Stretch Assembler

Reference Manual

2006.07

Last modified: 01/30/2007

Part #: RU-0013-0409-000

Version 2.11.2
Dean Elsner and Jay Fenlason

Before Using this Manual

Using st-as

This is a user guide to the GNU assembler `st-as` version 2.11.2. This version of the document describes `st-as` configured to generate code for Xtensa architectures.

This document is distributed under the terms of the GNU Free Documentation License. A copy of the license is included in the section entitled "GNU Free Documentation License".

Changes from Previous Versions

The following changes were made for Xtensa T1050 processors:

- Replaced the `.begin` literal and.end literal directives with a new `.literal` directive. See literal. The old literal directives are still recognized but will eventually be phased out. All new assembly code should use the `.literal` directive.
- Reorganized the chapter on features for Xtensa processors. See Features for Xtensa Processors.
- Upgraded from GNU as version 2.9.1 to version 2.11.2.

Contents

Before Using this Manual

Chapter 1 Overview

1.1 Structure of this Manual 1-4
1.2 The GNU Assembler 1-4
1.3 Object File Formats 1-5
1.4 Command Line 1-5
1.5 Input Files 1-5
1.5.1 File Names and Line Numbers 1-6
1.6 Output (Object) File 1-6
1.7 Error and Warning Messages 1-7

Chapter 2 Command-Line Options

2.1 Enable Listings: -a[cdhlns] 2-1
2.2 -D 2-2
2.3 Work Faster: -f 2-2
2.4 .include Search Path: -I path 2-2
2.5 Difference Tables: -K 2-3
2.6 Include Local Labels: -L 2-3
2.7 Dependency Tracking: --MD 2-3
2.8 Name the Object File: -o 2-4
2.9 Join Data and Text Sections: -R 2-4
2.10 Display Assembly Statistics: --statistics 2-4
2.11 Announce Version: -v 2-4
2.12 Control Warnings: -W, --warn, --no-warn, --fatal-warnings 2-5
2.13 Generate Object File in Spite of Errors: -Z 2-5

Chapter 3 Syntax

3.1 Preprocessing 3-1
3.2 White space 3-2
3.3 Comments 3-2
3.4 Symbols 3-3
3.5 Statements 3-3
3.6 Constants 3-3
3.6.1 Character Constants 3-4
3.6.1.1 Strings 3-4
3.6.1.2 Characters 3-5
3.6.2 Number Constants 3-5
3.6.2.1 Integers 3-5
3.6.2.2 Bignums 3-6
3.6.2.3 Flonums 3-6

Chapter 4 Sections and Relocation

4.1 Background 4-1
4.2 Linker Sections 4-2
4.3 Assembler Internal Sections 4-3

4.4 Subsections 4-4
4.5 bss Section 4-5

Chapter 5 Symbols

5.1 Labels 5-1
5.2 Giving Symbols Other Values 5-1
5.3 Symbol Names 5-1
5.3.1 Local Symbol Names 5-2
5.4 The Special Dot Symbol 5-3
5.5 Symbol Attributes 5-3
5.5.1 Value 5-3
5.5.2 Type 5-3

Chapter 6 Expressions

6.1 Empty Expressions 6-1
6.2 Integer Expressions 6-1
6.2.1 Arguments 6-1
6.2.2 Operators 6-2
6.2.3 Prefix Operator 6-2
6.2.4 Infix Operators 6-2

Chapter 7 Assembler Directives

7.1 .abort 7-1
7.2 .align abs-expr, abs-expr, abs-expr 7-1
7.3 .ascii "string"... 7-2
7.4 .asciz "string"... 7-2
7.5 .balign[wl] abs-expr, abs-expr, abs-expr 7-2
7.6 .byte expressions 7-3
7.7 .comm symbol, length 7-3
7.8 .double flonums 7-4
7.9 .eject 7-4
7.10 .else 7-4
7.11 .elseif 7-4
7.12 .end 7-5
7.13 .endfunc 7-5
7.14 .endif 7-5
7.15 .equ symbol, expression 7-5
7.16 .equiv symbol, expression 7-5
7.17 .err 7-6
7.18 .exitm 7-6
7.19 .extern 7-6
7.20 .fail expression 7-6
7.21 .file string 7-6
7.22 .fill repeat, size, value 7-7
7.23 .float flonums 7-7
7.24 .func name[,label] 7-7
7.25 .global symbol, .global symbol 7-8
7.26 .hidden names 7-8
7.27 .hword expressions 7-8
7.28 .ident 7-8
7.29 .if absolute expression 7-9
7.30 .include "file" 7-10
7.31 .int expressions 7-10
7.32 .internal names 7-10

7.33 .irp symbol,values... 7-11
7.34 .irpc symbol,values.... 7-11
7.35 .lcomm symbol, length 7-12
7.36 .lflags. 7-12
7.37 .line line-number. 7-12
7.38 .ln line-number 7-12
7.39 .list 7-12
7.40 .long expressions. 7-13
7.41 .macro 7-13
7.42 .nolist. 7-13
7.43 .octa bignums 7-14
7.44 .org new-lc, fill 7-15
7.45 .p2align[wl] abs-expr, abs-expr, abs-expr. 7-15
7.46 .previous. 7-16
7.47 .popsection 7-16
7.48 .print string 7-17
7.49 .protected names. 7-17
7.50 .psize lines, columns 7-17
7.51 .purgem name. 7-17
7.52 .pushsection name, subsection 7-18
7.53 .quad bignums 7-18
7.54 .rept count 7-18
7.55 .sbttl "subheading". 7-19
7.56 .section name (ELF version) 7-19
7.57 .set symbol, expression 7-20
7.58 .short expressions 7-20
7.59 .single flonums 7-20
7.60 .size name, expression (ELF Version). 7-20
7.61 .sleb128 expressions 7-20
7.62 .skip size, fill 7-21
7.63 .space size, fill 7-21
7.64 .stabd, .stabn, .stabs 7-21
7.65 .string "str" 7-22
7.66 .struct expression 7-22
7.67 .subsection name. 7-23
7.68 .symver. 7-23
7.69 .text subsection. 7-24
7.70 .title "heading" 7-24
7.71 .type name, type description (ELF Version) 7-24
7.72 .uleb128 expressions. 7-25
7.73 .version "string". 7-25
7.74 .vtable_entry table, offset 7-25
7.75 .vtable_inherit child, parent. 7-25
7.76 .weak names. 7-26
7.77 .word expressions 7-26
7.78 Deprecated Directives 7-26

Chapter 8 Features for Xtensa Processors

8.1 Command Line Options 8-1
8.2 Assembler Syntax 8-3
8.2.1 Opcode Names 8-3
8.2.2 Register Names 8-4
8.3 Xtensa Optimizations 8-4
8.3.1 Using Density Instructions 8-4
8.3.2 Automatic Instruction Alignment 8-5

8.4 Xtensa Relaxation 8-6
8.4.1 Conditional Branch Relaxation 8-6
8.4.2 Function Call Relaxation 8-6
8.4.3 Other Immediate Field Relaxation 8-7
8.5 Directives 8-8
8.5.1 density 8-9
8.5.2 relax 8-9
8.5.3 longcalls 8-10
8.5.4 generics 8-10
8.5.5 literal 8-10
8.5.6 literal_position 8-11
8.5.7 literal_prefix 8-12
8.5.8 freeregs 8-12
8.5.9 frame 8-12

Chapter 9 Acknowledgements

Appendix A GNU Free Documentation License

A.1 Preamble A-1
A.2 Applicability and Definitions A-1
A.3 Verbatim Copying A-3
A.4 Copying in Quantity A-3
A.5 Modifications A-4
A.6 Combining Documents A-6
A.7 Collections of Documents A-6
A.8 Aggregation with Independent Works A-6
A.9 Translation A-7
A.10 Termination A-7
A.11 Future Revisions of This License A-8
A.12 Addendum: How to use this License for your documents A-8

Appendix B History

Index

Tables

Table 3-1 Escape sequences 3-4
Table 4-1 Linker sections 4-3
Table 4-2 Internal sections 4-4
Table 5-1 Local symbol names 5-2
Table 6-1 Highest precedence operators 6-2
Table 6-2 Intermediate precedence operators 6-2
Table 6-3 Low precedence operators 6-3
Table 6-4 Lowest precedence operators 6-3
Table 7-1 Supported variants of .if 7-9
Table 7-2 Macro 7-14
Table 7-3 Special macro variables 7-14
Table 7-4 Optional flag characters 7-19
Table 8-1 Xtensa extensions to the assembler 8-1

Chapter 1 Overview

Here is a brief summary of how to invoke `st-as`. For details, see Chapter 2, "Command-Line Options".

```
st-as [ -a[cdhlns][=file] ]
      [ -D ]
      [ --defsym sym=val ]
      [ -f ]
      [ --gstabs ]
      [ --gdwarf2 ]
      [ --help ]
      [ -I dir ]
      [ -J ]
      [ -K ]
      [ -L ] [ --keep-locals ]
      [ -o objfile ]
      [ -R ]
      [ --statistics ]
      [ -v ] [ -version ]
      [ --version ]
      [ -W ] [ --warn ]
      [ --fatal-warnings ]
      [ -w ]
      [ -x ]
      [ -Z ]
      [ --target-help ]
      [ --[no-]density ]
      [ --[no-]relax ]
      [ --[no-]generics ]
      [ --[no-]text-section-literals ]
      [ --rename-section oldname=newname(
                                  :oldname2=newname2)* ]
      [ --[no-]target-align ]
      [ --[no-]longcalls ]
      [ --xtensa-core=name ]
      [ --xtensa-system=registry ]
      [ --xtensa-params=path ]
      [ -- | files ... ]
```

This option	does this
`-a[cdhlmns]`	Turn on listings, in any of a variety of ways:
`-ac`	omit false conditionals
`-ad`	omit debugging directives
`-ah`	include high-level source
`-al`	include assembly
`-am`	include macro expansions
`-an`	omit forms processing

This option	does this
`-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`.

This option	does this
`-D`	Ignored. This option is accepted for script compatibility with calls to other assemblers.
`--defsym sym=value`	Define the symbol `sym` to be *`value`* before assembling the input file. *`value`* must be an integer constant. As in C, a leading `0x` indicates a hexadecimal value, and a leading `0` indicates an octal value.
`-f`	Fast—skip white space and comment preprocessing (assume source is compiler output).
`--gstabs`	Generate stabs debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it.
`--gdwarf2`	Generate DWARF2 debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it. **NOTE:** This option is only supported by some targets, not all of them. It is not supported for Xtensa targets.
`--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`	This option is accepted, but has no effect on the Xtensa family.
`-L` `--keep-locals`	Keep (in the symbol table) local symbols. On traditional `a.out` systems these start with `L`, but different systems have different local label prefixes.
`-o objfile`	Name the object file output from `st-as` *`objfile`*.
`-R`	Fold the data section into the text section.
`--statistics`	Print the maximum space (in bytes) and total time (in seconds) used by assembly.

This option	does this
`--strip-local-absolute`	Remove local absolute symbols from the outgoing symbol table.
`-v` `-version`	Print the `as` version.
`--version`	Print the `as` version and exit.
`-W` `--no-warn`	Suppress warning messages.
`--fatal-warnings`	Treat warnings as errors.
`--warn`	Don't suppress warning messages or treat them as errors.
`-w`	Ignored
`-x`	Ignored
`-Z`	Generate an object file even after errors.
`-- | files ...`	Standard input, or source files to assemble.

The following options are available when st-as is configured for an Xtensa processor. See Chapter 2, “Command-Line Options”, for details.

This option	does this
`--density | --no-density`	Enable or disable use of instructions from the Xtensa code density option. This is enabled by default when the Xtensa processor supports the code density option.
`--relax | --no-relax`	Enable or disable instruction relaxation. This is enabled by default. **NOTE:** In the current implementation, these options also control whether assembler optimizations are performed, making these options equivalent to `--generics` and `--no-generics`.
`--generics | --no-generics`	Enable or disable all assembler transformations of Xtensa instructions. The default is `--generics`; `--no-generics` should be used only in the rare cases when the instructions must be exactly as specified in the assembly source.
`--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.
`--rename-section` *oldname=newname*`(:`*oldname2=newname2*`)*`	When generating output sections, rename the *`oldname`* section to *`newname`*.
`--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`.

This option	does this
`--longcalls | --no-longcalls`	Enable or disable transformation of call instructions to allow calls across a greater range of addresses. The default is `--no-longcalls`.
`--xtensa-core=name`	Specify the name of an Xtensa processor core configuration to use.
`--xtensa-system=`*registry*	Specify a directory to be used as the Xtensa core registry.
`--xtensa-params=`*path*	Specify the location of the parameter file in a TIE Development Kit (TDK) that was produced by running the TIE Compiler (tc).

1.1 Structure of this Manual

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

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

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

1.2 The GNU Assembler

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

Unlike older assemblers, `st-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 “.org new-lc, fill” on page 7-15).

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 “Symbol Attributes” on page 5-3 For the Xtensa target, `st-as` is configured to produce ELF format object files.

1.4 Command Line

After the program name `st-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, however, is significant.

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

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

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

If you give `st-as` no file names, it attempts to read one input file from the `st-as` standard input, which is normally your terminal. You may have to type `<ctl-D>` to tell `st-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, `st-as` produces a small, empty object file.

1.5.1 File Names 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 "Error and Warning Messages" on page 1-7.

Physical files are those files named in the command line given to `st-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 `st-as` source is itself synthesized from other files. `st-as` understands the # directives emitted by the `st-gcc` preprocessor. See also ".file string" on page 7-6.

1.6 Output (Object) File

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

1.7 Error and Warning Messages

`st-as` may write warnings and error messages to the standard error file (usually your terminal). This should not happen when a compiler runs `st-as` automatically. Warnings report an assumption made so that `st-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 ".file string" on page 7-6) it is used for the file name, otherwise the name of the current input file is used. If a logical line number was given (see ".line line-number" on page 7-12) 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.

Chapter 2 Command-Line Options

This chapter describes command-line options available in *all* versions of the GNU assembler; see Chapter 8, “Features for Xtensa Processors”, for options specific to the Xtensa version.

If you are invoking `st-as` via the GNU C compiler, or if you are using Tensilica's XCC 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:

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

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

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

2.1 Enable Listings: -a[cdhlns]

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

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

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

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

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

2.2 -D

This option has no effect whatsoever, but it is accepted to make it more likely that scripts written for other assemblers also work with `st-as`.

2.3 Work Faster: -f

`-f` should only be used when assembling programs written by a (trusted) compiler. `-f` stops the assembler from doing whitespace and comment preprocessing on the input file(s) before assembling them. See "Preprocessing" on page 3-1.

WARNING! *If you use -f when the files actually need to be preprocessed (if they contain comments, for example), st-as does not work correctly.*

2.4 .include Search Path: -I path

Use this option to add a *path* to the list of directories `st-as` searches for files specified in `.include` directives (see ".include "file"" on page 7-10). 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, `st-as` searches any `-I` directories in the same order as they were specified (left to right) on the command line.

2.5 Difference Tables: -K

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

2.6 Include Local Labels: -L

Labels beginning with `L` (upper case only) are called *local labels*. See "Symbol Names" on page 5-1. Normally you do not see such labels when debugging, because they are intended for the use of programs (like compilers) that compose assembler programs, not for your notice. Normally both `st-as` and `st-ld` discard such labels, so you do not normally debug with them.

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

By default, a local label is any label beginning with `L`, but each target is allowed to redefine the local label prefix.

For Xtensa processors, local labels begin with `.L`. In addition, Xtensa has a class of local debug labels that begin with `.Ln` and `.LM`. These local debug labels do not participate in target alignment when the `--target-align` option is used.

2.7 Dependency Tracking: --MD

`st-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.

2.8 Name the Object File: -o

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

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

2.9 Join Data and Text Sections: -R

`-R` tells `st-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”.)

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

This option is only useful if you use sections named `.text` and `.data`.

2.10 Display Assembly Statistics: --statistics

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

2.11 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.

2.12 Control Warnings: -W, --warn, --no-warn, --fatal-warnings

`st-as` should never give a warning or error message when assembling compiler output. But programs written by people often cause `st-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 `st-as` assembles your file. Errors, which stop the assembly, are still reported.

If you use the `--fatal-warnings` option, `st-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.

2.13 Generate Object File in Spite of Errors: -Z

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

n `errors,` *m* `warnings, generating bad object file.`

NOTE: This option currently does not work reliably for Xtensa targets.

Chapter 3 Syntax

This chapter describes the machine-independent syntax allowed in a source file. `st-as` syntax is similar to what many other assemblers use; it is inspired by the BSD 4.2 assembler.

3.1 Preprocessing

The `st-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 “.include “file”” on page 7-10). You can use the GNU C compiler driver or Tensilica's XCC compiler driver to get other “CPP” style preprocessing by giving the input file a `.S` suffix. See “Options Controlling the Kind of Output” in the *GNU C and C++ Compiler User's Guide*.

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 White space

White space is one or more blanks or tabs, in any order. White space is used to separate symbols, and to make programs neater for people to read. Unless within character constants (see “Character Constants” on page 3-4), any white space means the same as exactly one space.

3.3 Comments

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

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

```
/*
    The only way to include a newline ('\n') in a comment
    is to use this sort of comment.
*/
/* This sort of comment does not nest. */
```

Anything from the *line comment* character to the next newline is considered a comment and is ignored. The line comment character is # for Xtensa systems; see Chapter 8, “Features for Xtensa Processors”.

To be compatible with past assemblers, lines that begin with # have a special interpretation. Following the # should be an absolute expression (see “Expressions” on page 6-1): the logical line number of the *next* line. Then a string (see “Strings” on page 3-4) is allowed: if present it is a new logical file name. The rest of the line, if any, should be white space.

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 `st-as`.

3.4 Symbols

A *symbol* is one or more characters chosen from the set of all letters (both upper and lower case), digits and the three characters `_ . $`. No symbol may begin with a digit. Case is significant. There is no length limit: all characters are significant. Symbols are delimited by characters not in that set, or by the beginning of a file (since the source program must end with a newline, the end of a file is not a possible symbol delimiter). See “Symbols” on page 3-3.

3.5 Statements

A *statement* ends at a newline character (`\n`) or at a semicolon (`;`). The newline or semicolon is considered part of the preceding statement. Newlines and semicolons within character constants are an exception: they do not end statements.

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

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

A statement begins with zero or more labels, optionally followed by a key symbol which determines what kind of statement it is. The key symbol determines the syntax of the rest of the statement. If the symbol begins with a dot `.` then the statement is an assembler directive: typically valid for any computer. If the symbol begins with a letter the statement is an assembly language *instruction*: it assembles into a machine language instruction.

A label is a symbol immediately followed by a colon (`:`). White space before a label or after a colon is permitted, but you may not have white space between a label's symbol and its colon. See Chapter 5, “Labels”.

```
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-31415926535897932384626433832\
95028841971.693993751E-40              # - pi, a flonum.
```

3.6.1 Character Constants

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

3.6.1.1 Strings

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

Table 3-1 Escape sequences

This escape sequence	does this
`\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.

Table 3-1 Escape sequences

This escape sequence	does this
`\"`	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 `st-as` has no other interpretation, so `st-as` knows it is giving you the wrong code and warns you of the fact.

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

3.6.1.2 Characters

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

3.6.2 Number Constants

`st-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 Operator”, on pager 6-2).

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

A flonum is written by writing (in order)

- The digit `0`.
- A letter, to tell `st-as` the rest of the number is a flonum.
- An optional sign: either `+` or `-`.
- An optional integer part: zero or more decimal digits.
- An optional fractional part: `.` followed by zero or more decimal digits.
- An optional exponent, consisting of:
  - An `E` or `e`.
  - Optional sign: either `+` or `-`.
  - One or more decimal digits.

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

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

Chapter 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 `st-ld` reads many object files (partial programs) and combines their contents to form a program that can be run. When `st-as` emits an object file, the partial program is assumed to start at address 0. `st-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 `st-as` uses sections.

`st-ld` moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a *section*. Assigning run-time addresses to sections is called *relocation*. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses.

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

`st-as` can also generate whatever other named sections you specify using the `.section` directive. See ".section name (ELF version)" on page 7-19, for the ELF version. If you do not use any directives that place output in the `.text` or `.data` sections, these sections still exist, but are empty.

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

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

- To which section does the address refer? What is the numeric value of `(address) - (start-address of section)`?
- Is the reference to an address "Program-Counter relative"?

In fact, every address `st-as` ever uses is expressed as:

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

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

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

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

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

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

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

4.2 Linker Sections

`st-ld` deals with just four kinds of sections, summarized in Table 4-1.

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.

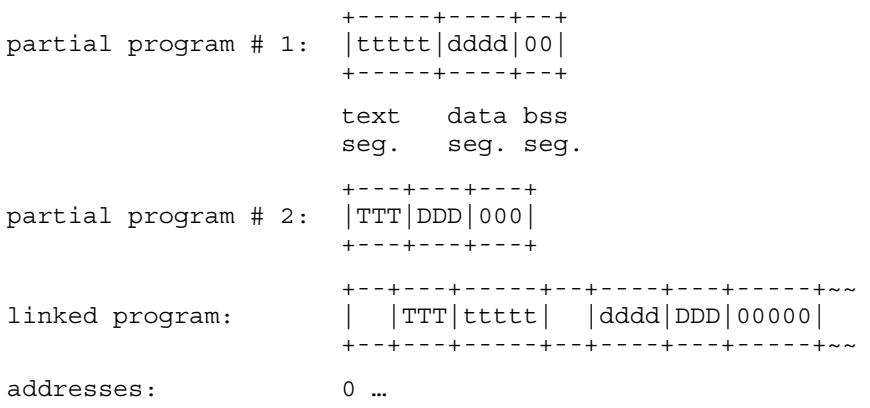

Table 4-1 Linker sections

This section	holds/does this
named sections	These sections hold your program. `st-as` and `st-ld` treat them as separate but equal sections. Anything you can say of one section is true of another. When the program is running, however, it is customary for the text section to be unalterable. The text section is often shared among processes: it contains instructions, constants and the like. The data section of a running program is usually alterable: for example, C variables would be stored in the data section.
bss section	This section contains zeroed bytes when your program begins running. It is used to hold uninitialized variables or common storage. The length of each partial program's bss section is important, but because it starts out containing zeroed bytes there is no need to store explicit zero bytes in the object file. The bss section was invented to eliminate those explicit zeros from object files.
absolute section	Address 0 of this section is always "relocated" to runtime address 0. This is useful if you want to refer to an address that `st-ld` must not change when relocating. In this sense we speak of absolute addresses being "unrelocatable": they do not change during relocation.
undefined section	This "section" is a catch-all for address references to objects not in the preceding sections.

4.3 Assembler Internal Sections

These sections are meant only for the internal use of `st-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 `st-as` warning messages, so it might be helpful to have an idea of their meanings to `st-as`. These sections are used to permit the value of every expression in your assembly language program to be a section-relative address.

Table 4-2 Internal sections

This section	holds/does this
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 Subsections

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. `st-as` allows you to use *subsections* for this purpose. Within each section, there can be numbered subsections with values from 0 to 8192. Objects assembled into the same subsection go into the object file together with other objects in the same subsection. For example, a compiler might want to store constants in the text section, but might not want to have them interspersed with the program being assembled. In this case, the compiler could issue a `.text 0` before each section of code being output, and a `.text 1` before each group of constants being output.

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

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

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a `.text` *expression* or a `.data` *expression* statement. You can also use an extra subsection argument with arbitrary named sections: `.section` *name*, *expression*. *Expression* should be an absolute expression. (See 6, “Expressions”.) 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 `st-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 ".lcomm symbol, length" on page 7-12.

The `.comm` pseudo-op may be used to declare a common symbol, which is another form of uninitialized symbol; see ".comm symbol, length" on page 7-3.

Chapter 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! *st-as does not place symbols in the object file in the same order they were declared. This may break some debuggers.*

5.1 Labels

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

5.2 Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign `=`, followed by an expression (see Chapter 6, “Expressions”). This is equivalent to using the `.set` directive. See Section 7.57, “.set symbol, expression”, on pager 7-20.

5.3 Symbol Names

Symbol names begin with a letter or with one of `._`. On most machines, you can also use `$` in symbol names; exceptions are noted in Chapter 8, “Features for Xtensa Processors”. That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted in Chapter 8), 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.

5.3.1 Local Symbol Names

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

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

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

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

Table 5-1 Local symbol names

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

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

5.4 The Special Dot Symbol

The special symbol `.` refers to the current address that `st-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, `st-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 `st-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 `st-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.

Chapter 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 `st-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. `st-as` aborts with an error message in this situation.

6.1 Empty Expressions

An empty expression has no value: it is just white space or null. Wherever an absolute expression is required, you may omit the expression, and `st-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 `st-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 white space.

6.2.3 Prefix Operator

`st-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.

Table 6-1 Highest precedence operators

This operator	is
*	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 >>

Table 6-2 Intermediate precedence operators

This operator	is
|	Bitwise Inclusive Or.
&	Bitwise And.

Table 6-2 Intermediate precedence operators

This operator	is
^	Bitwise Exclusive Or.
!	Bitwise Or Not.

Low Precedence

Table 6-3 Low precedence operators

This operator	is
+	Addition. If either argument is absolute, the result has the section of the other argument. You may not add together arguments from different sections.
-	Subtraction. If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections.
==	Is Equal To
<>	Is Not Equal To
<	Is Less Than
>	Is Greater Than
>=	Is Greater Than Or Equal To
<=	Is Less Than Or Equal To

The comparison operators can be used as infix operators. A true result has a value of -1 whereas a false result has a value of 0.

NOTE: These operators perform signed comparisons.

Table 6-4 Lowest precedence operators

This operator	is
&&	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 result 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.

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

7.2 .align abs-expr, abs-expr, abs-expr

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

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

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

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

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

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

7.3 .ascii "string"...

`.ascii` expects zero or more string literals (see "Strings" on page 3-4) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

7.4 .asciz "string"...

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

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

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

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

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

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

7.6 .byte expressions

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

7.7 .comm symbol, length

`.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 `st-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 `st-ld` sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

When using ELF, the `.comm` directive takes an optional third argument. This is the desired alignment of the symbol, specified as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address

should be zero). The alignment must be an absolute expression, and it must be a power of two. If `st-ld` allocates uninitialized memory for the common symbol, it will use the alignment when placing the symbol. If no alignment is specified, `st-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.

`.data` tells `st-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.

7.8 .double flonums

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

7.9 .eject

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

7.10 .else

`.else` is part of the `st-as` support for conditional assembly; see ".if absolute expression" on page 7-9. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

7.11 .elseif

`.elseif` is part of the `st-as` support for conditional assembly; see ".if absolute expression" on page 7-9. It is shorthand for beginning a new `.if` block that would otherwise fill the entire `.else` section.

7.12 .end

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

7.13 .endfunc

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

7.14 .endif

`.endif` is part of the `st-as` support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See ".if absolute expression" on page 7-9.

7.15 .equ symbol, expression

This directive sets the value of *symbol* to *expression*. It is synonymous with `.set`; see ".set symbol, expression" on page 7-20.

7.16 .equiv symbol, expression

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

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

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

7.17 .err

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

7.18 .exitm

Exit early from the current macro definition. See ".macro" on page 7-13.

7.19 .extern

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

7.20 .fail expression

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

7.21 .file string

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

7.22 .fill repeat, size, value

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

7.23 .float flonums

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

7.24 .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`.

7.25 .global symbol, .global symbol

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

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

7.26 .hidden names

This one of the ELF visibility directives. The other two are `.internal` (see ".internal names" on page 7-10) and `.protected` (see ".protected names" on page 7-17).

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.

7.27 .hword expressions

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

This directive is a synonym for `.short`.

7.28 .ident

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

7.29 .if absolute expression

`.if` marks the beginning of a section of code which is only considered part of the source program being assembled if the argument (which must be an *absolute expression*) is non-zero. The end of the conditional section of code must be marked by `.endif` (see ".endif" on page 7-5); optionally, you may include code for the alternative condition, flagged by `.else` (see ".elseif" on page 7-4). 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:

Table 7-1 Supported variants of .if

This variant	does this
`.ifdef` *symbol*	Assembles the following section of code if the specified symbol has been defined.
`.ifc` *string1,string2*	Assembles the following section of code if the two strings are the same. The strings may be optionally quoted with single quotes. If they are not quoted, the first string stops at the first comma, and the second string stops at the end of the line. Strings that contain white space should be quoted. The string comparison is case sensitive.
`.ifeq` *absolute expression*	Assembles the following section of code if the argument is zero.
`.ifeqs` *string1,string2*	Another form of `.ifc`. The strings must be quoted using double quotes.
`.ifge` *absolute expression*	Assembles the following section of code if the argument is greater than or equal to zero.
`.ifgt` *absolute expression*	Assembles the following section of code if the argument is greater than zero.
`.ifle` *absolute expression*	Assembles the following section of code if the argument is less than or equal to zero.
`.iflt` *absolute expression*	Assembles the following section of code if the argument is less than zero.
`.ifnc` *string1,string2.*	Like `.ifc`, but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.
`.ifndef` *symbol* `.ifnotdef` *symbol*	Assembles the following section of code if the specified symbol has not been defined. Both spelling variants are equivalent.

Table 7-1 Supported variants of .if

This variant	does this
`.ifne` *absolute expression*	Assembles the following section of code if the argument is not equal to zero (in other words, this is equivalent to `.if`).
`.ifnes` *string1,string2*	Like `.ifeqs`, but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.

7.30 .include "file"

This directive provides a way to include supporting files at specified points in your source program. The code from *file* is assembled as if it followed the point of the `.include`; when the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the `-I` command-line option (see 2, "Command-Line Options"). Quotation marks are required around *file*.

7.31 .int expressions

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

7.32 .internal names

This one of the ELF visibility directives. The other two are `.hidden` (see ".hidden names" on page 7-8) and `.protected` (see ".protected names" on page 7-17).

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` (that is, not visible to other components), and that some extra, processor specific processing must also be performed upon the symbols as well.

7.33 .irp symbol,values...

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

For example, assembling

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

is equivalent to assembling

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

7.34 .irpc symbol,values...

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

For example, assembling

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

is equivalent to assembling

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

7.35 .lcomm symbol, length

Reserve *length* (an absolute expression) bytes for a local common denoted by *symbol*. The section and value of *symbol* are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. *Symbol* is not declared global (see ".global symbol, .global symbol" on page 7-8), so is normally not visible to `st-ld`.

7.36 .lflags

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

7.37 .line line-number

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

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

7.38 .ln line-number

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

7.39 .list

Control (in conjunction with the `.nolist` directive) whether 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 2, “Command-Line Options”), the initial value of the listing counter is one.

7.40 .long expressions

`.long` is the same as `.int`, see “.int expressions” on page 7-10.

7.41 .macro

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

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

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

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

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`.

7.42 .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.

Table 7-2 Macro

This macro	does this

`.macro` *macname*
`.macro` *macname macargs …*
	Begin the definition of a macro called *macname*. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with `=deflt`. For example, these are all valid `.macro` statements:
	`.macro` *comm*
	Begin the definition of a macro called *comm*, which takes no arguments.
	`.macro plus1 p, p1`
	`.macro plus1 p p1`
	Either statement begins the definition of a macro called `plus1`, which takes two arguments; within the macro definition, write `\p` or `\p1` to evaluate the arguments.
	`.macro reserve_str p1=0 p2`
	Begin the definition of a macro called `reserve_str`, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as `reserve_str a,b` (with `\p1` evaluating to `a` and `\p2` evaluating to `b`), or as `reserve_str ,b` (with `\p1` evaluating as the default, in this case `0`, and `\p2` evaluating to `b`).

Table 7-3 Special macro variables

this variable	does this
`.endm`	Mark the end of a macro definition.
`.exitm`	Exit early from the current macro definition.
`\@`	`st-as` maintains a counter of how many macros it has executed in this pseudo-variable; you can copy that number to your output with `\@`, but only within a macro definition.

7.43 .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.

7.44 .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, `st-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 `st-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.

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

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

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.

7.46 .previous

This is one of the ELF section stack manipulation directives. The others are `.section` (see ".section name (ELF version)" on page 7-19), `.subsection` (see ".subsection name" on page 7-23), `.pushsection` (see ".pushsection name, subsection" on page 7-18), and `.popsection` (see ".popsection" on page 7-16).

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

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

7.47 .popsection

This is one of the ELF section stack manipulation directives. The others are `.section` (see ".section name (ELF version)" on page 7-19), `.subsection` (see ".subsection name" on page 7-23), `.pushsection` (see ".pushsection name, subsection" on page 7-18), and `.previous` (see ".previous" on page 7-16).

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.

7.48 .print string

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

7.49 .protected names

This one of the ELF visibility directives. The other two are `.hidden` (see ".hidden names" on page 7-8) and `.internal` (see ".internal names" on page 7-10).

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.

7.50 .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.

`st-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`.

7.51 .purgem name

Undefine the macro *name*, so that later uses of the string will not be expanded. See ".macro" on page 7-13.

7.52 .pushsection name, subsection

This is one of the ELF section stack manipulation directives. The others are `.section` (see ".section name (ELF version)" on page 7-19), `.subsection` (see ".subsection name" on page 7-23), `.popsection` (see ".popsection" on page 7-16), and `.previous` (see ".previous" on page 7-16).

This directive is a synonym for `.section`. It 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`.

7.53 .quad bignums

`.quad` expects zero or more bignums, separated by commas. For each bignum, it emits an 8-byte integer. If the bignum won't fit in 8 bytes, it prints a warning message; and just takes the lowest order 8 bytes of the bignum.

The term *quad* comes from contexts in which a word is two bytes; hence *quad*-word for 8 bytes.

7.54 .rept count

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

For example, assembling

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

is equivalent to assembling

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

7.55 .sbttl "subheading"

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

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

7.56 .section name (ELF version)

This is one of the ELF section stack manipulation directives. The others are `.subsection` (see ".subsection name" on page 7-23), `.pushsection` (see ".pushsection name, subsection" on page 7-18), `.popsection` (see ".popsection" on page 7-16), and `.previous` (see ".previous" on page 7-16).

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

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

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

Table 7-4 Optional flag characters

This character	means this
a	section is allocatable
w	section is writable
x	section is executable

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

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

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

7.57 .set symbol, expression

Set the value of *symbol* to *expression*. This changes *symbol*'s value and type to conform to *expression*. If *symbol* was flagged as external, it remains flagged (see "Symbol Attributes" on page 5-3).

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

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

7.58 .short expressions

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

7.59 .single flonums

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

7.60 .size name, expression (ELF Version)

This directive is used to set 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.

7.61 .sleb128 expressions

sleb128 stands for "signed little endian base 128." This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See ".uleb128 expressions" on page 7-25.

7.62 .skip size, fill

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

7.63 .space size, fill

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

7.64 .stabd, .stabn, .stabs

There are three directives that begin `.stab`. All emit symbols (see 5, "Symbols"), for use by symbolic debuggers. The symbols are not entered in the `st-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 `st-ld` and debuggers choke on silly bit patterns.

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

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

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

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

`.stabd type, other, desc`	The "name" of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings. The symbol's value is set to the location counter, relocatably. When your program is linked, the value of this symbol is the address of the location counter when the .stabd was assembled.
`.stabn type, other, desc, value`	The name of the symbol is set to the empty string "".
`.stabs string, type, other, desc, value`	All five fields are specified.

7.65 .string "str"

Copy the characters in *str* to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a 0 byte. You can use any of the escape sequences described in "Strings" on page 3-4.

7.66 .struct expression

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

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

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

7.67 .subsection name

This is one of the ELF section stack manipulation directives. The others are `.section` (see ".section name (ELF version)" on page 7-19), `.pushsection` (see ".pushsection name, subsection" on page 7-18), `.popsection` (see ".popsection" on page 7-16), and `.previous` (see ".previous" on page 7-16).

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.

7.68 .symver

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

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

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

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

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

Another usage of the `.symver` directive is:

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

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

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

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

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

7.69 .text subsection

Tells `st-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.

7.70 .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.

7.71 .type name, type description (ELF Version)

This directive is used to set 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. The syntaxes supported are:

```
.type <name>,#function
.type <name>,#object

.type <name>,@function
```

```
.type <name>,@object

.type <name>,%function
.type <name>,%object

.type <name>,"function"
.type <name>,"object"

.type <name> STT_FUNCTION
.type <name> STT_OBJECT
```

7.72 .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 “.sleb128 expressions” on page 7-20.

7.73 .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`.

7.74 .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`.

7.75 .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 the `*ABS*` section.

7.76 .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.

7.77 .word expressions

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

7.78 Deprecated Directives

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

```
.abort
.line
```

Chapter 8 Features for Xtensa Processors

This chapter covers features of the GNU assembler that are specific to the Xtensa architecture. For details about the Xtensa instruction set, please consult the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

8.1 Command Line Options

The Xtensa version of the GNU assembler supports these special options:

Table 8-1 Xtensa extensions to the assembler

This option	does this
`--density | --no-density`	Enable or disable use of the Xtensa code density option (16-bit instructions). See "Using Density Instructions" on page 8-4. If the processor is configured with the density option, this is enabled by default; otherwise, it is always disabled.
`--relax | --no-relax`	Enable or disable relaxation of instructions with immediate operands that are outside the legal range for the instructions. See "Xtensa Relaxation" on page 8-6. The default is `--relax` and this default should almost always be used. If relaxation is disabled with `--no-relax`, instruction operands that are out of range will cause errors. Note: In the current implementation, these options also control whether assembler optimizations are performed, making these options equivalent to `--generics` and `--no-generics`.
`--generics | --no-generics`	Enable or disable all assembler transformations of Xtensa instructions, including both relaxation and optimization. The default is `--generics`; `--no-generics` should only be used in the rare cases when the instructions must be exactly as specified in the assembly source. As with `--no-relax`, using `--no-generics` causes out of range instruction operands to be errors.

Table 8-1 Xtensa extensions to the assembler

This option	does this
`--text-section-literals | --no-text-section-literals`	Control the treatment of literal pools. The default is `--no-text-section-literals`, which places literals in a separate section in the output file. This allows the literal pool to be placed in a data RAM/ROM, and it also allows the linker to combine literal pools from separate object files to remove redundant literals and improve code size. With `--text-section-literals`, the literals are interspersed in the text section in order to keep them as close as possible to their references. This may be necessary for large assembly files.
`--rename-section oldname=newname(:oldname2=newname2)*`	When generating output sections, rename the oldname section to newname. This can be used for `.text`, `.data`, `.bss`, `.literal`, or any other section name. The output file will use the new names. This option can be used multiple times to rename multiple sections. This should not be used to rename multiple input sections to the same output section or to rename input sections to the predefined section names.
`--target-align | --no-target-align`	Enable or disable automatic alignment to reduce branch penalties at some expense in code size. See "Automatic Instruction Alignment" on page 8-5. This optimization is enabled by default. Note that the assembler will always align instructions like `LOOP` that have fixed alignment requirements.
`--longcalls | --no-longcalls`	Enable or disable transformation of call instructions to allow calls across a greater range of addresses. See "Function Call Relaxation" on page 8-6. This option should be used when call targets can potentially be out of range, but it degrades both code size and performance. The default is `--no-longcalls`.
`--xtensa-core=`*`name`*	Specify the name of an Xtensa processor core configuration to use. The configuration information is taken from the entry for name in the Xtensa core registry (see the `--xtensa-system` option). If this option is not specified, the Xtensa core name is either the value of the `XTENSA_CORE` environment variable or "default" if that variable is not set.

Table 8-1 Xtensa extensions to the assembler

This option	does this
`--xtensa-system=registry`	Specify a directory to be used as the Xtensa core registry. If this option is not set, the `XTENSA_SYSTEM` environment variable specifies the Xtensa registry, and if that is not set, the default registry, `<xttools_root>/config`, is used. Please see the *Xtensa Software Development Toolkit User's Guide* for more information about Xtensa core registries.
`--xtensa-params=path`	Specify the location of the parameter file in a TIE Development Kit (TDK) that was produced by running the TIE Compiler (tc). If path identifies a directory rather than a file, the parameters are read from a file named default-params if it exists in that directory. The parameter file may also be specified by setting the `XTENSA_PARAMS` environment variable. The `--xtensa-params` option takes precedence over the environment variable. See the *Tensilica Instruction Extension (TIE) Language User's Guide* for more information.

8.2 Assembler Syntax

Block comments are delimited by `/*` and `*/`. End of line comments may be introduced with either `#` or `//`.

Instructions consist of a leading opcode or macro name followed by whitespace and an optional comma-separated list of operands:

```
opcode [operand,...]
```

Instructions must be separated by a newline or semicolon.

8.2.1 Opcode Names

See the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for a complete list of opcodes and descriptions of their semantics.

The Xtensa assembler distinguishes between *generic* and *specific* opcodes. Specific opcodes correspond directly to Xtensa machine instructions. Prefixing an opcode with an underscore character (_) identifies it as a specific opcode. Opcodes without a leading underscore are generic, which means the assembler is required to preserve their semantics but may not translate them directly to the specific opcodes with the same names. Instead, the assembler may optimize a generic opcode and select a better instruction to use in its place (see “Xtensa

Optimizations" on page 8-4), or the assembler may relax the instruction to handle operands that are out of range for the corresponding specific opcode (see "Xtensa Relaxation" on page 8-6).

Only use specific opcodes when it is essential to select the exact machine instructions produced by the assembler. Using specific opcodes unnecessarily only makes the code less efficient, by disabling assembler optimization, and less flexible, by disabling relaxation.

Note that this special handling of underscore prefixes only applies to Xtensa opcodes, not to either built-in macros or user-defined macros. When an underscore prefix is used with a macro (e.g., `_NOP`), it refers to a different macro. The assembler generally provides built-in macros both with and without the underscore prefix, where the underscore versions behave as if the underscore carries through to the instructions in the macros. For example, `_NOP` expands to `_OR a1,a1,a1`.

The underscore prefix only applies to individual instructions, not to series of instructions. For example, if a series of instructions have underscore prefixes, the assembler will not transform the individual instructions, but it may insert other instructions between them (e.g., to align a `LOOP` instruction). To prevent the assembler from modifying a series of instructions as a whole, use the `no-generics` directive. See "generics" on page 8-10.

8.2.2 Register Names

An initial `$` character is optional in all register names. General purpose registers are named `a0`...`a15`. Additional registers may be added by processor configuration options. In particular, the MAC16 option adds a MR register bank. Its registers are named `m0`...`m3`.

As a special feature, `sp` is also supported as a synonym for `a1`.

8.3 Xtensa Optimizations

The optimizations currently supported by `st-as` are generation of density instructions where appropriate and automatic branch target alignment.

8.3.1 Using Density Instructions

The Xtensa instruction set has a code density option that provides 16-bit versions of some of the most commonly used opcodes. Use of these opcodes can significantly reduce code size. When possible, the assembler automatically

translates generic instructions from the core Xtensa instruction set into equivalent instructions from the Xtensa code density option. This translation can be disabled by using specific opcodes (see “Opcode Names” on page 8-3), by using the `--no-density` command-line option (see 2, “Command-Line Options”), or by using the `no-density` directive (see “density” on page 8-9).

It is a good idea *not* to use the density instructions directly. The assembler will automatically select dense instructions where possible. If you later need to avoid using the code density option, you can disable it in the assembler without having to modify the code.

8.3.2 Automatic Instruction Alignment

The Xtensa assembler will automatically align certain instructions, both to optimize performance and to satisfy architectural requirements.

When the `--target-align` command-line option is enabled (see 2, “Command-Line Options”), the assembler attempts to widen density instructions preceding a branch target so that the target instruction does not cross a 4-byte boundary. Similarly, the assembler also attempts to align each instruction following a call instruction. If there are not enough preceding safe density instructions to align a target, no widening will be performed. This alignment has the potential to reduce branch penalties at some expense in code size. The assembler will not attempt to align labels with the prefixes `.Ln` and `.LM`, since these labels are used for debugging information and are not typically branch targets.

The `LOOP` family of instructions must be aligned on either a 1 or 2 mod 4 byte boundary. The assembler knows about this restriction and inserts the minimal number of 2 or 3 byte no-op instructions to satisfy it. When no-op instructions are added, any label immediately preceding the original loop will be moved in order to refer to the loop instruction, not the newly generated no-op instruction.

Similarly, the `ENTRY` instruction must be aligned on a 0 mod 4 byte boundary. The assembler satisfies this requirement by inserting zero bytes when required. In addition, labels immediately preceding the `ENTRY` instruction will be moved to the newly aligned instruction location.

8.4 Xtensa Relaxation

When an instruction operand is outside the range allowed for that particular instruction field, `st-as` can transform the code to use a functionally-equivalent instruction or sequence of instructions. This process is known as *relaxation*. This is typically done for branch instructions because the distance of the branch targets is not known until assembly-time. The Xtensa assembler offers branch relaxation and also extends this concept to function calls, `MOVI` instructions and other instructions with immediate fields.

8.4.1 Conditional Branch Relaxation

When the target of a branch is too far away from the branch itself, i.e., when the offset from the branch to the target is too large to fit in the immediate field of the branch instruction, it may be necessary to replace the branch with a branch around a jump. For example,

```
  beqz      a2, L
```

may result in:

```
  bnez.n    a2, M
  j L
M:
```

(The `BNEZ.N` instruction would be used in this example only if the density option is available. Otherwise, `BNEZ` would be used.)

8.4.2 Function Call Relaxation

Function calls may require relaxation because the Xtensa immediate call instructions (`CALL0`, `CALL4`, `CALL8` and `CALL12`) provide a PC-relative offset of only 512 KBytes in either direction. For larger programs, it may be necessary to use indirect calls (`CALLX0`, `CALLX4`, `CALLX8` and `CALLX12`) where the target address is specified in a register. The Xtensa assembler can automatically relax immediate call instructions into indirect call instructions. This relaxation is done by loading the address of the called function into the callee's return address register and then using a `CALLX` instruction. So, for example:

```
  call8     func
```

might be relaxed to:

```
  .literal  .L1, func
  l32r      a8, .L1
  callx8    a8
```

Because the addresses of targets of function calls are not generally known until link-time, the assembler must assume the worst and relax all the calls to functions in other source files, not just those that really will be out of range. The linker can recognize calls that were unnecessarily relaxed, but it can only partially remove the overhead introduced by the assembler.

Call relaxation has a negative effect on both code size and performance, so this relaxation is disabled by default. If a program is too large and some of the calls are out of range, function call relaxation can be enabled using the `--longcalls` command-line option or the `longcalls` directive (see "longcalls" on page 8-10).

8.4.3 Other Immediate Field Relaxation

The `MOVI` machine instruction can only materialize values in the range from -2048 to 2047. Values outside this range are best materialized with `L32R` instructions. Thus:

```
movi a0, 100000
```

is assembled into the following machine code:

```
.literal .L1, 100000
l32r a0, .L1
```

The `L8UI` machine instruction can only be used with immediate offsets in the range from 0 to 255. The `L16SI` and `L16UI` machine instructions can only be used with offsets from 0 to 510. The `L32I` machine instruction can only be used with offsets from 0 to 1020. A load offset outside these ranges can be materialized with an `L32R` instruction if the destination register of the load is different than the source address register. For example:

```
l32i a1, a0, 2040
```

is translated to:

```
.literal .L1, 2040
l32r a1, .L1
addi a1, a0, a1
l32i a1, a1, 0
```

If the load destination and source address register are the same, an out-of-range offset causes an error.

The Xtensa `ADDI` instruction only allows immediate operands in the range from -128 to 127. There are a number of alternate instruction sequences for the generic `ADDI` operation. First, if the immediate is 0, the `ADDI` will be turned into a `MOV.N` instruction (or the equivalent `OR` instruction if the code density option is not available). If the `ADDI` immediate is outside of the range -128 to 127, but inside the range -32896 to 32639, an `ADDMI` instruction or

ADDMI/ADDI sequence will be used. Finally, if the immediate is outside of this range and a free register is available, an L32R/ADD sequence will be used with a literal allocated from the literal pool.

For example:

```
addi    a5, a6, 0
addi    a5, a6, 512
addi    a5, a6, 513
addi    a5, a6, 50000
```

is assembled into the following:

```
.literal .L1, 50000
mov.n    a5, a6
addmi    a5, a6, 0x200
addmi    a5, a6, 0x200
addi     a5, a5, 1
l32r     a5, .L1
add      a5, a6, a5
```

8.5 Directives

The Xtensa assembler supports a region-based directive syntax:

```
.begin directive [options]
...
.end directive
```

All the Xtensa-specific directives that apply to a region of code use this syntax.

The directive applies to code between the .begin and the .end. The state of the option after the .end reverts to what it was before the .begin. A nested .begin/.end region can further change the state of the directive without having to be aware of its outer state. For example, consider:

```
  .begin no-density
L:  add a0, a1, a2
  .begin density
M:  add a0, a1, a2
  .end density
N:  add a0, a1, a2
  .end no-density
```

The generic ADD opcodes at L and N in the outer no-density region both result in ADD machine instructions, but the assembler selects an ADD.N instruction for the generic ADD at M in the inner density region.

The advantage of this style is that it works well inside macros which can preserve the context of their callers.

When command-line options and assembler directives are used at the same time and conflict, the one that overrides a default behavior takes precedence over one that is the same as the default. For example, if the code density option is available, the default is to select density instructions whenever possible. So, if the above is assembled with the `--no-density` flag, which overrides the default, all the generic `ADD` instructions result in `ADD` machine instructions. If assembled with the `--density` flag, which is already the default, the `no-density` directive takes precedence and only one of the generic `ADD` instructions is optimized to be a `ADD.N` machine instruction. An underscore prefix identifying a specific opcode always takes precedence over directives and command-line flags.

The following directives are available:

- Density Directive: Disable Use of Density Instructions.
- Relax Directive: Disable Assembler Relaxation.
- Longcalls Directive: Use Indirect Calls for Greater Range.
- Generics Directive: Disable All Assembler Transformations.
- Literal Directive: Intermix Literals with Instructions.
- Literal Position Directive: Specify Inline Literal Pool Locations.
- Literal Prefix Directive: Specify Literal Section Name Prefix.
- Freeregs Directive: List Registers Available for Assembler Use.
- Frame Directive: Describe a stack frame.

8.5.1 density

The `density` and `no-density` directives enable or disable optimization of generic instructions into density instructions within the region. See “Using Density Instructions” on page 8-4.

```
.begin [no-]density
.end [no-]density
```

This optimization is enabled by default unless the Xtensa configuration does not support the code density option or the `--no-density` command-line option was specified.

8.5.2 relax

The `relax` directive enables or disables relaxation within the region. See “Xtensa Relaxation” on page 8-6.

NOTE: In the current implementation, these directives also control whether assembler optimizations are performed, making them equivalent to the `generics` and `no-generics` directives.

```
.begin [no-]relax
.end [no-]relax
```

Relaxation is enabled by default unless the `--no-relax` command-line option was specified.

8.5.3 longcalls

The `longcalls` directive enables or disables function call relaxation. See "Function Call Relaxation" on page 8-6.

```
.begin [no-]longcalls
.end [no-]longcalls
```

Call relaxation is disabled by default unless the `--longcalls` command-line option is specified.

8.5.4 generics

This directive enables or disables all assembler transformation, including relaxation (see "Xtensa Relaxation" on page 8-6) and optimization (see "Xtensa Optimizations" on page 8-4).

```
.begin [no-]generics
.end [no-]generics
```

Disabling generics is roughly equivalent to adding an underscore prefix to every opcode within the region, so that every opcode is treated as a specific opcode. See "Opcode Names" on page 8-3. In the current implementation of `st-as`, built-in macros are also disabled within a `no-generics` region.

8.5.5 literal

The `.literal` directive is used to define literal pool data, i.e., read-only 32-bit data accessed via `L32R` instructions.

```
.literal label, value[, value...]
```

This directive is similar to the standard `.word` directive, except that the actual location of the literal data is determined by the assembler and linker, not by the position of the `.literal` directive. Using this directive gives the assembler freedom to locate the literal data in the most appropriate place and possibly to combine identical literals. For example, the code:

```
entry sp, 40
.literal .L1, sym
l32r    a4, .L1
```

can be used to load a pointer to the symbol `sym` into register `a4`. The value of `sym` will not be placed between the `ENTRY` and `L32R` instructions; instead, the assembler puts the data in a literal pool.

By default literal pools are placed in a separate section; however, when using the `--text-section-literals` option (see 2, “Command-Line Options”), the literal pools are placed in the current section. These text section literal pools are created automatically before `ENTRY` instructions and manually after `.literal_position` directives (see “literal_position” on page 8-11). If there are no preceding `ENTRY` instructions or `.literal_position` directives, the assembler will print a warning and place the literal pool at the beginning of the current section. In such cases, explicit `.literal_position` directives should be used to place the literal pools.

8.5.6 literal_position

When using `--text-section-literals` to place literals inline in the section being assembled, the `.literal_position` directive can be used to mark a potential location for a literal pool.

```
.literal_position
```

The `.literal_position` directive is ignored when the `--text-section-literals` option is not used.

The assembler will automatically place text section literal pools before `ENTRY` instructions, so the `.literal_position` directive is only needed to specify some other location for a literal pool. You may need to add an explicit jump instruction to skip over an inline literal pool.

For example, an interrupt vector does not begin with an `ENTRY` instruction so the assembler will be unable to automatically find a good place to put a literal pool. Moreover, the code for the interrupt vector must be at a specific starting address, so the literal pool cannot come before the start of the code. The literal pool for the vector must be explicitly positioned in the middle of the vector (before any uses of the literals, of course). The `.literal_position` directive can be used to do this. In the following code, the literal for `M` will automatically be aligned correctly and is placed after the unconditional jump.

```
  .global M
code_start:
  j continue
  .literal_position
  .align 4
continue:
  movi    a4, M
```

8.5.7 literal_prefix

The `literal_prefix` directive allows you to specify different sections to hold literals from different portions of an assembly file. With this directive, a single assembly file can be used to generate code into multiple sections, including literals generated by the assembler.

```
.begin literal_prefix [name]
.end literal_prefix
```

For the code inside the delimited region, the assembler puts literals in the section *name*`.literal`. If this section does not yet exist, the assembler creates it. The *name* parameter is optional. If *name* is not specified, the literal prefix is set to the "default" for the file. This default is usually `.literal` but can be changed with the `--rename-section` command-line argument.

8.5.8 freeregs

This directive tells the assembler that the given registers are unused in the region.

```
.begin freeregs ri[,ri...]
.end freeregs
```

This allows the assembler to use these registers for relaxations or optimizations. (They are actually only for relaxations at present, but the possibility of optimizations exists in the future.)

Nested `freeregs` directives can be used to add additional registers to the list of those available to the assembler. For example:

```
.begin freeregs a3, a4
.begin freeregs a5
```

has the effect of declaring `a3`, `a4`, and `a5` all free.

8.5.9 frame

This directive tells the assembler to emit information to allow the debugger to locate a function's stack frame. The syntax is:

```
.frame reg, size
```

where *reg* is the register used to hold the frame pointer (usually the same as the stack pointer) and *size* is the size in bytes of the stack frame. The `.frame` directive is typically placed immediately after the `ENTRY` instruction for a function.

In almost all circumstances, this information just duplicates the information given in the function's `ENTRY` instruction; however, there are two cases where this is not true:

1. The size of the stack frame is too big to fit in the immediate field of the `ENTRY` instruction.
2. The frame pointer is different than the stack pointer, as with functions that call `alloca`.

Chapter 9 Acknowledgements

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

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

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

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

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

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

The Intel 80386 machine description was written by Eliot Dresselhaus.

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

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

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

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

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

Ian Lance Taylor of Cygnus Support merged the Motorola and MIT syntax for the 68k, completed support for some COFF targets (68k, i386 SVR3, and SCO Unix), added support for MIPS ECOFF and ELF targets, wrote the initial RS/6000 and PowerPC assembler, and made a few other minor patches.

Steve Chamberlain made `st-as` able to generate listings.

Hewlett-Packard contributed support for the HP9000/300.

Jeff Law wrote GAS and BFD support for the native HPPA object format (SOM) along with a fairly extensive HPPA test suite (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.

Many others have contributed large or small bug fixes 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 1.1, March 2000.

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

A.1 Preamble

The purpose of this License is to make a manual, textbook, or other written 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 noncommercially. 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.

A.2 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

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. (For example, 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.

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 “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and 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 has been designed to thwart or discourage subsequent modification by readers is not Transparent. 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 designed for human modification. Opaque formats include PostScript, PDF, 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 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.

A.3 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.

A.4 Copying in Quantity

If you publish printed copies 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 publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. 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.

A.5 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:

1. 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.
2. 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 less than five).
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. 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.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section entitled “History”, and 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.

10. 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.
11. In any section entitled “Acknowledgements” or “Dedications”, preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. 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.
13. Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.
14. Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

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.

A.6 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.

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”.

A.7 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.

A.8 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, does not as a whole count as a Modified Version of the Document,

provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they 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 quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.9 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 provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

A.10 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11 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. Go to 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.

A.12 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 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled ``GNU Free Documentation License''.

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being LIST”; likewise for Back-Cover Texts.

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.

Appendix B History

The original version of this document, entitled “Using AS, the GNU Assembler”, was written by Dean Elsner, Jay Fenlason and friends. The version for `as` 2.11.2 was released in 2001 and published by the Free Software Foundation.

Tensilica, Inc. changed the title to “GNU Assembler User's Guide” and modified the document to include features specific to Xtensa processors. The revised document was published by Tensilica, Inc. on the date shown in the inside cover page. The TeXinfo source files for this modified document are available from http://www.tensilica.com/gnudocs.

Index

Symbols

! 6-3
- 6-3
-- 1-5, 1-6
1-6, 3-2
#APP 3-1
#NO_APP 3-1
& 6-2
&& 6-3
. (dot symbol) 3-3
.abort directive 7-1
.align directive 7-1
.ascii directive 7-2
.asciz directive 7-2
.balign directive 7-2
.byte directive 7-3
.comm directive 7-3
.double directive 7-4
.eject directive 7-4
.else directive 7-4
.elseif directive 7-4
.end directive 7-5
.endfunc directive 7-5
.endif directive 7-5
.endm directive 7-14
.equ directive 7-5
.equiv directive 7-5
.err directive 7-6
.exitm directive 7-6, 7-14
.extern directive 7-6
.fail directive 7-6
.file directive 7-6
.fill directive 7-7
.float directive 7-7
.func directive 7-7
.global directive 7-8
.hidden directive 7-8
.hword directive 7-8
.ident directive 7-8
.if directive 7-9
.ifc directive 7-9
.ifdef directive 7-9
.ifeq directive 7-9
.ifeqs directive 7-9
.ifge directive 7-9
.ifgt directive 7-9
.ifle directive 7-9
.iflt directive 7-9
.ifnc directive 7-9
.ifndef directive 7-9
.ifne directive 7-10
.ifnes directive 7-10
.ifnotdef directive 7-9
.include directive 7-10
.inlcude directive search path 2-2
.int directive 7-10
.internal directive 7-10
.irp directive 7-11
.irpc directive 7-11
.lcomm directive 7-12
.lflags directive (ignored) 7-12
.line directive 7-12
.list directive 7-12
.ln directive 7-12
.long directive 7-13
.macro directive 7-13
.nolist directive 7-13
.o 1-6
.octa directive 7-14
.org directive 7-15
.p2align directive 7-15
.popsection directive 7-16
.previous directive 7-16
.print directive 7-17
.protected directive 7-17
.psize directive 7-17
.purgem directive 7-17
.section name (ELF) directive 7-19
.set directive 7-20
.short directive 7-20
.single directive 7-20
.size (ELF) directive 7-20
.skip directive 7-21
.sleb128 directive 7-20
.space directive 7-21
.stabd directive 7-21
.stabn directive 7-21
.stabs directive 7-21
.string directive 7-22
.struct directive 7-22
.subsection directive 7-23
.symver directive 7-23
.text directive 7-24
.title directive 7-24
.type directive (ELF version) 7-24
.uleb128 directive 7-25
.version directive 7-25
.vtable_entry directive 7-25
.vtable_inherit directive 7-25
.weak directive 7-26
.word directive 7-26
: (label) 3-3, 5-1
< 6-3
<= 6-3
<> 6-3
== 6-3
> 6-3
>= 6-3
\ 3-4, 3-5
\" 3-5
\@ directive 7-14
\ddd (octal character code) 3-4
\f (formfeed character) 3-4
\n 3-4
\n (newline character) 3-4
\r 3-4
\t 3-4

\xd ... (hex character code) 3-4
^ 6-3
_ opcode prefix 8-3
| 6-2
|| 6-3

A

-a 1-1, 2-1
a.out 1-6
absolute section 4-3
-ac 1-1, 2-1
-ad 1-1, 2-1
ADDI instructions, relaxation 8-7
addition, permitted arguments 6-3
addresses, format of 4-2
advancing location counter 7-15
-ah 1-1, 2-1
-al 1-1, 2-1
alignment of branch target 8-5
alignment of ENTRY instructions 8-5
alignment of LOOP instructions 8-5
-am 1-1
-an 1-1, 2-2
AND 6-2
arguments for addition 6-3
arguments to integer expressions 6-1
arithmetic functions 6-2
arithmetic operands 6-2
-as 1-2, 2-1
assembler and linker 4-1
assembler internal logic error 4-4
assembler internal sections 4-3
assembler version 2-4
assembly listings, enabling 2-1
assigning values to symbols 5-1, 7-5
attributes, symbol 5-3

B

b 3-4
backslash (\\) 3-4
backspace (\b) 3-4
bignums 3-6
binary integer 3-5
binary integers 3-6
branch instructions, relaxation 8-6
branch target alignment 8-5
bss section 4-3, 4-5

C

C-A 5-2
call instructions, relaxation 8-6
carriage return (\r) 3-4
character constant, single 3-4
character constants 3-4
character escape codes 3-4
character, single 3-5
Characters 3-5
characters used in symbols 3-3
command line conventions 2-1
comments 3-2
comments, removed by preprocessor 3-1
common symbols 7-3
common variable storage 4-5
comparison expressions 6-3
conditional assembly 7-9
constant, single character 3-5
Constants 3-3
constants 3-4
constants, bignum 3-6
constants, character 3-4
constants, converted by preprocessor 3-1
constants, floating point 3-6
constants, integer 3-5
constants, number 3-5
constants, string 3-4
current address 5-3
current address, advancing 7-15

D

-D 1-2, 2-2
data and text sections, joining 2-4
debuggers and symbol order 5-1
decimal integers 3-5
--density 1-3, 8-1, 8-9
density directive 8-9
density instructions 8-4
density option, Xtensa 8-1
dependency tracking 2-3
deprecated directive 7-26
directive and instructions 3-3
directives, machine independent 7-1
directives, precedence 8-9
directives, Xtensa 8-8
dot symbol 5-3
doublequote(\") 3-5

E

eight-byte integer 7-18
ELF symbol type 7-24
empty expressions 6-1
ENTRY instruction alignment 8-5
EOF, newline must precede 3-3
error messages 1-7
errors, caused by warnings 2-5
errors, continuing after 2-5
escape codes, character 3-4
exclusive OR 6-3
expr section 4-4
expression arguments 6-1
expressions 6-1
  arguments to integer 6-1
  empty 6-1
  operations on integers 6-2
expressions, comparison 6-3
expressions,integer 6-1

F

-f 1-2, 2-2, 3-1
f 3-4
faster processing (-f) 2-2
--fatal-warnings 2-5
file name, logical 7-6
files, including 7-10
files, input 1-5
filling memory 7-21
floating 7-20
floating point numbers 3-6
floating point numbers (double) 7-4
floating point numbers (single) 7-20
Flonums 3-6
format of error messages 1-7

format of warning message 1-7
formfeed (\f) 3-4
frame directive 8-12
freeregs directive 8-12
functions, in expressions 6-2

G

generic opcodes 8-3
--generics 8-1
generics directive 8-10
greater than 6-3
greater than or equal to 6-3
grouping data 4-4

H

hex character code (\xd...) 3-4
hexadecimal integers 3-6

I

-I 1-2, 2-2, 7-10
-I path 2-2
inclusive OR 6-2
infix operators 6-2
input file line numbers 1-6
input files 1-5
instructions and directives 3-3
integer expressions 6-1
integer, 16-byte 7-14, 7-18
integers 3-5
integers, 16-bit 7-8
integers, 32-bit 7-10
integers, binary 3-5
integers, decimal 3-5
integers, hexadecimal 3-6
integers, octal 3-5
internal assembler sections 4-3
internal sections in warning messages 4-3
invocation summary 1-1
is equal to 6-3
is not equal to 6-3

J

joining data and text sections 2-4

K

-K 1-2, 2-3

L

-L 1-2, 2-1, 2-3, 5-2
L16SI instructions, relaxation 8-7
L26UI instructions, relaxation 8-7
L32I instructions, relaxation 8-7
L32R instructions, relaxation 8-7
label ( ) 3-3
labels 5-1
ld 1-6
length of symbols 3-3
less than 6-3
less than or equal to 6-3
line comment character 3-2
line number, logical 7-12
line numbers in warnings and errors 1-7
line numbers, in input file 1-6
lines, starting with # 3-2
linker 1-6
linker and assembler 4-1
Linker Sections 4-2
listing control, new page 7-4
listing control, paper size 7-17
listing control, title line 7-24
listing control, turning off 7-13
listing control, turning on 7-12
listings, enabling 2-1
literal directive 8-10
literal_position directive 8-11
literal_prefix directive 8-12
local comman symbols 7-12
local labels, retaining in output 2-3
local symbol names 5-2
location counter 5-3
location counter, advancing 7-15
logical AND 6-3
logical file name 7-6
logical line number 3-2, 7-12
logical OR 6-3
--longcalls 1-4, 8-2, 8-7, 8-10
longcalls directive 8-10
LOOP instructions alignment 8-5

M

machine independent directives 7-1
machine instructions (not covered) 1-4
machine-independent syntax 3-1
macros 7-13
macros, count executed 7-14
make rules 2-3
manual, structure and purpose 1-4
--MD 2-3
merging text and data sections 2-4
messages from assembler 1-7
minus, permitted arguments 6-3
MOVI instructions, relaxation 8-7

N

name object file 2-4
named section 7-19
named sections 4-3
names, symbol 5-1
naming symbols 5-1
new page in listings 7-4
newline (\n) 3-4
newline, required at end of file 3-3
--no-density 1-3, 8-1, 8-5, 8-9
no-density directive 8-9
--no-generics 1-3, 8-1
--no-longcalls 1-4, 8-2
no-longcalls directive 8-10
--no-relax 1-3, 8-1, 8-10
no-relax directive 8-10
NOT 6-3
--no-target-align 1-3, 8-2
--no-text-section-literals 1-3, 8-2
--no-warn 1-3, 2-5
null-terminated strings 7-2
number constants 3-5
number of macros executed 7-14
numbered subsections 4-4
numbers, 16-bit 7-8
numeric values 6-1

O

-o 1-2, 1-6, 2-4
object file 1-6
object file after errors 2-5
object file format 1-5
object file name 2-4
octal character code (\ddd) 3-4
octal integers 3-5
opcode names, Xtensa 8-4
opcodes, specific 8-3
opdcodes, generic 8-3
operands in expressions 6-1
operator
  infix 6-2
  prefix 6-2
operator precedence 6-2
operators in expressions 6-2
operators on integers 6-2
operators, permitted arguments 6-2
optimizations 8-4
option summary 1-1
options, all versions of assembler 2-1
options, command line 2-1
options, summary of 1-1
OR 6-2, 6-3
OR NOT 6-3
output file 1-6

P

p2align1 directive 7-15
p2alignw directive 7-15
padding the location counter 7-1
padding the location counter a given number of bytes 7-1
padding the location with a given power of two 7-15
page, in listings 7-4
paper size for listings 7-17
paths for .include 2-2
patterns, writing in memory 7-7
plus, permitted arguments 6-3
precedence of directives 8-9
precedence of operators 6-2
precision, floating point 3-6
prefix operator 6-2
prefix operators 6-2
Preprocessing 3-1
preprocessing, turning on and off 3-1
pseudo-ops, machine independent 7-1

R

-R 1-2, 2-4
register names, Xtensa 8-4
--relax 1-3, 8-1
relax directive 8-9
relaxatio of L8UI instructions 8-7
relaxation 8-6
relaxation fo ADDI instructions 8-7
relaxation of branch instructions 8-6
relaxation of call instructions 8-6
relaxation of immediate fields 8-7
relaxation of L16SI instructions 8-7
relaxation of L16UI instructions 8-7
relaxation of L32I instructions 8-7
relaxation of L32R instructions 8-7
relaxation of MOVI instructions 8-7
relocation 4-1
relocation example 4-4
--rename-section 1-3, 8-2, 8-12

S

search path for .include 2-2
section
  absolute 4-3
  bss 4-3, 4-5
  expr 4-4
  undefined 4-3
section stack 7-16, 7-18, 7-19
section-relative addressing 4-1
sections 4-1
sections in warning messages, internal 4-3
sections, named 4-1, 4-3
single character constant 3-4
sixteen-bit numbers 7-8
sixteen-byte integer 7-14
source program 1-5
sp register 8-4
space used, maximum for assembly 3-1
specific opcodes 8-3
standard assembler sections 4-1
standard input as input file 1-5
statement separator character 3-3
statements, structure of 3-3
--statistics 1-2, 2-4
statistics, about assembly 2-4
stopping the assembly 7-1
string constants 3-4
string literals 7-2
string, copying to object file 7-22
strings 3-4
subections 4-4
subexpressions 6-2
subtitles for listings 7-19
subtraction 6-3
summary of options 1-1
supporting files, including 7-10
suppressing warnings 2-5
symbol
  giving other values 5-1
  names
    local 5-2
  type 5-3
symbol attributes 5-3
symbol names 5-1
symbol names, temporary 5-1
symbol type, ELF 7-23
symbol value 5-3
symbol value, setting 7-20
symbol versioning 7-23
symbol, common 7-3
symbol, making visible to linker 7-8
symbolic debuggers, information for 7-21
Symbols 3-3
symbols, assigning values to 7-5
symbols, local common 7-3, 7-12
symbols,length of 3-3
syntax, machine-independent 3-1
syntax, Xtensa assembler 8-3

T

tab (\t) 3-4
--target-align 1-3, 2-3, 8-2, 8-5
text and data sections, joining 2-4
--text-section-literals 1-3, 8-2, 8-11
time, total for assembly 2-4
trusted compiler 2-2
turning preprocessing on and off 3-1
type of a symbol 5-3

U

undefined section 4-3

V

-v 1-3, 2-1, 2-4
value of a symbol 5-1, 5-3
-version 1-3, 2-4
version of the assembler 2-4
versions of symbols 7-23
visibility 7-8, 7-10, 7-17

W

-W 1-3, 2-5
--warn 1-3, 2-5
warning messages 1-7
warnings, causing error 2-5
warnings, suppressing 2-5
warnings, switching on 2-5
White space 3-2
whitespace, removed by preprocessor 3-1
work faster 2-2
writing patterns in memory 7-7

X

Xtensa architecture 8-1
Xtensa assembler syntax 8-3
Xtensa command-line options 8-1
Xtensa density option 8-1
Xtensa directives 8-8
Xtensa opcode names 8-3
Xtensa register names 8-4
--xtensa-core 1-4, 8-2
--xtensa-params 1-4, 8-3
--xtensa-system 1-4, 8-3

Z

-Z 2-5
zero-terminated strings 7-22