... 51
.hidden names ...................................................... 51
.hword expressions ................................................ 52
.ident ............................................................ 52
.if absolute expression .......................................... 52
.inclin "file"[,skip[,count]] ..................................... 53
.include "file" ..................................................... 53
.int expressions .................................................. 54
Using as

7.64 .internal names ........................................ 54
7.65 .irp symbol/values .................................... 54
7.66 .irpc symbol/values .................................. 54
7.67 .lcomm symbol , length ................................. 55
7.68 .lflags .................................................. 55
7.69 .line line-number ...................................... 55
7.70 .linkonce [type]. ..................................... 55
7.71 .list ..................................................... 56
7.72 .ln line-number ........................................ 56
7.73 .loc fileno lineno [column] [options] .......... 56
7.74 .loc_mark_labels enable . .......................... 57
7.75 .local names ............................................ 57
7.76 .long expressions ..................................... 57
7.77 .macro ................................................ 57
7.78 .mri val ............................................... 60
7.79 .noaltmacro ............................................ 60
7.80 .nolist ................................................ 60
7.81 .octa bignums ......................................... 60
7.82 .org new-lc , fill .................................. 60
7.83 .p2align[w] abs-expr , abs-expr , abs-expr ....... 60
7.84 .popsection .......................................... 61
7.85 .previous ............................................. 61
7.86 .print string ......................................... 62
7.87 .protected names ..................................... 62
7.88 .psize lines , columns ................................ 62
7.89 .purgem name ......................................... 62
7.90 .pushsection name [, subsection] [, "flags"[,
   @type [, arguments]]] ................................. 62
7.91 .quad bignums ......................................... 63
7.92 .reloc offset, reloc_name[, expression] ....... 63
7.93 .rept count ........................................... 63
7.94 .sbttl "subheading" .................................. 63
7.95 .scl class ............................................ 64
7.96 .section name ........................................ 64
7.97 .set symbol, expression ............................. 66
7.98 .short expressions .................................. 66
7.99 .single flonums ...................................... 67
7.100 .size ............................................... 67
7.101 .skip size , fill ................................... 67
7.102 .sleb128 expressions ................................ 67
7.103 .space size , fill .................................. 67
7.104 .stab , .stabn , .stabs ................................ 67
7.105 .string "str" , .string8 "str" , .string16 ........ 68
7.106 .struct expression .................................. 69
7.107 .subsection name .................................... 69
7.108 .symver .............................................. 69
7.109 .tag structname .................................... 70
7.110 .text subsection ..................................... 70
9.15.2 Floating Point ........................................ 149
9.15.3 i960 Machine Directives ............................. 149
9.15.4 i960 Opcodes ........................................ 150
  9.15.4.1 callj ........................................ 150
  9.15.4.2 Compare-and-Branch ............................ 150
9.16 IA-64 Dependent Features ............................... 152
  9.16.1 Options ........................................ 152
  9.16.2 Syntax .......................................... 153
    9.16.2.1 Special Characters .......................... 153
    9.16.2.2 Register Names ............................... 153
    9.16.2.3 IA-64 Processor-Status-Register (PSR) Bit Names
                      ........................................ 153
  9.16.3 Opcodes .......................................... 154
9.17 IP2K Dependent Features ................................. 155
  9.17.1 IP2K Options .................................... 155
9.18 LM32 Dependent Features ................................ 156
  9.18.1 Options ........................................ 156
  9.18.2 Syntax .......................................... 156
    9.18.2.1 Register Names ............................... 156
    9.18.2.2 Relocatable Expression Modifiers .......... 157
  9.18.3 Opcodes .......................................... 158
9.19 M32C Dependent Features ................................ 159
  9.19.1 M32C Options .................................... 159
  9.19.2 Symbolic Operand Modifiers ...................... 159
9.20 M32R Dependent Features ................................ 161
  9.20.1 M32R Options .................................... 161
  9.20.2 M32R Directives ................................ 162
  9.20.3 M32R Warnings ................................... 163
9.21 M680x0 Dependent Features ............................... 165
  9.21.1 M680x0 Options .................................. 165
  9.21.2 Syntax .......................................... 168
  9.21.3 Motorola Syntax ................................ 169
  9.21.4 Floating Point ................................... 170
  9.21.5 680x0 Machine Directives ....................... 170
  9.21.6 Opcodes .......................................... 171
    9.21.6.1 Branch Improvement ......................... 171
    9.21.6.2 Special Characters ......................... 172
9.22 M68HC11 and M68HC12 Dependent Features ................. 173
  9.22.1 M68HC11 and M68HC12 Options .................... 173
  9.22.2 Syntax .......................................... 174
  9.22.3 Symbolic Operand Modifiers ...................... 175
  9.22.4 Assembler Directives ............................ 176
  9.22.5 Floating Point ................................... 176
  9.22.6 Opcodes .......................................... 177
    9.22.6.1 Branch Improvement ......................... 177
9.23 MicroBlaze Dependent Features ........................... 178
  9.23.1 Directives ...................................... 178
9.24 MIPS Dependent Features ................................ 179
9.24  Assembler options ........................................... 179
9.24.1  Assembler options .......................................... 179
9.24.2  MIPS ECOFF object code .................................. 183
9.24.3  Directives for debugging information ................... 183
9.24.4  Directives to override the size of symbols ............. 183
9.24.5  Directives to override the ISA level .................... 184
9.24.6  Directives for extending MIPS 16 bit instructions ...... 184
9.24.7  Directive to mark data as an instruction ............... 185
9.24.8  Directives to save and restore options ................. 185
9.24.9  Directives to control and restore options ............. 185
9.24.10 Directives to override floating-point options ......... 186

9.25  MMIX Dependent Features ..................................... 187
9.25.1  Command-line Options .................................... 187
9.25.2  Instruction expansion ...................................... 188
9.25.3  Syntax ...................................................... 188
  9.25.3.1  Special Characters .................................... 188
  9.25.3.2  Symbols ................................................. 189
  9.25.3.3  Register names .......................................... 189
  9.25.3.4  Assembler Directives .................................. 190
9.25.4  Differences to \texttt{mmixal} .................................. 192

9.26  MSP 430 Dependent Features ................................ 194
9.26.1  Options .................................................. 194
9.26.2  Syntax ...................................................... 194
  9.26.2.1  Macros .................................................. 194
  9.26.2.2  Special Characters .................................... 194
  9.26.2.3  Register Names .......................................... 194
  9.26.2.4  Assembler Extensions .................................. 194
9.26.3  Floating Point .............................................. 195
9.26.5  Opcodes .................................................... 196
9.26.6  Profiling Capability ........................................ 196

9.27  PDP-11 Dependent Features ................................ 198
9.27.1  Options .................................................. 198
  9.27.1.1  Code Generation Options ............................... 198
  9.27.1.2  Instruction Set Extension Options .................... 198
  9.27.1.3  CPU Model Options ..................................... 199
  9.27.1.4  Machine Model Options .................................. 199
9.27.2  Assembler Directives ...................................... 200
9.27.3  PDP-11 Assembly Language Syntax .......................... 200
9.27.4  Instruction Naming ........................................ 200
9.27.5  Synthetic Instructions ..................................... 201

9.28  picoJava Dependent Features ................................ 202
9.28.1  Options .................................................. 202

9.29  PowerPC Dependent Features ................................ 203
9.29.1  Options .................................................. 203
9.29.2  PowerPC Assembler Directives .................................. 204

9.30  IBM S/390 Dependent Features ................................ 205
9.30.1  Options .................................................. 205
9.30.2 Special Characters ................................................. 205
9.30.3 Instruction syntax .................................................. 205
  9.30.3.1 Register naming ............................................ 206
9.30.3.2 Instruction Mnemonics ...................................... 206
9.30.3.3 Instruction Operands ........................................ 207
9.30.3.4 Instruction Formats ......................................... 209
9.30.3.5 Instruction Aliases .......................................... 212
9.30.3.6 Instruction Operand Modifier ............................... 215
9.30.3.7 Instruction Marker ........................................... 217
9.30.3.8 Literal Pool Entries ......................................... 217
9.30.4 Assembler Directives ............................................ 218
9.30.5 Floating Point .................................................... 219
9.31 SCORE Dependent Features ........................................ 220
  9.31.1 Options ......................................................... 220
  9.31.2 SCORE Assembler Directives ................................ 220
9.32 Renesas / SuperH SH Dependent Features ......................... 222
  9.32.1 Options ......................................................... 222
  9.32.2 Syntax .......................................................... 222
    9.32.2.1 Special Characters ..................................... 222
    9.32.2.2 Register Names ......................................... 222
    9.32.2.3 Addressing Modes ...................................... 223
  9.32.3 Floating Point ................................................ 223
  9.32.4 SH Machine Directives ...................................... 224
  9.32.5 Opcodes ....................................................... 224
9.33 SuperH SH64 Dependent Features .................................. 225
  9.33.1 Options ......................................................... 225
  9.33.2 Syntax .......................................................... 225
    9.33.2.1 Special Characters ..................................... 225
    9.33.2.2 Register Names ......................................... 226
    9.33.2.3 Addressing Modes ...................................... 226
  9.33.3 SH64 Machine Directives .................................... 226
  9.33.4 Opcodes ....................................................... 227
9.34 SPARC Dependent Features ......................................... 228
  9.34.1 Options ......................................................... 228
  9.34.2 Enforcing aligned data ........................................ 228
  9.34.3 Sparc Syntax .................................................. 229
    9.34.3.1 Special Characters ..................................... 229
    9.34.3.2 Register Names ......................................... 229
    9.34.3.3 Constants ................................................ 231
    9.34.3.4 Relocations ............................................... 232
    9.34.3.5 Size Translations ...................................... 234
  9.34.4 Floating Point ................................................ 235
  9.34.5 Sparc Machine Directives .................................... 235
9.35 TIC54X Dependent Features ....................................... 237
  9.35.1 Options ......................................................... 237
  9.35.2 Blocking ....................................................... 237
  9.35.3 Environment Settings ......................................... 237
  9.35.4 Constants Syntax ............................................... 237
9.35.5 String Substitution ........................................... 237
9.35.6 Local Labels ..................................................... 238
9.35.7 Math Builtins ..................................................... 238
9.35.8 Extended Addressing ........................................... 240
9.35.9 Directives ......................................................... 240
9.35.10 Macros .......................................................... 245
9.35.11 Memory-mapped Registers ................................. 246
9.36 Z80 Dependent Features ........................................ 247
9.36.1 Options .......................................................... 247
9.36.2 Syntax ............................................................ 247
  9.36.2.1 Special Characters ...................................... 247
  9.36.2.2 Register Names ......................................... 248
  9.36.2.3 Case Sensitivity ........................................ 248
9.36.3 Floating Point .................................................. 248
9.36.4 Z80 Assembler Directives .................................... 248
9.36.5 Opcodes ........................................................ 249
9.37 Z8000 Dependent Features ..................................... 250
9.37.1 Options .......................................................... 250
9.37.2 Syntax ............................................................ 250
  9.37.2.1 Special Characters ...................................... 250
  9.37.2.2 Register Names ......................................... 250
  9.37.2.3 Addressing Modes ...................................... 250
9.37.3 Assembler Directives for the Z8000 ......................... 251
9.37.4 Opcodes ........................................................ 252
9.38 VAX Dependent Features ........................................ 252
9.38.1 VAX Command-Line Options ................................. 252
9.38.2 VAX Floating Point ............................................ 253
9.38.3 Vax Machine Directives ....................................... 253
9.38.4 VAX Opcodes ................................................... 253
9.38.5 VAX Branch Improvement ..................................... 254
9.38.6 VAX Operands .................................................. 255
9.38.7 Not Supported on VAX ......................................... 256
9.39 v850 Dependent Features ....................................... 256
9.39.1 Options .......................................................... 256
9.39.2 Syntax ............................................................ 256
  9.39.2.1 Special Characters ...................................... 256
  9.39.2.2 Register Names ......................................... 257
9.39.3 Floating Point .................................................. 259
9.39.4 V850 Machine Directives ..................................... 259
9.39.5 Opcodes ........................................................ 259
9.40 Xtensa Dependent Features .................................... 262
9.40.1 Command Line Options ....................................... 262
9.40.2 Assembler Syntax .............................................. 263
  9.40.2.1 Opcode Names ............................................ 263
  9.40.2.2 Register Names ......................................... 264
9.40.3 Xtensa Optimizations ......................................... 264
  9.40.3.1 Using Density Instructions ............................... 264
  9.40.3.2 Automatic Instruction Alignment ........................ 264
Chapter 1: Overview

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 17.

```
as [-a[cdghlns]=file] [-alternate] [-D]
   [-debug-prefix-map old=new]
   [-defsym sym=val] [-f] [-g] [-gstabs]
   [-K] [-L] [-listing-lhs-width=NUM]
   [-listing-lhs-width2=NUM] [-listing-rhs-width=NUM]
   [-target-help] [target-options]
   [-files ...]
```

Target Alpha options:

```
[-mcpu]
[-mdebug | -no-mdebug]
[-replace | -noreplace]
[-relax] [-g] [-Gsize]
[-F] [-32addr]
```

Target ARC options:

```
[-march[5|6|7|8]]
[-EB|EL]
```

Target ARM options:

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

Target CRIS options:

```
[-underscore | -no-underscore]
[-pic] [-N]
[-emulation=criself | -emulation=crisout]
[-march=v0|v10 | -march=v10 | -march=v32 | -march=common_v10_v32]
```

Target D10V options:

```
[-O]
```

Target D30V options:

```
[-O|-n|-N]
```

Target H8/300 options:

```
[-h-tick-hex]
```

Target i386 options:
...
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 PDP11 options:
[mpic | -mno-pic] [-mall] [-mno-extensions]
[-mextension | -mno-extension]
[-mcpu | -mmachine]

Target picoJava options:
[-mb | -me]

Target PowerPC options:
[mpwrx | mpwr2] [-mpwr | -m601 | -mppc | -mppc32 | -m603 | -m604]
[-m403 | -m405 | -mppc64 | -m620 | -mppc64bridge | -mbooke]
[-mcom | -many | -maltivec | -mvx | -memb]
[-mregnames | -mno-regnames]
[-mrelocatable | -mrelocatable-lib]
[-mlittle | -mlittle-endian | -mbig | -mbig-endian]
[-msolaris | -mno-solaris]

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

Target SCORE options:
[-EB] [-EL] [-FIXDD] [-NWARN]
[-SCORE0] [-SCORE5U] [-SCORE7] [-SCORE3]
[-march=score7] [-march=score3]
[-USE_R1] [-KPIC] [-O0] [-G num] [-V]

Target SPARC options:
[-Av6 | -Ax7 | -Av8 | -Asparcler | -Asparclite]
[-Av8plus | -Av8plusa | -Av9 | -Av9a]
[-xarch=v8plus | -xarch=v8plusa] [-bump]
[-32 | -64]

Target TIC54X options:
[-mfar-to-file <filename>] [-me <filename>]

Target Z80 options:
[-z80] [-r800]
[-ignore-undocumented-instructions] [-Wund]
[-ignore-unportable-instructions] [-Wup]
[-warn-undocumented-instructions] [-Wud]
[-warn-unportable-instructions] [-Wup]
[-forbid-undocumented-instructions] [-Fud]
[ -forbid-unportable-instructions] [-Fup]

**Target Xtensa options:**
- [no-][text-section-literals] [-[no-][absolute_literals]]
- [no-][target-align] [-[no-][longcalls]]
- [no-][transform]
- [rename-section oldname=newname]

@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 44.

-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.

--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 35.

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

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

--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
-v
   Print the `as` version.

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

-W
   Suppress warning messages.

--fatal-warnings
   Treat warnings as errors.

--warn
   Don’t suppress warning messages or treat them as errors.
-w
-w
   Ignored.
-x
-x
   Ignored.
-Z
   Generate an object file even after errors.

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

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

-marc[5]678
   This option selects the core processor variant.

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

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.

-k
    Specify that PIC code has been generated.

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

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

-O
    Optimize output by parallelizing instructions.

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

-O
    Optimize output by parallelizing instructions.

-n
    Warn when nops are generated.

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

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

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

-b
    Add code to collect statistics about branches taken.

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

The following options are available when as is configured for the 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 | -m68090 | -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 co-

For details about the PDP-11 machine dependent features options, see Section 9.27.1 [PDP-11-Options], page 198.

-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.

-mextention | -mno-extension Enable (or disable) a particular instruction set extension.
-mcpu
Enable the instruction set extensions supported by a particular CPU, and disable all other extensions.

-mmachine
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.

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

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

-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-instruction-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 | -Asparcle | -Asparclite
-Av8plus | -Av8plusa | -Av9 | -Av9a
Explicitly select a variant of the SPARC architecture.
‘-Av8plus’ and ‘-Av8plusa’ select a 32 bit environment. ‘-Av9’ and ‘-Av9a’ select a 64 bit environment.

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

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

-bump
Warn when the assembler switches to another architecture.

The following options are available when as is configured for 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
-mips64
-mips64r2
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’, ‘-mips64’, and ‘-mips64r2’ correspond to generic ‘MIPS V’, ‘MIPS32’, ‘MIPS32 Release 2’, ‘MIPS64’, and ‘MIPS64 Release 2’ 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.

-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.

-mips16
-no-mips16

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

-msmartmips
-mno-smartmips

Enables the SmartMIPS extension to the MIPS32 instruction set. This
is equivalent to putting .set 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 in-
structions. ‘-mno-dspr2’ turns off this option.

-mm
-mno-mm

Generate code for the MT Application Specific Extension. This tells the as-
sembler to accept MT instructions. ‘-mno-mm’ turns off this option.
--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.

--emulation=name

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

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

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

-nocpp

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

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

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

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

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

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

-m64

-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, ‘g6’, ‘g6’, ‘z900’, ‘z990’,
 ‘z9-109’, ‘z9-ec’, or ‘z10’.

-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.

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

--text-section-literals | --no-text-section-literals
With ‘--text-section-literals’, literal pools are interspersed in the text
section. The default is ‘--no-text-section-literals’, which places literals in
a separate section in the output file. These options only affect literals referenced
via PC-relative L32R instructions; literals for absolute mode L32R instructions
are handled separately.

--absolute-literals | --no-absolute-literals
Indicate to the assembler whether L32R instructions use absolute or PC-relative
addressing. The default is to assume absolute addressing if the Xtensa processor
includes the absolute L32R addressing option. Otherwise, only the PC-relative
L32R mode can be used.

--target-align | --no-target-align
Enable or disable automatic alignment to reduce branch penalties at the expense
of some code density. The default is ‘--target-align’.

--longcalls | --no-longcalls
Enable or disable transformation of call instructions to allow calls across a
greater range of addresses. The default is ‘--no-longcalls’.

--transform | --no-transform
Enable or disable all assembler transformations of Xtensa instructions. The
default is '--transform'; '--no-transform' should be used only in the rare
cases when the instructions must be exactly as specified in the assembly source.

--rename-section oldname=newname
When generating output sections, rename the oldname section to newname.

The following options are available when as is configured for a Z80 family processor.

-z80 Assemble for Z80 processor.
-r800 Assemble for R800 processor.
-ignore-undocumented-instructions
Ignore undocumented Z80 instructions that also work on R800 without warn-
ing.

-ignore-unportable-instructions
Ignore all undocumented Z80 instructions without warning.

-warn-undocumented-instructions
Issue a warning for undocumented Z80 instructions that also work on R800.

-warn-unportable-instructions
Issue a warning for undocumented Z80 instructions that do not work on R800.

-forbid-undocumented-instructions
Treat all undocumented instructions as errors.

-forbid-unportable-instructions
Treat undocumented Z80 instructions that do not work on R800 as errors.

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
try to introduce the machine architecture; we do not describe the instruction set,
standard mnemonics, registers or addressing modes that are standard to a particular archi-
tecture. 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 77). 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 7.82 [.org], page 60).

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 37.

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 16.

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 7.51 [.file], page 50.

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 a logical file name has been given (see Section 7.51 [.file], page 50) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (see Section 7.69 [.line], page 55) then it is used to calculate the number printed, otherwise the actual line in the current source file is printed. The message text is intended to be self explanatory (in the grand Unix tradition).

Error messages have the format

file_name:NNN:FATAL:Error Message Text

The file name and line number are derived as for warning messages. The actual message text may be rather less explanatory because many of them aren’t supposed to happen.
2 Command-Line Options

This chapter describes command-line options available in all versions of the GNU assembler; see Chapter 9 [Machine Dependencies], page 77, 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 44.
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 23.

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 7.62 [.include], page 53). 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 7.121 [.word], page 72. 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 35. 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 17). 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 7.71 [List], page 56), .title (see Section 7.111 [Title], page 70), .sbttl (see Section 7.94 [Sbttl], page 63), .psize (see Section 7.88 [Psize], page 62), and .eject (see Section 7.36 [Eject], page 48) 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 or the ASM960 (depending upon the configured target) assembler from Microtec Research. The exact nature of the MRI syntax will not be documented here; see the MRI manuals for more information. Note in particular that the handling of macros and macro arguments is somewhat different. The purpose of this option is to permit assembling existing MRI assembler code using as.

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

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

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

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

- IDNT, .ident and NAME pseudo-ops
  The MRI IDNT, .ident and NAME pseudo-ops assign a module name to the output file. This is not supported by other object file formats.
• ORG pseudo-op
The m68k MRI ORG pseudo-op begins an absolute section at a given address. This differs from the usual as .org pseudo-op, which changes the location within the current section. Absolute sections are not supported by other object file formats. The address of a section may be assigned within a linker script.

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

• EBCDIC strings
EBCDIC strings are not supported.

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

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

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

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

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

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

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

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

• .debug pseudo-op
The i960 .debug pseudo-op is not supported.

• .extended pseudo-op
The i960 .extended pseudo-op is not supported.

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

• .optimize pseudo-op
The i960 .optimize pseudo-op is not supported.

• .output pseudo-op
The i960 .output pseudo-op is not supported.
• **.setreal** pseudo-op

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

### 2.10 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 Name the Object File: `--o`

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

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

### 2.12 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 29.)

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**.

### 2.13 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).

### 2.14 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.

### 2.15 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.
2.16 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.

2.17 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 7.62 [.include], page 53). 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 25), 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.

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

```c
/* This sort of comment does not nest. */
```
Anything from the line comment character to the next newline is considered a comment and is ignored. The line comment character is ‘;’ on the ARC; ‘@’ on the ARM; ‘;’ for the H8/300 family; ‘;’ for the HPPA; ‘#’ on the i386 and x86-64; ‘#’ on the i960; ‘;’ for the PDP-11; ‘;’ for picoJava; ‘#’ for Motorola PowerPC; ‘#’ for IBM S/390; ‘#’ for the Sunplus SCORE; ‘!’ for the Renesas / SuperH SH; ‘!’ on the SPARC; ‘!’ on the ip2k; ‘!’ on the m32c; ‘!’ on the m32r; ‘!’ on the 680x0; ‘!’ on the 68HC11 and 68HC12; ‘!’ on the Vax; ‘;’ for the Z80; ‘!’ for the Z8000; ‘!’ on the V850; ‘!’ for Xtensa systems; see Chapter 9 [Machine Dependencies], page 77.

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

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

‘--’;

To be compatible with past assemblers, lines that begin with ‘#’ have a special interpretation. Following the ‘#’ should be an absolute expression (see Chapter 6 [Expressions], page 39): the logical line number of the next line. Then a string (see Section 3.6.1.1 [Strings], page 25) 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 77. No symbol may begin with a digit. Case is significant. There is no length limit: all characters are significant. Symbols are delimited by characters not in that set, or by the beginning of a file (since the source program must end with a newline, the end of a file is not a possible symbol delimiter). See Chapter 5 [Symbols], page 35.

### 3.5 Statements

A statement ends at a newline character (‘\n’) or line separator character. (The line separator is usually ‘;’, unless this conflicts with the comment character; see Chapter 9 [Machine Dependencies], page 77.) The newline or separator character is considered part of the preceding statement. Newlines and separators within character constants are an exception: they do not end statements.

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

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

A label is a symbol immediately followed by a colon (:`). Whitespace before a label or after a colon is permitted, but you may not have whitespace between a label’s symbol and its colon. See Section 5.1 [Labels], page 35.

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\9502841971.693993751E-40 # - pi, a flonum.
```

### 3.6.1 Character Constants

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

#### 3.6.1.1 Strings

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

- `\b` Mnemonic for backspace; for ASCII this is octal code 010.
- `\f` Mnemonic for FormFeed; for ASCII this is octal code 014.
- `\n` Mnemonic for newline; for ASCII this is octal code 012.
- `\r` Mnemonic for carriage-Return; for ASCII this is octal code 015.
- `\t` Mnemonic for horizontal Tab; for ASCII this is octal code 011.
\digit digit digit  
An octal character code. The numeric code is 3 octal digits. For compatibility 
with other Unix systems, 8 and 9 are accepted as digits: for example, \008 has 
the value 010, and \009 the value 011.

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

\  
Represents one ‘\’ character.

\  
Represents one ‘\’ character. Needed in strings to represent this character, 
because an unescaped ‘\’ would end the string.

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

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

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

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

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

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

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

A hexadecimal integer is ‘0x’ or ‘0X’ followed by one or more hexadecimal digits chosen 
from ‘0123456789abcdefABCDEF’.
Integers have the usual values. To denote a negative integer, use the prefix operator ‘-’ discussed under expressions (see Section 6.2.3 [Prefix Operators], page 39).

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, Renesas / SuperH SH, and AMD 29K architectures, the letter must be one of the letters ‘DFPRSX’ (in upper or lower case).

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

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

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

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

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

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

4.1 Background

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

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

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

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

When it generates COFF or ELF output, `as` can also generate whatever other named sections you specify using the `'.section'` directive (see Section 7.96 [.section], page 64). 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

\[(\text{section}) + (\text{offset into section})\]

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 \(\{\text{secname } N\}\) to mean “offset \(N\) into section secname.”

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

The idea of sections is extended to the \textit{undefined} section. Any address whose section is unknown at assembly time is by definition rendered \(\{\text{undefined } U\}\)—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 \textit{undefined}.

By analogy the word \textit{section} is used to describe groups of sections in the linked program. \texttt{ld} puts all partial programs’ text sections in contiguous addresses in the linked program. It is customary to refer to the \textit{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 \texttt{ld}; others are invented for use of \texttt{as} and have no meaning except during assembly.

\section*{4.2 Linker Sections}

\texttt{ld} deals with just four kinds of sections, summarized below.

\begin{itemize}
  \item \textbf{named sections}
  \item \textbf{text section}
  \item \textbf{data section}
  \item \textbf{bss section}
\end{itemize}

These sections hold your program. \texttt{as} and \texttt{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.

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 must not change when relocating. In this sense we speak of absolute addresses being “unrelocatable”: they do not change during relocation.

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

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

Partial program #1:
```
text   data   bss
    ttttt    ddd 00
```

Partial program #2:
```
text   data   bss
       TTT      DDDD 000
```

linked program:
```
text   data   bss
       TTT    ttttt    ddd    DDDD    00000
```

addresses:
0...

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

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

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

4.4 Sub-Sections
Assembled bytes conventionally fall into two sections: text and data. You may have separate groups of data in named sections that you want to end up near to each other in the object file, even though they are not contiguous in the assembler source. `as` allows you to use subsections for this purpose. Within each section, there can be numbered subsections with values from 0 to 8192. Objects assembled into the same subsection go into the object file
together with other objects in the same subsection. For example, a compiler might want to store constants in the text section, but might not want to have them interspersed with the program being assembled. In this case, the compiler could issue a `.text 0` before each section of code being output, and a `.text 1` before each group of constants being output.

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

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

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

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a `.text expression` or a `.data expression` statement. When generating COFF output, you can also use an extra subsection argument with arbitrary named sections: `.section name, expression`. When generating ELF output, you can also use the `.subsection directive (see Section 7.107 [SubSection], page 69) to specify a subsection: `.subsection expression`. Expression should be an absolute expression (see Chapter 6 [Expressions], page 39). 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 7.67 [.lcomm], page 55.

The `.comm` pseudo-op may be used to declare a common symbol, which is another form of uninitialized symbol; see Section 7.30 [.comm], page 47.
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 7.96 [.section], page 64. You may only assemble zero values into the section. Typically the section will only contain symbol definitions and .skip directives (see Section 7.101 [.skip], page 67).
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 39). This is equivalent to using the `.set` directive. See Section 7.97 [.set], page 66. In the same way, using a double equals sign `==` here represents an equivalent of the `.eqv` directive. See Section 7.45 [.eqv], page 49.

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 77. 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`.

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: ‘-L’], page 18) to retain the local symbols in the object files.
Local Labels
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 positive integer). To refer to the most recent previous definition of that label write ‘Nb’, using the same number as when you defined the label. To refer to the next definition of a local label, write ‘Nf’—the ‘b’ stands for “backwards” and the ‘f’ stands for “forwards”.

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

`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 ‘.L6\text{-}C-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 7.33 [.desc], page 48). 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 Greater Than
   <          Is Less Than
   >          Is Greater Than Or Equal To
   >=         Is Less Than Or Equal To

The comparison operators can be used as infix operators. A true result 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.
Chapter 7: Assembler Directives

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

This chapter discusses directives that are available regardless of the target machine configuration for the GNU assembler. Some machine configurations provide additional directives. See Chapter 9 [Machine Dependencies], page 77.

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.

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

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

The way the required alignment is specified varies from system to system. For the arc, hppa, i386 using ELF, i860, iq2000, m68k, or32, s390, sparc, tic4x, tic80 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 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 25) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

7.6 .asciz "string"...

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

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

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment request in bytes. For example ‘.balign 8’ advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

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

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

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

7.8 .byte expressions

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

7.9 .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.

7.10 .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.

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

7.11 .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.

7.12 .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.

7.13 .cfi_lsyna encoding [, exp]

.cfi_lsyna 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), second argument is not present, otherwise second argument should be a constant or a symbol name. The default after .cfi_startproc is .cfi_lsyna 0xff, no LSDA.
7.14 .cfi_def_cfa register, offset

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

7.15 .cfi_def_cfa_register register

.cfie_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.

7.16 .cfi_def_cfa_offset offset

.cfie_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.

7.17 .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.

7.18 .cfi_offset register, offset

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

7.19 .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.

7.20 .cfi_register register1, register2

Previous value of register1 is saved in register register2.

7.21 .cfi_restore register

.cfie_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.

7.22 .cfi_undefined register

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

7.23 .cfi_same_value register

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

7.24 .cfi_remember_state,

First save all current rules for all registers by .cfi_remember_state, then totally screw them up by subsequent .cfi_* directives and when everything is hopelessly bad, use .cfi_restore_state to restore the previous saved state.
7.25 .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.

7.26 .cfi_signal_frame
Mark current function as signal trampoline.

7.27 .cfi_window_save
SPARC register window has been saved.

7.28 .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.

7.29 .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.

7.30 .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 PE1.

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

### 7.31 `.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.

### 7.32 `.def name`

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

### 7.33 `.desc symbol, abs-expression`

This directive sets the descriptor of the symbol (see Section 5.5 [Symbol Attributes], page 37) 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.

### 7.34 `.dim`

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

### 7.35 `.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 77.

### 7.36 `.eject`

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

### 7.37 `.else`

`.else` is part of the `as` support for conditional assembly; see Section 7.60 [`.if`], page 52. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

### 7.38 `.elseif`

`.elseif` is part of the `as` support for conditional assembly; see Section 7.60 [`.if`], page 52. It is shorthand for beginning a new `.if` block that would otherwise fill the entire `.else` section.
7.39 .end
.end marks the end of the assembly file. as does not process anything in the file past the .end directive.

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

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

7.42 .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 7.60 [.if], page 52.

7.43 .equ symbol, expression
This directive sets the value of symbol to expression. It is synonymous with ‘.set’; see Section 7.97 [.set], page 66.

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 7.44 [Equiv], page 49.

7.44 .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.

7.45 .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.

7.46 .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.
7.47 .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 16.

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

7.48 .exitm

Exit early from the current macro definition. See Section 7.77 [Macro], page 57.

7.49 .extern

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

7.50 .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.

7.51 .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.
7.52 .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.

7.53 .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 77.

7.54 .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.

7.55 .global symbol, .globl symbol

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

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

On the HPPA, *.global is not always enough to make it accessible to other partial programs. You may need the HPPA-only .EXPORT directive as well. See Section 9.11.5 [HPPA Assembler Directives], page 128.

7.56 .gnu_attribute tag,value

Record a GNU object attribute for this file. See Chapter 8 [Object Attributes], page 75.

7.57 .hidden names

This is one of the ELF visibility directives. The other two are .internal (see Section 7.64 [.internal], page 54) and .protected (see Section 7.87 [.protected], page 62).

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.

7.58 .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'.

7.59 .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.

7.60 .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 7.42 [.endif], page 49); optionally, you may include code for the alternative condition, flagged by `.else` (see Section 7.37 [.else], page 48). 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.

7.61 .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 17). 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.

7.62 .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 17). Quotation marks are required around file.
7.63 `.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.

7.64 `.internal names`

This is one of the ELF visibility directives. The other two are `.hidden` (see Section 7.57 [.hidden], page 51) and `.protected` (see Section 7.87 [.protected], page 62).

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.

7.65 `.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 7.77 [Macro], page 57.

7.66 `.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@
```
7.67 .lcomm symbol, length

Reserve \textit{length} (an absolute expression) bytes for a local common denoted by \textit{symbol}. The section and value of \textit{symbol} are those of the new local common. The addresses are allocated in the \texttt{bss} section, so that at run-time the bytes start off zeroed. \textit{Symbol} is not declared global (see Section 7.55 [.global], page 51), so is normally not visible to \texttt{ld}.

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

The syntax for .lcomm differs slightly on the HPPA. The syntax is `\texttt{symbol .lcomm, length}`; \textit{symbol} is optional.

7.68 .lflags

\texttt{as} accepts this directive, for compatibility with other assemblers, but ignores it.

7.69 .line line-number

Change the logical line number. \textit{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 \textit{line-number} − 1. One day \texttt{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 \texttt{a.out} or \texttt{b.out} object-code formats, \texttt{as} still recognizes it when producing COFF output, and treats `\texttt{.line}` as though it were the COFF `\texttt{.ln}` if it is found outside a \texttt{.def/.endef} pair.

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

7.70 .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 \textit{type} argument is optional. If specified, it must be one of the following strings. For example:

\begin{verbatim}
    .linkonce same_size
\end{verbatim}

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

7.71 .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 17), the initial value of the listing
counter is one.

7.72 .ln line-number
‘.ln’ is a synonym for ‘.line’.

7.73 .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 true.
  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.

7.74 .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.

7.75 .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 7.67 [Lcomm], page 55) 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 7.30 [Comm], page 47) to define aligned local common data.

7.76 .long expressions
.long is the same as `.int`. See Section 7.63 [.int], page 54.

7.77 .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:

```
.include "macro.asm"

#include "macro.asm"
ex

#include "macro.asm"
```

With that definition, ‘SUM 0,5’ is equivalent to this assembly input:

```
.include "macro.asm"
```

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 7.89 [Purgem], page 62) 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 7.65 [Irp], page 54) and `.irpc` (see Section 7.66 [Irpc], page 54) 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 44.
7.78 .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 19.

7.79 .noaltmacro

Disable alternate macro mode. See Section 7.4 [Altmacro], page 44.

7.80 .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.

7.81 .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.

7.82 .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.

7.83 .p2align[wl] abs-expr, abs-expr, abs-expr

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the number of low-order zero bits the location counter must have after advancement. For example ‘.p2align 3’ advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.
The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

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

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

7.84 .popsection

This is one of the ELF section stack manipulation directives. The others are .section (see Section 7.96 [Section], page 64), .subsection (see Section 7.107 [SubSection], page 69), .pushsection (see Section 7.90 [PushSection], page 62), and .previous (see Section 7.85 [Previous], page 61).

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.

7.85 .previous

This is one of the ELF section stack manipulation directives. The others are .section (see Section 7.96 [Section], page 64), .subsection (see Section 7.107 [SubSection], page 69), .pushsection (see Section 7.90 [PushSection], page 62), and .popsection (see Section 7.84 [PopSection], page 61).

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
```
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.

7.86 .print string

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

7.87 .protected names

This is one of the ELF visibility directives. The other two are .hidden (see Section 7.57 [Hidden], page 51) and .internal (see Section 7.64 [Internal], page 54).

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.

7.88 .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.

7.89 .purgem name

Undefine the macro name, so that later uses of the string will not be expanded. See Section 7.77 [Macro], page 57.

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

This is one of the ELF section stack manipulation directives. The others are .section (see Section 7.96 [Section], page 64), .subsection (see Section 7.107 [SubSection], page 69),
.popsection (see Section 7.84 [PopSection], page 61), and .previous (see Section 7.85 [Previous], page 61).

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 7.96 [Section], page 64) directive.

7.91 .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.

7.92 .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.

7.93 .rept count

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

For example, assembling

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

is equivalent to assembling

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

7.94 .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.
7.95 .scl class

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

7.96 .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
- **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)

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 sub-section number (see Section 4.4 [Sub-Sections], page 31).

ELF Version

This is one of the ELF section stack manipulation directives. The others are .subsection (see Section 7.107 [SubSection], page 69), .pushsection (see Section 7.90 [PushSection], page 62), .popsection (see Section 7.84 [PopSection], page 61), and .previous (see Section 7.85 [Previous], page 61).

For ELF targets, the .section directive is used like this:
The optional flags argument is a quoted string which may contain any combination of the following characters:

- `a`: section is allocatable
- `w`: section is writable
- `x`: section is executable
- `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

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

Many targets only support the first three section types.

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

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 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
- #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.

### 7.97 .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 37).

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

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

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

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

### 7.98 .short expressions

`.short` is normally the same as `.word`. See Section 7.121 [.word], page 72.

In some configurations, however, `.short` and `.word` generate numbers of different lengths. See Chapter 9 [Machine Dependencies], page 77.
7.99 .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 77.

7.100 .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.

7.101 .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’.

7.102 .sleb128 expressions
sleb128 stands for “signed little endian base 128.” This is a compact, variable length rep-
resentation of numbers used by the DWARF symbolic debugging format. See Section 7.113
[.uleb128], page 71.

7.103 .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 Refer-
ence Manual (HP 92432-90001) for the meaning of the .space directive. See
Section 9.11.5 [HPPA Assembler Directives], page 128, for a summary.

7.104 .stabd, .stabn, .stabs
There are three directives that begin ‘.stab’. All emit symbols (see Chapter 5 [Symbols],
page 35), 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, .stabe, 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.

.stabe 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.

7.105 .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 25.

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. */
```
7.106 .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:

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

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.

7.107 .subsection name

This is one of the ELF section stack manipulation directives. The others are .section (see Section 7.96 [Section], page 64), .pushsection (see Section 7.90 [PushSection], page 62), .popsection (see Section 7.84 [PopSection], page 61), and .previous (see Section 7.85 [Previous], page 61).

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.

7.108 .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:

```asm
.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:

```asm
.symver name, name20@nodename
```

When `name` is not defined within the file being assembled, it is treated as `name20@nodename`. When `name` is defined within the file being assembled, the symbol `name`, `name`, will be changed to `name20@nodename`.

### 7.109 .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.

### 7.110 .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.

### 7.111 .title "heading"

Use `heading` as the title (second line, immediately after the source file name and page number) 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.

### 7.112 .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:

```asm
.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:

```asm
.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 Linux targeted assemblers).

- **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 Linux targeted assemblers).

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

### 7.113 .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 7.102 [.sleb128], page 67.

### 7.114 .val addr

This directive, permitted only within .def/.endef pairs, records the address *addr* as the value attribute of a symbol table entry.
7.115 .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.

7.116 .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.

7.117 .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.

7.118 .warning "string"
Similar to the directive .error (see Section 7.47 [.error "string"], page 50), but just emits a warning.

7.119 .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.

7.120 .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.

7.121 .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 77), 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.

### 7.122 Deprecated Directives

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

`.abort`

`.line`
Chapter 8: Object Attributes

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 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 hardware floating-point ABI with 64-bit wide double-precision floating-point registers and 32-bit wide general purpose 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.

8.2 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 \texttt{as} runs. Floating point representations vary as well, and \texttt{as} often supports a few additional directives or command-line options for compatibility with other assemblers on a particular platform. Finally, some versions of \texttt{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.
9.13 80386 Dependent Features

The i386 version as supports both the original Intel 386 architecture in both 16 and 32-bit mode as well as AMD x86-64 architecture extending the Intel architecture to 64-bits.

9.13.1 Options

The i386 version of as has a few machine dependent options:

`--32 | --64`

Select the word size, either 32 bits or 64 bits. Selecting 32-bit implies Intel i386 architecture, while 64-bit implies AMD x86-64 architecture.

These options are only available with the ELF object file format, and require that the necessary BFD support has been included (on a 32-bit platform you have to add `--enable-64-bit-bfd` to configure enable 64-bit usage and use x86-64 as target platform).

`-n`

By default, x86 GAS replaces multiple nop instructions used for alignment within code sections with multi-byte nop instructions such as `leal 0(%esi,1),%esi`. This switch disables the optimization.

`--divide`

On SVR4-derived platforms, the character `'/'` is treated as a comment character, which means that it cannot be used in expressions. The `--divide` option turns `'/'` into a normal character. This does not disable `'/'` at the beginning of a line starting a comment, or affect using `'#'` for starting a comment.

`-march=CPU[+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: i8086, i186, i286, i386, i486, i586, i686, pentium, pentiumpro, pentiumii, pentiummii, pentium4, prescott, nocona, core, core2, corei7, l1om, k6, k6_2, athlon, opteron, k8, amdfam10, generic32 and generic64.

In addition to the basic instruction set, the assembler can be told to accept various extension mnemonics. For example, `--march=i686+sse4+vmx` extends i686 with sse4 and vmx. The following extensions are currently supported: 8087, 287, 387, no87, mmx, nommx, sse, sse2, sse3, sse3, sse4.1, sse4.2, sse4, nosse, avx, noavx, vmx, smx, xsave, aes, pclmul, fma, movbe, ept, clflush, syscall, rdtsscp, 3dnow, 3dnowa, sse4a, sse5, svme, abm and padlock. Note that rather than extending a basic instruction set, the extension mnemonics starting with `no` revoke the respective functionality.

When the `.arch` directive is used with `--march`, the `.arch` directive will take precedent.

`-mtune=CPU`

This option specifies a processor to optimize for. When used in conjunction with the `--march` option, only instructions of the processor specified by the `--march` option will be generated.

Valid `CPU` values are identical to the processor list of `--march=CPU`.
-msse2avx
This option specifies that the assembler should encode SSE instructions with VEX prefix.

-msse-check=none
-msse-check=warning
-msse-check=error
These options control if the assembler should check SSE instructions. ‘-msse-check=none’ will make the assembler not to check SSE instructions, which is the default. ‘-msse-check=warning’ will make the assembler issue a warning for any SSE instruction. ‘-msse-check=error’ will make the assembler issue an error for any SSE instruction.

-mmemonic=att
-mmemonic=intel
This option specifies instruction mnemonic for matching instructions. The .att_mnemonic and .intel_mnemonic directives will take precedent.

-msyntax=att
-msyntax=intel
This option specifies instruction syntax when processing instructions. The .att_syntax and .intel_syntax directives will take precedent.

-mnaked-reg
This option specifies that registers don’t require a ‘%’ prefix. The .att_syntax and .intel_syntax directives will take precedent.

9.13.2 x86 specific Directives
.lcomm symbol, length[, alignment]
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. Since symbol is not declared global, it is normally not visible to ld. The optional third parameter, alignment, specifies the desired alignment of the symbol in the bss section.

This directive is only available for COFF based x86 targets.

9.13.3 AT&T Syntax versus Intel Syntax
as now supports assembly using Intel assembler syntax. .intel_syntax selects Intel mode, and .att_syntax switches back to the usual AT&T mode for compatibility with the output of gcc. Either of these directives may have an optional argument, prefix, or noprefix specifying whether registers require a ‘%’ prefix. AT&T System V/386 assembler syntax is quite different from Intel syntax. We mention these differences because almost all 80386 documents use Intel syntax. Notable differences between the two syntaxes are:

- AT&T immediate operands are preceded by ‘$’; Intel immediate operands are undelimited (Intel ‘push 4’ is AT&T ‘pushl $4’). AT&T register operands are preceded by ‘%’; Intel register operands are undelimited. AT&T absolute (as opposed to PC relative) jump/call operands are prefixed by ‘*’; they are undelimited in Intel syntax.
• AT&T and Intel syntax use the opposite order for source and destination operands. Intel ‘add eax, 4’ is ‘addl $4, %eax’. The ‘source, dest’ convention is maintained for compatibility with previous Unix assemblers. Note that ‘bound’, ‘invlpga’, and instructions with 2 immediate operands, such as the ‘enter’ instruction, do not have reversed order. Section 9.13.13 [i386-Bugs], page 143.

• In AT&T syntax the size of memory operands is determined from the last character of the instruction mnemonic. Mnemonic suffixes of ‘b’, ‘w’, ‘l’ and ‘q’ specify byte (8-bit), word (16-bit), long (32-bit) and quadruple word (64-bit) memory references. Intel syntax accomplishes this by prefixing memory operands (not the instruction mnemonics) with ‘byte ptr’, ‘word ptr’, ‘dword ptr’ and ‘qword ptr’. Thus, Intel ‘mov al, byte ptr foo’ is ‘movb foo, %al’ in AT&T syntax.

• Immediate form long jumps and calls are ‘lcall/ljmp $section, $offset’ in AT&T syntax; the Intel syntax is ‘call/jmp far section:offset’. Also, the far return instruction is ‘lret $stack-adjust’ in AT&T syntax; Intel syntax is ‘ret far stack-adjust’.

• The AT&T assembler does not provide support for multiple section programs. Unix style systems expect all programs to be single sections.

9.13.4 Instruction Naming

Instruction mnemonics are suffixed with one character modifiers which specify the size of operands. The letters ‘b’, ‘w’, ‘l’ and ‘q’ specify byte, word, long and quadruple word operands. If no suffix is specified by an instruction then as tries to fill in the missing suffix based on the destination register operand (the last one by convention). Thus, ‘mov %ax, %bx’ is equivalent to ‘movw %ax, %bx’; also, ‘mov $1, %bx’ is equivalent to ‘movw $1, %bx’. Note that this is incompatible with the AT&T Unix assembler which assumes that a missing mnemonic suffix implies long operand size. (This incompatibility does not affect compiler output since compilers always explicitly specify the mnemonic suffix.)

Almost all instructions have the same names in AT&T and Intel format. There are a few exceptions. The sign extend and zero extend instructions need two sizes to specify them. They need a size to sign/zero extend from and a size to zero extend to. This is accomplished by using two instruction mnemonic suffixes in AT&T syntax. Base names for sign extend and zero extend are ‘movs...’ and ‘movz...’ in AT&T syntax (‘movsx’ and ‘movzx’ in Intel syntax). The instruction mnemonic suffixes are tacked on to this base name, the from suffix before the to suffix. Thus, ‘movsbl %al, %edx’ is AT&T syntax for “move sign extend from %al to %edx.” Possible suffixes, thus, are ‘bl’ (from byte to long), ‘bw’ (from byte to word), ‘wl’ (from word to long), ‘bq’ (from byte to quadruple word), ‘wq’ (from word to quadruple word), and ‘lq’ (from long to quadruple word).

Different encoding options can be specified via optional mnemonic suffix. ‘.s’ suffix swaps 2 register operands in encoding when moving from one register to another.

The Intel-syntax conversion instructions
• ‘cbw’ — sign-extend byte in ‘%al’ to word in ‘%ax’,
• ‘cwde’ — sign-extend word in ‘%ax’ to long in ‘%eax’,
• ‘cwd’ — sign-extend word in ‘%ax’ to long in ‘%dx:%ax’,
• ‘cdq’ — sign-extend dword in ‘%eax’ to quad in ‘%edx:%eax’,
• ‘cdqe’ — sign-extend dword in ‘%eax’ to quad in ‘%rax’ (x86-64 only),
• ‘cqo’ — sign-extend quad in ‘%rax’ to octuple in ‘%rdx:%rax’ (x86-64 only),

are called ‘cbtw’, ‘cwtl’, ‘cwtd’, ‘cltd’, ‘cltq’, and ‘cqto’ in AT&T naming. as accepts either naming for these instructions.

Far call/jump instructions are ‘lcall’ and ‘ljmp’ in AT&T syntax, but are ‘call far’ and ‘jump far’ in Intel convention.

9.13.5 AT&T Mnemonic versus Intel Mnemonic


9.13.6 Register Naming

Register operands are always prefixed with ‘%’. The 80386 registers consist of
• the 8 32-bit registers ‘%eax’ (the accumulator), ‘%ebx’, ‘%ecx’, ‘%edx’, ‘%edi’, ‘%esi’, ‘%ebp’ (the frame pointer), and ‘%esp’ (the stack pointer).
• the 8 16-bit low-ends of these: ‘%ax’, ‘%bx’, ‘%cx’, ‘%dx’, ‘%di’, ‘%si’, ‘%bp’, and ‘%sp’.
• the 8 8-bit registers: ‘%ah’, ‘%al’, ‘%bh’, ‘%bl’, ‘%ch’, ‘%cl’, ‘%dh’, and ‘%dl’ (These are the high-bytes and low-bytes of ‘%ax’, ‘%bx’, ‘%cx’, and ‘%dx’)
• the 6 section registers ‘%cs’ (code section), ‘%ds’ (data section), ‘%ss’ (stack section), ‘%es’, ‘%fs’, and ‘%gs’.
• the 3 processor control registers ‘%cr0’, ‘%cr2’, and ‘%cr3’.
• the 6 debug registers ‘%db0’, ‘%db1’, ‘%db2’, ‘%db3’, ‘%db6’, and ‘%db7’.
• the 2 test registers ‘%tr6’ and ‘%tr7’.
• the 8 floating point register stack ‘%st’ or equivalently ‘%st(0)’, ‘%st(1)’, ‘%st(2)’, ‘%st(3)’, ‘%st(4)’, ‘%st(5)’, ‘%st(6)’, and ‘%st(7)’. These registers are overloaded by 8 MMX registers ‘%mm0’, ‘%mm1’, ‘%mm2’, ‘%mm3’, ‘%mm4’, ‘%mm5’, ‘%mm6’ and ‘%mm7’.
• the 8 SSE registers registers ‘%xmm0’, ‘%xmm1’, ‘%xmm2’, ‘%xmm3’, ‘%xmm4’, ‘%xmm5’, ‘%xmm6’ and ‘%xmm7’.

The AMD x86-64 architecture extends the register set by:
• enhancing the 8 32-bit registers to 64-bit: ‘%rax’ (the accumulator), ‘%rbx’, ‘%rcx’, ‘%rdx’, ‘%rdi’, ‘%rsi’, ‘%rbp’ (the frame pointer), ‘%rsp’ (the stack pointer)
• the 8 extended registers ‘%r8’–‘%r15’.
• the 8 32-bit low ends of the extended registers: ‘%r8d’–‘%r15d’
• the 8 16-bit low ends of the extended registers: ‘%r8w’–‘%r15w’
• the 8 8-bit low ends of the extended registers: ‘%r8b’–‘%r15b’
• the 4 8-bit registers: ‘%sil’, ‘%dil’, ‘%bpl’, ‘%spl’.
• the 8 debug registers: ‘%db8’–‘%db15’.
• the 8 SSE registers: ‘%xmm8’–‘%xmm15’.
9.13.7 Instruction Prefixes

Instruction prefixes are used to modify the following instruction. They are used to repeat string instructions, to provide section overrides, to perform bus lock operations, and to change operand and address sizes. (Most instructions that normally operate on 32-bit operands will use 16-bit operands if the instruction has an “operand size” prefix.) Instruction prefixes are best written on the same line as the instruction they act upon. For example, the ‘scas’ (scan string) instruction is repeated with:

```plaintext
repne scas %es:(%edi),%al
```

You may also place prefixes on the lines immediately preceding the instruction, but this circumvents checks that as does with prefixes, and will not work with all prefixes.

Here is a list of instruction prefixes:


- Operand/Address size prefixes ‘data16’ and ‘addr16’ change 32-bit operands/addresses into 16-bit operands/addresses, while ‘data32’ and ‘addr32’ change 16-bit ones (in a `.code16` section) into 32-bit operands/addresses. These prefixes must appear on the same line of code as the instruction they modify. For example, in a 16-bit `.code16` section, you might write:

```plaintext
addr32 jmpl *(%ebx)
```

- The bus lock prefix ‘lock’ inhibits interrupts during execution of the instruction it precedes. (This is only valid with certain instructions; see a 80386 manual for details).

- The wait for coprocessor prefix ‘wait’ waits for the coprocessor to complete the current instruction. This should never be needed for the 80386/80387 combination.

- The ‘rep’, ‘repe’, and ‘repne’ prefixes are added to string instructions to make them repeat ‘%ecx’ times (‘%cx’ times if the current address size is 16-bits).

- The ‘rex’ family of prefixes is used by x86-64 to encode extensions to i386 instruction set. The ‘rex’ prefix has four bits — an operand size overwrite (64) used to change operand size from 32-bit to 64-bit and X, Y and Z extensions bits used to extend the register set.

You may write the ‘rex’ prefixes directly. The ‘rex64xyz’ instruction emits ‘rex’ prefix with all the bits set. By omitting the 64, x, y or z you may write other prefixes as well. Normally, there is no need to write the prefixes explicitly, since gas will automatically generate them based on the instruction operands.

9.13.8 Memory References

An Intel syntax indirect memory reference of the form

```plaintext
section:[base + index*scale + disp]
```

is translated into the AT&T syntax

```plaintext
section:disp(base, index, scale)
```

where `base` and `index` are the optional 32-bit base and index registers, `disp` is the optional displacement, and `scale`, taking the values 1, 2, 4, and 8, multiplies `index` to calculate the address of the operand. If no `scale` is specified, `scale` is taken to be 1. `section` specifies the optional section register for the memory operand, and may override the default section register (see a 80386 manual for section register defaults). Note that section overrides in
AT&T syntax must be preceded by a ‘%’. If you specify a section override which coincides with the default section register, as does not output any section register override prefixes to assemble the given instruction. Thus, section overrides can be specified to emphasize which section register is used for a given memory operand.

Here are some examples of Intel and AT&T style memory references:

**AT&T**: ‘-4(ebp)’, Intel: ‘[ebp - 4]’
- base is ‘%ebp’; disp is ‘-4’. section is missing, and the default section is used (‘%ss’ for addressing with ‘%ebp’ as the base register). index, scale are both missing.

**AT&T**: ‘foo(,%eax,4)’, Intel: ‘[foo + eax*4]’
- index is ‘%eax’ (scaled by a scale 4); disp is ‘foo’. All other fields are missing. The section register here defaults to ‘%ds’.

**AT&T**: ‘foo(,1)’; Intel: ‘[foo]’
- This uses the value pointed to by ‘foo’ as a memory operand. Note that base and index are both missing, but there is only one ‘,’. This is a syntactic exception.

**AT&T**: ‘%gs:foo’; Intel ‘gs:foo’
- This selects the contents of the variable ‘foo’ with section register section being ‘%gs’.

Absolute (as opposed to PC relative) call and jump operands must be prefixed with ‘*’. If no ‘*’ is specified, as always chooses PC relative addressing for jump/call labels.

Any instruction that has a memory operand, but no register operand, must specify its size (byte, word, long, or quadruple) with an instruction mnemonic suffix (‘b’, ‘w’, ‘l’ or ‘q’, respectively).

The x86-64 architecture adds an RIP (instruction pointer relative) addressing. This addressing mode is specified by using ‘rip’ as a base register. Only constant offsets are valid. For example:

**AT&T**: ‘1234(%rip)’, Intel: ‘[rip + 1234]’
- Points to the address 1234 bytes past the end of the current instruction.

**AT&T**: ‘symbol(%rip)’, Intel: ‘[rip + symbol]’
- Points to the symbol in RIP relative way, this is shorter than the default absolute addressing.

Other addressing modes remain unchanged in x86-64 architecture, except registers used are 64-bit instead of 32-bit.

### 9.13.9 Handling of Jump Instructions

Jump instructions are always optimized to use the smallest possible displacements. This is accomplished by using byte (8-bit) displacement jumps whenever the target is sufficiently close. If a byte displacement is insufficient a long displacement is used. We do not support word (16-bit) displacement jumps in 32-bit mode (i.e. prefixing the jump instruction with the ‘data16’ instruction prefix), since the 80386 insists upon masking ‘%eip’ to 16 bits after the word displacement is added. (See also see Section 9.13.14 [i386-Arch], page 143)
Note that the ‘jcxz’, ‘jecxz’, ‘loop’, ‘loopz’, ‘loope’, ‘loopnz’ and ‘loopne’ instructions only come in byte displacements, so that if you use these instructions (gcc does not use them) you may get an error message (and incorrect code). The AT&T 80386 assembler tries to get around this problem by expanding ‘jcxz foo’ to

```
jcxz cx_zero
jmp cx_nonzero
```

cx_zero: jmp foo
cx_nonzero:

9.13.10 Floating Point

All 80387 floating point types except packed BCD are supported. (BCD support may be added without much difficulty). These data types are 16-, 32-, and 64-bit integers, and single (32-bit), double (64-bit), and extended (80-bit) precision floating point. Each supported type has an instruction mnemonic suffix and a constructor associated with it. Instruction mnemonic suffixes specify the operand’s data type. Constructors build these data types into memory.

- Floating point constructors are ‘.float’ or ‘.single’, ‘.double’, and ‘.tfloat’ for 32-, 64-, and 80-bit formats. These correspond to instruction mnemonic suffixes ‘s’, ‘l’, and ‘t’. ‘t’ stands for 80-bit (ten byte) real. The 80387 only supports this format via the ‘fldt’ (load 80-bit real to stack top) and ‘fstpt’ (store 80-bit real and pop stack) instructions.

- Integer constructors are ‘.word’, ‘.long’ or ‘.int’, and ‘.quad’ for the 16-, 32-, and 64-bit integer formats. The corresponding instruction mnemonic suffixes are ‘s’ (single), ‘l’ (long), and ‘q’ (quad). As with the 80-bit real format, the 64-bit ‘q’ format is only present in the ‘fildq’ (load quad integer to stack top) and ‘fistpq’ (store quad integer and pop stack) instructions.

Register to register operations should not use instruction mnemonic suffixes. ‘fstl %st, %st(1)’ will give a warning, and be assembled as if you wrote ‘fst %st, %st(1)’, since all register to register operations use 80-bit floating point operands. (Contrast this with ‘fstl %st, mem’, which converts ‘%st’ from 80-bit to 64-bit floating point format, then stores the result in the 4 byte location ‘mem’)

9.13.11 Intel’s MMX and AMD’s 3DNow! SIMD Operations

as supports Intel’s MMX instruction set (SIMD instructions for integer data), available on Intel’s Pentium MMX processors and Pentium II processors, AMD’s K6 and K6-2 processors, Cyrix’ M2 processor, and probably others. It also supports AMD’s 3DNow! instruction set (SIMD instructions for 32-bit floating point data) available on AMD’s K6-2 processor and possibly others in the future.

Currently, as does not support Intel’s floating point SIMD, Katmai (KNI).

The eight 64-bit MMX operands, also used by 3DNow!, are called ‘%mm0’, ‘%mm1’, ..., ‘%mm7’. They contain eight 8-bit integers, four 16-bit integers, two 32-bit integers, one 64-bit integer, or two 32-bit floating point values. The MMX registers cannot be used at the same time as the floating point stack.

See Intel and AMD documentation, keeping in mind that the operand order in instructions is reversed from the Intel syntax.
9.13.12 Writing 16-bit Code

While as normally writes only “pure” 32-bit i386 code or 64-bit x86-64 code depending on the default configuration, it also supports writing code to run in real mode or in 16-bit protected mode code segments. To do this, put a `.code16` or `.code16gcc` directive before the assembly language instructions to be run in 16-bit mode. You can switch as back to writing normal 32-bit code with the `.code32` directive.

`.code16gcc` provides experimental support for generating 16-bit code from gcc, and differs from `.code16` in that `call`, `ret`, `enter`, `leave`, `push`, `pop`, `pusha`, `popa`, `pushf`, and `popf` instructions default to 32-bit size. This is so that the stack pointer is manipulated in the same way over function calls, allowing access to function parameters at the same stack offsets as in 32-bit mode. `.code16gcc` also automatically adds address size prefixes where necessary to use the 32-bit addressing modes that gcc generates.

The code which as generates in 16-bit mode will not necessarily run on a 16-bit pre-80386 processor. To write code that runs on such a processor, you must refrain from using any 32-bit constructs which require as to output address or operand size prefixes.

Note that writing 16-bit code instructions by explicitly specifying a prefix or an instruction mnemonic suffix within a 32-bit code section generates different machine instructions than those generated for a 16-bit code segment. In a 32-bit code section, the following code generates the machine opcode bytes ‘66 6a 04’, which pushes the value ‘4’ onto the stack, decrementing `%esp` by 2.

```
pushw $4
```

The same code in a 16-bit code section would generate the machine opcode bytes ‘6a 04’ (i.e., without the operand size prefix), which is correct since the processor default operand size is assumed to be 16 bits in a 16-bit code section.

9.13.13 AT&T Syntax bugs

The UnixWare assembler, and probably other AT&T derived ix86 Unix assemblers, generate floating point instructions with reversed source and destination registers in certain cases. Unfortunately, gcc and possibly many other programs use this reversed syntax, so we’re stuck with it.

For example

```
fsub %st,%st(3)
```

results in ‘%st(3)’ being updated to ‘%st - %st(3)’ rather than the expected ‘%st(3) - %st’. This happens with all the non-commutative arithmetic floating point operations with two register operands where the source register is ‘%st’ and the destination register is ‘%st(i)’.

9.13.14 Specifying CPU Architecture

as may be told to assemble for a particular CPU (sub-)architecture with the .arch cpu_type directive. This directive enables a warning when gas detects an instruction that is not supported on the CPU specified. The choices for cpu_type are:

- ‘i8086’
- ‘i186’
- ‘i286’
- ‘i386’
- ‘i486’
- ‘i586’
- ‘i686’
- ‘pentium’
- ‘pentiumpro’
- ‘pentiumii’
- ‘pentiumiii’
- ‘pentium4’
- ‘prescott’
- ‘nocona’
- ‘core’
- ‘core2’
### 9.13.15 Notes

There is some trickery concerning the `mul` and `imul` instructions that deserves mention. The 16-, 32-, 64- and 128-bit expanding multiplies (base opcode ‘0xf6’; extension 4 for `mul` and 5 for `imul`) can be output only in the one operand form. Thus, ‘imul %ebx, %eax’ does not select the expanding multiply; the expanding multiply would clobber the ‘%edx’ register, and this would confuse gcc output. Use ‘imul %ebx’ to get the 64-bit product in ‘%edx:%eax’.

We have added a two operand form of `imul` when the first operand is an immediate mode expression and the second operand is a register. This is just a shorthand, so that, multiplying ‘%eax’ by 69, for example, can be done with ‘imul $69, %eax’ rather than ‘imul $69, %eax, %eax’.
10 Reporting Bugs

Your bug reports play an essential role in making \texttt{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 \texttt{as} work better. Bug reports are your contribution to the maintenance of \texttt{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 \texttt{as} bug. Reliable assemblers never crash.
- If \texttt{as} produces an error message for valid input, that is a bug.
- If \texttt{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 \texttt{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 \texttt{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 ‘\texttt{etc/SERVICE}’ in the GNU Emacs distribution.

In any event, we also recommend that you send bug reports for \texttt{as} to \url{http://www.sourceforge.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.
- 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.
- 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.

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 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 Ken Raeburn (email address raeburn@cygnus.com).

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

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

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

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

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

The Intel 80386 machine description was written by Eliot Dresselhaus.

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

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

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

Support for the Zilog Z8k and 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
http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

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.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--no-stubs</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--no-target-align</code></td>
<td>command line option, ARM</td>
</tr>
<tr>
<td><code>--no-text-section-literals</code></td>
<td>command line option, ARC</td>
</tr>
<tr>
<td><code>--no-transform</code></td>
<td>command line option, MIPS</td>
</tr>
<tr>
<td><code>--no-underscore</code></td>
<td>command line option, CRIS</td>
</tr>
<tr>
<td><code>--no-warn</code></td>
<td>command line option, M32R</td>
</tr>
<tr>
<td><code>--pcrel</code></td>
<td>command line option, i960</td>
</tr>
<tr>
<td><code>--pic</code></td>
<td>command line option, CRIS</td>
</tr>
<tr>
<td><code>--print-insn-syntax</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>--register-prefix-optional</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>--relax</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>--relax</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--rename-section</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--renesas</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--short-branches</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--small</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--statistics</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--strict-direct-mode</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--target-align</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--text-section-literals</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--traditional-format</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--transform</code></td>
<td>command line option, MMIX</td>
</tr>
<tr>
<td><code>--underscore</code></td>
<td>command line option, CRIS</td>
</tr>
<tr>
<td><code>--warn</code></td>
<td>command line option, VAX/VMS</td>
</tr>
<tr>
<td><code>--32addr</code></td>
<td>command line option, Alpha</td>
</tr>
<tr>
<td><code>--a</code></td>
<td>command line option, i960</td>
</tr>
<tr>
<td><code>--A</code></td>
<td>command line option, i960</td>
</tr>
<tr>
<td><code>--ad</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>--ag</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>--ah</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>--al</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>--an</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>--as</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>Asparclet</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>Asparclite</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>Av6</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>Av8</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>Av9</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>Av9a</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>b</code></td>
<td>command line option, i960</td>
</tr>
<tr>
<td><code>big</code></td>
<td>command line option, M32R</td>
</tr>
<tr>
<td><code>D</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>D</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>eabi=</code></td>
<td>command line option, ARM</td>
</tr>
<tr>
<td><code>EB</code></td>
<td>command line option, ARC</td>
</tr>
<tr>
<td><code>EB</code></td>
<td>command line option, ARM</td>
</tr>
<tr>
<td><code>EB</code></td>
<td>command line option, MIPS</td>
</tr>
<tr>
<td><code>EB</code></td>
<td>command line option, M32R</td>
</tr>
<tr>
<td><code>EL</code></td>
<td>command line option, ARC</td>
</tr>
<tr>
<td><code>EL</code></td>
<td>command line option, ARM</td>
</tr>
<tr>
<td><code>EL</code></td>
<td>command line option, MIPS</td>
</tr>
<tr>
<td><code>EL</code></td>
<td>command line option, M32R</td>
</tr>
<tr>
<td><code>-f</code></td>
<td>command line option, Alpha</td>
</tr>
<tr>
<td><code>-F</code></td>
<td>command line option, Alpha</td>
</tr>
<tr>
<td><code>-g</code></td>
<td>command line option, Alpha</td>
</tr>
<tr>
<td><code>-G</code></td>
<td>command line option, Alpha</td>
</tr>
<tr>
<td><code>-G</code></td>
<td>option (MIPS)</td>
</tr>
<tr>
<td><code>-h</code></td>
<td>option, VAX/VMS</td>
</tr>
<tr>
<td><code>-H</code></td>
<td>option, VAX/VMS</td>
</tr>
<tr>
<td><code>-I</code></td>
<td>option, VAX/VMS</td>
</tr>
<tr>
<td><code>-I</code></td>
<td>path, VAX/VMS</td>
</tr>
<tr>
<td><code>-I</code></td>
<td>option, M32RX</td>
</tr>
<tr>
<td><code>-Ip</code></td>
<td>option, M32RX</td>
</tr>
<tr>
<td><code>-J</code></td>
<td>ignored on VAX</td>
</tr>
<tr>
<td><code>-K</code></td>
<td>command line option, ARM</td>
</tr>
<tr>
<td><code>-KPIC</code></td>
<td>option, M32R</td>
</tr>
<tr>
<td><code>-KPIC</code></td>
<td>option, MIPS</td>
</tr>
<tr>
<td><code>-L</code></td>
<td>option, VAX/VMS</td>
</tr>
<tr>
<td><code>-l</code></td>
<td>option, M680x0</td>
</tr>
<tr>
<td><code>-little</code></td>
<td>option, M32R</td>
</tr>
<tr>
<td><code>-M</code></td>
<td>option, M680x0</td>
</tr>
<tr>
<td><code>-m[no-]68851</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m[no-]68881</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m[no-]div</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m[no-]mac</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m[no-]float</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m[no-]mac</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m[no-]usp</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/03</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/04</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/05</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/10</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/15</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/20</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/21</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/23</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/24</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/34</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/35</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/40</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/44</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/45</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/50</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/53</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/55</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/60</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/70</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/73</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/83</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/84</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/93</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m11/94</code></td>
<td>command line option, M680x0</td>
</tr>
<tr>
<td><code>-m16c</code></td>
<td>option, M16C</td>
</tr>
<tr>
<td><code>-m31</code></td>
<td>option, s390</td>
</tr>
<tr>
<td><code>-m32c</code></td>
<td>option, M32C</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Blackfin syntax</td>
<td>107</td>
</tr>
<tr>
<td>block</td>
<td>251</td>
</tr>
<tr>
<td>branch improvement, M680x0</td>
<td>171</td>
</tr>
<tr>
<td>branch improvement, M68HC11</td>
<td>177</td>
</tr>
<tr>
<td>branch improvement, VAX</td>
<td>254</td>
</tr>
<tr>
<td>branch instructions, relaxation</td>
<td>265</td>
</tr>
<tr>
<td>branch recording, i960</td>
<td>148</td>
</tr>
<tr>
<td>branch statistics table, i960</td>
<td>148</td>
</tr>
<tr>
<td>branch target alignment</td>
<td>264</td>
</tr>
<tr>
<td>break directive, TIC54X</td>
<td>242</td>
</tr>
<tr>
<td>BSD syntax</td>
<td>200</td>
</tr>
<tr>
<td>bss directive, i960</td>
<td>149</td>
</tr>
<tr>
<td>bss directive, TIC54X</td>
<td>240</td>
</tr>
<tr>
<td>bss section</td>
<td>30,32</td>
</tr>
<tr>
<td>bug criteria</td>
<td>271</td>
</tr>
<tr>
<td>bug reports</td>
<td>271</td>
</tr>
<tr>
<td>bugs in assembler</td>
<td>271</td>
</tr>
<tr>
<td>Built-in symbols, CRIS</td>
<td>113</td>
</tr>
<tr>
<td>builtin math functions, TIC54X</td>
<td>238</td>
</tr>
<tr>
<td>builtin subsym functions, TIC54X</td>
<td>245</td>
</tr>
<tr>
<td>bus lock prefixes, i836</td>
<td>140</td>
</tr>
<tr>
<td>bval</td>
<td>251</td>
</tr>
<tr>
<td>byte directive</td>
<td>45</td>
</tr>
<tr>
<td>byte directive, TIC54X</td>
<td>241</td>
</tr>
</tbody>
</table>

C

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_mode directive, TIC54X</td>
<td>241</td>
</tr>
<tr>
<td>'C54XDSP_DIR' environment variable, TIC54X</td>
<td></td>
</tr>
<tr>
<td>call instructions, i386</td>
<td>139</td>
</tr>
<tr>
<td>call instructions, relaxation</td>
<td>266</td>
</tr>
<tr>
<td>call instructions, x86-64</td>
<td>139</td>
</tr>
<tr>
<td>callj, i960 pseudo-opcode</td>
<td>150</td>
</tr>
<tr>
<td>carriage return (x)</td>
<td>25</td>
</tr>
<tr>
<td>case sensitivity, Z80</td>
<td>248</td>
</tr>
<tr>
<td>cfi_endproc directive</td>
<td>45</td>
</tr>
<tr>
<td>cfi_sections directive</td>
<td>45</td>
</tr>
<tr>
<td>cfi_startproc directive</td>
<td>45</td>
</tr>
<tr>
<td>char directive, TIC54X</td>
<td>241</td>
</tr>
<tr>
<td>character constant, Z80</td>
<td>247</td>
</tr>
<tr>
<td>character constants</td>
<td>25</td>
</tr>
<tr>
<td>character escape codes</td>
<td>25</td>
</tr>
<tr>
<td>character escapes, Z80</td>
<td>247</td>
</tr>
<tr>
<td>character, single</td>
<td>26</td>
</tr>
<tr>
<td>characters used in symbols</td>
<td>24</td>
</tr>
<tr>
<td>clink directive, TIC54X</td>
<td>241</td>
</tr>
<tr>
<td>code16 directive, i386</td>
<td>143</td>
</tr>
<tr>
<td>code16gcc directive, i386</td>
<td>143</td>
</tr>
<tr>
<td>code32 directive, i386</td>
<td>143</td>
</tr>
<tr>
<td>code64 directive, i386</td>
<td>143</td>
</tr>
<tr>
<td>code64 directive, x86-64</td>
<td>143</td>
</tr>
<tr>
<td>COFF auxiliary symbol information</td>
<td>48</td>
</tr>
<tr>
<td>COFF structure debugging</td>
<td>70</td>
</tr>
<tr>
<td>COFF symbol attributes</td>
<td>38</td>
</tr>
<tr>
<td>COFF symbol descriptor</td>
<td>48</td>
</tr>
<tr>
<td>COFF symbol storage class</td>
<td>64</td>
</tr>
<tr>
<td>COFF symbol type</td>
<td>70</td>
</tr>
<tr>
<td>COFF symbols, debugging</td>
<td>48</td>
</tr>
<tr>
<td>COFF value attribute</td>
<td>71</td>
</tr>
<tr>
<td>COMDAT</td>
<td>55</td>
</tr>
<tr>
<td>comm directive</td>
<td>47</td>
</tr>
<tr>
<td>command line conventions</td>
<td>15</td>
</tr>
<tr>
<td>command line options, V850</td>
<td>256</td>
</tr>
<tr>
<td>command-line options ignored, VAX</td>
<td>252</td>
</tr>
<tr>
<td>comments</td>
<td>23</td>
</tr>
<tr>
<td>comments, removed by preprocessor</td>
<td>23</td>
</tr>
<tr>
<td>common directive, SPARC</td>
<td>235</td>
</tr>
<tr>
<td>common sections</td>
<td>55</td>
</tr>
<tr>
<td>common variable storage</td>
<td>32</td>
</tr>
<tr>
<td>compare and jump expansions, i960</td>
<td>150</td>
</tr>
<tr>
<td>compare/branch instructions, i960</td>
<td>150</td>
</tr>
<tr>
<td>comparison expressions</td>
<td>40</td>
</tr>
<tr>
<td>conditional assembly</td>
<td>52</td>
</tr>
<tr>
<td>constant, single character</td>
<td>26</td>
</tr>
<tr>
<td>constants</td>
<td>25</td>
</tr>
<tr>
<td>constants, bignum</td>
<td>27</td>
</tr>
<tr>
<td>constants, character</td>
<td>25</td>
</tr>
<tr>
<td>constants, converted by preprocessor</td>
<td>23</td>
</tr>
<tr>
<td>constants, floating point</td>
<td>27</td>
</tr>
<tr>
<td>constants, integer</td>
<td>26</td>
</tr>
<tr>
<td>constants, number</td>
<td>26</td>
</tr>
<tr>
<td>constants, Sparc</td>
<td>231</td>
</tr>
<tr>
<td>constants, string</td>
<td>25</td>
</tr>
<tr>
<td>constants, TIC54X</td>
<td>237</td>
</tr>
<tr>
<td>conversion instructions, i386</td>
<td>138</td>
</tr>
<tr>
<td>conversion instructions, x86-64</td>
<td>138</td>
</tr>
<tr>
<td>coprocessor wait, i386</td>
<td>140</td>
</tr>
<tr>
<td>copy directive, TIC54X</td>
<td>241</td>
</tr>
<tr>
<td>cpu directive, M680x0</td>
<td>170</td>
</tr>
<tr>
<td>CR16 Operand Qualifiers</td>
<td>110</td>
</tr>
<tr>
<td>CR16 support</td>
<td>110</td>
</tr>
<tr>
<td>crash of assembler</td>
<td>271</td>
</tr>
<tr>
<td>CRIS '--emulation=crisainit' command line option</td>
<td>112</td>
</tr>
<tr>
<td>CRIS '--emulation=crisainit' command line option</td>
<td>112</td>
</tr>
<tr>
<td>CRIS '--arch=architecture' command line option</td>
<td>112</td>
</tr>
<tr>
<td>CRIS '--mul=architecture' command line option</td>
<td>112</td>
</tr>
<tr>
<td>CRIS '--no-mul=architecture' command line option</td>
<td>112</td>
</tr>
<tr>
<td>CRIS '--no-underscore=architecture' command line option</td>
<td>112</td>
</tr>
<tr>
<td>CRIS '--no-pic' command line option</td>
<td>112</td>
</tr>
<tr>
<td>CRIS '--underscore' command line option</td>
<td>112</td>
</tr>
<tr>
<td>CRIS '--undertilde' command line option</td>
<td>112</td>
</tr>
<tr>
<td>CRIS architecture variant option</td>
<td>112</td>
</tr>
<tr>
<td>CRIS assembler directive .arch</td>
<td>116</td>
</tr>
<tr>
<td>CRIS assembler directive .dword</td>
<td>115</td>
</tr>
<tr>
<td>CRIS assembler directive .syntax</td>
<td>115</td>
</tr>
<tr>
<td>CRIS assembler directives</td>
<td>115</td>
</tr>
<tr>
<td>CRIS built-in symbols</td>
<td>113</td>
</tr>
<tr>
<td>CRIS instruction expansion</td>
<td>113</td>
</tr>
</tbody>
</table>
CRIS line comment characters .......................... 114
CRIS options ............................................. 112
CRIS position-independent code ......................... 112
CRIS pseudo-op .arch ................................. 116
CRIS pseudo-op .dword ............................... 115
CRIS pseudo-op .syntax ............................... 115
CRIS pseudo-ops ........................................ 115
CRIS register names ..................................... 115
CRIS support .............................................. 112
CRIS symbols in position-independent code ............. 114
tbp register, V850 ..................................... 250
toff pseudo-op, V850 .................................... 256
tpc register, V850 ..................................... 250
tpsw register, V850 .................................... 250
current address ......................................... 37
current address, advancing ................................ 60

d
D10V @word modifier ...................................... 120
D10V addressing modes .................................. 119
D10V floating point ...................................... 120
D10V line comment character ............................ 118
D10V opcode summary ................................... 120
D10V optimization ........................................ 7
D10V options ............................................. 117
D10V registers ............................................ 118
D10V size modifiers ...................................... 117
D10V sub-instruction ordering ......................... 118
D10V sub-instructions ................................... 117
D10V support ............................................. 117
D10V syntax .............................................. 117
D30V addressing modes .................................. 124
D30V floating point ..................................... 124
D30V Guarded Execution ................................ 122
D30V line comment character ............................ 121
D30V nops ................................................ 7
D30V nops after 32-bit multiply ........................ 7
D30V opcode summary ................................... 124
D30V optimization ........................................ 7
D30V options ............................................. 121
D30V registers ............................................ 123
D30V size modifiers ...................................... 121
D30V sub-instruction ordering ......................... 121
D30V sub-instructions ................................... 121
D30V support ............................................. 121
D30V syntax .............................................. 121
data alignment on SPARC ................................ 228
data and text sections, joining .......................... 21
data directive ............................................. 48
data directive, TIC54X ................................... 241
data relocations, ARM .................................... 92
data section .............................................. 30
data1 directive, M680x0 ................................ 170
data2 directive, M680x0 ................................ 170
datalabel, SH64 ........................................... 226
dbpc register, V850 ..................................... 259
dbpsw register, V850 .................................... 259
d debuggers, and symbol order ......................... 35
d debugging COFF symbols ............................. 48
d DEC syntax .............................................. 200
d decimal integers ....................................... 26
def directive ............................................. 48
def directive, TIC54X ................................... 242
density instructions ...................................... 264
dependency tracking ...................................... 21
deprecated directives .................................... 73
desc directive ............................................. 48
descrictor, of a.out symbol .............................. 38
dfloat directive, VAX .................................. 253
difference tables altered ................................ 73
difference tables, warning ............................... 18
differences, mmixal ...................................... 192
dim directive ............................................. 48
directives and instructions .............................. 24
directives for PowerPC ................................. 204
directives for SCORE .................................. 220
directives, Blackfin ..................................... 109
directives, M32R ......................................... 162
directives, M680x0 ...................................... 170
directives, machine independent ....................... 43
directives, Xtensa ....................................... 267
directives, Z8000 ........................................ 251
Disabling floating-point instructions ................. 186
Disable single-precision floating-point operations .... 186
displacement sizing character, VAX .................... 255
dollar local symbols ..................................... 37
dot (symbol) ............................................. 37
double directive ........................................ 48
double directive, i386 ................................ 142
double directive, M680x0 ................................ 170
double directive, M68HC11 ................................ 176
double directive, TIC54X ................................ 241
double directive, VAX .................................. 253
double directive, x86-64 ................................ 142
doublequote (" ) ......................................... 26
drlist directive, TIC54X ................................ 241
drmolist directive, TIC54X .............................. 241
dual directive, i860 ..................................... 145

e
EOFF sections .............................................. 183
ecr register, V850 ....................................... 258
eight-byte integer ....................................... 63
eipc register, V850 ..................................... 258
eipsw register, V850 .................................... 258
eject directive ........................................... 48
ELF symbol type ......................................... 70
e else directive ......................................... 48
e elseif directive ....................................... 48
empty expressions ........................................ 39
e msg directive, TIC54X ................................ 241
emulation .................................................. 12
encoding options, i386 ................................ 138
| gp register, V850                        | 257 |
| grouping data                           | 31  |
|**H**                                    |     |
| H8/300 addressing modes                 | 125 |
| H8/300 floating point (IEEE)             | 126 |
| H8/300 line comment character           | 125 |
| H8/300 line separator                   | 125 |
| H8/300 machine directives (none)         | 127 |
| H8/300 opcode summary                   | 127 |
| H8/300 options                          | 125 |
| H8/300 registers                        | 125 |
| H8/300 size suffixes                    | 127 |
| H8/300 support                          | 125 |
| H8/300H, assembling for                 | 127 |
| half directive, ARC                     | 88  |
| half directive, SPARC                   | 235 |
| half directive, TIC54X                   | 242 |
| hex character code (\xzd)               | 26  |
| hexadecimal integers                    | 26  |
| hexadecimal prefix, Z80                 | 247 |
| hfloat directive, VAX                   | 253 |
| hi pseudo-op, V850                      | 260 |
| hi0 pseudo-op, V850                     | 259 |
| hidden directive                        | 51  |
| high directive, M32R                    | 162 |
| hilo pseudo-op, V850                    | 260 |
| HPPA directives not supported           | 128 |
| HPPA floating point (IEEE)              | 128 |
| HPPA Syntax                             | 128 |
| HPPA-only directives                    | 129 |
| hword directive                         | 52  |
|**I**                                    |     |
| i370 support                            | 133 |
| i386 16-bit code                        | 143 |
| i386 arch directive                     | 143 |
| i386 att_syntax pseudo op               | 137 |
| i386 conversion instructions           | 138 |
| i386 floating point                     | 142 |
| i386 immediate operands                 | 137 |
| i386 instruction naming                 | 138 |
| i386 instruction prefixes               | 140 |
| i386 intel_syntax pseudo op             | 137 |
| i386 jump optimization                  | 141 |
| i386 jump, call, return                 | 138 |
| i386 jump/call operands                 | 137 |
| i386 memory references                  | 140 |
| i386 mnemonic compatibility             | 139 |
| i386 mul, imul instructions             | 144 |
| i386 options                            | 136 |
| i386 register operands                  | 137 |
| i386 registers                          | 139 |
| i386 sections                           | 138 |
| i386 size suffixes                      | 138 |
| i386 source, destination operands       | 137 |
| i386 support                            | 136 |
| i386 syntax compatibility               | 137 |
| i80386 support                          | 136 |
| i800 machine directives                 | 145 |
| i800 opcodes                            | 146 |
| i800 support                            | 145 |
| i800 architecture options               | 148 |
| i800 branch recording                   | 148 |
| i800 callj pseudo-opcode                | 150 |
| i800 compare and jump expansions        | 150 |
| i800 compare/branch instructions        | 150 |
| i800 floating point (IEEE)               | 149 |
| i800 machine directives                 | 149 |
| i800 opcodes                            | 150 |
| i800 options                            | 148 |
| i800 support                            | 148 |
| 1A-64 line comment character            | 153 |
| 1A-64 line separator                    | 153 |
| 1A-64 options                           | 152 |
| 1A-64 Processor-status-Register bit names| 153 |
| 1A-64 registers                         | 153 |
| 1A-64 support                           | 152 |
| 1A-64 Syntax                            | 153 |
| ident directive                         | 52  |
| identifiers, ARM                        | 92  |
| identifiers, MSP 430                    | 194 |
| if directive                            | 52  |
| ifb directive                           | 52  |
| ifc directive                           | 52  |
| ifdef directive                         | 52  |
| ifeq directive                          | 52  |
| ifeqs directive                         | 52  |
| ifge directive                          | 52  |
| ifgt directive                          | 52  |
| ifle directive                          | 53  |
| iflt directive                          | 53  |
| ifnb directive                          | 53  |
| ifnc directive                          | 53  |
| ifndef directive                       | 53  |
| ifne directive                          | 53  |
| ifnes directive                         | 53  |
| ifndefdef directive                    | 53  |
| immediate character, ARM                | 92  |
| immediate character, M680x0              | 172 |
| immediate character, VAX                | 255 |
| immediate fields, relaxation            | 266 |
| immediate operands, i386                | 137 |
| immediate operands, x86-64              | 137 |
| imul instruction, i386                   | 144 |
| imul instruction, x86-64                | 144 |
| incbin directive                        | 53  |
| include directive                       | 53  |
| include directive search path           | 18  |
| indirect character, VAX                 | 255 |
| infix operators                         | 40  |
| inhibiting interrupts, i386              | 140 |
| input                                   | 15  |
| input file linenumbers                  | 16  |
jump optimization, x86-64 141
jump/call operands, i386 137
jump/call operands, x86-64 137

L
L16SI instructions, relaxation 266
L16UI instructions, relaxation 266
L32I instructions, relaxation 266
L8UI instructions, relaxation 266
label () 25
label directive, TIC54X 242
labels 35
lcomm directive 55
lcomm directive, COFF 137
ld 16
ldfloat directive 170
ldfloat directive, M680x0 176
ldfloat directive, M68HC11 176
ldfloat directive, TIC54X 241
LDR reg,=<label> pseudo op, ARM 97
leafproc directive, i960 149
length directive, TIC54X 242
length of symbols 24
lflags directive (ignored) 55
line comment character 23
line comment character, Alpha 79
line comment character, ARM 92
line comment character, AVR 102
line comment character, D10V 118
line comment character, D30V 121
line comment character, H8/300 125
line comment character, IA-64 153
line comment character, M680x0 172
line comment character, MSP 430 194
line comment character, s390 205
line comment character, SH 222
line comment character, SH64 225
line comment character, Sparc 229
line comment character, V850 256
line comment character, Z80 247
line comment character, Z8000 250
line comment characters, CRIS 114
line comment characters, MMIX 188
line directiv 55
line directive, MSP 430 195
line numbers, in input files 16
line numbers, in warnings/errors 16
line separator character 24
line separator, Alpha 79
line separator, ARM 92
line separator, AVR 102
line separator, H8/300 125
line separator, IA-64 153
line separator, SH 222
line separator, SH64 225
line separator, Sparc 229
line separator, Z8000 250
lines starting with # 24
| M32C options | M32C support | M32R architecture options | M32R directives | M32R options | M32R support | M32R warnings | M680x0 addressing modes | M680x0 architecture options | M680x0 branch improvement | M680x0 directives | M680x0 floating point | M680x0 immediate character | M680x0 line comment character | M680x0 opcodes | M680x0 pseudo-opcodes | M680x0 size modifiers | M680x0 support | M680x0 syntax | M68HC11 addressing modes | M68HC11 and M68HC12 support | M68HC11 assembler directive .far | M68HC11 assembler directive .interrupt | M68HC11 assembler directive .mode | M68HC11 assembler directives | M68HC11 branch improvement | M68HC11 floating point | M68HC11 imodifiers | M68HC11 opcodes | M68HC11 options | M68HC11 pseudo-opcodes | M68HC11 syntax | M68HC12 assembler directives | machine dependencies | machine directives, ARC | machine directives, ARM | machine directives, H8/300 (none) | machine directives, i860 | machine directives, i960 | machine directives, MSP 430 | machine directives, SH | machine directives, SH64 | machine directives, SPARC | machine directives, TIC54X | machine directives, VAX | machine directives, x86 | machine independent directives | machine-independent syntax | macro directive | macro directive, TIC54X | macros | macros, count executed | Macros, MSP 430 |
no-schedule directive ...................................... 268
no-transform directive ..................................... 268
nolist directive ............................................. 60
nolist directive, TIC54X ..................................... 242
NGP pseudo op, ARM .......................................... 97
notes for Alpha ............................................... 78
null-terminated strings ...................................... 44
number constants ........................................... 26
number of macros executed ................................. 59
numbered subsections ....................................... 31
numbers, 16-bit ............................................. 52
numeric values ............................................... 39
nword directive, SPARC ....................................... 235

O
object attributes ........................................... 75
object file .................................................. 16
object file format .......................................... 15
object file name ............................................ 21
object file, after errors ................................... 22
obsolete directives .......................................... 73
octa directive ............................................... 60
octal character code ($ddd) ................................ 25
octal integers .............................................. 26
offset directive, V850 ....................................... 259
opcode mnemonics, VAX .................................... 253
opcode names, Xtensa ...................................... 263
opcode summary, AVR ....................................... 103
opcode summary, D10V ..................................... 120
opcode summary, D30V ..................................... 124
opcode summary, H8/300 ................................... 127
opcode summary, LM32 ..................................... 158
opcode summary, SH ......................................... 224
opcode summary, SH64 ..................................... 227
opcode summary, Z8000 .................................... 252
opcodes for ARC ............................................. 88
opcodes for ARM ............................................ 97
opcodes for MSP 430 ...................................... 196
opcodes for V850 ........................................... 259
opcodes, i860 ............................................... 146
opcodes, i960 ............................................... 150
opcodes, M680x0 ............................................ 171
opcodes, M68HC11 ........................................... 177
operand delimiters, i386 ................................... 137
operand delimiters, x86-64 ................................ 137
operand notation, VAX ....................................... 255
operands in expressions .................................... 39
operator precedence ........................................ 40
operators, in expressions ................................... 39
operators, permitted arguments ......................... 40
optimization, D10V ......................................... 7
optimization, D30V ......................................... 7
optimizations ............................................... 264
option directive, ARC ....................................... 88
option directive, TIC54X .................................... 243
option summary ............................................ 1
options for Alpha ........................................... 78
options for ARC (none) ..................................... 85
options for ARM (none) ..................................... 85
options for AVR (none) ..................................... 101
options for Blackfin (none) ............................... 107
options for i386 ............................................. 136
options for IA-64 ........................................... 152
options for LM32 (none) .................................... 156
options for MSP430 (none) .............................. 194
options for PDP-11 .......................................... 198
options for PowerPC ......................................... 203
options for s390 .............................................. 205
options for SCORE ........................................... 220
options for SPARC ......................................... 228
options for V850 (none) ................................... 256
options for VAX/VMS ........................................ 252
options for x86-64 .......................................... 136
options for Z80 .............................................. 247
options, all versions of assembler ....................... 17
options, command line ..................................... 15
options, CRIS ............................................... 112
options, D10V .............................................. 117
options, D30V .............................................. 121
options, H8/300 ............................................ 125
options, i960 ............................................... 148
options, IP2K ............................................... 155
options, M32C ............................................... 159
options, M32R ............................................... 161
options, M680x0 ............................................ 165
options, M68HC11 .......................................... 173
options, MMIX ............................................... 187
options, P.J .................................................. 202
options, SH .................................................. 222
options, SH64 ............................................... 225
options, TIC54X ............................................. 237
options, Z8000 .............................................. 250
org directive ............................................... 60
other attribute, of .out symbol ......................... 38
output file .................................................. 16

P
p2align directive ........................................... 60
p2alignl directive .......................................... 61
p2alignw directive .......................................... 61
padding the location counter .............................. 43
padding the location counter given a power of two . 60
padding the location counter given number of bytes . 44
page, in listings ............................................ 48
paper size, for listings .................................... 62
paths for .include .......................................... 18
patterns, writing in memory ............................... 51
PDP-11 comments ........................................... 200
PDP-11 floating-point register syntax .................. 200
PDP-11 general-purpose register syntax ................ 200
PDP-11 instruction naming ................................ 200
PDP-11 support ............................................. 198
PDP-11 syntax ............................................. 200
PIC code generation for ARM ......................... 91
PIC code generation for M32R .......................... 161
PIC selection, MIPS .................................. 179
PJ endianness ........................................... 9
PJ options ............................................. 202
PJ support ............................................ 202
plus, permitted arguments ............................. 40
popssection directive .................................. 61
Position-independent code, CRIS ...................... 112
Position-independent code, symbols in, CRIS ........ 114
PowerPC architectures ................................ 203
PowerPC directives ................................... 204
PowerPC options ....................................... 203
PowerPC support ...................................... 203
precedence of operators ............................... 40
precision, floating point ............................... 27
prefix operators ....................................... 39
prefixes, i386 .......................................... 140
preprocessing .......................................... 23
preprocessing, turning on and off .................... 23
previous directive ...................................... 61
primary attributes, COFF symbols .................... 38
print directive ......................................... 62
proc directive, SPARC ................................ 235
profiler directive, MSP 430 ............................ 195
profiling capability for MSP 430 ..................... 196
protected directive ..................................... 62
pseudo-op .arch, CRIS ................................ 116
pseudo-op .dword, CRIS ................................ 115
pseudo-op .syntax, CRIS ................................ 115
pseudo-op BSPEC, MMIX ............................... 192
pseudo-op BYTE, MMIX ................................. 191
pseudo-op ESPEC, MMIX ............................... 192
pseudo-op GREG, MMIX ................................. 190
pseudo-op IS, MMIX .................................... 190
pseudo-op LOC, MMIX .................................. 190
pseudo-op LOCAL, MMIX ............................... 190
pseudo-op OCTA, MMIX ................................. 191
pseudo-op PREFIX, MMIX .............................. 192
pseudo-op TETRA, MMIX ............................... 191
pseudo-op WYDE, MMIX ............................... 191
pseudo-opcodes, M680x0 .............................. 171
pseudo-opcodes, M68HC11 ............................ 177
pseudo-ops for branch, VAX .......................... 254
pseudo-ops, CRIS ...................................... 115
pseudo-ops, machine independent .................... 43
pseudo-ops, MMIX ..................................... 190
prize directive ......................................... 62
PSR bits ................................................ 153
pstring directive, TIC54X ............................. 244
psw register, V850 ..................................... 258
purgen directive ........................................ 62
purpose of GNU assembler .............................. 14
pushsection directive .................................. 62
Q
quad directive .......................................... 63
quad directive, i386 .................................... 142
quad directive, x86-64 ................................ 142
R
real-mode code, i386 .................................. 143
ref directive, TIC54X .................................. 242
register directive, SPARC ............................. 236
register names, Alpha .................................. 79
register names, ARC .................................... 85
register names, ARM .................................... 92
register names, AVR .................................... 102
register names, CRIS ................................... 115
register names, H8/300 ................................ 125
register names, IA-64 .................................. 153
register names, LM32 ................................... 156
register names, MMIX ................................... 189
register names, MSP 430 ................................ 194
register names, Sparc ................................... 229
register names, V850 ................................... 257
register names, VAX .................................... 255
register names, Xtensa .................................. 264
register names, Z80 ..................................... 248
register naming, x390 .................................. 206
register operands, i386 ................................ 137
register operands, x86-64 ................................ 137
registers, D10V ......................................... 118
registers, D30V ......................................... 123
registers, i386 .......................................... 139
registers, SH .......................................... 222
registers, SH64 .......................................... 226
registers, TIC54X memory-mapped ..................... 246
registers, x86-64 ....................................... 139
registers, Z8000 ........................................ 250
relaxation .............................................. 265
relaxation of ADDI instructions ....................... 267
relaxation of branch instructions ..................... 265
relaxation of call instructions ......................... 266
relaxation of immediate fields ......................... 266
relaxation of L16SI instructions ....................... 266
relaxation of L16UI instructions ....................... 266
relaxation of L32I instructions ......................... 266
relaxation of L8UI instructions ......................... 266
relaxation of MOV1 instructions ....................... 266
reloc directive ......................................... 63
relocation ............................................. 29
relocation example ..................................... 31
relocations, Alpha ...................................... 79
relocations, Sparc ...................................... 232
repeat prefixes, i386 .................................. 140
reporting bugs in assembler ............................ 271
rept directive .......................................... 63
reserve directive, SPARC .............................. 236
return instructions, i386 ................................ 138
return instructions, x86-64 ............................. 138
REX prefixes, i386 ..................................... 140
<table>
<thead>
<tr>
<th>SH64 line separator</th>
<th>226</th>
</tr>
</thead>
<tbody>
<tr>
<td>SH64 opcode summary</td>
<td>227</td>
</tr>
<tr>
<td>SH64 options</td>
<td>225</td>
</tr>
<tr>
<td>SH64 registers</td>
<td>226</td>
</tr>
<tr>
<td>SH64 support</td>
<td>225</td>
</tr>
<tr>
<td>shhigh directive, M32R</td>
<td>162</td>
</tr>
<tr>
<td>shshort directive</td>
<td>66</td>
</tr>
<tr>
<td>shshort directive, ARC</td>
<td>88</td>
</tr>
<tr>
<td>shshort directive, TIC54X</td>
<td>242</td>
</tr>
<tr>
<td>SIMD, i386</td>
<td>142</td>
</tr>
<tr>
<td>SIMD, x86-64</td>
<td>142</td>
</tr>
<tr>
<td>single character constant</td>
<td>26</td>
</tr>
<tr>
<td>single directive</td>
<td>67</td>
</tr>
<tr>
<td>single directive, i386</td>
<td>142</td>
</tr>
<tr>
<td>single directive, x86-64</td>
<td>142</td>
</tr>
<tr>
<td>single quote, Z80</td>
<td>247</td>
</tr>
<tr>
<td>sixteen bit integers</td>
<td>52</td>
</tr>
<tr>
<td>sixteen byte integer</td>
<td>60</td>
</tr>
<tr>
<td>size directive (COFF version)</td>
<td>67</td>
</tr>
<tr>
<td>size directive (ELF version)</td>
<td>67</td>
</tr>
<tr>
<td>size modifiers, D10V</td>
<td>117</td>
</tr>
<tr>
<td>size modifiers, D30V</td>
<td>121</td>
</tr>
<tr>
<td>size modifiers, M680x0</td>
<td>168</td>
</tr>
<tr>
<td>size prefixes, i386</td>
<td>140</td>
</tr>
<tr>
<td>size suffixes, H8/300</td>
<td>127</td>
</tr>
<tr>
<td>size, translations, Sparc</td>
<td>234</td>
</tr>
<tr>
<td>sizes operands, i386</td>
<td>138</td>
</tr>
<tr>
<td>sizes operands, x86-64</td>
<td>138</td>
</tr>
<tr>
<td>skip directive</td>
<td>67</td>
</tr>
<tr>
<td>skip directive, M680x0</td>
<td>170</td>
</tr>
<tr>
<td>skip directive, SPARC</td>
<td>236</td>
</tr>
<tr>
<td>s1e128 directive</td>
<td>67</td>
</tr>
<tr>
<td>small objects, MIPS ECOFF</td>
<td>183</td>
</tr>
<tr>
<td>SmartMIPS instruction generation override</td>
<td>185</td>
</tr>
<tr>
<td>SOM symbol attributes</td>
<td>38</td>
</tr>
<tr>
<td>source program</td>
<td>15</td>
</tr>
<tr>
<td>source, destination operands; i386</td>
<td>137</td>
</tr>
<tr>
<td>source, destination operands; x86-64</td>
<td>137</td>
</tr>
<tr>
<td>sp register</td>
<td>264</td>
</tr>
<tr>
<td>sp register, V850</td>
<td>257</td>
</tr>
<tr>
<td>space directive</td>
<td>67</td>
</tr>
<tr>
<td>space directive, TIC54X</td>
<td>243</td>
</tr>
<tr>
<td>space used, maximum for assembly</td>
<td>21</td>
</tr>
<tr>
<td>SPARC architectures</td>
<td>228</td>
</tr>
<tr>
<td>Sparc constants</td>
<td>231</td>
</tr>
<tr>
<td>SPARC data alignment</td>
<td>228</td>
</tr>
<tr>
<td>SPARC floating point (IEEE)</td>
<td>235</td>
</tr>
<tr>
<td>Sparc line comment character</td>
<td>229</td>
</tr>
<tr>
<td>Sparc line separator</td>
<td>229</td>
</tr>
<tr>
<td>SPARC machine directives</td>
<td>235</td>
</tr>
<tr>
<td>SPARC options</td>
<td>228</td>
</tr>
<tr>
<td>Sparc registers</td>
<td>229</td>
</tr>
<tr>
<td>Sparc relocations</td>
<td>232</td>
</tr>
<tr>
<td>Sparc size translations</td>
<td>234</td>
</tr>
<tr>
<td>SPARC support</td>
<td>228</td>
</tr>
<tr>
<td>SPARC syntax</td>
<td>228</td>
</tr>
<tr>
<td>special characters, ARC</td>
<td>85</td>
</tr>
<tr>
<td>special characters, M680x0</td>
<td>172</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>as</td>
<td>Assembly language for various microprocessors</td>
</tr>
<tr>
<td>COFF</td>
<td>Common Object File Format</td>
</tr>
<tr>
<td>SOM</td>
<td>Symbol attributes</td>
</tr>
<tr>
<td>symbol descriptor</td>
<td>COFF</td>
</tr>
<tr>
<td>symbol attributes</td>
<td>SOM</td>
</tr>
<tr>
<td>standard assembler sections</td>
<td>29</td>
</tr>
<tr>
<td>standard input, as input file</td>
<td>15</td>
</tr>
<tr>
<td>statement separator character</td>
<td>24</td>
</tr>
<tr>
<td>statement separator, Alpha</td>
<td>79</td>
</tr>
<tr>
<td>statement separator, ARM</td>
<td>92</td>
</tr>
<tr>
<td>statement separator, AVR</td>
<td>102</td>
</tr>
<tr>
<td>statement separator, H8/300</td>
<td>125</td>
</tr>
<tr>
<td>statement separator, IA-64</td>
<td>153</td>
</tr>
<tr>
<td>statement separator, SH</td>
<td>222</td>
</tr>
<tr>
<td>statement separator, SH64</td>
<td>225</td>
</tr>
<tr>
<td>statement separator, Sparc</td>
<td>229</td>
</tr>
<tr>
<td>statement separator, Z8000</td>
<td>250</td>
</tr>
<tr>
<td>statements, structure of</td>
<td>24</td>
</tr>
<tr>
<td>statistics, about assembly</td>
<td>21</td>
</tr>
<tr>
<td>stopping the assembly</td>
<td>43</td>
</tr>
<tr>
<td>string constants</td>
<td>25</td>
</tr>
<tr>
<td>string directive</td>
<td>68</td>
</tr>
<tr>
<td>string directive on HPPA</td>
<td>130</td>
</tr>
<tr>
<td>string directive, TIC54X</td>
<td>244</td>
</tr>
<tr>
<td>string literals</td>
<td>44</td>
</tr>
<tr>
<td>string, copying to object file</td>
<td>68</td>
</tr>
<tr>
<td>string16 directive</td>
<td>68</td>
</tr>
<tr>
<td>string16, copying to object file</td>
<td>68</td>
</tr>
<tr>
<td>string32 directive</td>
<td>68</td>
</tr>
<tr>
<td>string32, copying to object file</td>
<td>68</td>
</tr>
<tr>
<td>string64 directive</td>
<td>68</td>
</tr>
<tr>
<td>string64, copying to object file</td>
<td>68</td>
</tr>
<tr>
<td>string8 directive</td>
<td>68</td>
</tr>
<tr>
<td>string8, copying to object file</td>
<td>68</td>
</tr>
<tr>
<td>struct directive</td>
<td>69</td>
</tr>
<tr>
<td>struct directive, TIC54X</td>
<td>244</td>
</tr>
<tr>
<td>structure debugging, COFF</td>
<td>70</td>
</tr>
<tr>
<td>sub-instruction ordering, D10V</td>
<td>118</td>
</tr>
<tr>
<td>sub-instruction ordering, D30V</td>
<td>121</td>
</tr>
<tr>
<td>sub-instructions, D10V</td>
<td>117</td>
</tr>
<tr>
<td>sub-instructions, D30V</td>
<td>121</td>
</tr>
<tr>
<td>subexpressions</td>
<td>39</td>
</tr>
<tr>
<td>subsection directive</td>
<td>69</td>
</tr>
<tr>
<td>subsym builtins, TIC54X</td>
<td>245</td>
</tr>
<tr>
<td>subtitles for listings</td>
<td>63</td>
</tr>
<tr>
<td>subtraction, permitted arguments</td>
<td>40</td>
</tr>
<tr>
<td>summary of options</td>
<td>1</td>
</tr>
<tr>
<td>support</td>
<td>128</td>
</tr>
<tr>
<td>supporting files, including</td>
<td>53</td>
</tr>
<tr>
<td>suppressing warnings</td>
<td>22</td>
</tr>
<tr>
<td>symbol attributes</td>
<td>37</td>
</tr>
<tr>
<td>symbol attributes, a.out</td>
<td>37</td>
</tr>
<tr>
<td>symbol attributes, COFF</td>
<td>38</td>
</tr>
<tr>
<td>symbol attributes, SOM</td>
<td>38</td>
</tr>
<tr>
<td>symbol descriptor, COFF</td>
<td>48</td>
</tr>
<tr>
<td>symbol modifiers</td>
<td>102, 157, 159, 175</td>
</tr>
<tr>
<td>symbol names</td>
<td>35</td>
</tr>
<tr>
<td>symbol names, $ in</td>
<td>118, 122, 222, 225</td>
</tr>
<tr>
<td>symbol names, local</td>
<td>35</td>
</tr>
<tr>
<td>symbol names, temporary</td>
<td>36</td>
</tr>
<tr>
<td>symbol storage class (COFF)</td>
<td>64</td>
</tr>
<tr>
<td>symbol type</td>
<td>37</td>
</tr>
<tr>
<td>symbol type, COFF</td>
<td>70</td>
</tr>
<tr>
<td>symbol type, ELF</td>
<td>70</td>
</tr>
<tr>
<td>symbol value</td>
<td>37</td>
</tr>
<tr>
<td>symbol value, setting</td>
<td>66</td>
</tr>
<tr>
<td>symbol values, assigning</td>
<td>35</td>
</tr>
<tr>
<td>symbol versioning</td>
<td>69</td>
</tr>
<tr>
<td>symbol, common</td>
<td>47</td>
</tr>
<tr>
<td>symbol, making visible to linker</td>
<td>51</td>
</tr>
<tr>
<td>symbolic debuggers, information for</td>
<td>67</td>
</tr>
<tr>
<td>symbols</td>
<td>35</td>
</tr>
<tr>
<td>Symbols in position-independent code, CRIS</td>
<td>114</td>
</tr>
<tr>
<td>symbols with uppercase, VAX/VMS</td>
<td>252</td>
</tr>
<tr>
<td>symbols, assigning values to</td>
<td>49</td>
</tr>
<tr>
<td>Symbols, built-in, CRIS</td>
<td>113</td>
</tr>
<tr>
<td>Symbols, CRIS, built-in</td>
<td>113</td>
</tr>
<tr>
<td>symbols, local common</td>
<td>55</td>
</tr>
<tr>
<td>symver directive</td>
<td>69</td>
</tr>
<tr>
<td>syntax compatibility, i386</td>
<td>137</td>
</tr>
<tr>
<td>syntax compatibility, x86-64</td>
<td>137</td>
</tr>
<tr>
<td>syntax, AVR</td>
<td>102</td>
</tr>
<tr>
<td>syntax, Blackfin</td>
<td>107</td>
</tr>
<tr>
<td>syntax, D10V</td>
<td>117</td>
</tr>
<tr>
<td>syntax, D30V</td>
<td>121</td>
</tr>
<tr>
<td>syntax, LM32</td>
<td>157</td>
</tr>
<tr>
<td>syntax, M32C</td>
<td>159</td>
</tr>
<tr>
<td>syntax, M680x0</td>
<td>168</td>
</tr>
<tr>
<td>syntax, M68HC11</td>
<td>174, 175</td>
</tr>
<tr>
<td>syntax, machine-independent</td>
<td>23</td>
</tr>
<tr>
<td>syntax, SPARC</td>
<td>228</td>
</tr>
<tr>
<td>syntax, Xtensa assembler</td>
<td>263</td>
</tr>
<tr>
<td>sysproc directive, i960</td>
<td>149</td>
</tr>
</tbody>
</table>

**T**

- tab (\t) | 25
- tab directive, TIC54X | 244
- tag directive | 70
- tag directive, TIC54X | 244
- tdaoff pseudo-op, V850 | 260
- temporary symbol names | 36
- text and data sections, joining | 21
- text directive | 70
- text section | 30
- tfloat directive, i386 | 142
- tfloat directive, x86-64 | 142
- Thumb support | 89
- TIC54X builtin math functions | 238
- TIC54X machine directives | 240
- TIC54X memory-mapped registers | 246
- TIC54X options | 237
- TIC54X subsym builtins | 245
TIC54X support ........................................... 237
TIC54X-specific macros ................................ 245
time, total for assembly ................................ 21
title directive ............................................. 70
tp register, V850 ......................................... 257
transform directive ...................................... 268
trusted compiler ......................................... 18
turning preprocessing on and off ....................... 23
type directive (COFF version) ......................... 70
type directive (ELF version) ............................ 70
type of a symbol ........................................ 37

U
u_long directive, SH ................................... 224
u_word directive, SH ................................... 224
ubyte directive, TIC54X ................................. 241
uchar directive, TIC54X ................................. 241
uhalf directive, TIC54X ................................. 242
uint directive, TIC54X ................................. 242
uleb128 directive .................................... 71
ulong directive, TIC54X ................................. 242
undefined section .................................... 31
union directive, TIC54X ................................. 244
unsegm ................................................. 251
usect directive, TIC54X ................................. 245
ushort directive, TIC54X ................................. 242
u_word directive, TIC54X ................................. 242

V
V850 command line options ............................. 256
V850 floating point (IEEE) ............................... 259
V850 line comment character ............................ 256
V850 machine directives ................................. 259
V850 opcodes ........................................... 259
V850 options (none) .................................... 256
V850 register names ................................... 257
V850 support ............................................ 256
val directive ............................................. 71
value attribute, COFF .................................. 71
value of a symbol ...................................... 37
var directive, TIC54X ................................. 245
VAX bitfields not supported ............................ 256
VAX branch improvement ................................ 254
VAX command-line options ignored .................... 252
VAX displacement sizing character ..................... 255
VAX floating point ...................................... 253
VAX immediate character ................................ 255
VAX indirect character .................................. 255
VAX machine directives ................................ 253
VAX opcode mnemonics ................................ 253
VAX operand notation .................................. 255
VAX register names .................................... 255
VAX support ............................................. 252
VAX-11 C compatibility ................................ 252
VAX/VMS options ....................................... 252
version directive ....................................... 72
version directive, TIC54X ............................. 245
version of assembler .................................. 21
versions of symbols .................................... 69
visibility ............................................... 51, 54, 62
VMS (VAX) options ..................................... 252
vtable_entry directive .................................. 72
vtable_inherit directive ................................ 72

W
warning directive ....................................... 72
warning for altered difference tables ................ 18
warning messages ...................................... 16
warnings, causing error ............................... 22
warnings, M32R ......................................... 163
warnings, suppressing ................................. 22
warnings, switching on ............................... 22
weak directive .......................................... 72
weakref directive ....................................... 72
whitespace .............................................. 23
whitespace, removed by preprocessor ................. 23
wide floating point directives, VAX .................. 253
width directive, TIC54X ............................... 242
Width of continuation lines of disassembly output . 19
Width of first line disassembly output ................ 19
Width of source line output ........................... 19
wmsg directive, TIC54X ............................... 241
word directive .......................................... 72
word directive, ARC .................................. 88
word directive, H8/300 ................................ 127
word directive, i386 ................................... 142
word directive, SPARC ................................. 236
word directive, TIC54X ................................. 242
word directive, x86-64 ................................ 142
writing patterns in memory ............................. 51
wval ............................................... 251

X
x86 machine directives .................................. 137
x86-64 arch directive .................................. 143
x86-64 att_syntax pseudo op ........................... 137
x86-64 conversion instructions ......................... 138
x86-64 floating point .................................. 142
x86-64 immediate operands ............................. 137
x86-64 instruction naming ............................... 138
x86-64 intel_syntax pseudo op ......................... 137
x86-64 jump optimization .............................. 141
x86-64 jump, call, return ................................ 138
x86-64 jump/call operands .............................. 137
x86-64 memory references ............................. 140
x86-64 options .......................................... 136
x86-64 register operands ............................... 137
x86-64 registers .......................................... 139
x86-64 sections .......................................... 138
x86-64 size suffixes .................................... 138
x86-64 source, destination operands ................... 137
x86-64 support ........................................ 136
x86-64 syntax compatibility ...................... 137
xfloat directive, TIC54X ......................... 241
xlong directive, TIC54X .......................... 242
Xtensa architecture ............................... 262
Xtensa assembler syntax ........................... 263
Xtensa directives ................................... 267
Xtensa opcode names ............................... 263
Xtensa register names ............................. 264
xword directive, SPARC ......................... 236

Z
Z80 $ ................................................. 247
Z80 ' ................................................ 247
Z80 floating point .................................. 248
Z80 line comment character ...................... 247

Z80 options ......................................... 247
Z80 registers ........................................ 248
Z80 support .......................................... 247
Z80 Syntax ........................................... 247
Z80, ................................................. 247
Z80, case sensitivity ............................... 248
Z80-only directives ................................. 248
Z800 addressing modes ............................. 250
Z8000 directives .................................... 251
Z8000 line comment character ................. 250
Z8000 line separator ............................... 250
Z8000 opcode summary ............................. 252
Z8000 options ....................................... 250
Z8000 registers ..................................... 250
Z8000 support ....................................... 250
zdaoff pseudo-op, V850 .......................... 261
zero register, V850 ................................ 257
zero-terminated strings ......................... 44