cadence®

Xtensa® C and C++ Compiler

User's Guide

For Xtensa Tools Version 12

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Issue Date:10/2015

RG-2015.2
PD-15--0330-10-01

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

1. Introduction 1
1.1 XCC Highlights 1
1.2 Migrating from GCC 2
1.2.1 Command-Line Options 2
1.2.2 Exception Handling 2
1.2.3 Floating Point Optimization 3
1.3 CLANG 3
1.3.1 Preprocessor Defines 3
1.3.2 Versions 3
1.3.3 Error Messages 4
1.3.4 Unsupported Clang Features 4
1.3.5 Inlining 4
1.3.6 Compiler Warning Options 5
1.4 Input File Handling 5
1.5 Preprocessor 6
1.6 Host Platform Differences 6
2. Command-Line Options 7
2.1 Compiler Output Options 7
2.2 Preprocessor Options 8
2.3 Language Dialect Options 11
2.4 Warning Control Options 13
2.5 Debugging Options 18
2.6 Optimization Options 19
2.7 Inlining Options 21
2.8 Code Generation Options 22
2.9 Assembler Options 24
2.10 Linker Options 25
2.11 Xtensa-Specific Options 27
3. Extensions to C and C++ 29
3.1 Intrinsic Functions for Xtensa Instructions 29
3.1.1 Variable Shifts 31
3.1.2 MAC16 31
3.2 Built-in Functions 32
3.3 Memory Consistency 32
3.3.1 TIE Ports 33
TIE Ports and Deadlock 34
3.4 C++ Style Comments 35
3.5 Inline Functions in C 35
3.6 Support for long long Variables 36
3.7 Restrict Pointers 36
3.8 Pointer Arithmetic 36
3.9 Variable-Length Arrays 36
3.10 Zero-Length Arrays 37

3.11 Attributes of Functions, Types and Variables 37
3.12 Inline Assembly 41
3.13 Xtensa Boolean Types 45
3.14 Operator Overloading in C or C++ Programs 46
4. Advanced Optimization Topics 49
4.1 Viewing the Effects of Compiler Optimizations 49
4.2 Optimizing Functions Individually 51
4.3 Controlling Miscellaneous Optimizations with the -OPT Option Group 52
4.3.1 Controlling Alias Analysis 54
4.4 Using Profiling Feedback 57
4.5 Super Software Pipelining 61
4.6 Interprocedural Analysis and Optimization 62
4.6.1 Building Libraries with IPA 64
4.7 Inexact imaps 65
4.8 Loop Pragmas 65
#pragma no_unroll 65
4.9 Speculation 66
4.10 SIMD Vectorization 67
4.10.1 Viewing the Results of Vectorizing Transformations 68
4.10.2 Aligning Data for Vectorization 69
4.10.3 Controlling Vectorization through Pragmas 72
#pragma concurrent 72
#pragma simd_if_convert 72
#pragma simd 73
4.10.4 Features and Limitations 73
Limitation in the Stride of Memory Accesses 74
Outer Loop Vectorization 74
Guard Bits 74
4.10.5 Vectorization Analysis Report 75
SIMD Analysis Messages 76
4.11 Managing Memory Bank Conflicts 83
A. Command-Line Option List with -help 87
B. Summary of Compiler Pragmas 101

List of Tables

Table 1–1. Input File Handling 5
Table 2–2. Compiler Output Options 7
Table 2–3. Preprocessor Options 8
Table 2–4. Language Dialect Options 11
Table 2–5. Warning Options 13
Table 2–6. Debugging Options 18
Table 2–7. Optimization Options 19
Table 2–8. Inlining Options 22
Table 2–9. Code Generation Options 23
Table 2–10. Assembler Options 25
Table 2–11. Linker Options 25
Table 2–12. Xtensa-Specific Options 27
Table 3–13. Xtensa Header Files 29
Table 3–14. Declaration Attributes 38
Table 3–15. asm Operand Constraints 42
Table 3–16. asm Constraint Modifiers 43
Table 3–17. Supported Operators 47
Table 4–18. -OPT Option Group 53
Table 4–19. Vectorization Options 67
Table A-20. Summary of Command-Line Options 87
Table B-21. Summary of Compiler Pragmas 101

Preface

This document describes the features of the Xtensa® C and C++ Compiler for the Xtensa Tools Version 12, and the command-line options that control it.

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- `$` represents your shell prompt, in user-session examples.
- **`literal_input`** indicates literal command-line input.
- *`variable`* indicates a user parameter.
- `literal_keyword` (in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- *`... output ...`* indicates unspecified program output.
- *`[optional-variable]`* indicates an optional parameter.
- **`[`***`variable`***`]`** indicates a parameter within literal square-braces.
- **`{`***`variable`***`}`** indicates a parameter within literal curly-braces.
- **`(`***`variable`***`)`** indicates a parameter within literal parentheses.
- `|` means *OR*.
- *`(var1 | var2)`* indicates a required choice between one of multiple parameters.
- *`[var1 | var2]`* indicates an optional choice between one of multiple parameters.
- *`var1 [, varn]*`* indicates a list of 1 or more parameters (0 or more repetitions).

Terms

- *0x* at the beginning of a value indicates a hexadecimal value.
- *b* means bit.
- *B* means byte.
- *Mb* means megabit.
- *MB* means megabyte.
- *PC* means program counter.
- *word* means 4 bytes.

Changes from the Previous Version

The following clarification was made to this document for the Xtensa Tools Version 12.0.2 released with the Cadence Tensilica RG-2015.2 release.

- Clarified the description for SIMD_IF_UNSAFE_ACCESS in Chapter 4

The following clarification was made to this document for the Xtensa Tools Version 12.0.0 released with the Cadence Tensilica RG-2015.0 release.

- Added support for Clang, as described in section Section 1.3 on page 3

Processor Version Compatibility for Xtensa Tools Version 12

The Xtensa Tools version 12 released with the Cadence Tensilica RG-2015.0 or later releases supports these versions of Tensilica processors:

- Xtensa LX processors (Releases RA-200X.x to RG-201X.x)
- Xtensa processors (Releases RA-200X.x to RF-201X.x)
- Diamond Standard Series processors (Rev. A to Rev. D hardware)

Xtensa Tools Version 12 and Xtensa LX7

The primary target for the use of Xtensa Tools version 12 is Xtensa LX7 processors built from the RG-201X.X release versions. All modes of use are supported, from system architecture analysis to code development, hardware co-verification, and silicon bringup. Unless otherwise stated, all features and use modes described in this document are applicable for use with these processors.

Throughout this guide, many references to Xtensa refer generally to any Tensilica processor (Xtensa LX, Xtensa) that implements the Xtensa instruction set architecture, unless otherwise noted.

- Features that are available only for Xtensa LX processors are preceded with the symbol **For LX cores** .

Xtensa Tools Version 12 and Software Upgrades for Xtensa 6, 7, 8, 9, 10, and 11 and LX1, LX2, LX3, LX4, LX5 and LX6 Processors

Xtensa Tools Version 12 is compatible and tested for use with these previous versions (that is, releases prior to this RG-2015.0 release) of Xtensa and Diamond Standard processors family (RA-200X.x to RD-201x.x) for code generation, profiling, and debug for post-silicon processors, via the software upgrade process.

1. Introduction

The Xtensa C and C++ Compiler (XCC) is an advanced optimizing compiler for all the Xtensa processors. XCC augments the standard Xtensa GNU software development toolchain, the assembler, linker, debugger, libraries and binary utilities. While XCC's operation is similar to standard GNU C and C++ compilers (GCC), XCC provides support for TIE (Tensilica Instruction Extension language) as well as superior execution performance and smaller size of the compiled code through improved optimization and code generation technology. This guide gives an overview of features available in XCC and describes command-line options that control its behavior.

Content in this guide assumes that you have experience with software development in either a UNIX or Windows environment and are comfortable using command-line tools and makefiles. You should be familiar with the *Xtensa Software Development Toolkit User's Guide*.

Note: Throughout this guide, there are many references to *Xtensa* that refer generally to any Xtensa processor that implements the Xtensa Instruction Set Architecture.

1.1 XCC Highlights

XCC represents both the C compiler, called `xt-xcc`, and the C++ compiler, called `xt-xc++`. Typical phases in the overall compilation process with XCC include preprocessing, compiling, assembly and linking. XCC uses a GNU preprocessor, assembler (see the *GNU Assembler User's Guide*) and linker (see the *GNU Linker User's Guide*). It maintains a high level of compatibility with GCC by supporting most GCC command-line options and language extensions. The main difference is that XCC incorporates a large number of advanced optimization techniques that may reduce both the execution time and the code size. These techniques include:

- SSA-based global optimizations, such as constant propagation, dead code elimination, partial redundancy elimination and strength reduction
- Loop-nest transformations based on dependence analysis and automatic C/C++ source code vectorization for DSP coprocessors provided by Cadence.
- A software pipeliner that can overlap operations from multiple iterations of a loop
- Function inlining based on heuristics that improve execution performance without a significant increase in code size
- New code generation methods specifically designed to reduce code size
- Interprocedural (whole program) analyses (IPA) and optimizations, such as dead function and variable elimination, cross-file inlining and more precise alias analysis

- Feedback optimizations that use profile information from executing the actual code to improve execution speed, and especially, code size

1.2 Migrating from GCC

XCC uses version 4.2.0 of GCC as a front end, the phase of the compiler responsible for parsing the input program. As such, XCC is highly compatible with GCC 4.2.0 and supports many of the language extensions and compiler flags. Usually, the only noticeable difference in switching from a version of GCC to XCC is the substantially faster or, with the space-saving option, smaller code produced by XCC. However, certain minor differences do exist between the two compilers, as outlined in the following sections.

1.2.1 Command-Line Options

Because XCC and GCC share the same preprocessor, assembler, and linker, command-line options that apply to those compilation phases have the same effect with both compilers.

The XCC front end (the phase that takes the output of the preprocessor and performs syntax and semantic analysis of the source code before converting into an intermediate format used in later phases) is built from the GNU code base, therefore, options applicable to the front end, such as those for controlling the language dialect or requesting and suppressing warnings, work the same way as in GCC.

The XCC back end (the phase that performs extensive optimizations and produces assembly code) is completely different from the GCC back end. While XCC accepts GCC back-end related options for compatibility in makefiles, some of these options have no effect on the behavior of the XCC back end. For example, the `-frerun-loop-opt` option instructs the GCC back end to run its loop optimizer twice. Although XCC does not reject this option, its presence does not change the actions taken by the compiler.

1.2.2 Exception Handling

Enabling exception handling adds substantially to code size, whether or not the exceptions are ever used. Therefore, `xt-xc++` has exception handling turned *off* by default to avoid penalizing programs that do not use exceptions. This is the opposite of standard GCC, which has exception handling enabled. If your program uses exceptions, you can enable exception handling in `xt-xc++` by supplying the `-fexceptions` option on the command line. Otherwise, `xt-xc++` will produce an error for a source file that contains exception constructs.

If you need exception handling in any of your source files, it is recommended that you compile all your source files with the `-fexceptions` option.

1.2.3 Floating Point Optimization

When compiling with `-O3`, XCC is more aggressive with respect to floating point optimizations than GCC. In particular, XCC will reorder operations in an expression and XCC will replace a floating point divide or sqrt operations with its reciprocal. These optimizations mean that with XCC, floating point code compiled at `-O3` might not be bit-exact with code compiled at lower optimization levels. The more aggressive optimization can be disable by using the `-fno-unsafe-math-optimizations` flag.

1.3 CLANG

Starting with the RG-2015.0 release and version 12 of the Xtensa tools, XCC offers a beta version of replacing the GCC front end with Clang version 3.4 from the LLVM project See http://clang.llvm.org for details about Clang.In future versions, Clang will replace GCC.

Clang provides superior error handling compared to GCC. By default Clang supports the C99 standard and optionally supports the C11 standard. Otherwise, Clang is mostly compatible with GCC, so many users will not see any differences. However, there are some differences as highlighted below.

Clang is selected using the `-clang` compiler flag.

1.3.1 Preprocessor Defines

Clang defines the following macros in addition to the ones defined by the default GCC front end.

- `__clang__ = 1`
- `__clang_major = 3`
- `__clang_minor__` = 4

1.3.2 Versions

Clang defaults to support the C99 version of C, compared to C89 for the default GCC front end. C11 is supported through the `-std=c11` flag.C89 is not supported.

1.3.3 Error Messages

Clang provides cleaner and more precise error and warning messages than GCC, including column positions. Note that the format of the messages is different so any tools that rely on the exact format might need to be updated.

1.3.4 Unsupported Clang Features

Clang does not support the GNU extension of nested functions.

Clang does not support the GCC-style syntax for allocating a global variable into a specified hardware register. With GCC, everywhere the variable is used, XCC will use the assigned register in place of the variable.

```
register int x asm("register name");
```

The Clang beta has known bugs supporting standard GNU asms.

Clang does not support #pragma weak.

1.3.5 Inlining

The C99 standard added inline to the C language. There are three types of inline declarations, `static inline`, `inline` and `extern inline`. With all three, the compiler may or may not actually inline a call to the function.

- With `static inline`, if some call to the function is not inlined, the body of the function is emitted as a normal static function. The emitted function is locally emitted so that only calls from the same file will invoke it. If multiple files contain calls to the same static inline function that ends up not being inlined, the final binary will contain multiple copies of the function.
- `Extern inline` is meant to be used in only one file. A global copy of the function is emitted. If multiple files contain the same `extern inline` definition, the linker will complain of duplicate functions.
- With `inline` by itself, the function body will never be emitted. If a particular function is only marked as inline, and if some call to that function is not inlined, the linker will complain that the function is undefined.

In normal usage, functions to be inlined are marked as `inline` in header files and `extern inline` in a single C file. That way, if ever the function is not inlined, a single copy is created in the single C file.

Clang by default supports the C99 standard. XCC with GCC supports an earlier GCC behavior. `Static inline` behaves identically to the standard. `Extern inline` does not emit the function body while `inline` by itself does emit the function: the opposite behavior of the standard. Using the flag `-fgnu89-inline` gives the old behavior.

Note that both GCC and Clang have the same behavior for C++ programs, matching the C++ standard which is different than C. In C++, `inline` by itself emits a copy of the function but marks it specially so that the linker can remove duplicate definitions.

1.3.6 Compiler Warning Options

Some warning options are different between the two compilers:

`-Wdiv-by-zero` in GCC is replaced by `-Wdivision-by-zero` in Clang

`-Woverride-init` in GCC is replaced by `-Winitializer-overrides` in Clang.

`-Wmultichar` in GCC is replaced by `-Wfour-char-constants` in Clang. Also, GCC by default warns whereas Clang only warns when the option is given.

`-Wno-overflow` is not supported.

1.4 Input File Handling

XCC determines the proper manner in which to handle an input file based on the file name extension. Table 1–1 lists the extensions recognized by XCC and describes how they are handled. If an extension is not recognized, the input file is passed directly to the linker.

Table 1–1. Input File Handling

File Name Extension	XCC treats the input file as
`.c`	A C language file that needs preprocessing. However, if the compiler is invoked as `xt-xc++`, the input file is treated as a C++ language file.
`.C .cc .c++ .cpp .cxx`	A C++ language file that needs preprocessing.
`.i`	An already preprocessed C language file. However, if the compiler is invoked as `xt-xc++`, the input file is treated as a preprocessed C++ language file.
`.ii`	An already preprocessed C++ language file.
`.s`	An already preprocessed assembly file.
`.S`	An assembly file that needs preprocessing.
`.o`	An object file that will be passed to the linker.

A C language file compiled with `xt-xc++` has C++ style linkage. The C++ style linkage may cause an error if a C language file is compiled with `xt-xc++` and linked with a C language file compiled with `xt-xcc`. Link errors can occur because the two files have different linkage conventions. To avoid this error, do not mix the two compilers, instead, use either `xt-xcc` or `xt-xc++` for all your C language files.

1.5 Preprocessor

XCC defines the following preprocessor assertions and macros, which can be used in applications to tailor the code to the specific processor configuration:

- `cpu(xtensa)`

  Asserts that the target CPU is an Xtensa processor implementing the Xtensa Instruction Set Architecture.

- `__GNUC__=4 __GNUC_MINOR__=2`

  Indicates that the compiler is compatible with GNU C version 4.2.

- `machine(xtensa)`

  Asserts that the target machine is an Xtensa system.

- `__XTENSA__`

  Specifies that the target is an Xtensa processor implementing the Xtensa Instruction Set Architecture.

- `__XCC__`

  Indicates that the program is being compiled with `xt-xcc` or `xt-xc++`.

- `__XTENSA_EL__ or __XTENSA_EB__`

  Specifies that the target processor is little-endian or big-endian. Only one of these is defined, depending on your configuration.

- `__XTENSA_SOFT_FLOAT__`

  Specifies that *the target processor was configured without the floating-point engine, and floating-point operations will be emulated in software*.

1.6 Host Platform Differences

XCC uses some random search algorithms as part of its heuristics. The random seeds used on different host platforms are different. Therefore, there may be small differences in the code generated on different host platforms.

2. Command-Line Options

As in GCC, most of the command-line options that start with `-W`, `-f` or `-m` have two forms, for example, `-f`*`option`* and `-fno-`*`option`*. Unless noted otherwise, the following tables describe only the non-default form of each option.

2.1 Compiler Output Options

Table 2–2 describes the options that control the type of output XCC generates.

Table 2–2. Compiler Output Options

Option	Description
`-c` `--compile`	Compiles or assembles the input files without linking. Output files are named by replacing the input file's extension (such as `.c` or `.cpp`) with `.o`. If a single input file is specified on the command line, you can override the output file name with the `-o` option.
`-dumpversion`	Prints the version of the compiler front end.
`-E`	Preprocesses the input files without any further compilation steps. The preprocessed source code is sent to standard output. If there is a single input file on the command line, you can use the `-o` option to direct the output to a file. This option is useful to see how preprocessor macros are expanded.
`-fsyntax-only`	Checks the code for syntax errors only.
`-help` `--help`	Prints information about options described in these tables.
`-o` *`file`* `--output`	Places the compiler output in the named file. When multiple input files are specified on the command line, this option is useful only if the output is an executable file (by default, XCC names it `a.out`).
`-S` `--assemble`	Compiles the input files without assembling. Output files contain assembler code and are named by replacing the input file's extension with `.s`. If a single input file is specified on the command line, you can override the output file name with the `-o` option.
`-show`	Prints the compilation phases as they execute.

Table 2–2. Compiler Output Options (continued)

Option	Description
`-v` `--verbose`	Prints the compiler version and compilation phases as they execute.
`-version` `--version`	Prints the compiler version.

`-x` *`language`*	Uses *`language`* as the source language for the input files that follow. This option applies to all the input files that follow it until the next `-x` option is seen on the command line. Possible values for *`language`* are:
	`c`
	`c-header`
	`cpp-output`
	`c++`
	`c++-header`
	`c++-cpp-output`
	`assembler`
	`assembler-with-cpp`
	The special value `none` turns off any previous language specification.

2.2 Preprocessor Options

Table 2–3 describes the options that control the GNU C preprocessor.

Table 2–3. Preprocessor Options

Option	Description
`-A`*`question`*`(`*`answer`*`)` `-A`*`question`*`=`*`answer`	Asserts the *`answer`* to *`question`*, so it can be tested in the source code using preprocessor directive `#if #`*`question`*`(`*`answer`*`)`. For example, `#if #cpu(xtensa)` tests if the target of the compilation is an Xtensa processor.
`-B` *`dir`*	Equivalent to `-Ldir -isystem dir/include`.
`-C` `--comments`	Retains C and C++ comments after preprocessing. Normally the preprocessor strips comments before producing its output.
`-CC`	Retain all comments, including comments inside macros.
`-dD`	Prints macro names and their expansions in the preprocessor output. You must use `-E` in order to use `-dD`.

Table 2–3. Preprocessor Options (continued)

Option	Description
-dM	Prints all macros in effect at the end of preprocessing, including predefined macros. You must use -E in order to use -dM. xt-xcc -E -dM *empty-file* will print all predefined macros.
-dN	Like -dD, but prints only macro names and not their expansions. You must use -E in order to use -dN.
-D*name*=*value*	Defines macro *name* as *value*. Using this option is equivalent to having the line #define *name value* appear before any other line in the source file.
-D*name* --define-macro *name*	Equivalent to -D*name*=1.
-H --trace-includes	Prints to standard error the names of all header files used during preprocessing.
-I*dir* --include-directory=*dir*	Adds *dir* to the list of directories that are searched for header files before the standard system directories. If more than one -I option is specified, directories are searched in the order they appear on the command line.
-idirafter *dir* --include-directory-after *dir*	Adds *dir* to the header file search path, but only after all directories specified with -I and the standard system directories.
-imacros *file* --imacros *file*	Preprocesses *file* for any macro definitions. Any code in *file* is discarded, and the only effect of this option is to define or undefine macros. All -U and -D options are processed before any -imacros options, so macros defined with -D or undefined with -U are applied to *file*.
-include *file* --include *file*	Acts as if the line #include *file* appears before any other line in the source file. Note that -D and -U options are processed before any -include options. This means that any macros defined with -D or undefined with -U are applied to *file*.
-iprefix *prefix* --include-prefix *prefix*	Specifies *prefix* as the prefix for subsequent -iwithprefix and -iwithprefixbefore options.
-iquote *dir*	Adds *dir* to the header file search path before all directories specified with -I and the standard system directories, but only for header files requested with #include "file" and not for those requested with #include <file>.

Table 2–3. Preprocessor Options (continued)

Option	Description
`-isystem` *`dir`*	Adds *`dir`* to the header file search path after all directories specified with `-I`, but before the standard system directories.
`-iwithprefix` *`dir`* `--include-with-prefix` *`dir`*	Appends *`dir`* to the prefix specified previously with `-iprefix`, and adds the resulting directory to the header file search path in the same place as `-idirafter`.
`-iwithprefixbefore` *`dir`* `--include-with-prefix-before` *`dir`*	Appends *`dir`* to the prefix specified previously with `-iprefix`, and adds the resulting directory to the header file search path in the same place as `-I`.

`-M`
`--dependencies`
 	
`-MM`
`--user-dependencies`
	Prints a rule that describes dependencies of the input file and header files it includes. The rule is sent to the standard output in a format that can be used in makefiles. With `-M`, all included header files are listed. With `-MM`, header files found in system header directories are skipped. These options imply the `-E` option, and the compilation ends after preprocessing.

`-MD`
`--write-dependencies`
 	
`-MMD`
`--write-user-dependencies`
	Similar to `-M` and `-MM`, except that the output is written to a file named by replacing the input file's extension with `.d`. Therefore, these options do not alter the rest of the compilation process.

`-MF`	Specifies the output file for make dependencies
`-MG` `--print-missing-file-dependencies`	Treats missing header files as generated files and assumes they are in the source directory. They must be used together with`-M` or `-MM`. Not supported with `-MD` or `-MMD`.
`-MP`	Adds phony targets to make dependencies.
`-MQ`	Specifies target name with quoting for make dependencies.
`-MT`	Specifies target name for make dependencies.
`-nostdinc` `--no-standard-includes`	Does not search the standard system directories for header files.
`-nostdinc++`	Does not search the standard C++ directories for header files. However, the standard C directories are still searched.
`-P` `--no-line-commands`	Does not generate `#line` directives during preprocessing. Note that this option might cause error messages to report the wrong line number or source file paths.

Table 2–3. Preprocessor Options (continued)

Option	Description
`-traditional-cpp` `--traditional-cpp`	Attempts to support some aspects of traditional C preprocessors.
`-U`*`name`* `--undefine-macro` *`name`*	Undefines macro *`name`* to the preprocessor. Using this command is equivalent to having the line `#undef` *`name`* before any other line in the source file. When `-D`*`name`* and `-U`*`name`* both appear on the same command line, the one appearing later takes precedence.
`-Wp,`*`option`*	Passes *`option`* directly to the preprocessor. Use this for those preprocessor options that are not recognized by the XCC driver.

2.3 Language Dialect Options

Table 2–4 describes the options that control the dialects of C or C++ accepted by the compiler.

Table 2–4. Language Dialect Options

Option	Description
`-ansi` `--ansi`	In C mode, disables GNU extensions, such as `asm` and `inline` keywords and C++ style `//` comments; enables ANSI trigraph feature; predefines the preprocessor macro `__STRICT_ANSI__`.
`-fcheck-new`	Checks that the pointer returned by `operator new` is non-null. This check is normally unnecessary.
`-fconserve-space`	In C++, places uninitialized global variables in the common block, not in the `.bss` section.
`-fdiagnostics-show-location=`	Indicates how often source location information should be emitted [once|every-line].
`-fdiagnostics-show-option`	Show related command line options in diagnostic messages
`-felide-constructors`	In C++, removes constructors when this seems plausible. This may result in incorrect code when constructors have side effects.
`-fexceptions`	In C++ mode, enables support for exception handling. This option is off by default to minimize code size and improve performance. You are alerted to turn it on, if needed.
`-fextended-identifiers`	Allow universal characters in identifier names
`-ffor-scope` `-fno-for-scope`	In C++ mode, if `-ffor-scope` is specified, limits the scope of a variable declared in the initialization of a `for` loop to the `for` loop. If neither is specified, limits the scope, warns, but allows old style code that would otherwise be invalid.

Table 2–4. Language Dialect Options (continued)

Option	Description
`-ffreestanding` `-fno-freestanding`	Asserts that compilation takes place in a freestanding environment where the standard library might not be available.
`-fhosted` `-fno-hosted`	Asserts that compilation does not take place in a hosted environment where the entire standard library is available. `-fhosted` is equivalent to `-fno-freestanding` and `-fno-hosted` is equivalent to `ffreestanding`.
`-fms-extensions`	Accepts some non-standard Microsoft extensions.
`-fno-asm`	In C mode, disables recognition of `asm`, `inline` or `typeof` keywords. In C++ mode, disables recognition of `typeof`.
`-fno-default-inline`	Do not assume `inline' for functions defined inside a class scope (C++).
`-fno-dollars-in-identifiers`	Disallows `$` in identifier names.
`-fno-implicit-templates`	Does not implicitly instantiate templates.
`-fno-signed-bitfields` `-funsigned-bitfields`	Treats bitfields as unsigned, which are signed by default.
`-fno-rtti`	Do not generate C++ run-time type information.
`-fno-threadsafe-statics` `-fthreadsafe-statics`	Do not emit extra code for thread-safe initialization of local statics
`-fpack-struct`	Packs all structure members together without holes. This option results in less efficient code being generated. Code compiled with this option will not be binary compatible with code not compiled with this option, including system libraries.
`-fpermissive`	Allow some nonconforming code to compile.
`-fpreprocessed`	Tell preprocessor that input has already been preprocessed.
`-frepo`	Enables automatic template instantiation.
`-fshort-enums`	Allocates to an enum only as many bytes as needed to hold the enum.
`-fsigned-char` `-fno-unsigned-char`	Makes `char` type signed, like `signed char`, which is unsigned by default.

Table 2–4. Language Dialect Options (continued)

Option	Description
-std=c++98	Support ISO C++ from 1998
-std=c89	Support ISO C from 1990
-std=c99	Support ISO C from 1999
-std=c9x	Support ISO C from 1999
-std=gnu++98	Support ISO C++ from 1998, with GNU extensions
-std=gnu89	Support ISO C from 1990, with GNU extensions
-std=gnu99	Support ISO C from 1999, with GNU extensions
-std=gnu9x	Support ISO C from 1999, with GNU extensions
-std=iso9899:1990	Support ISO C from 1990
-std=iso9899:199409	Support ISO C from 1990, with 1994 amendments
-std=iso9899:1999	Support ISO C from 1999
`-trigraphs` `--trigraphs`	Supports ANSI C trigraphs.

2.4 Warning Control Options

The XCC compiler can produce warnings for numerous situations. To request warnings required by strict ANSI standard C, use the `-pedantic` option; to request warnings about code constructs that are generally considered questionable, use the `-Wall` (and for even more checks the `-W`) option.

Table 2–5 lists the options for controlling compiler warnings.

Table 2–5. Warning Options

Option	Description
`-pedantic` `--pedantic`	Issues warnings required by strict ANSI standard C.
`-pedantic-errors` `--pedantic-errors`	Same as `-pedantic`, but treats warnings as errors.

`-w`
`-woffall`
`--no-warnings`
	Disables all warnings. These options override any other warning options, regardless of their order.

`-Werror`	Treats all warnings as errors.

Table 2–5. Warning Options (continued)

Option	Description
-Wall --all-warnings	Enables most warning messages. It implies: -Wchar-subscripts -Wcomment -Wformat -Wimplicit -Wmain -Wparentheses -Wswitch -Wunused
-W --extra-warnings	Enables extra warning messages: when comparing signed and unsigned values, may generate a wrong result, when an unsigned value is compared < 0 or >= 0, when a function may return with or without a value, when values returned by a conditional expression have different types, when a storage class specifier is not the first thing in a declaration. Note that these extra warnings are not enabled by -Wall.
-Waddress	Warn about suspicious use of memory addresses.
-Waggregate-return	Warns about functions that return structures, unions or arrays.
-Wbad-function-cast	Warns when a function call is cast to a non-matching type.
-Wc++-compat	Warn about code that is not valid C++.
-Wcast-align	Warns about pointer casts that increase the required alignment, as in casting a char * into an int *.
-Wcast-qual	Warns about pointer casts that discard type qualifiers.
-Wchar-subscripts	Warns about array subscripts whose type is char.
-Wcomment	Warns about nested comments.
-Wconversion	Warns about possibly confusing type conversions: when a type conversion for a function argument is different with or without a prototype, or when a negative integer number is converted into an unsigned type.
-Wctor-dtor-privacy	Warn when all the constructors or destructors of a class are private (C++).
-Wdeclaration-after-statement	Warn when a declaration is found after a statement in a block.
-Weffc++	Warns about violation of some style rules from Effective C++.
-Werror=	Treat the specified warnings as errors.
-Wextra	Enable extra warnings.
-Wfatal-errors	Stop compiling at the first error.
-Wfloat-equal	Warn about floating-point equality comparisons.

Table 2–5. Warning Options (continued)

Option	Description
-Wformat	Warns about format mismatches in printf, scanf and related functions.
-Wformat-nonliteral	Warn about printf/scanf formats that are not string literals.
-Wformat-security	Warn about printf/scanf formats that may be security problems
-Wformat-y2k	Warn about strftime formats which may yield a two-digit year.
-Wformat=2	Enable additional format warnings.
-Wimplicit	Implies -Wimplicit-function-declaration and -Wimplicit-int.
-Wimplicit-function-declaration	Warns when a function is declared implicitly.
-Wimplicit-int	Warns when a type is not specified in a declaration, and it defaults to int.
-Wimport	Warn about use of #import.
-Wlarger-than-	Warn if an object larger than the specified size is defined.
-Wmain	Warns if main has a non-conforming type; it should have external linkage, return int, and accept zero, two or three appropriately typed arguments.
-Wmissing-braces	Warns if an aggregate initializer is not fully bracketed.
-Wmissing-declarations	Warns if the definition of a global function is not preceded by a previous declaration. This option is useful for detecting global functions that are not declared in header files.
-Wmissing-prototypes	Same as -Wmissing-declarations, but the previous function declaration must provide a prototype.
-Wmissing-field-initializers	Warn if a structure initializer is missing some fields.
-Wmissing-format-attribute	Warn about function pointers that might be candidates for format attributes.
-Wmmissing-include-dirs	Warn if a user-supplied include directory does not exist.
-Wno-attributes	Do not warn about unexpected attributes.
-Wnested-externs	Warns about extern declarations that are not at the file scope level.
-Wno-deprecated	In C++, does not warn about uses of deprecated features.
-Wno-deprecated-declarations	Do not warn about uses of __attribute__((deprecated)) declarations.
-Wno-div-by-zero	Do not warn about integer division by zero.
-Wno-endif-labels	Do not warn about text following #else and #endif.
-Wno-error=	Do not treat the specified warnings as errors.
-Wno-format-extra-args	Do not warn about extra format arguments.

Table 2–5. Warning Options (continued)

Option	Description
`-Wno-format-zero-length`	Do not warn about zero-length formats
`-Wno-int-to-pointer-cast`	Suppress warnings for casts to pointer type f rom an integer of a different size.
`-Wno-invalid-offsetof`	Suppress warnings for using the offsetof macro with non-POD types.
`-Wno-long-long`	Does not warn about the use of `long long` variables, even if the `-pedantic` option is used.
`-Wno-multichar`	Do not warn about multicharacter constants.
`-Wno-non-template-friend`	In C++, disables warnings when non-templatized friend functions are declared within a template.
`-Wno-pmf-conversions`	Does not warn about converting a bound pointer to a member function to a plain pointer.
`-Wno-pointer-to-int-cast`	Suppress warnings for casts from a pointer type to an integer of a different size.
`-Wno-pragmas`	Do not warn about misuses of pragmas.
`-Wno-variadic-macros`	Do not Warn about variadic macros used in pedantic mode.
`-Wno-overflow`	Do not warn about overflow in constant expressions.
`-Wnon-virtual-dtor`	Warns if a class has virtual functions, but is a non-virtual destructor.
`-Wnonnull`	Warn about passing null for arguments marked with a nonnull attribute.
`-Wnormalized=`	Control warnings about normalization of characters in identifier names.
`-Wold-style-cast`	Warn if an old-style (C-style) cast to a non-void type is used (C++).
`-Wold-style-definition`	Warn when an old-style function definition is used.
`-Woverlength-strings`	Warn about string constants too long to be portable
`-Woverloaded-virtual`	Warns when a derived class function declaration may be an error in defining a virtual function; i.e., the derived virtual function has the same name, but not the same signature of a virtual function declared in the base class.
`-Woverride-init`	Warn about initialized field overridden when using designated initializers.
`-Wpacked`	Warn about structures where packed attribute does not reduce size.
`-Wpadded`	Warn if padding is included in a structure.
`-Wparentheses`	Warns if missing parentheses may lead to confusing code constructs.

Table 2–5. Warning Options (continued)

Option	Description
-Wpointer-arith	Warns if a pointer to a function or to void is used in arithmetic.
-Wpointer-sign	Warn about pointer assignments or argument passing with different signedness.
-Wredundant-decls	Warns about multiple declarations of the same object.
-Wreorder	Warn and rearrange the order of member initializers to match their declaration order.
-Wreturn-type	Warns if a function definition does not specify a return type (it defaults to int) or if a function whose return type is not void returns with no value (including the implicit return at the end of function).
-Wsequence-point	Warn about order of evaluation that is not defined by sequence points.
-Wshadow	Warns when one local variable shadows another.
-Wsign-compare	Warns when comparing signed and unsigned values may lead to a wrong result.
-Wsign-promo	Warns when an unsigned or enumerated type is promoted to a signed type over an unsigned type.
-Wstrict-aliasing	Warn about code that may violate rules for -fstrict-aliasing.
-Wstrict-prototypes	Warns about function declarations and definitions that do not include the argument types.
-Wswitch	Warns when a switch has an index of an enumerated type and not all values of the enumeration are covered by the switch or when some values outside of the enumeration are covered by the switch.
-Wswitch-default	Warn about switch statements without a default case.
-Wswitch-enum	Warn about switch cases that do not match enum type values.
-Wsystem-headers	Do not suppress warnings from system header files.
-Wtraditional	Warns about certain constructs that behave differently in traditional C from ANSI C.
-Wtrigraphs	Warns if any trigraphs are encountered.
-Wundef	Warns if an undefined identifier is defined in an #if directive.
-Wunknown-pragmas	Warn about unknown pragmas.
-Wunused-function	Warns if a static function is used or declared, but not defined.
-Wunused-label	Warns if a label is defined but not used.
-Wunused-macros	Warn if a macro is defined but not used.
-Wunused-parameter	Warns if a function parameter is not used.

Table 2–5. Warning Options (continued)

Option	Description
`-Wunused-tie-intrinsic-result`	Warn if TIE intrinsic result is not used
`-Wunused-variable`	Warns if a local or static variable is not used.
`-Wunused-value`	Warns when a statement computes a value that is not used.
`-Wunused`	Implies -Wunused-function -Wunused-label -Wunused-variable -Wunused-value In conjunction with `-W`, it also implies -Wunused-parameter.
`-Wvolatile-register-var`	Warn about volatile register variables
`-Wwrite-strings`	Gives string constants the type `const char *`, and performs type checking accordingly.

2.5 Debugging Options

Table 2–6 describes the options that control information generated by the compiler to aid you in debugging your application.

Table 2–6. Debugging Options

Option	Description
`-clist`	Produces a `.w2c.c` file that contains a C-level view of code transformations performed by XCC. This option is not applicable to C++ input files. For more information, see Section 4.10.1.
`-fmessage-length=`*n*	Tries to format error messages so that they fit in lines of approximately *n* characters. If *n* is 0, the default value, then each error message appears on a single line.

`-g`
`-gdwarf-2`
`--debug`
	Generates debugging information in DWARF version 2 format. Object and executable files will be larger, but easier to debug. For most precise source-level debugging, this option should be used with optimizations disabled (`-O0`). Aggressive code transformations performed by XCC at higher optimization levels, such as `-O2` and `-O3` (especially when used in conjunction with `-ipa`), may obscure the debugging information.

`-g`*level*	Generates different amounts of debugging information. With `-g0`, does not generate any debugging information. With `-g1`, generates minimal debugging information. With `-g2` and `-g3`, generates full debugging information. The option `-g` is equivalent to `-g2`.

Table 2–6. Debugging Options (continued)

Option	Description

`-keep`
`-keep_min`
`-save-temps`
`--save-temps`
	Saves intermediate files generated by the compiler; by default, they are discarded. Intermediate files after preprocessing are named by replacing the input file's extension with "`.i`" for C input files and `.ii` for C++ input files. Assembly code generated by the compiler is saved in the same way as with the `-S` option. By using these options you can have both an executable to run and a human-readable assembly file to examine with just one compilation step. The option "-keep_min" keeps only the preprocessed, assembly and object files. The others also keep intermediate files that are not generally useful to the programmer.

`-print-file-name=`*`library`* `--print-file-name=`*`library`*	Prints the full absolute name of the library that would be used when linking. The compiler does nothing else.
`-print-libgcc-file-name` `--print-libgcc-file-name`	Equivalent to `-print-file-name=libgcc.a`.
`-print-prog-name=`*`program`* `--print-prog-name=`*`program`*	Like `-print-file-name`, except searches for a program.

2.6 Optimization Options

Table 2–7 describes the options that control the level and type of optimizations performed by the compiler.

Table 2–7. Optimization Options

Option	Description
`-O0`	Performs no optimization. Does not inline any functions (but, by default, removes unused static functions). This is the default when no other optimization level is specified.
`-O1`	Performs local (single basic block) optimizations: constant folding and propagation, common subexpression elimination, peephole optimizations and local register allocation. Considers for inlining only those functions that are explicitly marked with the `inline` specifier or defined inside the class scope.

`-O`
`-O2`
`--optimize`
	Does all the optimizations implied by `-O1`, plus global (function level) optimizations based on data-flow analysis: dead code elimination, partial redundancy elimination, strength reduction, global register allocation, instruction scheduling, loop unrolling. Performs the heuristic-based inlining of static functions in addition to functions explicitly marked as `inline`. Optimizations at this level are virtually guaranteed to improve performance relative to `-O0` and `-O1`.

Table 2–7. Optimization Options (continued)

Option	Description
-O3	Does all the optimizations implied by -O2, plus additional loop transformations based on dependence analysis, such as interchange and outer unrolling. This optimization level is required to enable the -LNO option group and the automatic vectorization feature. Optimizations performed at -O3 are generally more aggressive and may, in some cases, degrade performance relative to -O2. This optimization level may cause changes in floating-point results due to the relaxation of operation ordering rules and the automatic use of recip and rsqrt instructions.
-Os	Optimizes for space. This option can be used in conjunction with any optimization level, but the optimizations are guided by the goal of minimizing the code size. If no other optimization level is specified, -Os implies -O2. It also implies -mno-target-align. Note that even with -Os, the compiler will generate FLIX (VLIW) instructions if available to improve performance. Use -OPT:space_flix=1 to limit the use of FLIX to cases where code size is not impacted.
-OPT:	Option group for controlling miscellaneous optimizations. This option group applies only to -O2 and -O3 optimization levels. For more details, refer to Section 4.3.
-fassociative-math	Allow re-association of floating point operations. This is the default at -O3
-fb_create *filename* -fb_create_32 *filename* -fb_create_64 *filename*	Instruments the code to generate feedback information into *filename* using 32- or 64-bit counters. The default uses 32-bit counters. For more details refer to Section 4.4. Note that in order to use this feature you must both compile and link with this flag.
-fb_create_HW *filename*	Instruments the code to generate feedback information using 64-bit counters on real hardware. Resultant code must be run via Xplorer or standalone Xplorer. For more details refer to Section 4.4. Note that in order to use this feature you must both compile and link with this flag.
-fb_opt *filename*	Uses feedback information in *filename* to improve performance and code size. For more details refer to Section 4.4.
-fb_reorder	Enables function reordering based on the feedback information when used in conjunction with -fb_opt. This option implies -ffunction-sections.
-ffast-math	Allows optimizations that may violate ANSI or IEEE arithmetic rules.
-fopt-gen	Generate a compiler option file that describes the optimization level for every function.
-fopt-use	Specify a file that lists functions along with the optimization level to use for those functions. See Section 4.2.
-fno-pragma-loop-count	Disables the use of #pragma loop_count described in Section 4.8.
-fstrict-aliasing	Equivalent to -OPT:alias=typed. See Table 4–1.
-fno-strict-aliasing	Equivalent to -OPT:alias=any. See Table 4–1.

Table 2–7. Optimization Options (continued)

Option	Description
`-fno-strict-overflow`	Disables optimizations that assume strict signed overflow rules or pointer semantics.
`-fno-unroll-loops`	Disables unrolling of inner loops. Note that at `-O3`, outer loops might still be unrolled and in rare cases inner loops can be interchanged into outermore positions and then be unrolled.
`-fno-unsafe-math-optimizations`	Do not enable floating point optimizations that do not strictly conform to language or IEEE floating point rules. This option is the default at all optimization levels except `-O3`.
`-freciprocal-math`	Allow the reciprocal of a value to be used instead of dividing by the value. This is the default at `-O3`.
`-funsafe-math-optimizations`	Enable floating point optimizations that do not strictly conform to language or IEEE floating point rules. This option, for example, allows the reordering of floating point expressions even though floating point arithmetic is not associative. It allows the replacing of a divide or sqrt operation with their reciprocal. This option is the default at `-O3` and off by default otherwise.
`-hwpg` `-hwpg=`*n*	Instruments the code for profiling when running on real hardware, using timer *n* or performance counters if *n* is not specified. Resultant code must be run via Xplorer or `xt-gdb`. When this option is used only for linking, it causes periodical program interruption at run time to record the current program counter allowing `xt-gprof` or Xplorer to generate a flat profile of the application. When the option is also used for compiling, it instruments every call to generate a dynamic call graph allowing the profiler to estimate hierarchical profiles but also perturbing the executable.
`-ipa` `-IPA`	Performs interprocedural analysis and optimization. This option can be used in conjunction with any optimization level, but its benefits are most visible when it is combined with `-O2` or `-O3`, with or without `-Os`. For more details, refer to Section 4.6.
`-LNO:`	Option group for controlling loop-nest optimizations. This option group requires `-O3` optimization level. For more details, refer to Section 4.10.
`-mcoproc`	Enables automatic use of register files from standard C/C++ code other than the standard AR register file. For example, the compiler might infer the use of HiFi or ConnX instructions that access HiFi or ConnX register files. Higher performance but perhaps not safe when compiling interrupt handlers.

2.7 Inlining Options

XCC includes a phase that performs function inlining and removal of unused functions. Table 2–8 summarizes options that can be used to control the inliner.

Table 2–8. Inlining Options

Option	Description
`-fkeep-inline-functions`	Generates separate code for a function marked as `inline`, even if all calls to that function are inlined. Does not affect unmarked functions even if they are inlined.
`-fkeep-static-functions`	Generates separate code for a static function, even if all calls to that function are inlined.
`-fno-inline`	Ignores the `inline` specifier and does not inline any functions. This is the default only at `-O0`.
`-fno-inline-functions`	Does not perform the heuristic-based inlining of static functions (functions explicitly marked as `inline` are not affected). This is the default at `-O0` and `-O1`. It has no effect if you compile with `-ipa`.
`-INLINE:=off`	Skips the inlining and dead function removal phase completely. This may increase the code size, especially for C++ programs.
`-INLINE:aggressive=off` `-INLINE:aggressive=on`	Makes the heuristic-based inlining less or more aggressive. More aggressive inlining often improves performance at the expense of the increased code size. The default is `on` with interprocedural optimization (`-ipa`) except when optimizing for space (`-Os`), and `off` otherwise.
`-INLINE:dve=off`	Disables deletion of static variables that are never used, an optimization for saving memory. By default, they are deleted. This option only applies to compilations without `-ipa`.
`-INLINE:must=`*`function`*	Forces inlining of *`function`*, if possible.
`-INLINE:never=`*`function`*	Prevents inlining of *`function`*.
`-INLINE:preemptible`	Includes global (in addition to static) functions as candidates for heuristic-based inlining. This is the default only with interprocedural optimization (`-ipa`).
`-INLINE:requested`	Overrides the compiler's inlining heuristics and forces inlining of those functions that are explicitly marked with the `inline` specifier or defined inside the class scope. The compiler may also inline additional functions. Note that in previous versions of XCC, `-INLINE:requested` was equivalent to the current behavior of `-INLINE:requested_only`.
`-INLINE:requested_only`	Overrides the compiler's inlining heuristics and forces inlining of those and only those functions that are explicitly marked with the `inline` specifier or defined inside the class scope.

2.8 Code Generation Options

Table 2–9 describes options that control the code generation of the compiler.

Table 2–9. Code Generation Options

Option	Description
`-fcommon`	Allocates uninitialized, non static, global variables in the common block and not in the `.bss` section. This avoids a multiple definition link error when a variable is declared without the `extern` specifier in two different files.
`-ffunction-sections`	Emits each function in a separate section named `.text.`*`function_name`*. When used in conjunction with the `-Wl,-gc-sections` linker option, this allows for removal of unused functions.
`-fno-builtin`	Does not replace built-in functions with inlined code. See Section 3.2.
`-fno-zero-initialized-in-bss`	Disables placing of zero-initialized variables in the `.bss` section.
`-fpic` `-fPIC`	Generates position-independent code. This option must be used for code that might be dynamically loaded at an arbitrary location, such as code that will be linked into a shared-library with the `-shared` option.
`-mcbox`	Prevents bundling of two load operations in the same FLIX instruction unless one of the load addresses is marked with `#pragma ymemory` and the other one is not. See Section 4.11 for more details.
`-mflush-tieport`	Serializes TIE port references by generating `EXTW` instructions.
`-mfused-madd` `-mno-fused-madd`	Controls generation of floating-point multiply/add (`MADD.S`) or multiply/subtract (`MSUB.S`) instructions. With `-fused-madd`, the compiler tries to combine floating-point multiply and add/subtract operations. The fused multiply add/subtract instructions do not round the intermediate result and may produce results with *more* bits of precision than specified by the IEEE 754 standard. The default is `-mfused-madd` at the `-O3` optimization level and `-mno-fused-madd` at all other levels. This option has no effect in configurations without the floating-point coprocessor.
`-mno-generate-flix`	Disables generation of any FLIX instructions, but allows them in `asm` statements.
`-mno-flix`	Disables generation and use of any FLIX instructions.
`-mno-l32r-flix`	Prevent generation of `L32R` in anything other than slot 0 of a multi-slot FLIX instruction and prevent bundling `L32R` together with any other load or store. This can be used to avoid exceptions described in Section "Instruction RAM Load and Store" (18.3.1) of the *Xtensa Microprocessor Data Book*. This flag is off by default except for Xtensa processor configurations with two load/store units and an Instruction RAM.

Table 2–9. Code Generation Options (continued)

Option	Description
`-mno-mul6, -mno-mul32, -mno-div32`	Suppress the generation of code that utilizes the MUL16, MUL32, and 32-bit integer divider options, respectively. The compiler will generate code that does not depend on these hardware configuration options even if they are present in the core configuration. These options are useful if you wish to provide a compiled library which will work on multiple core configurations, regardless of whether these hardware options exist. NOTE: These options only control generated code for the current compilation. Any pre-compiled libraries or modules that you link in may or may not contain these instructions. You may need to recompile modules from source code and/or modify and rebuild your configuration to obtain the desired result.
`-mno-reorder-tieport`	Prevents reordering of TIE port references.
`-mno-serialize-volatile`	Does not separate volatile references with MEMW instructions. See Section 3.3 for more details.
`-mno-zero-cost-loop`	Disables use of the zero-overhead loop instructions. On configurations that support zero-overhead loop instructions, by default, the compiler tries to use these instructions at optimization levels `-O2` and `-O3`.
`-mshift32`	Forces the result of shifting an `int` value by `32` to be well defined: `0` for a left shift, an unsigned right shift, or a signed right shift of a non-negative value; `0xFFFFFFFF` for a signed right shift of a negative value. Note that both C and C++ standards declare shifting of a `32`-bit value by `32` as undefined.
`-mzero-init-data`	Same as `-fno-zero-initialized-in-bss`.

2.9 Assembler Options

Options described in Table 2–10 only affect the assembly phase. Therefore, their effects will not be seen in the compiler-generated assembly code, but only in the disassembled object code.

Table 2–10. Assembler Options

Option	Description
`-mlongcalls`	Enables transformation of direct calls into indirect calls to allow calls across a greater address range. When the assembler cannot determine that the target of a direct call is within the effective offset range of the `CALL` instruction, it translates the `CALL` instruction into an `L32R` instruction—to load the target address into the return address register—followed by a `CALLX` instruction.
`-mno-target-align`	Disables automatic alignment of branch targets. By default, the assembler tries to reduce branch penalties by widening density instructions to align branch targets and instructions following calls. This option is also enabled with the `-Os` option.
`-mrename-section-`*`old=new`	Renames the section old to new when generating an object file. This option cannot be used in conjunction with the -ipa option; instead, use `__attribute__((section("`*`name`*`"))`, which is described in Section 3.11.
`-mtext-section-literals`	Generates literals interspersed in the text section in order to keep them as close as possible to their references. When using configurations with PC-relative `L32R` instructions, this may be necessary for very large functions. By default, literals are placed in a separate section (`.literal`). This option is ignored on configurations with absolute `L32R` instructions.
`-Wa,`*`option`	Passes *`option`* directly to the assembler. This is used for those assembler options that are not recognized by the XCC driver. For more information, see the *GNU Assembler User's Guide.*

2.10 Linker Options

Table 2–11 describes options that control the GNU linker. Refer to the *GNU Linker User's Guide* for more details.

Table 2–11. Linker Options

Option	Description
`-e`*`ADDRESS`*	Sets the start address.
`-B` *`dir`*	Equivalent to `-Ldir -isystem dir/include`.
`--gc-sections`	Remove unused functions. Should be used in conjunction with -ffunction-sections.
`-L`*`dir`* `--library-directory` *`dir`*	Adds directory *`dir`* to the list of directories to be searched for libraries specified with the `-l` option.
`-l`*`name`*	Searches the library `lib`*`name.a`* during linking. Directories searched are standard system library directories and those specified with the `-L` option.

Table 2–11. Linker Options (continued)

Option	Description
`-nostdlib` `--no-standard-libraries`	Implies `-nodefaultlibs` and `-nostartfiles`.
`-s`	Removes all symbol table and relocation information from the executable.
`-shared` `--shared`	Creates a shared library. Currently, this option cannot be used in the interprocedural compilation mode (`-ipa`). The option is relevant only when compiling for the Linux operating system.
`-static` `--static`	Does not link with shared libraries. The option is relevant only when compiling for the Linux operating system.
`-T` *`scriptfile`*	Uses *`scriptfile`* as the linker script rather than the default files.
`-u` *`symbol`* `--force-link` *`symbol`*	Pretends *`symbol`* is undefined so as to force linking of library modules to define it.
`-Wl,`*`option`*	Passes *`option`* directly to the linker. This is used for those linker options that are not recognized by the XCC driver. For more information, see the *GNU Linker User's Guide.*

2.11 Xtensa-Specific Options

Table 2–12 describes Xtensa-specific options dealing with configuration and platform management.

Table 2–12. Xtensa-Specific Options

Option	Description
`-mlsp=`*`lspname`*	Uses *`lspname`* as the linker support package. For more information about linker support packages, see the *Xtensa Linker Support Packages (LSPs) Reference Manual.*
`--xtensa-core=`*`core`*	Uses *`core`* as the target processor configuration. For more information about selecting a processor configuration, see the *Xtensa Software Development Toolkit User's Guide.*
`--xtensa-params=`*`tdk`*	Uses *`tdk`* as the TIE development kit directory. For more information, see the *Xtensa Software Development Toolkit User's Guide and Tensilica Instruction Extension (TIE) Language User's Guide*.
`--xtensa-system=`*`registry`*	Uses *`registry`* as the Xtensa processor core registry. For more information about Xtensa core registries, see the *Xtensa Software Development Toolkit User's Guide*.

3. Extensions to C and C++

XCC supports many extensions to ANSI standard C (C99) and C++. Features that require new syntax (`//` comments in C) or keywords (`inline` in C or) are turned off with the `-ansi` option, however, the alternate keywords `__inline__` and `__asm__` will continue to work. Use the `-pedantic` option to request warnings that are required by strict ANSI standards.

3.1 Intrinsic Functions for Xtensa Instructions

Sometimes you might want to ensure that the compiler uses a particular instruction or series of instructions. You can do this by writing the program in assembly language or using inline assembly in a C or C++ program, but these solutions can be tedious. As an alternative, XCC provides intrinsic functions for most Xtensa instructions and for all TIE prototypes. The compiler translates a call to one of these intrinsics into the corresponding Xtensa instruction or sequence of instructions.

These intrinsic functions are defined in header files. You must include the appropriate header file before you can use an intrinsic function. The various header files are shown in Table 3–13 below. The header file describing most core instructions can be found in `xtensa/tie/xt_core.h`. The header file describing TIE instructions from the user TIE file `tiefile.tie` can be found in `xtensa/tie/tiefile.h`. The other header files are only available when the corresponding option is included in the configuration. These files are installed in either the default `include` directory (`<`*`xtensa_root`*`>/xtensa-elf/arch/include`) or in the TIE development kit (`<`*`tdk`*`>/include`) directory. As both directories are on the default search path for header files, you do not have to specify absolute paths for the header files.

For brevity, the header files for vertical DSP coprocessors are not listed. See the individual user's guides for them.

Table 3–13. Xtensa Header Files

Header File Name	Description
`xtensa/tie/xt_booleans.h`	Boolean instructions in Coprocessor Option
`xtensa/tie/xt_core.h`	Core ISA instructions
`xtensa/tie/xt_density.h`	16-bit density instructions
`xtensa/tie/xt_DFP.h`	Double-precision floating-point Option
`xtensa/tie/xt_DFP_assist.h`	Double-precision floating point assist instructions
`xtensa/tie/xt_FP.h`	Single precision floating-point Option
`xtensa/tie/xt_ioports.h`	Diamond compatible ports and queue instructions

Table 3–13. Xtensa Header Files (continued)

Header File Name	Description
xtensa/tie/xt_MAC16.h	MAC16 Option
xtensa/tie/xt_misc.h	Miscellaneous Operations Option
xtensa/tie/xt_mul.h	16-bit and 32-bit Multiplier Option
xtensa/tie/xt_sync.h	Multiprocessor Synchronization Option
xtensa/tie/<*tiefile*>.h	User-defined TIE. The name <*tiefile*> may come from the tie file named <*tiefile*>.tie or from the name given in tc -name <*tiefile*> or from the TDB name when attaching the TIE file to a configuration in Xplorer.

Besides the intrinsic functions, the header files also define data types used by the intrinsics. The intrinsic names for all opcodes other than user TIE and DSP coprocessor opcodes are prefixed with XT_ .

If an instruction produces a single result (that is, if there is exactly one out operand and there are no inout operands), the intrinsic function returns that result as the return value, and any other in operands are the function parameters. Otherwise, if there are any inout operands, or if there are multiple out operands, the intrinsic function returns void and takes all the operands as parameters. The actual parameters provided for out and inout operands must be lvalues. (An lvalue is an expression referring to a named region of storage.) For example, when invoking an intrinsic, the out and inout arguments can be variable names, but cannot be literal constants or expressions like "x + 1".

Also, the type of out and inout arguments must match the type expected by the instruction or prototype; explicit casts or implicit type conversions are not allowed. For source operands that are immediate operands in the corresponding instructions, the intrinsic arguments must be constants with values that are known to the compiler. For example, in the code:

```
#include <xtensa/tie/xt_misc.h>
extern int x, y;
void sign_extend (int a, int b) {
     x = XT_SEXT (a, 12); // valid
     y = XT_SEXT (a, b); // invalid
}
```

The first statement, x = XT_SEXT (a, 12) is valid, and the compiler translates the XT_SEXT macro into a single instruction, such as:

```
sext a4, a2, 12/* a2 = a; a4 = result to be stored into x */
```

However, the second statement, `y = XT_SEXT (a, b)`, is not valid because the `sext` instruction requires an immediate operand and the second argument to `XT_SEXT, b,` is not a constant. The compiler reports this statement as an error.

3.1.1 Variable Shifts

Variable shifts on Xtensa processors use a pair of instructions. For example, the C expression `a = b << c` is implemented by issuing an `SSL` instruction that sets the `SAR` register followed by an `SLL` instruction that does the shift based on the value in the `SAR` register. This can potentially lead to problems when using intrinsics to do variable shifts. If the user invokes two intrinsics, one for each instruction, there is no way to guarantee that the compiler will not need to do another shift in between the two intrinsic calls. If that other shift also writes the `SAR` register, the intrinsics might not function properly. Therefore, the compiler also supports six intrinsics that each implement a different pair of instructions needed to perform a different type of variable shift. When using these paired intrinsics, the compiler is guaranteed to preserve the correct value of SAR prior to performing a shift.

```
int SSAI_SRC(int src1, int src2, immediate amount);
int SSR_SRC(int src1, int src2, int amount);
int WSR_SAR_SRC(int src1, int src2, int amount);
int SSR_SRA(int src, int amount);
unsigned SSR_SRL(unsigned src, int amount);
int SSL_SLL(int src, int amount);
```

These intrinsics are all included in `xtensa/tie/xt_core.h`.

3.1.2 MAC16

Many MAC16 instructions use the `MR` register file, which has irregular properties. For example, the `mx` operand of the `MULA.DD.LL.LDDEC` instruction can only designate either `MR` register `m0` or `m1`, while the `my` operand can only designate either `MR` register `m2` or `m3`. Because of this irregularity, the compiler cannot automatically allocate variables to the `MR` register file. You must directly specify the `MR` register numbers when using an intrinsic for one of these instructions. An intrinsic operand for an `MR` register must be an immediate value that is the `MR` register number. For example, `MULA.DD.LL.LDDEC(0,as,0,2)` generates an instruction that uses `m` registers `m0`, `m0`, and `m2` respectively. It is your responsibility to keep track of which values are in each `MR` register.

Many MAC16 instructions implicitly use the accumulator state. This state is set by some instructions and read later by other instructions. The compiler might also generate instructions that use this state to implement standard C multiply operations, and the compiler cannot always know if you, the application programmer, are currently using the

state to hold the result of an intrinsic call. In order to avoid such situations, only use the intrinsics locally. That is, make sure that there are no multiplies between setting the state via an intrinsic and reading the state.

3.2 Built-in Functions

XCC recognizes certain C library functions as built-in, and at an optimization level higher than `-O0`, XCC may replace calls to them with special code sequences. This often results in faster and sometimes smaller code, but it prevents linking against different implementations of these functions. The following functions are affected:

```
abs, fabs, labs, ffs, div, ldiv, ffloor, fceil, fmod, frem, memcpy,
memcmp, memset, bzero, bcmp, strcmp, strcpy, strlen, fsqrt, sin, cos
```

Unlike some older versions of GCC, XCC always treats `alloca` as a built-in function and replaces calls to it with simple instructions that adjust the stack pointer.

Special treatment of these functions is disabled with the flag `-fno-builtin`. A version of each function prefixed with `__builtin_` is also provided, and those versions are treated special even when the `-fno-builtin` flag is used.

3.3 Memory Consistency

C/C++ provides a sequential programming model in which every statement happens in the order written. In reality, to improve performance, the compiler can change the order, and, in particular, the relative order of two memory references that refer to distinct locations. Similarly, because the Xtensa processor is pipelined and contains internal buffers, the hardware might also change the relative order of two memory references, as seen by an outside agent, whenever the two memory references refer to different memory locations. Normally, these optimizations are safe, as well as effective. However, in a real system with multiple processor cores or independent devices, at times you may want to preserve memory ordering in certain portions of your program. For example, in a multiprocessor system, one processor might compute data into shared memory and then set a shared memory flag variable to indicate to another processor that the data is available. If either the compiler or the hardware reorders memory references, the second processor might see the flag being set before the data is actually available.

Programmers often try to guarantee sequential semantics by using the `volatile` attribute. This is different from using `volatile` to guarantee that a memory reference to a memory mapped device with side effects is not deleted. Unfortunately, the use of `volatile` to preserve ordering is inefficient and potentially dangerous. The C and C++ standards guarantee that two volatile references are not reordered with respect to each other, but they do not guarantee that a volatile reference is not reordered with respect to

a non-volatile one. Thus, to be safe, both flags and data must be marked as `volatile`, but doing so can be inefficient. Because `volatile` can be used for memory consistency, the compiler is forced to be overly conservative when volatile is used for devices with side effects. The Xtensa hardware may reorder memory references unless separated with a `MEMW` or other synchronization instruction. XCC separates all volatile references with a `MEMW` instruction by default, even though this is usually unnecessary for devices with side effects. The flag `-mno-serialize-volatile` instructs XCC to omit the `MEMW` instructions.

A safer and more efficient way to guarantee consistency is through the use of a pragma, as follows:

```
#pragma flush_memory
```

XCC insures that all data is effectively flushed to or from memory at the point of the pragma, so that all memory references before the pragma occur before any memory references after the pragma. XCC also places a `MEMW` instruction at the point of the pragma to insure that the hardware does not reorder references across the pragma.

Consider the following example.

```
for (i=0; i<n; i++) {
     data[i] = ...
}
#pragma flush_memory
flag_data_ready = 1;
```

All the data is guaranteed to be written before the flag, `flag_data_ready`, is written.

3.3.1 TIE Ports For LX cores

TIE ports (`lookups`, `queue`, `import_wire` and state with the `export` attribute) potentially have similar consistency issues. Refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* or the *Tensilica Instruction Extension (TIE) Language User's Guide* for details about these features. You might, for example, use a TIE output queue to produce data and then use a shared memory flag to signal that the data has been computed. By default, XCC assumes that references to different TIE ports are unrelated to each other or to memory, and hence can potentially be reordered. Similarly, by default, the Xtensa hardware might reorder a TIE port reference with respect to another, or to memory in the sense that the effects of a reference from a later instruction might become externally visible before the effects of an earlier reference. Note that neither XCC nor the hardware will reorder multiple references to the same TIE port. In addition, related TIE ports such as a QUEUE and its associated NOTRDY interface are considered to be one port, so that references to one of them are not reordered with respect to references to another.

XCC provides an option, `-mflush-tieport`, that guarantees that neither XCC nor the hardware will reorder a TIE port reference with respect to another or to memory. Given this option, XCC will serialize all TIE port references with respect to each other and to memory, and XCC will insert an `EXTW` instruction before and after each TIE port reference. Note that `EXTW` may take an arbitrary amount of cycles, as it must wait for all writes to output queues and all TIE lookup accesses to be completed.

The use of the `-mflush-tieport` flag is potentially expensive overkill in terms of slowing down the performance of the application. Instead, we recommend using the pragma `#pragma flush`. This pragma works similarly to `#pragma flush_memory`, except that it also affects the ordering of TIE ports. All memory or TIE port references issued before the pragma are guaranteed to complete before any memory or TIE port references issued after the pragma. The compiler will insert a single `EXTW` instruction at the point of the pragma. As the pragma affects memory as well as TIE ports, there is no need to use both pragmas.

Note that if you are not using any TIE ports or do not require them to be sequentially consistent, it is more efficient to use `#pragma flush_memory`. The use of `#pragma flush` generates an `EXTW` instruction rather than the `MEMW` instruction generated with `#pragma flush_memory`, and the `EXTW` instruction is potentially more expensive than the `MEMW` instruction.

Note that the use of the compiler option or the pragma guarantees that earlier accesses complete before later accesses, from the point of view of the processor. However, there is no way to guarantee that all system effects from earlier accesses are complete. If an output queue is connected to a deep external queue, an `EXTW` instruction will cause the processor to stall until all pushed data enters the external queue, not until all pushed data leaves the other end of the external queue.

TIE Ports and Deadlock

There are potential deadlock conditions when designing or programming a system with queues. If one processor is blocked waiting for a signal from an independent agent, and the independent agent is not sending the signal because it is in turn waiting for some signal from the processor, the system will deadlock. In addition to situations that might inherently lead to deadlock, deadlock might also result because the compiler or hardware reorders references to TIE ports and memory with respect to the original program order.

Consider a simple two processor system as shown in Figure 3–1. Each processor is sending data to the other via a small queue The first processor writes into a queue and then tries to get a response back from the second queue. The second processor reads data from the first queue and then sends back a response to the second queue. If the compiler for the first processor rearranges the references so that the first processor tries to read its input queue before it writes its output queue, the read will cause the proces-

sor to block while waiting for data before it has a chance to write its output data. The system will deadlock because the second processor will never write its queue until it receives its data from the first processor.

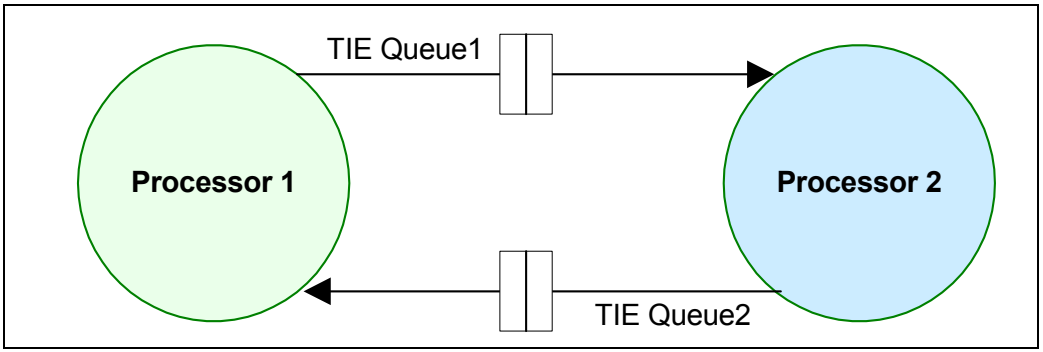

Figure 3–1. TIE Ports and Deadlock

To avoid this situation, you can add a `#pragma flush` in between the two queue references, or you can use the compiler flag `-mflush-tieport`. Both these choices, however, will cause the compiler to generate an `EXTW` instruction. For this example, you want to prevent the compiler from reordering references so that a later reference will not cause the processor to stall without allowing an earlier reference to complete. However, you do not care if a later reference becomes externally visible before an earlier one. In such situations, an `EXTW` instruction is not necessary. Instead you can use `#pragma no_reorder` or the compiler flag `-mno-reorder-tieport`. These options prevent the compiler from reordering references, but do not generate `EXTW` instructions. [1]

3.4 C++ Style Comments

C++ style comments, which begin with "`//`" and extend to the end of the line, may be used in C programs.

3.5 Inline Functions in C

You can use the `inline` function specifier to encourage the compiler to inline a particular function into its callers. This is a standard keyword in C++, but XCC also allows it in C programs. For more details on function inlining, see Table 2–7 on page 19 and Table 2–8 on page 22.

1. **Note:** For completeness the compiler also supports `#pragma no_reorder_memory`, which affects memory references but not TIE port references. However, if you do not have TIE port references, there is no performance penalty for using `#pragma no_reorder` instead of `#pragma no_reorder_memory`.

3.6 Support for `long long` Variables

Variables declared with type `long long` will be 64-bit integers. To make a constant of type `long long`, append `LL` to the constant. Similarly, you can specify that a constant have the type `unsigned long long` by appending `ULL`. For example,

```
long long foo = 0x123456789LL;

unsigned long long bar = 0xFFFFFFFFULL;
```

Variables of type `long long` may be used in expressions. They are treated in the same manner as any other built-in type. Because the base Xtensa core is a 32-bit architecture, operations with `long long` variables execute more slowly than with shorter integer types.

3.7 Restrict Pointers

XCC allows pointers to be declared with a `__restrict` or a `__restrict__` type modifier, which enables the compiler to better optimize the use of these pointers. See Section 4.3.1 “Controlling Alias Analysis” on page 7 for details.

3.8 Pointer Arithmetic

ANSI C does not allow pointer arithmetic with pointers to `void` and pointers to functions. XCC supports the GNU extension that allows addition and subtraction operations on such pointers. The `sizeof` operator may be applied to a function or `void` type with the return value one.

3.9 Variable-Length Arrays

You can declare automatic arrays with a variable number of elements. For example:

```
char *
add_underscores (char * name)
{
     char underscored[strlen (name) + 5];
     sprintf (underscored, "__%s__", name);
     return strdup (underscored);
}
```

You can also achieve a similar effect with similar performance using `alloca`. However, one advantage of using a variable-length array is that its size is computed when the storage is allocated and can be retrieved with the `sizeof` operator. Also, the convenience that variable-length arrays provide is even greater for arrays with multiple dimensions. For example:

```
void
make_copies (int n, char * name)
{
     int i;
     char copy[n][strlen (name) + 1];

     for (i = 0; i < n; i++)
          strcpy (copy[i], name);
     ...
}
```

3.10 Zero-Length Arrays

You can declare arrays to be of size zero. Use this feature when defining structures that describe variable length objects for the last member of the struct. For example:

```
typedef struct network_packet {
          header_t header;
          char payload[0];
} net_packet;

void build_packet(header_t header, char * payload, int payload_size)
{
          net_packet * p = malloc(sizeof(net_packet) + payload_size);
          p->header = header;
          memcpy(p->payload, payload, payload_size);
}
```

3.11 Attributes of Functions, Types and Variables

You can annotate functions, types and variables with `__attribute__` directives, which provide additional information to the compiler. Follow the `__attribute__` directive with a double-parenthesized list of attributes, which is placed immediately before the semicolon that ends the declaration or at the beginning of the definition.

For example, the following are valid:

```
int array[100] __attribute__ ((aligned (16), common));__attribute__
((aligned (16), common)) int array[100];
```

```
struct packed_struct { char c; int i; } __attribute__ ((packed));

void func () __attribute__ ((section (".my_text")));

__attribute__ ((section (".my_text"))) int func (int a)
{
     return a + 1;
};
```

Place attributes for functions before any references to the function. Otherwise, the attribute can be ignored.

XCC supports the attributes in declarations as shown in Table 3–14. Note that XCC, like GCC, issues a warning rather than an error when encountering an unsupported (or misspelled) attribute, whether or not that attribute is supported by GCC. Please review these warnings appropriately.

Table 3–14. Declaration Attributes

Attribute	Description
`aligned (`*`alignment`*`)`	Requires *`alignment`* as the minimum alignment (in bytes) for the function, variable or structure field being declared (*`alignment`* must be a power of 2). If specified in a type declaration, it applies to all variables of that type.
`always_inline`	Use on a function declaration. If possible, always inline calls to the attributed function regardless of heuristics.
`cleanup (`*`cleanup_function`*`)`	*`cleanup_function`* is invoked when the attributed automatic local variable goes out of scope. The cleanup function must take a single argument with a pointer type compatible with the attributed variable.
`common`	Allocates the variable being declared to the common block. This applies only to uninitialized global variables, which are by default placed in the `.bss` section. You can apply common block semantics to all uninitialized global variables using the command-line option `-fcommon`. This attribute is ignored in C++.
`const`	The attributed function does not have any effect other than setting its return value. The return value depends only on the parameter and not on any global variables or indirection of the parameters. A `const` function may not access volatile variables. `Const` is a stricter version of `pure`. Using this attribute allows the compiler to more aggressively optimize calls to the attributed function.
`constructor`	The attributed function is called before `main` as if it were a static constructor.
`deprecated`	Warn if the attributed function is called.
`destructor`	The attributed function is called after exiting `main` as it were a static destructor.
`init_priority (`*`priority`*`)`	Controls the order in which global objects in C++ are initialized. Lower priority items are initialized first and destructed last. *`priority`* can be a value from 101 to 65535 inclusive.

Table 3–14. Declaration Attributes (continued)

Attribute	Description
`malloc`	The attributed function can be assumed by the compiler to return a pointer to unique memory that is not aliased with other pointers valid at the time the function is called. Use of this attribute allows the compiler to better optimize particularly when using custom memory allocators.
`mode (`*`this_mode`*`)`	Specify the data type of the attributed variable according to the machine mode *`this_mode`*.
`nocommon`	Allocates the variable being declared to the `.bss` section, which provides initialization to zero. This applies only to uninitialized global variables. The `-fno-common` option, which is the default in XCC, has the effect of applying this attribute to all uninitialized global variables. This attribute is ignored in C++.
`noinline`	Use on a function declaration. Do not inline any calls to the attributed function.
`noreturn`	The compiler is free to assume that calls to the attributed function never return.

`optimize ("`*`opt_level`*`")`	Set the optimization level for the attributed function to opt_level, where opt_level is one of -O0, -O1, -O2 or -O3, possibly paired with -Os. If XCC is invoked with one of the command line options -O1, -O2 or -O3, this attribute instructs the compiler to instead compile this function at the optimization level specified in the attribute. Compiling without any command line optimization options, or with -O0 flag, will override any attribute specifications, allowing easier debugging.
	For example:
	`__attribute__((optimize ("-Os")))`
	`void func0()`
	`{`
	`  int i;`
	`  for(i = 0; i < 100; i++) {`
	`  a[i] = b[i] + c[i];`
	`  }`
	`}`

`overlay(N)`	Place the attributed function in overlay number "N". For convenience, a macro is defined so that the user can just write OVERLAY(N) after the function declaration when including xtensa/overlay.h. The Automatic Xtensa Overlay Manager is described in detail in the *Xtensa System Software Reference Manual* `void foo0(int n) OVERLAY(0);`

Table 3–14. Declaration Attributes (continued)

Attribute	Description

packed	Does not pad structure fields to satisfy the default alignment requirements for their types (however, alignment requested with attribute `aligned` is always enforced). This attribute may be used for an individual structure field to indicate that the field need not be aligned, or for an entire structure type, which has the effect of applying the attribute to each of its fields. For example:
	
	`struct unpacked {`
	`  char c1; int i1; char c2; int i2;`
	`};`
	`struct packed_one {`
	`  char c1;`
	`  int  i1 __attribute__ ((packed));`
	`  char c2;`
	`  int  i2;`
	`};`
	`struct packed_all {`
	`  char c1; int i1; char c2; int i2;`
	`} __attribute__ ((packed));`
	
	In the first case, the compiler inserts three bytes of padding between `c1` and `i1`, and between `c2` and `i2`, to enforce four-byte alignment for `int` fields, and `sizeof(struct unpacked)` is `16`. In the second case, there is no padding between `c1` and `i1`, and `sizeof(struct packed_one)` is `12`. In the last case, no padding is inserted, and `sizeof(struct packed_all)` is `10`.
	Using this attribute may reduce the amount of memory occupied by variables of packed types, but references to such variables will be slower because the compiler must insert the additional code to handle unaligned memory accesses.

pure	The attributed function does not have any effect other than setting its return value. The return value depends only on the parameters and/or global variables. A `pure` function may not access volatile variables. Using this attribute allows the compiler to more aggressively optimize calls to the attributed function.

Table 3–14. Declaration Attributes (continued)

Attribute	Description
`section ("name")`	Places the function or variable being declared in the named section. When placing a function in a named section, the compiler generates literals to hold global addresses or other large literal constants. When a function is placed in section *name* using the attribute, the compiler will place the literals associated with the function in section *name*`.literal` for PC-relative mode L32R instructions and in *name*`.lit4` for absolute mode L32R instructions. There are two exceptions to the rule. If *name* ends in `.text`, the `.text` suffix is dropped in the literal section name. If *name* begins with `.gnu.linkonce.t`, then the name of the literal section is formed by replacing the `.t` substring with `.literal` or `.lit4`. For example, if *name* is `.gnu.linkonce.t.func`, the literals will be placed in `.gnu.linkonce.literal.func` or `.gnu.linkonce.lit4.func`. Note that variables placed in a `.bss` section will be initialized to *0*, overriding any program initialization. Note that assigning a variable or a code section to a preexisting section used by the compiler, such as `.text` or `.data`, might lead to unpredictable results.
`rodata_section ("name")`	For a function with this attribute, any function-scope `static` read-only variables and compiler generated read-only data, such as jump tables, are placed in the named section. Typically, this attribute is used in conjunction with the `section` attribute. For example, if you are placing your most important code inside `.iram0.text`, you might want to use this attribute to place the generated jump tables in a local data memory.
`unused`	Do not warn if an attributed function appears to be unused.
`used`	Emits code for a function, even if it appears that the function is not used.
`visibility ("visibility_type")`	Set the linkage of the attributed declaration to one of "default", "hidden", "protected" or "internal."
`weak`	Emits the declaration as a weak symbol instead of global. Weak symbols will be overridden by other non-weak symbols of the same name.
`weakref` `weakref ("target")`	Mark a declaration as a weak reference.

3.12 Inline Assembly

XCC supports GCC-style inline assembly with the `asm` keyword[2].

```
asm("assembly" [: output_args [: input_args [: clobber_list]]]);
```

2. When compiling with the `-ansi` flag, the keyword `asm` is not recognized. An alternative form, `___asm___`, is always recognized.

assembly is a quoted string to be passed directly to the assembler. For example, to insert an `isync` instruction use the following:

```
asm("isync");
```

C variables can be used by inline assembly through the optional use of *output_args* and *input_args*. Each argument list is a comma-separated list of quoted constraints followed by parenthesized C variables. Arguments are placed inside `assembly` using *%n* where *n* is the numeric position of the argument. For example, to add the three C variables `in1`, `in2` and `out` using the `add` instruction, you can use the following:

```
asm("add %0, %1, %2" : "=a" (out) : "a" (in1), "a" (in2) );
```

The list *output_args* contains the single argument for the C variable `out`. Because it is the zero'th argument, it is placed in the assembly immediately after the `add` in place of the `%0`. The constraint `a` instructs the compiler to use a general-purpose register for the variable `out`. The constraint modifier `"="` instructs the compiler that the `asm` modifies the variable `out` and therefore, for example, the compiler cannot move later uses of the variable `out` ahead of the `asm`. Note that the compiler does not analyze the `asm` string and would otherwise not know that the `add` instruction modifies its first argument.

Table 3–15 lists supported `asm` operand constraints.

Table 3–15. `asm` Operand Constraints

Constraint	Description

`a`
`r`
`v`
	General-purpose register operand or TIE register operand is allowed.

`b`	Boolean register operand is allowed.
`f`	Floating-point register operand is allowed.
`i` `n`	Integer constant operand is allowed.
`0..9`	An operand that matches the specified operand number is allowed.

Table 3–16 describes supported constraint modifiers.

Table 3–16. `asm` Constraint Modifiers

Modifier	Description
=	This is an output (write only) operand.
+	This operand is both input and output (read and written by the instruction). Operands without = or + modifiers are assumed to be input only.
&	This operand is clobbered early (modified before the instruction is finished using its input operands), and it may not be in the same register as an input operand.

As previously mentioned, the "=" modifier instructs the compiler that the argument is defined by the `asm`. The compiler assumes that the `asm` does not read the argument and hence does not need to load it into a register before the `asm`.

Use the "+" modifier for arguments that are both read and written. Consider the following example.

```
asm("add %0, %0, %1" : "+a" (inout) : "a" (in) );
```

The compiler will increment the variable `inout` by the value `in` using the same register to read `inout` and then write it.

The compiler assumes that the assembly instruction will consume all input arguments before any output arguments are written, and therefore the compiler might use the same register for both an output argument and unrelated input argument. If this is not the case, use "=&" in place of "=". Consider the following example with a two instruction sequence to set a variable `out1` to the value `3` and then increment `out2` by the value of `in`.

```
asm("movi %0, 3 \n\tadd %1, %1, %2" : "=&a" (out1), "+a" (out2) : "a"
(in));
```

Without the use of the "=&" modifier, the compiler would be free to incorrectly use the same register for `out1` and `in`.

If there are no input arguments, "*: input_args*" can be omitted. If there are no output arguments, you must use two consecutive colons in place of the output arguments. For example, to add `in1` to `in2` and place the result in the hardwired register `a11`, use the following:

```
asm("add a11, %0, %1" :: "a" (in1), "a" (in2): "a11");
```

Because the compiler cannot analyze the assembly instruction inside the `asm`, the compiler would not, by default, know that the above `asm` modifies the register `a11`, and therefore the compiler might use the same register for other uses. Adding `a11` to the *`clobber_list`* as shown above tells the compiler that the hardwired registers in the list are clobbered by the `asm`.

If your assembly instruction has side effects that are not expressed through the output operands, XCC does not know about them and might be overly aggressive in optimizing; XCC might delete an `asm` or might move an `asm` relative to other C code. Adding the keyword `volatile` immediately after the `asm` keyword instructs XCC to neither delete the `asm` nor move the `asm` with respects to another volatile `asm` or volatile C variable access. Consider, for example, using a pair of `asm` instructions to do a variable shift.

```
asm volatile("ssl %0" : : "a" (shift));
asm volatile("sll %0, %1": "=a" (out) : "a" (in));
```

Without the volatile attribute, the compiler might, for example, switch the order of the `ssl` and `sll` instructions.

The compiler might still move a volatile `asm` with respect to a non-volatile memory instruction. If the `asm` is doing a load or store from memory, this might be problematic. You can add the quoted keyword `"memory"` to the list of clobbered registers to instruct the compiler that the `asm` is modifying an unknown memory location. This way XCC will never delete the `asm` nor move it with respect to any other memory operation.

You can put multiple assembly instructions together in the same `asm`. This is the only way to guarantee that multiple instructions will be grouped together without any intervening instructions. Using a single `asm`, the previous shift example can be rewritten as follows.

```
asm volatile("ssl %1 \n \sll %0, %2": "=a" (out) : "a" (shift), "a"
(in));
```

XCC also supports the GCC-style syntax for allocating a global variable into a specified hardware register. Everywhere the variable is used, XCC will use the assigned register in place of the variable.

```
register int x asm("register name");
```

On a final note, because XCC includes intrinsic functions that represent TIE and many configuration-specific instructions, most users will not need to write inline assembly code.

XCC currently does not support user TIE register files and TIE states in the ASMs. Using TIE instructions which use or modify TIE register files or TIE states in ASMs may cause xt-xcc to change the sequence of how the TIE register files or TIE states are read/written.

3.13 Xtensa Boolean Types

If your Xtensa processor configuration includes the Boolean register file option, you can access a register file that holds special Boolean type values. These Boolean types are useful in conjunction with custom TIE, DSP, and HiFi coprocessors and floating-point instructions.

To declare a variable of type `xtbool`, use the following syntax:

```
#include <xtensa/tie/xt_booleans.h>
xtbool  variable_name;
```

`xtbool` represents a single bit. Use `xtbool2`, `xtbool4`, `xtbool8`, `xtbool16` respectively to represent 2, 4, 8 or 16 wide Boolean bit vectors.

User TIE or coprocessors can define instructions that set or read these registers while the Boolean register file option defines instructions to query or branch on these registers.

For example, the Vectra LX DSP engine coprocessor includes instructions that compare two vectors of size 4 or 8 and produce `xtbool4` or `xtbool8` results. You can reduce these results to `xtbool` values using one of the `XT_ANY4`, `XT_ALL4`, `XT_ANY8`, and `XT_ALL8` intrinsics. You can then test the `xtbool` values in control flow statements (if, for, while).

Also, you can use the `xtbool` type in logical expressions (and, or, not). Alternatively, use the `xtbool4` or `xtbool8` comparison results in instructions that perform conditional data movement. Conversion between `int` and all `xtbool` types is supported. Each type occupies 1 byte in memory except for `xtbool16`, which occupies 2 bytes. Passing pointers to `xtbool` variables as function arguments is supported, but XCC does not currently support functions with `xtbool` parameters.

Note: Using the `xtbool` type for normal Boolean values is not efficient if TIE intrinsics or Vectra LX or floating-point instructions do not produce them.

Following are two examples using `xtbool` variables. The first example has a function that accepts an `xtbool` array and checks for occurrence of consecutive TRUE values. The second example, which uses the Vectra LX DSP engine, checks for the first element that differs between two input Vectra LX arrays.

Example 1:

```
int foo(xtbool ba[])
{
     int i = 1;
     xtbool last_b = ba[0];

     while (ba[i] || last_b) {
          last_b = ba[i++];
     }
     return i;
}
```

Example 2:

```
short v_diff(vec8x16 v1[], vec8x16 v2[], short len)
{
     short i = 0;
     while (XT_ALL8(EQ20(v1[i], v2[i])) && i < len) i++;
     return i;
}
```

3.14 Operator Overloading in C or C++ Programs

TIE intrinsics allow TIE instructions to be used in user C or C++ programs. To make C or C++ programs easier to understand, operators such as '+' can be used in place of the TIE intrinsic function calls. This is accomplished with the TIE operator construct as described in the Operation Sections chapter in the *Tensilica Instruction Extension (TIE) Language Reference Manual*, which provides a way to replace a C operator with a TIE intrinsic when used with TIE ctype operands.

An extension to the TIE operator feature is to define a C function that implements an overloaded operator in C or C++ programs. Unlike the case of the TIE operator, which is implemented with straight line of instructions, the C operator functions are regular functions that can contain control flow constructs such as loops, if-then-else statements, or subroutine calls.

Consider the following TIE:

```
regfile VR 32 16 v
ctype Vtype 32 32 VR
operation VADD { out VR c, in AR a, in VR b } {} {
     assign c = a + b;
}
proto add_short_av { out Vtype c, in int16 a, in Vtype b } {} {
     VADD c, a, b;
}
```

The following simple example defines a C function which implements the C operator '+' and can be used with operands of one signed short and one Vtype ctypes.

```
inline Vtype __xt_operator_PLUS(signed short a, Vtype b)
{
     return add_short_av(a, b);
}
```

To use the overloaded '+' operator, the program needs to include the function declaration above, which can be conveniently achieved with a single header file for all overload functions definitions. And in the program the user can write, for example,

```
Vtype v = 3;
signed short s = 2;
v = s + v; // overloaded '+' operator
```

The effect is the same as calling the intrinsic 'add_short_av' directly on operands 's' and 'v'.

The inline keyword in the overload function decollation is optional. The overload function is essentially a normal C function, except that the name has a special pattern that is recognized by `xt-xcc` to replace an overloaded operator.

Below is another slightly more complicated operator overload function example where an if-statement is used to warn about bad shift amounts. Otherwise, the Vtype value of 'v' is casted to an integer and a right shift is performed.

```
Vtype __xt_operator_RSHIFT(Vtype v, int b) {
     if (b>32) {
          printf("warning: right-shift amount %d > 32\n", b);
          return 0;
     } else
          return (int)v>>b; // this is using integer shift
}
```

Currently, the following operators can be replaced with overload functions:

Table 3–17. Supported Operators

Operator	C function name	Input Arguments
+	__xt_operator_PLUS	2
-	__xt_operator_MINUS	2
*	__xt_operator_MULT	2
/	__xt_operator_TRUNC_DIV	2
%	__xt_operator_TRUNC_MOD	2

Table 3–17. Supported Operators

Operator	C function name	Input Arguments
<<	__xt_operator_LSHIFT	2
>>	__xt_operator_RSHIFT	2
|	__xt_operator_BIT_IOR	2
^	__xt_operator_BIT_XOR	2
&	__xt_operator_BIT_AND	2
<	__xt_operator_LT	2
<=	__xt_operator_LE	2
>	__xt_operator_GT	2
>=	__xt_operator_GE	2
==	__xt_operator_EQ	2
!=	__xt_operator_NE	2
-	__xt_operator_NEGATE	1
~	__xt_operator_BIT_NOT	1
!	__xt_operator_TRUTH_NOT	1

By specifying the operators above, the following operators are derived and can be used in the C application:

+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

Note these operators require that the first input argument and the return value of the function are of the same type.

At least one of the operands has to be a TIE ctype. Multiple overloading functions with different operand types can be used to overload the same operator. The order of operand types for binary operators is used to determine the overloading functions used. In the case that the operand types differ, two versions may be needed to allow different ordering of operand types. When the same operator is overloaded with both a TIE operator construct and a C functions, the C function version will be used.

4. Advanced Optimization Topics

If the performance goals for your program are not achieved by simply using `-O2` or `-O3` (with or without `-IPA`) command-line options, you can further guide the optimizations performed by XCC. In this chapter we describe how you can view the results of compiler transformations and use additional command-line options or source code directives to improve the effectiveness of XCC optimizations.

4.1 Viewing the Effects of Compiler Optimizations

If you use the `-S` or the `-keep` command-line option, the compiler will save the generated assembly code in files with the `.s` extension. With the optimization level higher than `-O0`, the compiler annotates the assembly code with additional data about loops in your program. This information can be useful to determine which areas of your code need additional optimization. For example, consider the following simple function:

```
int sum(int a[])
{
     int i, s = 0;
     for (i = 0; i < 100; i++)
          s += a[i];
     return s;
}
```

If XCC is invoked with the `-O2` optimization level, it produces the following assembly code for the loop body and informational notes about it.

```
#<loop> Loop body line 2, nesting depth: 1, iterations: 12
#<loop> unrolled 8 times
#<swps>
#<swps>  17 cycles per pipeline stage in steady state with unroll=8
#<swps>   1 pipeline stages
#<swps>  17 real ops (excluding nop)
#<swps>
#<swps>      min 17 cycles required by resources
#<swps>      min 8 cycles required by recurrences
#<swps>      min 17 cycles required by resources/recurrence
#<swps>      min 10 cycles required for non-pipelined critical path
#<swps>         17 cycles non-pipelined schedule length
#<swps>
     l32i.n  a8,a9,0                 # [0*II+0]  id:15
     l32i.n  a11,a9,4                # [0*II+1]  id:15
     add.n   a8,a8,a10                       # [0*II+2]
     l32i.n  a10,a9,8                # [0*II+3]  id:15
```

```
        add.n   a8,a11,a8                       # [0*II+4]
        l32i.n  a11,a9,12               # [0*II+5]  id:15
        add.n   a8,a10,a8                       # [0*II+6]
        l32i.n  a10,a9,16               # [0*II+7]  id:15
        add.n   a8,a11,a8                       # [0*II+8]
        l32i.n  a11,a9,20               # [0*II+9]  id:15
        add.n   a8,a10,a8                       # [0*II+10]
       l32i.n  a10,a9,24               # [0*II+11]  id:15
        add.n   a8,a11,a8                       # [0*II+12]
        add.n   a8,a10,a8                       # [0*II+13]
        l32i.n  a11,a9,28               # [0*II+14]  id:15
        addi    a9,a9,32                        # [0*II+15]
        add.n   a10,a11,a8                      # [0*II+16]
```

The compiler unrolled the loop eight times. Eight iterations of the loop execute in 17 cycles.

Each number in brackets to the right of an instruction represents the cycle (within a loop iteration) in which that instruction executes. This allows you to find delays due to pipeline stalls. The software pipeliner interleaves operations from multiple iterations of the loop. The number multiplying the `II` symbol represents the iteration being executed. In this example, all operations come from the same iteration, so all multipliers are 0.

If the compiler was instructed to optimize for space with the `-Os` option, the loop will not be software pipelined or unrolled because this would increase the code size. The generated assembly code would look like this:

```
#<loop> Loop body line 2, nesting depth: 1, iterations: 100
#<sched>
#<sched> Loop schedule length: 3 cycles (ignoring nested loops)
#<sched>
#<sched>     1 mem refs     ( 33% of peak)
#<sched>     2 integer op   ( 66% of peak)
#<sched>     3 instructions (100% of peak)
#<sched>
     l32i.n  a10,a9,0    # [0]
     addi.n  a9,a9,4     # [1]
     add.n   a2,a10,a2   # [2]
```

The generated loop will require 3 cycles per iteration instead of the 2.25 cycles per the original iteration when compiling with `-O2`. However, the code is significantly smaller.

You can also use the `-clist` command-line option to generate `.w2c.c` files that contain a high level view of code transformations performed by the compiler. This option is primarily useful in conjunction with the XCC automatic vectorization feature and is described in more detail in Section 4.10.1. `-clist` can also be helpful in observing the effects of function inlining, especially with the `-ipa` option.

4.2 Optimizing Functions Individually

Compiler flags allow the user to control optimization but only at the file level. Each flag applies to all of the functions in a file. It is often useful to compile different functions in the same file using different compiler flags. This can be done using an external editable text file known as an optimization file, using the `-fopt-gen` and `-fopt-use` flags.

The `-fopt-use` flag may be used to specify function optimization levels by placing a list of one line entries in a text file and supplying that file name through the `-fopt-use` flag.The flag usage is as follows:

`-fopt-use=<optimization_file>,`

where the `<optimization_file>` is a text file with a list of lines, each with the general format:

`[<direct_path>]<filename>:<procedure_name>:<optimization_string>.`

The optional `<direct_path>` may be needed to disambiguate file name collisions. Lines beginning with # are ignored as comments. The maximum line length allowed is 1024 characters. The format for the `<optimization_string>` for function-level optimization is as follows: `[-O{0,1,2,3}] [-Os]` to indicate compiling a certain function at levels 0 through 3, or to optimize for space. The optimization levels specified in the file will override the command line optimization options `-O1`, `-O2` or `-O3`. Compiling without any command line optimization options, or equivalently with the `-O0` flag, overrides any file specifications, allowing easier debugging.

For example, create a text file name `foo.opt`, with the following contents:

```
test.c:test_foo: -O1
test.c:test_bar: -O3 -Os
main.c:main_foo: -O0
```

Then, compiling with the command line: `xt-xcc -O2 test.c main.c -fopt-use=foo.opt` would result in the following: `test_foo()` in `test.c` compiled at `-O1`, `test_bar()` in `test.c` compiled at `-O3` and optimized for space, `main_foo()` in main.c compiled at `-O0` and any other function in `test.c` and `main.c` compiled at `-O2`, as specified on the command line.

The compiler option `-fopt-gen=<optimization_file>` creates an output file for later use by `-fopt_use`. Once generated, the files may be edited to control optimization per-function. Be sure to remove the `-fopt-gen` option or subsequent compiles will overwrite any changes you make.

The `-fopt-gen` option with no file name creates a file named `<source_basename>.opt`, during a compile step and `<executable_basename>.opt` during a link step. For most functions, the optimization will be set to whatever was given in the command line. However, if feedback compilation is used, the compiler will automatically compile certain functions for space. By generating and then keeping the generated file, the user can take advantage of the decisions made by the feedback optimization without having to keep compiling the application using feedback. The use of the `-fopt-use` flag is meant to be robust to changes in the applications. If a function is later deleted, the compiler will ignore its entry in the file. If a function is added and no entry is added to the file, the function will be compiled with whatever flags are given on the command line.

C++ functions must be specified using their mangled names. By starting with the `-fopt-gen` flag, the file will contain the correct mangled name of every function. The standard utility `xt-c++-filt` can be used to demangle the names in the file to get their unmangled C++ name.

Note that similar functionality can be achieved using the optimized attribute described in Table 3–14 Note also that if a function is inlined into a caller, the function will be optimized according to the optimization level of the caller.

4.3 Controlling Miscellaneous Optimizations with the `-OPT` Option Group

You can direct various analysis and code transformations that XCC performs during the optimization process by using command-line options from the `-OPT` option group. These options are of the form `-OPT:option_name=value`. Option names and the corresponding values are described in Table 4–1.

Of particular importance are the options for guiding alias analysis in the compiler. Two memory references in the program are said to be aliased with each other if they refer to the same memory location. Memory references can be either direct (named scalar variables) or indirect (through pointers). In cases where the compiler is unable to fully analyze two memory references and prove that they cannot be aliased, it makes assumptions in accordance with the source language rules. Using the command-line options, you can make these assumptions more or less conservative, thus making the compiler optimizations less or more aggressive.

Table 4–1. −OPT Option Group

Option	Description
`-OPT:alias=any`	The compiler assumes that any two memory references may be aliased. This is the most conservative aliasing model, and it should be used only if your program violates the ANSI type-based aliasing rules. GCC-compatible option `-fno-strict-aliasing` has the same effect.
`-OPT:alias=typed`	The compiler assumes ANSI type-based aliasing rules (this is the default). According to these rules, two memory references of different types will not alias, with the following exceptions: types may differ in signedness (for example, an `int *` pointer may point to an `unsigned int` variable), types may have different qualifiers (such as `const` or `volatile`), one type may be an aggregate type that contains a member of the other type (for example, an `int *` pointer and a pointer to a struct that has an `int` member may point to overlapping memory locations), and a pointer of type `char *` may point to any other type. The GCC-compatible option with the same meaning is `-fstrict-aliasing`.
`-OPT:alias=restrict`	The compiler assumes that memory references with one level of indirection through different *named* pointers do not alias with each other, nor with any direct memory references.
`-OPT:alias=disjoint`	The compiler assumes that memory references indirecting through different *named* pointers (possibly with multi-level indirection) do not alias with each other, nor with any direct memory references.
`-OPT:Olimit=`*`size`*	Does not optimize functions that exceed the specified *`size`*. When processing a very large function, XCC may decide to skip the optimization phase in order to avoid a significant increase in the compilation time. If this happens, the compiler will also issue a warning about the `Olimit` value that you can use to force the optimization. Specifying `Olimit=0` tells to compiler to optimize all functions, regardless of their sizes.
`-OPT:roundoff=0`	Does not perform transformations that may change the round-off semantics of floating-point operations. This is the default with the optimization level `-O2` or lower. At `-O3` the compiler may apply some mathematically valid transformations (such as reassociating or reordering of expressions) that affect the precision of floating-point operations.

Table 4–1. `-OPT` Option Group (continued)

Option	Description
`-OPT:space_flix=0`	If set to 1, do not bundle operations into FLIX instructions in situations where bundling will increase code size. Note that operations that only exist in FLIX formats will still be bundled.
`-OPT:space_opt=n`	For use together with feedback optimizations. Optimizes for space, rather than time whenever a function takes less than $\frac{n}{10}$ % of the total execution time. By default, *n* is set to `10` so that all routines that take up less than `1%` of the execution time are optimized for space. For more details, see Section 4.4.
`-OPT:unroll=`*times*	Does not unroll any inner loop more than the specified number of *times* (this does not apply to loops with small constant trip counts that the compiler chooses to fully unroll). `-OPT:unroll=1` disables loop unrolling. Note that at `-O3`, the compiler might unroll outer loops or interchange outer and inner loops irrespective of this option.

4.3.1 Controlling Alias Analysis

The XCC compiler performs various optimizations aimed at reducing the impact of memory references on the overall program performance. For example, register allocation decreases the number of memory operations by keeping values in registers instead of accessing them from memory, and instruction scheduling may hide memory latency by overlapping memory and arithmetic operations. When determining the safety of these optimizations, the compiler relies on the results of alias analysis. In this section, we illustrate how different memory aliasing models affect the behavior of the optimizer.

For the first example, consider the following function:

```
short typed(int i, int *p, short *s)
{
     *p = *s;
     return *s;
}
```

If this function is compiled with `-O2 -OPT:alias=any`, the generated assembly code (without the entry and return instructions) will look like this:

```
        l16si     a2,a4,0
        s32i.n    a2,a3,0
        l16si     a2,a4,0
```

Note that the value corresponding to `*s` is loaded twice: once before and once after the store to `*p`. Because `-OPT:alias=any` implies the most conservative aliasing model, the compiler must assume that pointers `p` and `s` may point to overlapping memory locations, and therefore the store to `*p` may change the value of `*s`.

If the same function is compiled with just `-O2`, which implies `-OPT:alias=typed`, the following code is generated:

```
        l16si     a2,a4,0
        s32i.n    a2,a3,0
```

Because pointers `p` and `s` point to different types (`int` and `short`), the compiler can conclude that memory references `*p` and `*s` do not alias with each other, which implies that it is safe to load the value `*s` only once.

In a slightly modified example, pointers `p` and `s` point to the same `int` type:

```
int restrict(int i, int *p, int *s)
{
     *p = *s;
     return *s;
}
```

With `-O2`, the type-based aliasing rules are not sufficient to establish that memory references `*p` and `*s` do not overlap, and the value `*s` must be loaded from memory twice:

```
        l32i.n     a2,a4,0
        s32i.n     a2,a3,0
        l32i.n     a2,a4,0
```

Performance can be improved by using the `__restrict` or `__restrict__` type qualifier.

```
int restrict(int i, int * __restrict p, int * __restrict s)
{
     *p = *s;
     return *s;
}
```

The `restrict` type qualifier is based on the ISO C 1999 standard, although XCC supports it for C++ programs as well as C programs. The compiler is allowed to assume that modified memory accessed through a `restrict` pointer is only accessed through that pointer and not through another pointer nor directly as a global variable. Therefore, in the example above, `*p` and `*s` are not allowed to overlap. The compiler can generate more efficient code by removing the second load of `*s`:

```
        l32i.n     a2,a4,0
        s32i.n     a2,a3,0
```

Note that it would be sufficient to declare either pointer as `restrict`. The use of `restrict` pointers can potentially make a large difference to performance. In particular, but not exclusively, it is often very useful for vectorization. However, you must be very careful to only use the qualifiers in cases where the pointer has no aliases. Otherwise, the use of the `restrict` pointer might change the behavior of your program.

The `restrict` modifier can be placed on function formals, global pointers and pointers declared at the function scope, as shown in the following example:

```
int func (int *a, int *b)
{
  int * __restrict p = a;
  int * __restrict q = b;
  *p = *q;
  return *q;
}
```

In the above example, `*p` and `*q` are not allowed to overlap and the compiler generates more efficient code by removing the second load of `*q`.

Note that XCC does not efficiently support the use of `restrict` on pointers declared inside a nested scope. For such pointers, XCC will conservatively ignore the `restrict` qualifier.

To ensure correct code generation, if the `restrict` modifier is used for a function's formal parameters or within the function's body, XCC will not inline the function.

Alternatively, you can compile your program using `-O2 -OPT:alias=restrict`. This option indicates that pointers with different names do not point to overlapping memory locations. The use of the type qualifier is in general safer than the use of the command-line option since the type qualifier only applies to a single variable, while the option applies to every variable in the compilation unit. However, the use of the type attribute does require you to modify your code.

The `restrict` aliasing model is not the most aggressive available. The shortcoming of the `restrict` aliasing model is demonstrated by the following example:

```
int disjoint(int i, int **p, int **s)
{
     **p = **s;
     return **s;
}
```

Because `restrict` only disambiguates memory references created by a single level of indirection through named pointers, the compiler cannot assume that pointers `*p` and `*s` do not point to the same location, and it generates two loads of `**s`:

```
l32i.n      a2,a4,0
l32i.n      a9,a3,0
l32i.n      a8,a2,0
s32i.n      a8,a9,0
l32i.n      a2,a2,0
```

You can use the option `-OPT:alias=disjoint` to tell the compiler that memory references `**p` and `**s` do not alias. The resulting code will contain only one load of `**s`:

```
l32i.n      a2,a4,0
l32i.n      a8,a3,0
l32i.n      a2,a2,0
s32i.n      a2,a8,0
```

As illustrated by these examples, using the command-line options to instruct the compiler to make stronger aliasing assumptions may allow for more aggressive optimizations, thus leading to improved performance. However, you must ensure that your source code conforms to the constraints of the specified memory aliasing model. In general, ensuring this conformity may be rather difficult for the `restrict` and `disjoint` memory models.

These models should be used only under carefully controlled circumstances, that is, only with code that has been specifically written to satisfy the constraints described above.

4.4 Using Profiling Feedback

Various compiler optimizations may benefit from information about profiling and branch frequencies. At a micro level, for example, when the Xtensa processor executes a conditional branch instruction, it incurs additional overhead when the branch is taken compared to falling through to the next instruction. Therefore, it is desirable to have as many branches be fall-through as possible, which XCC can achieve by reordering the code and changing the branch directions. At a more macro level, for example, the compiler should not unroll a loop that in practice often only executes for one iteration, and the inliner should, in general, only inline functions that are frequently called.

XCC has three mechanisms to estimate the frequency of different regions of code: heuristics, pragmas and automatic feedback from profiling information.

In the absence of other means, XCC uses heuristics for guessing branch directions. For example, XCC assumes that loops execute multiple iterations. Of course, heuristics are just guesses, and XCC can do a much better job with more accurate information. XCC has a mechanism to feed back information taken from an actual run back into the compilation process, using a three-step process.

1. In the first step, invoke the compiler with one of the following options:

`xt-xcc ... -fb_create` *`filename`* `....`

The compiler instruments the code to count the frequency of all branches. By default, the compiler uses 32-bit counters. If your code is sufficiently long running so that any particular region is executed more than $2^{31} - 1$ times, the counters will overflow and generate inaccurate counts. This will not cause your program to execute incorrectly, but it will degrade optimization. For such long-running programs, invoke the compiler including the following option to use 64-bit counters.

`-fb_create_64` *`filename`*

By default, the compiler does not use 64-bit counters because usage of 64-bit counters significantly slows down the second step.

Note that for this first step, you must both compile and link your application with this option. Also note that the compiler instrumentation uses standard library functions to do memory allocation and file I/O. If you redefine any of the library functions used by the instrumentation library, listed below, you cannot compile any file containing those redefined functions that use `-fb_create`.

```
atexit
close
delete
exit
fclose
fflush
fopen
fprintf
fputc
free
fseek
fwrite
malloc
new
open
realloc
```

XCC gives an error message in such situations. If you redefine any library function in a way that redefines its externally visible behavior, even if you do not compile the function with `-fb_create`, you may get unpredictable results in the instrumented version.

2. In the second step, run your application with a representative input set. You may run the application multiple times with different representative sets. Each run creates a new profiling file prefixed by *`filename`*. The program can be run on any system that supports basic file I/O, including the simulator. This run will take significantly longer than a normal run, often several factors longer. The input sets do not have to be the same as the ones used in production (often, shorter running input sets are used).

However, the closer the run is to the production run, the more accurate the information will be. If a particular run only invokes some subset of the common modes expected in production, it is possible and desirable to execute multiple runs.

If your target hardware does not support a file system, you can use the following option.

```
-fb_create_HW filename
```

The compiler instruments the code as in the `-fb_create_64` case but also includes software to allow the data to be retrieved on real hardware automatically through the debugger. Code compiled and linked with this option must be executed on the hardware with the debugger attached, either through Xplorer or through xt-gdb. Once the file is retrieved, it can be used identically as files produced with the other options. See the *Xtensa Software Development Toolkit User's Guide* for more information.

3. In the third step, reinvoke the compiler including the following option:

```
xt-xcc -fb_opt filename
```

The compiler uses the profiles generated in all the files prefixed by *`filename`*. Multiple profiles are averaged together, weighted by their execution time. This second invocation of the compiler must use the same sources and similar options as the first invocation using `-fb_create` *`filename`*. In particular, you must either use `-ipa` in both compilations, or neither. If you change the sources, you must regenerate your profiling files by going back to the first step. Be sure to delete your old profile files. If you do not delete the old profile files, the compiler will give you an error message saying that the profiling files no longer match.

Note that the profiling file is created in the same directory in which the application is run. If you compile and run it in separate directories, you must either move the profiling files, or use a full path to the run directory when performing the third step.

If you add the option

```
-fb_reorder
```

in step 3, the compiler will reorder functions based on the profile data in an attempt to improve instruction cache performance. The compiler can also use feedback data for placing frequently used code into IRAM. In conjunction with `-fb_reorder` you use the link option

```
-Wo,--use-iram
```

to enable IRAM code placement, and the compiler will determine the free space available in IRAM and will fill it with frequently used functions. You can also override the Xtensa core configuration parameters and instruct the compiler to use less than the total available IRAM size by specifying one or more link options

```
-Wo,--iram=SECTION:[LITERAL_SECTION:]SIZE
```

where *SECTION* is the IRAM output section name, *LITERAL_SECTION* is the section name for literals if different from *SECTION*, and *SIZE* is the amount of space to use, in bytes. For example: `-Wo,--iram=.iram0.text:1024` will instruct the compiler to put 1024 byte of code and literals into `.iram0.text` section.

Note that to use `-fb_reorder` option, you must use the XCC compiler driver (`xt-xcc` or `xt-xc++`) to perform the link step. You need to pass `-fb_reorder` at both compile and link steps. `-Wo` options have effect at the link step only.

Using automatic feedback can significantly improve run-time performance, but it makes an even larger impact on code size, because it enables XCC to automatically decide which functions to compile for speed and which functions to compile for size. When using automatic feedback, XCC compiles every routine that takes at least one percent of the execution time for speed and every other routine for size by default. This ratio can be adjusted using the following option:

```
xt-xcc -OPT:space_opt=n
```

Using this option, XCC optimizes a function for space whenever that function takes less than $\frac{n}{10}$ % of the total execution time.

Note that to take advantage of this feature of feedback, your configuration must have at least one timer or you must compile with IPA.

When you cannot use automatic profile information, or when you know that a branch is almost always taken (or not taken) regardless of feedback information, XCC provides two pragmas:

```
#pragma frequency_hint NEVER
#pragma frequency_hint FREQUENT
```

When placed right after the conditional test of an `if` statement (at the beginning of the *then* block), these directives indicate that the conditional branch is almost never, or very frequently taken. In the following example,

```
int a_lt_b(int a, int b)
{
     if (a < b) {
          #pragma frequency_hint NEVER
          return 1;
     }
     return 0;
}
```

the `frequency_hint` directive indicates that the branch is almost never taken, and XCC produces the following assembly code at the `-O2` optimization level:

```
        blt       a2,a3,.LABEL
        movi.n    a2,0
        retw.n
    .LABEL:
        movi.n    a2,1
        retw.n
```

If the frequency hint is changed from `NEVER` to `FREQUENT`, the generated code changes to:

```
        bge       a2,a3,.LABEL
        movi.n    a2,1
        retw.n
    .LABEL:
        movi.n    a2,0
        retw.n
```

The use of pragmas allow more control than automatic profiling feedback. However, automatic profiling feedback provides much finer grained information.

Note: The compiler only reorganizes branches when compiler optimization levels `-O2` and above are used.

4.5 Super Software Pipelining

For many algorithms, particularly in DSP, performance can be dominated by the performance of inner loops. As discussed in the *C Application Programmer's Guide* Section 2.2.5, the compiler will try to software pipeline every inner loop. To software pipeline a loop, the compiler needs to come up with an ordering, or schedule, for all the operations in the loop. Finding the optimal ordering is an NP-complete problem, meaning that it is impossible to always find the best schedule in a tractable amount of time. Instead, the compiler uses heuristics to try many but not all potentially good orderings. Looking at an inner loop, you may feel that the compiler should have scheduled it better. Sometimes you are wrong, sometimes the problem is unrelated to scheduling, but sometimes the problem is simply that the compiler did not find the best schedule.

With super software pipelining, you can ask the compiler to exhaustively, but intelligently, search all possible schedules. To use this feature, add the following `#pragma super_swp` immediately preceding the inner loop. In many cases, the compiler will complete quickly, but in some cases the compiler will never complete. The process can be made somewhat faster by guiding the compiler and telling it how much to unroll the

inner loop and how many cycles (after unrolling) to try to schedule. Following is an example where the software pipeliner is asked to not unroll an inner loop and schedule it in 28 cycles.

```
#pragma super_swp ii=28, unroll=1
for (i=0; i<n; i++} {
     ...
```

Even with the additional hints, there are occasions where the compiler will attempt to compile essentially forever.

If the compiler succeeds in a large, but tractable, amount of time, you may not want to repeat the search every time you recompile your application. Instead, compile the code with the additional `-SWP:Op_Info=1` option. With this option, the compiler will place inside the generated `.s` file a string such as the following:

```
#pragma swp_schedule ii=3, unroll=1, sched[4]= 0 1 2 5
```

Moving this pragma into the source program immediately preceding the loop will allow the software pipeliner to again find the schedule very quickly. If the schedule is no longer valid, perhaps the code has been changed, the software pipeliner will try its normal heuristics.

4.6 Interprocedural Analysis and Optimization

One of the distinguishing features of the XCC compiler is its ability to perform interprocedural analysis (IPA) and optimization. While most traditional compiler optimizations are done within a single function, interprocedural optimizations are applied to the whole program. Various compiler analyses yield more precise information when performed across the entire program, and this in turn enables additional optimizations that may not be possible within the separate compilation model.

XCC implements the following interprocedural optimizations:

- Improved function inlining, with heuristics based on the analysis of the call graph for the whole program.
- Constant propagation for function parameters that are always passed the same constant value.
- Dead function and variable elimination to reduce the program size (especially when used in conjunction with the `-Os` option).
- Identification of global variables that are initialized to constant values, and never modified.
- More precise alias analysis based on the whole program view, which often may eliminate the need to use the restrictive memory aliasing models described in Section 4.3.

XCC supports interprocedural analysis for generating both normal executables and relocatable libraries, which are linked with `-ipalib` on the link line. This feature provides interprocedural analysis on libraries shipped to third parties who may not be using interprocedural analysis in their development. Section 4.6.1 contains more information about linking libraries with `-ipalib`.

To enable interprocedural analysis and optimization, you must use the `-ipa` (or `-IPA`) command-line option both when generating object (`.o`) files and when linking them. When this option is in effect, the `.o` files produced by the compiler will not be the normal object files; instead, they will contain the information about the original code, summarized in a form that is suitable for interprocedural analysis. During the link step, rather than invoking the standard linker (`xt-ld`), the XCC driver invokes the interprocedural module, which digests the summary `.o` files and performs various analyses and transformations. This module then generates intermediate files and writes them into a temporary directory *`executable_name`*`.ipakeep`, whose contents can be saved with `-keep`. In the final step, it invokes the compiler to transform the intermediate files into real object files, and then the standard linker to create the final output.

Because the interprocedural optimizer mimics the traditional steps of compiling and linking, most makefiles for normal executables will continue to work with a simple addition of `-ipa` to the command-line options. However, you should be aware of the following caveats:

- The `-ipa` option should be used for both compiling and linking. Although the interprocedural module will accept normal `.o` object files (produced without `-ipa`) and generate correct code, it cannot derive information useful for its analyses from such files. An attempt to link interprocedural summary `.o` files without the `-ipa` option on the link line will result in an error message produced by the linker.
- You must use the XCC compiler driver (`xt-xcc` or `xt-xc++`) during the link step with IPA. Using the standard linker (`xt-ld`) will result in an error message.
- The linking step with the interprocedural module hides the entire back-end compilation and optimization, in addition to the processing of summary `.o` files. Therefore, re-linking even after changing just a single `.o` file takes much more time than without IPA.
- Compiler optimization options apply to files. The interprocedural module rearranges functions across file boundaries. If different command line options are used for different files compiled with IPA, which options is used for any particular function is not defined. Therefore, it is recommended (although not required) that you use the same command-line options (especially the optimization level) for all the files compiled with IPA.
- Because of the dramatic extent of code changes performed during interprocedural optimization, using `-g` in conjunction with `-ipa` will produce extremely limited debugging information. This information will allow you to do line profiling with

`xt-gprof`, as well as set breakpoints and step through the code in a debugger (such as `xt-gdb`). However, examining source-level data in a debugger will not be possible.

- The `-ipa` option has no effect if it is used in conjunction with a command-line option that terminates the compilation process before producing object files (`-E` or `-S`).

To use interprocedural analysis when generating a final executable, compile the `.o` files with the `-ipa` option, and use the `-ipa` option on the link line.

To use interprocedural analysis on archives or `.a` files, compile the `.o` files with the `-ipa` option, and use the `-ipa` option on the link line.

4.6.1 Building Libraries with IPA

An archive built using the archiver *xt-ar* on object files compiled with IPA is not truly compiled until link time. The library itself, before link time, contains a representation of the preprocessed source files in the library. Therefore, IPA is required to be used at link time. This is useful because with IPA both the application and the functions it calls from the library can be optimized together. For example, IPA can inline a function from the library into the main application, an optimization that is not possible with non-IPA compiled libraries.

However, sometimes it is useful to build a normal, compiled, library using IPA. Such libraries can be distributed and then be linked into an application without requiring the final link to use IPA. When building the library, the compiler might inline functions if both the caller and the callee are within the library, but functions in the library will never be inlined into the application. Applications linked against these libraries can still use IPA to optimize the application interprocedurally, but the library will not be recompiled.

To safely optimize a program, IPA needs to know which functions access the global state. When building an executable, IPA sees all the object files, whether real or IPA objects. Therefore, IPA is able to automatically compute the information it needs in order to safely perform optimizations. When building a normal, compiled, library, IPA has no way of knowing what variables and functions in the library might be referenced by an application that will be linked to that library. For example, IPA can delete unused functions. But a function in a library might be designed to be called from a library's client, and not within the library itself. That function will appear to be unused and normally IPA would delete it. Therefore, when building a compiled library, you must explicitly tell IPA which variables and functions are externally accessible.

To build a normal, compiled, library with IPA, compile the source files with `-ipa` added to the compilation options. Linking the library requires two options: `-ipalib` and `-ipaentry=symbol_name` repeated for each symbol (function or variable) that is ex-

ternally accessible. For optimization purposes, IPA will assume that all the symbols marked with an `-ipaentry=` option may be accessed arbitrarily outside the library and will also assume that no additional global symbols are accessed.

4.7 Inexact imaps

TIE supports the concept of an *imap*, a generalized mechanism for allowing XCC to replace a sequence of operations with a single output operation. For example, using the ConnX D2 DSP Engine, you can write, via operator overloading or intrinsics, a multiply instruction followed by an add, and the compiler will automatically convert the pair of instructions into a single multiply-add instruction.

Sometimes, it may be desirable for the output operation's behavior to not match exactly the behavior of the sequence of input operations. For example, the multiply-add instruction might be more accurate than the sequence of input operations if the multiply and the add instruction each round their results while the multiply-add does a single round at the end. The TIE developer can specify that an imap has such inexact behavior by adding the property `imap_nonExact` to the imap in the TIE file. ConnX D2, for example, uses this mechanism. If this mechanism is used, XCC will try to use the imap by default only at optimization level `-O3`. Regardless of optimization level, this behavior can be explicitly turned on with the compiler option `-menable-non-exact-imaps` and can be turned off explicitly with `-mno-enable-non-exact-imaps`.

4.8 Loop Pragmas

XCC supports a set of pragmas to guide the compiler in optimizing loops. These pragmas impact the vectorizer, described in the next section, but also other portions of the compiler. A loop pragma must be placed immediately preceding the loop to which it applies.

#pragma no_unroll

This pragma disables unrolling of the loop it immediately precedes.

#pragma loop_count min=<level>, max=<level>, factor=<level>, avg=<level>

This pragma asserts the minimum trip count (**min**), the maximum trip count (**max**), and the trip count's even divisibility (**factor**) of the loop it immediately precedes. A given pragma may contain only some of the clauses

The minimum and factor values must be exactly correct as the compiler may use this information for optimizations such as omitting unrolling remainder iterations. Incorrect values for these parameters could result in incorrect code.

You can disable the use of this pragma with the `-fno-pragma-loop-count` command-line option.

Additionally, the average value (**avg**) provides a way to tell the compiler an estimate of the average trip count for this loop. The compiler will try to use this in heuristics that require an estimate of the amount of work done each time the loop is entered.

4.9 Speculation

Performance can sometimes be improved by allowing the compiler to speculate operations, generating code that will be invoked even if the original semantics specify that the code is not invoked. Consider the following example.

```
int main()
{
     int i;
     for (i=0; i<100; i++) {
          if (cond[i] > 0) {
               a[i] = b[i] + c*d;
          }
     }
}
```

The code, as written, will multiply *c*d* 100 times.It would be much more efficient to compute *c*d* once, outside of the loop bounds. If the variables are all integral, there is no harm in *speculatively* doing the multiply since the multiply has no side effects. If however, the variables are floating point, the multiply operation might set an exception flag. If the application is checking the flags, speculation might not be desired. If the application is not, speculating is advantageous.

The compiler flag, `-TENV:X=n` controls whether the compiler is allowed to speculate operations.

n=0: No speculation is allowed.

n=1: Only speculation of operations without side effects is allowed.

n=2: Allow speculation of floating point compute operations. Do not speculate either integer or floating point divides.

n=3: Allow speculation of integer and floating point divides.

n=4: Allow speculation of loads. If a load is to an inaccessible address, use of this flag might cause a load exception.

At optimization level -O2 or below, XCC will default to n=1. At -O3, XCC will default to n=2.

4.10 SIMD Vectorization

For Xtensa processor configurations that support the Vectra LX, HiFi3 or various ConnX processors, XCC can automatically vectorize certain loops, resulting in greatly improved performance.

This section describes how to enable vectorization and how to write your code in a way that makes it easier for XCC to perform this optimization. Table 4–2 summarizes the options that control the vectorizer. The remainder of this section describes the use of the vectorizer in more detail.

Table 4–2. Vectorization Options

Option	Description
`-LNO:outer_unroll=`*times*	Does not unroll any outer loop by a factor greater than *times*. `-LNO:outer_unroll=1` disables outer loop unrolling.
`-LNO:simd`	Enables vectorization. This option is only valid in conjunction with the `-O3` option.
`-LNO:simd_v`	Print s a summary report of compiler vectorization efforts to standard output
`-LNO:simd_agg_if_conv`	Vectorizes a loop containing an *IF* statement even if vectorizing the statement might cause the system to load a value that otherwise would not have been loaded. Equivalent to -TENV:X=4
`-LNO:aligned_pointers=on`	Treats all pointers used as array bases as if they are aligned correctly for use with the vectorizer.
`-LNO:aligned_formal_pointers=on`	Treats all pointers passed as formal parameters to functions as if they are aligned correctly for use in the vectorizer.

Automatic vectorization, and the `-LNO` option group that controls it, are only available in conjunction with the `-O3` option. At this optimization level, the compiler performs loop-nest dependence analysis that is used to evaluate validity and profitability of vectorizing transformations.

To invoke the automatic vectorization feature, you need to pass options `-O3 -LNO:simd` to the compiler, as follows:

```
xt-xcc -O3 -LNO:simd filename
```

There are three possible approaches to using Vector instructions within your C or C++ program:

1. Do not modify your code at all, and simply use the automatic vectorization feature in XCC. This is the easiest method, but it also may not take full advantage of all hardware capabilities.

2. Modify your code to meet one of the constrained memory models described in Section 4.3. Depending on the source code, these modifications may be relatively easy or quite difficult, but can result in dramatically improved performance.
3. Rewrite the code completely using Xtensa processor's TIE intrinsic functions. This is the most time-consuming method (although it is not very difficult), but allows you total control and has the most potential for improving the performance of time-critical portions of your application.

Combinations of these methods can be applied appropriately to the various algorithms.

4.10.1 Viewing the Results of Vectorizing Transformations

When writing and debugging SIMD code, it is often useful to see how the compiler has optimized the code you originally wrote. As described in Section 4.1, you can look at the `.s` assembly file that the compiler generates if the `-S` or `-keep` command-line option is in effect. But a higher, C-level view may be much easier to understand.

To see how the vectorizer has transformed your code, use the `-clist` option on the command line. XCC produces a C language translation of the optimized version of your code in files <*`filename`*>`.w2c.c` and <*`filename`*>`.w2c.h`. (If you use `-clist` in conjunction with `-ipa`, the translated files are named `1.w2c.c`, `2.w2c.c`, and so on, corresponding to the intermediate files created by the interprocedural module in the <*`executable_name`*>`.ipakeep` directory.) These files contain readable code that you can examine to find out which parts of your program have been vectorized. Although the translated code is usually, but not always, compilable, this is not the intended use of this option—there is no guarantee that the generated C code is semantically equivalent to the original source code.

Consider the following example.

```
int a[100], b[100], c[100];

int main()
{
     int i;
     for (i=0; i<100; i++) {
          a[i] = b[i] + c[i];
     }
}
```

Compile the example using `xt-xcc -O3 -LNO:simd -clist main.c`.

As a by-product of compiling `main.c`, the compiler generates two files: `main.w2c` and `main.w2c.h`. A slightly edited version of `main.w2c.c` follows.

```
#include "main.w2c.h"

_INT32 main()
{
     vec4x40 V_00;
     vec4x40 V_;
     vec4x40 V_0;
     vec4x40 V_4;
     _INT32 reg2;
     _INT32 i;

     for (i=0; i<=99; i+=4) {
          V_00 = *(vec4x40 *)(&b[i]);
          V_ = *(vec4x40 *)(&c[i]);
          V_0 = ADD40(V_00, V_);
          V_4 = V_0;
          * (vec4x32 *)(&a[i]) = (vec4x32)(V_4);
     }
     return reg2;
} /* main */
```

Translating the compiler's internal representation into C creates the `.w2c.c` files. As the code in these files is readable, you can see what portions of the original program were vectorized. You can use the information to manually vectorize the source or to aid its automatic vectorization. The `.w2c.c` files represent the stage of the compilation process immediately after vectorization. Further optimizations, including loop unrolling, happen at later states in the compilation process and are therefore not seen in the `-w2c.c` files.

Note: `-clist` is not currently available for C++ programs.

4.10.2 Aligning Data for Vectorization

The Xtensa architecture requires all core memory references to be aligned. MostDSP coprocessors provided by Cadence, through the use of alignment registers, provide some support for unaligned accesses. However, using the alignment registers is less efficient than handling aligned references. Often, references are aligned, but the compiler cannot prove they are aligned. You can help improve the result through a combination of command-line options and `#pragma` directives in the source code.

Some of the compiler issues with alignment are highlighted using a vector sum example:

```
int a[N], b[N], c[N];

void sum(int n)
{
     int i;
```

```
        for (i=0; i<n; i++) {
            a[i] = b[i] + c[i];
        }
}
```

In this example, `a`, `b` and `c` are global arrays and the compiler easily determines that the loop is vectorizable. The compiler aligns all arrays (global and local) to a boundary compatible with the vectorized data types. The vectorizer takes into account the alignment of the array and the lower bound of the loop (in this case 0) and generates regular, aligned vector loads. Using ConnX Vectra LX as an example, the inner loop of the generated code looks like this in the assembly file:

```
lvs32.iu v0, a8, 16; nop; nop;
lvs32.iu v1, a9, 16; nop; nop;
svs32.iu v2, a10, 16; nop; add40 v2, v0, v1;
```

Now consider a version of the vector sum function that takes the arrays as formal parameters:

```
void sum(int *a, int *b, int *c, int n)
{
        int i;
        for (i=0; i<n; i++) {
            a[i] = b[i] + c[i];
        }
}
```

Assuming that you relax the memory model by using a proper aliasing option (see Section 4.3.1), the vectorizer will be able to transform the loop. However, as it does not have enough information about the alignment of the arrays, it will assume that they are not aligned. An excerpt from the `.w2c.c` file generated by using the `-clist` command on this example follows. While the inner loop is as efficient as the provably aligned example, there is a significant outer loop overhead in using alignment registers.

```
ali_adr_2 = (_INT32)(&(b)[0]) + -16;
LVS32A_P(A_, ali_adr_2);
ali_adr_5 = (_INT32)(&(c)[0]) + -16;
LVS32A_P(A_0, ali_adr_5);
ali_adr_8 = (_INT32)(&(a)[0]) + -16;
A_2 = ZALIGN();
for(i = 0; i <= (n + -4); i = i + 4)
{
    LVS32A_IU(V_00, A_, ali_adr_2, 16);
    LVS32A_IU(V_, A_0, ali_adr_5, 16);
    V_0 = ADD40(V_00, V_);
    V_1 = V_0;
    SVS32A_IU(V_1, A_2, ali_adr_8, 16);
}
SVS32A_F(A_2, ali_adr_8); ;
```

For other examples, there may not be a sufficient number of alignment registers, resulting in less efficient code in the inner loop as well as the outer loop.

There are several ways to inform the compiler that arrays are correctly aligned and allow it to improve a loop's vectorization. The most local and restrictive way of doing this is through a pragma directive in the source code. If you are sure that a certain pointer refers to a location aligned at a vector boundary, you can specify this in the scope where the pointer is declared using:

```
#pragma aligned (<pointer_id>, <alignment_byte_boundary>)
```

The `alignment_byte_boundary` must be a power of 2, and you cannot use this directive for global pointers.

For example, assuming that `a`, `b` and `c` are aligned at 16-byte boundaries, our example can be modified as follows to allow better vectorization:

```
void sum(int *a, int *b, int *c, int n)
{
#pragma aligned (a, 16)
#pragma aligned (b, 16)
#pragma aligned (c, 16)
     int i;
     for (i=0; i<n; i++) {
          a[i] = b[i] + c[i];
     }
}
```

Note that `#pragma aligned` can only be used in a function block and the pointer should be defined in this block. We recommend the demonstrated usage.

Alignment assumptions can also be changed through these two command-line options:

`-LNO:aligned_pointers=on` (the default is `off`)

`-LNO:aligned_formal_pointers=on` (the default is `off`)

The first option instructs the compiler to treat *all pointers* used as array bases as if they are aligned correctly for use with the vectorizer. The exact alignment required depends on the target processor's configuration. If you use this option, it is your responsibility to ensure that the alignment constraints are met or otherwise the compiled program may behave incorrectly.

The second option tells the compiler to treat *all pointers passed as formal parameters to functions* as if they are aligned correctly for use in the vectorizer.

Note that the compiler automatically aligns all global arrays correctly. In addition, for users of the default Xtensa runtime, XTOS, local arrays and arrays returned from malloc are also aligned correctly. Therefore, it is usually safe to use the pointer alignment options as long as the program does not contain any pointer arithmetic. Users using other runtime libraries or operating systems must ensure that their stack is aligned to 16 bytes and that their version of malloc aligns all arrays to 16 bytes.

4.10.3 Controlling Vectorization through Pragmas

#pragma concurrent

When XCC is unable to resolve the data dependences in an otherwise vectorizable loop, `#pragma concurrent` allows the designer to mark the loop to indicate that each iteration of the loop is independent of all other iterations. This pragma will often make a loop vectorizable. `#pragma concurrent` is placed just before the loop's `for` statement, as shown below:

```
void copy (int *a, int *b, int n)
{
  int i;

#pragma concurrent
  for (i = 0; i < n; i++)
    a[i] = b[i];
}
```

You must be careful to only use #`pragma concurrent` in cases where the loop iterations are independent. Otherwise, the use of this pragma might change the behavior of your program.

#pragma simd_if_convert

XCC is able to vectorize loops containing conditionals using a technique called `if conversion`. Using this technique, all the operations inside a conditional are executed unconditionally but the results are committed conditionally using conditional move instructions. Consider the following example.

```
#pragma simd_if_convert
  for (i = 0; i < n; i++)
    if (cond[i] == 3)a[i] = b[i]+1;
```

XCC can vectorize the code using `if conversion` as can be seen in the following pseudo assembly code.

```
loop
      lv   v2, a[i]
      lv   v3, b[i]
      addv v4, v3, 1
      lv   v5, cond[i]
      eqv  b0, v5, 3
      movt v2, v4, b0
      sv   v2, a[i]
```

In every iteration of the loop we load all three arrays and then store into `a[]` some combination of elements from the sum and elements from the original value of `a[]` depending on the values of the conditional move instruction.

Note that if the condition is never true, we reference addresses that would not have otherwise been loaded or stored. If the condition is guarding against a `NULL` pointer dereference, for example, the process of `if conversion` might cause a memory exception. Therefore, the compiler will not by default vectorize a conditional unless it can prove that all memory addresses accessed under the conditional are always accessed regardless of the value of the conditional move instruction.

In many cases the programmer knows that it is perfectly safe to speculatively load or store from these memory addresses, but the compiler can not be sure. For such cases, the use of `#pragma simd_if_convert` by the designer instructs the compiler to perform the `if conversion` optimization even if the optimization may generate memory references to otherwise unreferenced addresses. Use of the pragma is similar to the use of the `-TENV:X=4` option, except that it applies only to the immediately following loop.

#pragma simd

Use of `#pragma simd` is equivalent to the use of both `#pragma concurrent` and `#pragma simd_if_convert`.

#pragma no_simd

This pragma disables vectorization of the loop it immediately precedes. Note that this pragma effects the automatic vectorization feature of the compiler. It has no effect on code written manually using vector intrinsics.

4.10.4 Features and Limitations

To use XCC's vectorization feature effectively, it is helpful to know and understand some of its limitations. This SIMD feature works best when the program is written in a vectorizable form. Some programs may need to be rewritten to use a different algorithm or different data organization to take full advantage of the vectorizer. The most common limi-

tations relate to aliasing as described in Section 4.3.1. Other limitations and features follow.

Limitation in the Stride of Memory Accesses

The vectorizer is only capable of vectorizing loops where successive memory accesses are to nearby locations. An array access of the form `[i+c]`, where `i` is the loop variable and `c` is a literal constant, is called a stride-1 access because successive accesses to the array are one element apart. Stride-1 accesses can be mostly easily vectorized since the memory system can load an entire vector of successive elements using a single load. The vectorizer is also capable of vectorizing loops with small positive strides by issuing multiple loads and then using select instructions to extract the appropriate data. Large strided references and negatively strided references cannot currently be vectorized.

Outer Loop Vectorization

The vectorizer is capable of vectorizing outer loops as well as inner ones. When there is more than one choice, the vectorizer will use a cost model to decide which loop to vectorize. Often only one loop contains stride-1 or small strided references, and the vectorizer can only choose that one.

Guard Bits

The XCC vectorizing feature does not obey the standard C/C++ integer overflow semantics when targeting the Vectra LX, ConnX Baseband, ConnX BBE and Imaging coprocessors. Some of their register files contain additional guard bits to allow for increased precision on intermediate arithmetic, for example, 20 bits for `short` data types and 40 bits for `int` data types. Results are saturated to 16 or 32 bits respectively when stored back to memory. The vectorizer automatically uses the extra precision, thereby changing the behavior of programs that would otherwise suffer from overflow.

It is possible that a program relies on C/C++ overflow semantics. For example, a program might rely on the fact that adding one to a `short` variable with the value of 32767 will result in the sum of -32768. The XCC vectorization feature should not be used on such programs.

4.10.5 Vectorization Analysis Report

If you use the `-LNO:simd_verbose` (or `-LNO:simd_v)` command-line option, the compiler prints the summary report for the vectorization analysis to the standard output. This information shows which loops are vectorized, and for those loops that are not, it indicates the issues that prevent vectorization.

Analysis information is reported using the following format:

```
<source_file_name>:<source_line_number> (<simd_message_id>):
<explanation>
```

Consider the following example compiled for the ConnX Vectra LX coprocessor using `xt-xcc -O3 -LNO:simd -LNO:simd_v -c`:

```
void alias(int *a, int *b, unsigned char *c, int n)
{
     int i;
     for (i=0; i<n; i++) {
          a[i] += b[i];
     }
     for (i=0; i<n; i++) {
          c[i]++;
     }
}
```

The following analysis file is generated by the compiler.

```
t.c:2 (SIMD_PROC_BEGIN): Vectorization analysis for function 'alias'.
t.c:5 (SIMD_ARRAY_ALIAS): Array base 'a' is aliased with array base 'b'
at line 5.
t.c:4 (SIMD_LOOP_BEGIN): Vectorization analysis begins with a new
loop.
t.c:4 (SIMD_DATA_DEPENDENCE): Data dependences prevent vectorization.
t.c:4 (SIMD_LOOP_NON_VECTORIZABLE): Loop is not vectorizable.
t.c:7 (SIMD_LOOP_BEGIN): Vectorization analysis begins with a new
loop.
t.c:8 (SIMD_NO_VECTOR_TYPE): Processor configuration does not support
vector unsigned char.
t.c:7 (SIMD_LOOP_NON_VECTORIZABLE): Loop is not vectorizable.
```

The first loop in the example is not vectorizable because arrays `a` and `b` are potentially aliased. The second loop is not vectorizable because the Vectra LX coprocessor does not support vectorization of operations of type `unsigned char`.

The alias messages are emitted by the data dependence analysis phase of the compiler. Since this phase is applied to the whole function before any optimizations, you may see many alias messages for a function.

Each message shows the line numbers and the names of the two aliasing array accesses. One of the accesses in the message must be a store operation. Occasionally, the compiler does not have the original name of a variable and prints an anonymous name instead.

After the dependence analysis, the vectorizer is invoked on each loop. For each loop, the analysis messages begin with SIMD_LOOP_BEGIN and ends with SIMD_LOOP_VECTORIZED or SIMD_LOOP_NOT_VECTORIZED.

To save compilation time, the vectorizer stops analyzing a loop as soon as it finds a problem. Hence, you will only see one problem per loop.

SIMD Analysis Messages

The following is a list of the analysis messages currently produced by the vectorizer.

SIMD_ACCESS_GAPS:Gaps in memory access sequence (base %s). The compiler is unable to vectorize array accesses with gaps. A loop can only be vectorized if it can be transformed so that it accesses sequential array elements on each loop iteration. The example loop below can't be vectorized because it accesses only the even elements of the arrays.

```
for (i = 0; i < N; i += 2)
     a[i] = b[i] + c[i];
```

SIMD_ANALYSIS_OP: Unable to analyze %s.

The current version of the compiler does not support vectorization of this type of operation.

SIMD_ANALYSIS_REDUCTION: Unable to analyze reduction of *%s*.

The current version of the compiler does not support this type of vector reduction.

SIMD_ARRAY_ALIAS: Array base *%s* is aliased with array base *%s* at line *%d*.

The compiler conservatively assumes that the two array bases may point to the same memory location. See Section 4.3.1.

SIMD_ARRAY_DEPENDENCE: Dependence between array access '%s' and array access '%s' at line %d.

Vectorization requires that loop iterations can be executed in parallel. This message indicates that data dependences make parallel execution illegal. The example loop below can't be vectorized because the data computed in each iteration (i) depends on the data computed in the previous iteration (i-1).

```
for (i = 1; i < N; i++)
     b[i] = b[i - 1] + a[i];
```

SIMD_BAD_ACCESS: *%s* is not a simple array access.

The compiler is unable to vectorize loops that contain complex indirect memory accesses.

SIMD_BAD_ACCESS_STRIDE: Bad array access stride.

The compiler is unable to vectorize array accesses with negative or non-small strides.

SIMD_BAD_CALL: Loop contains call to functions.

The compiler is unable to vectorize loops that contain function calls.

SIMD_BAD_LOOP: Non-array indirect memory accesses, calls, or unhandled control-flow.

Loops with non-array indirect memory accesses, function calls, non-vectorizable input TIE instructions or unsupported control-flow such as GOTO statements and loops with non-computable trip counts cannot be vectorized.

SIMD_BAD_LOOP_UPPER_BOUND: Unsupported loop upper bound expression.

The compiler tries to standardize all *for* loop upper bounds to *index* <= *bound* expressions. Vectorization is not possible if the compiler is unable to perform this transformation.

SIMD_BAD_TIE_OP: Non-vectorizable input TIE instruction %s.

The compiler is unable to vectorize loops containing TIE instructions that access states, memory or queues.

SIMD_BOOLEAN: Boolean coprocessor required.

SIMD_DATA_DEPENDENCE: Data dependences prevent vectorization.

Vectorization requires that loop iterations can be executed in parallel. This message indicates that data dependences make parallel execution illegal. The `i` loop in the sample loop nest below cannot be vectorized because the data computed in each iteration i depends on the data computed in the previous iteration `i-1`.

```
for (i = 0; i < N; i++)
     a[i] += a[i-1];
```

SIMD_DISABLED: To enable vectorization use `-O3 -LNO:simd`.

The verbosity option was set (`-LNO:simd_v`) without setting the appropriate optimization options (`-O3 -LNO:simd`).

SIMD_IF_DIFF_VL: Mismatched if-statement vector lengths.

The *then* and *else* branch of an *if* statement must be vectorized by the same amount.

SIMD_IF_HAS_LOOP: Loop in if-statement.

The current version of the compiler is unable to vectorize a loop containing an *IF* statement if the *if* statement in turn contains another loop.

SIMD_IF_UNSAFE_ACCESS: Unsafe memory accesses in if-statement.

Vectorization vectorizes an *if* statement using a technique called if-conversion that executes both sides of the conditional and then conditionally merges the results. Since if-conversion executes the *then* and the *else* branches of the *if* statement before testing the *if* condition, safe transformation requires that each operation or memory access is present on both branches of the *if* statement. If the operation or memory access is not on both sides of the *if* statement but it can be speculated safely, you can use the `-TENV:X=?` or `-LNO:simd_agg_if_conv` compiler option, or the `simd_if_convert` pragma to force the transformation.

SIMD_LOOP_BEGIN: Vectorization analysis begins with a new loop.

Before vectorizing a loop, the compiler performs complete operator and data dependence analysis on the loop to check the legality of the transformation.

SIMD_LOOP_COST_MODEL: Vectorization of this loop is not beneficial.

The compiler may choose not to vectorize a loop because the estimated performance of the vectorized loop is worse than the original, non-vector loop performance or than the performance achieved by vectorizing another loop in the same loop nest.

SIMD_LOOP_NON_VECTORIZABLE: Loop is not vectorizable.

The compiler is unable to vectorize this loop.

SIMD_LOOP_STEP: Non-unit loop step *%d*.

Vectorization is supported only if the loop step is equal to 1 or can be normalized by the compiler.

SIMD_LOOP_TOO_DEEP: Loop nest is too deep.

Vectorization analysis is not performed on outer loops that contain inner loops that are too deeply nested.

SIMD_LOOP_VECTORIZABLE: Loop is vectorizable.

SIMD_LOOP_VECTORIZATION_TRY: XCC is trying to vectorize the loop.

SIMD_LOOP_VECTORIZATION_RETRY: XCC is trying again to vectorize the loop.

The vectorizer may try to vectorize the same loop using different vectorization factors. This message will be output for each tried vectorization factor.

SIMD_NEGATIVE_ACCESS_STRIDE: Negative memory access stride (base %s).

The compiler is unable to vectorize array accesses with negative stride. For example, the loop below can't be vectorized because of array 'b' is accessed with a negative stride.

```
for (i = 0; i < N; i ++)
     a[i] = b[N - i - 1];
```

SIMD_NO_COMMON_VL: No common vectorization length is available.

All operations in a loop must be vectorized by the same amount.

SIMD_NO_COND_MOVE: No conditional move instruction for vector type *%s*.

Vector if conversion requires a conditional move instruction for merging a result of the specified vector type, but no such instruction is available in the current processor configuration.

SIMD_NO_COPROC: Vectorization turned off by `-mno-use-coproc`.

Vectorization may generate co-processor instructions which is disabled with `-mno-use-coproc`. If it should be allowed, remove `-mno-use-coproc` from the command line.

SIMD_NO_GUARDS: Insufficient guard bits.

SIMD_NO_LOAD_CONV: Processor configuration does not support load conversion from *%s* to *%s*.

SIMD_NO_LOOP: No well formed loops in function.

The compiler can only vectorize loops with computable bounds.

SIMD_NO_REDUCTION: Processor configuration does not support vector reduction of %s.

SIMD_NO_SCALAR_CONV: Processor configuration does not support scalar to vector conversion of *%s*.

SIMD_NO_SELECT: No vector select instruction available to transform *%s* accesses.

In certain cases, the data needs to be rearranged to vectorize a loop successfully. However, the required vector select instructions are not available in the current processor configuration.

SIMD_NO_SNL: Function contains no well-formed simply-nested for-loops.

The compiler can vectorize inner or outer simply-nested *for* loops. In a simply-nested loop, there is only one loop at each nesting depth. For example:

```
for (i = 0; i < N; i++) {
  ...
  for (j = 0; j < M; j++) { ... }
  ...
  for (k = 0; k < M; k++) { ... }
  ...
}
```

Loop `j` and `k` are at the same nesting depth, so loop `i` is not simply-nested. The compiler may transform the loop nest by fusing the `j` and `k` loop into one or by fissioning the `i` loop into two simply nested loops -- one for `i` and `j`, and another one for `i` and `k`. If an outer loop cannot be transformed to simply-nested loops, it will not be vectorized.

SIMD_NO_STORE_CONV: Processor configuration does not support store conversion from *%s* to *%s*.

SIMD_NO_TYPE_CONV: Processor configuration does not support type conversion from *%s* to *%s*.

SIMD_NO_VECTOR_OP: Processor configuration does not support vector *%s*.

SIMD_NO_VECTOR_TYPE: Processor configuration does not support vector *%s*.

SIMD_NO_VECTOR_TYPE_SIZE: Processor configuration does not support %d-way vector *%s*.

SIMD_NON_COUNTABLE_LOOP: The loop is not a countable for-loop.

The compiler can vectorize only single induction variable countable for-loops, or loops that can be transformed automatically into this form. Use of non-int induction variables or pointer accesses may prevent the compiler from recognizing countable for-loops.

SIMD_PRAGMA_IGNORED: Unable to apply the #pragma at line %d to a for-loop.

#pragma 'concurrent', 'simd' or 'simd_if_convert' must be placed immediately before the countable for-loop it applies to. For example,

```
#pragma simd
     for (i = 0; i < N; i++)
          a[i] = b[i];
```

SIMD_PROC_BEGIN: Vectorization analysis for function *%s*.

SIMD_SCALAR_ARRAY_STORE: Scalar array store %s prevents vectorization.

A non-vector store to an array element prevents vectorization. For example, the loop below can't be vectorized because of the use of array 'temp'. I

```
int temp[2];
for (i = 0; i < 100; i += 2) {
     temp[0] = a[i];
     temp[1] = a[i + 1];
     b[i] = temp[0] + temp[1];
     b[i + 1] = temp[0] - temp[1];
}
```

SIMD_SCALAR_DEPENDENCE: Bad scalar dependences (variable *%s*).

Data dependences on a scalar variable prevent vectorization. For example, the loop below cannot be vectorized because of the data dependence on `partial_sum`.

```
int partial_sum = 0;
for (i = 0; i < 100; i++) {
  partial_sum += a[i];
  b[i] = partial_sum;
}
```

SIMD_SIGNED_POW_TWO_DIV: Unsupported division of a signed value by a power-of-two amount.

To vectorize a loop by converting a power-of-two division operation into an equivalent right-shift operation, the type of the shifted value must be unsigned. The sample expressions below demonstrate the difference between signed and unsigned division:

```
1 / 2 = 0
1 >> 1 = 0
(-1) / 2 = 0
(-1) >> 1 = -1
```

One possible conversion from division to right-shift operations is illustrated by the following example:

```
unsigned int ui;
    signed int si;

    ui = ui / 4;
    si = si / 4;
```

The equivalent shift expressions for the two variables are:

```
ui = ui >> 2;
si = ((si >= 0) ? si : (si + 3)) >> 2;
```

SIMD_SMALL_TRIP_COUNT: Loop trip count is too small.

Vectorization is not supported if the smallest available vectorization factor is less than the loop iteration count.

SIMD_TRAPEZOIDAL_INNER_LOOP: Inner loop bounds depend on outer loop index.

Loop nests where the loop bounds of an inner loop depend on an outer loop index are called trapezoidal. The compiler may vectorize trapezoidal loops of depth 2. In this example, loops `j` and `k` can be vectorized but loop `i` cannot because it contains multiple trapezoidal inner loops.

```
for (i = 0; i < 100; i++)
  for (j = 0; j < i; j++)
    for (k = 0; k < i; k++)
      s += a[i + j + k];
```

SIMD_UNALIGNED_LOAD: Processor configuration does not support unaligned *%s* loads.

SIMD_UNALIGNED_STORE: Processor configuration does not support unaligned *%s* stores.

SIMD_UNSIGNED_LOOP_UPPER_BOUND: Unsigned < upper bound expression may prevent loop vectorization.

The compiler tries to standardize all *for* loop upper bounds to *index* <= *bound* expressions. When an unsigned < loop bound expression is transformed to <=, the extra code required to guarantee correctness cannot be vectorized. Therefore, unsigned < upper bounds may prevent vectorization of enclosing loops.

SIMD_VARIANT_SHIFT: Shift by loop-variant shift amount not supported (*%s*).

The compiler is unable to vectorize expressions that shift by an amount that varies across loop iterations. For example, if `i` is the loop index, the left-shift *a[i]* << *b[i]* has a loop-variant shift-amount *b[i]*.

SIMD_VAR_STRIDE: Processor configuration does not support variable stride accesses.

In certain cases, variable stride accesses can be vectorized using indexed updating scalar-to-vector loads (LS.XU) and vector select instructions. However, these instructions are not available in the current processor configuration.

4.11 Managing Memory Bank Conflicts

Xtensa processors support the use of one or two load/store units. Dual load/store units potentially allow your application to issue two load/stores every cycle. Data caches on dual load/store configurations must have at least two banks. The memory is banked so that successive data memory access width size references go to different banks. The processor can not issue multiple memory references to the same bank in the same cycle. If a program tries to issue two loads to the same bank, the hardware will stall for one cycle. Stores are buffered and will be issued asynchronously to avoid stalls. The compiler will try to compile code to avoid bank conflicts. It will try to never schedule in the same cycle two loads that are not to adjacent memory locations. Sometimes, this will cause the compiler to generate longer schedules in cases where two loads could be issued together to different banks, but the compiler does not know their relative alignment. This optimization can be turned off using `-fno-memory-bank`.

The memory bank optimization is only applied be default to configurations with multiple memory banks. For local memory, a customer can choose to implement banking themselves, outside of the core supplied by Cadence. In such cases, the programmer can explicitly tell the compiler that the memory is banked using the `-mmemory-banks=n` option.

For data-cacheless configurations with two load/store units, banking is optional. Either with or without banking, the processor can be connected to two single-ported local data memories using the Xtensa Connection Box (CBOX) (see the Hardware Overview chapter in the *Xtensa Microprocessor Data Book*). With the CBOX, there may be a one-cycle stall if two loads are issued in a FLIX bundle targeting the same bank of the same local memory. If either the two loads target different memories or different banks of the same memory, there will be no stalls. To avoid stalls like this, users can annotate their program with ymemory pragmas and optionally add `-mcbox` option when invoking `xt-xcc`. Here is an example for using the ymemory pragma.

```
void foo(int x1[], int x2[], int y1[], int n) {
  int i;
  for (i=0; i<n; i++) {
    x1[i] = y1[i]+2;
    x2[i] = y1[i];
  }
}
```

To illustrate, we use `xt-xcc` options `-O2 -OPT:alias=restrct -OPT:unroll=2`.

With no ymemory pragmas, the inner loop may generate:

```
        .frequency 0.971 48.040
```

```
{       # format fmtAlngA64
        l32i    a3,a9,0                         # [0*II+0]  id:25
        l32i    a6,a9,4                         # [0*II+0]  id:25
}
{       # format fmtAlngA64
        s32i    a3,a8,0                         # [0*II+1]  id:27
        s32i    a6,a8,4                         # [0*II+1]  id:27
}
{       # format fmtBlngA64
        addi    a3,a3,2                         # [0*II+2]
        addi    a6,a6,2                         # [0*II+2]
}
{       # format fmtBlngA64
        s32i    a3,a5,0                         # [0*II+3]  id:26
        addi    a9,a9,8                         # [0*II+3]
}
{       # format fmtBlngA64
        s32i    a6,a5,4                         # [0*II+4]  id:26
        addi    a8,a8,8                         # [0*II+4]
}
        addi    a5,a5,8                         # [0*II+5]
```

Note the dual loads being bundled in the same cycle which will result in stalls during execution.

To avoid such bundling, we can add ymemory pragmas, which annotate arrays or pointers in a subroutine, as below:

```
void foo(int x1[], int x2[], int y1[], int n) {
  int i;
#pragma ymemory (y1)
  for (i=0; i<n; i++) {
    x1[i] = y1[i]+2;
    x2[i] = y1[i];
  }
}
```

The new code compiled with the same options results in the following inner loop schedule:

```
        .frequency 0.971 48.040
        l32i.n  a3,a9,0                         # [0*II+0]  id:25
{       # format fmtAlngA64
        s32i    a3,a8,0                         # [0*II+1]  id:27
```

```
        l32i    a6,a9,4                         # [0*II+1]  id:25
}
{       # format fmtBlngA64
        addi    a3,a3,2                         # [0*II+2]
        s32i    a6,a8,4                         # [0*II+2]  id:27
}
{       # format fmtBlngA64
        addi    a6,a6,2                         # [0*II+3]
        addi    a9,a9,8                         # [0*II+3]
}
{       # format fmtAlngA64
        s32i    a3,a5,0                         # [0*II+4]  id:26
        s32i    a6,a5,4                         # [0*II+4]  id:26
}
{       # format fmtBlngA64
        addi    a8,a8,8                         # [0*II+5]
        addi    a5,a5,8                         # [0*II+5]
}
```

which has the same number of cycles per iteration but with no stall from load conflicts at run-time.

Please note that the programs that can gain the most benefit from ymemory pragmas are those with plenty of instruction-level parallelism to bundle each load with other operations. Otherwise, some FLIX bundle may be partially filled in order to avoid pairing conflicting loads.

With ymemory pragmas, the loads are divided into two categories: those loading from ymemory and those loading from non-ymemory. `xt-xcc` will separate ymemory loads by default, leaving no restrictions for the non-ymemory loads. With additional xt-xcc `-mcbox` option, non-ymemory loads will also be separated into different FLIX bundles as in the case for ymemory loads. This is useful for configurations with two load/store units and two single-ported local data memories. With -mcbox, two loads can still be bundled by xt-xcc if one is a ymemory and the other one non-ymemory loads.

A. Command-Line Option List with `-help`

Invoking `xt-xcc` or `xt-xc++` with `-help` will generate the summary of command-line options shown in Table A-20.

Table A-20. Summary of Command-Line Options

Option	Description
`--all-warnings`	Equivalent to `-Wall`
`--ansi`	Equivalent to `-ansi`
`--assemble`	Equivalent to `-S`
`--comments`	Equivalent to `-C`
`--compile`	Equivalent to `-c`
`--debug`	Equivalent to `-g`
`--define-macro`	Equivalent to `-D`
`--dependencies`	Equivalent to `-M`
`--extra-warnings`	Equivalent to `-W`
`--force-link`	Equivalent to `-u`
`--help`	Equivalent to `-help`
`--imacros`	Equivalent to `-imacros`
`--include`	Equivalent to `-include`
`--include-directory`	Equivalent to `-I`
`--include-directory-after`	Equivalent to `-idirafter`
`--include-prefix`	Equivalent to `-iprefix`
`--include-with-prefix`	Equivalent to `-iwithprefix`
`--include-with-prefix-before`	Equivalent to `-iwithprefixbefore`
`--library-directory`	Equivalent to `-L`
`--no-line-commands`	Equivalent to `-P`
`--no-standard-includes`	Equivalent to `-nostdinc`
`--no-standard-libraries`	Equivalent to `-nostdlib`
`--no-warnings`	Equivalent to `-w`
`--optimize`	Equivalent to `-O`
`--output`	Equivalent to `-o`
`--pedantic`	Equivalent to `-pedantic`
`--pedantic-errors`	Equivalent to `-pedantic-errors`
`--print-file-name`	Equivalent to `-print-file-name`
`--print-libgcc-file-name`	Equivalent to `-print-file-name=libgcc.a`

Table A-20. Summary of Command-Line Options (continued)

Option	Description
--print-missing-file-dependencies	Equivalent to -MG
--print-prog-name	Equivalent to -print-prog-name
--rename-section	Rename a section
--save-temps	Equivalent to -keep
--shared	Equivalent to -shared
--static	Equivalent to -static
--trace-includes	Equivalent to -H
--traditional-cpp	Equivalent to -traditional-cpp
--trigraphs	Equivalent to -trigraphs
--undefine-macro	Equivalent to -U
--user-dependencies	Equivalent to -MM
--verbose	Equivalent to -v
--version	Equivalent to -version
--write-dependencies	Equivalent to -MD
--write-user-dependencies	Equivalent to -MMD
--xtensa-core=	Specify the processor configuration
--xtensa-params=	Specify the TIE development kit directory
--xtensa-system=	Specify the processor core registry
-A	Add specified preprocessor assertion
-B	-B *dir* is equivalent to -isystem *dir*/include
-C	Retain C/C++ comments after preprocessing
-CC	Retain all comments, including comments inside macros
-D	Define specified preprocessor macro
-E	Run only preprocessor and send result to standard output
-H	Print names of all header files used during preprocessing
-I	Add named directory to the include search path list
-INLINE:	Option group to control function inlining
-IPA	Perform inter-procedural analysis and optimization
-L	Add named directory to the library search path list
-LNO:	Option group to control loop-nest optimizations
-M	Run preprocessor and print list of make dependencies
-MD	Generate a `.d' file that contains make dependencies
-MF	Specify output file for make dependencies

Table A-20. Summary of Command-Line Options (continued)

Option	Description
-MG	Treat missing header files as generated files in the source directory
-MM	Same as -M, but ignore system header files
-MMD	Same as -MD, but ignore system header files
-MP	Add phony targets to make dependencies
-MQ	Specify target name with quoting for make dependencies
-MT	Specify target name for make dependencies
-O	Same as -O2
-O0	Do not optimize
-O1	Perform local (basic block) optimizations
-O2	Perform global (function level) optimizations
-O3	Perform global optimizations and loop-nest transformations
-OPT:	Option group to control various optimizations
-Os	Optimize for space
-P	Do not generate #line directives during preprocessing
-S	Produce a `.s' assembly file and stop
-T	Use scriptfile as the linker script
-TENV::	With -TENV:X=n, option group to control speculation
-U	Undefine specified preprocessor macro
-W	Enable extra warnings
-Waddress	Warn about suspicious use of memory addresses
-Waggregate-return	Warn about returning structures, unions or arrays
-Wall	Enable most warnings
-Wbad-function-cast	Warn when a function call is cast to a non-matching type
-Wc++-compat	Warn about code that is not valid C++
-Wcast-align	Warn about pointer casts that increase alignment
-Wcast-qual	Warn about casts that discard qualifiers
-Wchar-subscripts	Warn about subscripts whose type is `char'
-Wcomment	Warn if nested comments are detected
-Wconversion	Warn about possibly confusing type conversions
-Wctor-dtor-privacy	Warn when all the constructors or destructors of a class are private
-Wdeclaration-after-statement	Warn when a declaration is found after a statement in a block

Table A-20. Summary of Command-Line Options (continued)

Option	Description
-Weffc++	Warn about violation of some style rules from Effective C++
-Werror	Treat all warnings as errors
-Werror=	Treat the specified warnings as errors
-Wextra	Enable extra warnings
-Wfatal-errors	Stop compiling at the first error
-Wfloat-equal	Warn about floating-point equality comparisons
-Wformat	Warn about printf/scanf format anomalies
-Wformat-nonliteral	Warn about printf/scanf formats that are not string literals
-Wformat-security	Warn about printf/scanf formats that may be security problems
-Wformat-y2k	Warn about strftime formats which may yield a two-digit year
-Wformat=2	Enable additional format warnings
-Wimplicit	Same as -Wimplicit-int -Wimplicit-function-declaration
-Wimplicit-function-declaration	Warn when function is declared implicitly
-Wimplicit-int	Warn when declaration does not specify type
-Wimport	Warn about use of #import
-Wlarger-than-	Warn if an object larger than the specified size is defined
-Wmain	Warn about suspicious declarations of main
-Wmissing-braces	Warn about missing braces in aggregate initializers
-Wmissing-declarations	Warn about functions without previous declarations
-Wmissing-field-initializers	Warn if a structure initializer is missing some fields
-Wmissing-format-attribute	Warn about function pointers that might be candidates for format attributes
-Wmissing-include-dirs	Warn if a user-supplied include directory does not exist
-Wmissing-prototypes	Warn about functions without previous prototypes
-Wnested-externs	Warn about extern declarations not at file scope level
-Wno-attributes	Do not warn about unexpected attributes
-Wno-deprecated	Do not warn about uses of deprecated features
-Wno-deprecated-declarations	Do not warn about uses of __attribute__((deprecated))
-Wno-div-by-zero	Do not warn about integer division by zero
-Wno-endif-labels	Do not warn about text following #else and #endif
-Wno-error=	Do not treat the specified warnings as errors
-Wno-format-extra-args	Do not warn about extra format arguments

Table A-20. Summary of Command-Line Options (continued)

Option	Description
-Wno-format-zero-length	Do not warn about zero-length formats
-Wno-int-to-pointer-cast	Do not warn about casts to pointer type from an integer of a different size
-Wno-invalid-offsetof	Do not warn about using the offsetof macro with non-POD types
-Wno-long-long	Do not warn about long long variables even if -pedantic is used
-Wno-multichar	Do not warn about multicharacter constants
-Wno-non-template-friend	Do not warn about non-templatized friend functions declared within a template
-Wno-overflow	Do not warn about overflow in constant expressions
-Wno-pmf-conversions	Do not warn about converting a bound pointer to a member function to a plain pointer
-Wno-pointer-to-int-cast	Do not warn about casts from a pointer type to an integer of a different size
-Wno-pragmas	Do not warn about misuses of pragmas
-Wno-variadic-macros	Do not warn about variadic macros used in pedantic mode
-Wnon-virtual-dtor	Warn if a class has virtual functions but a non-virtual destructor
-Wnonnull	Warn about passing null for arguments marked with a nonnull attribute
-Wnormalized=	Control warnings about normalization of characters in identifier names
-Wold-style-cast	Warn if an old C-style cast to a non-void type is used
-Wold-style-definition	Warn if an old-style function definition is used
-Woverlength-strings	Warn about string constants too long to be portable
-Woverloaded-virtual	Warn when a derived class function declaration may be an error in defining a virtual function
-Woverride-init	Warn if an initialized field is overridden when using designated initializers
-Wpacked	Warn about structures where packed attribute does not reduce size
-Wpadded	Warn if padding is included in a structure
-Wparentheses	Warn about possibly confusing omitted parentheses
-Wpointer-arith	Warn about function and void pointer arithmetic
-Wpointer-sign	Warn about pointer assignments or argument passing with different signedness

Table A-20. Summary of Command-Line Options (continued)

Option	Description
-Wredundant-decls	Warn about multiple declarations of the same object
-Wreorder	Warn and rearrange the order of member initializers to match their declaration order
-Wreturn-type	Warn about function return type inconsistencies
-Wsequence-point	Warn about order of evaluation that is not defined by sequence points
-Wshadow	Warn when one local variable shadows another
-Wsign-compare	Warn about possibly incorrect signed/unsigned comparisons
-Wsign-promo	Warn when an unsigned or enumerated type is promoted to a signed type over an unsigned
-Wstrict-aliasing	Warn about code that may violate rules for -fstrict-aliasing
-Wstrict-pototypes	Warn about non-prototyped function declarations
-Wswitch	Warn when switch has an index of an enumerated type and not all values are covered
-Wswitch-default	Warn about switch statements without a default case
-Wswitch-enum	Warn about switch cases that do not match enum type values
-Wsystem-headers	Do not suppress warnings from system header files
-Wtraditional	Warn about certain constructs that behave differently in traditional C from ANSI C
-Wtrigraphs	Warn if any trigraphs are encountered
-Wunaligned	Warn about memory references that are provably unaligned
-Wundef	Warn if an undefined identifier is evaluated in #if directive
-Wunknown-pragmas	Warn about unknown pragmas
-Wunused	Warn about unused functions, labels, parameters, variables and values
-Wunused-function	Warn if a static function is declared or used but not defined
-Wunused-label	Warn if label is defined but not used
-Wunused-macros	Warn if a macro is defined but not used
-Wunused-parameter	Warn if parameter is not used
-Wunused-tie-intrinsic-result	Warn if TIE intrinsic result is not used
-Wunused-value	Warn if statement computes value that is not used
-Wunused-variable	Warn if variable is not used
-Wvolatile-register-var	Warn about volatile register variables
-Wwrite-strings	Give string constants `const char *' type

Table A-20. Summary of Command-Line Options (continued)

Option	Description
-ansi	Disable GNU extensions to ANSI C
-c	Produce a `.o' object file and stop
-clist	Generate a `.w2c.c' file
-dD	Print macro names and expansions in the preprocessor output (requires -E)
-dM	Print macro definitions in effect after preprocessing (requires -E)
-dN	Print macro names in the preprocessor output (requires -E)
-dumpversion	Print the front end version
-e	Set the start address
-fPIC	Generate position-independent code
-fassociative-math	Allow re-association of floating-point operations. This is the default at -O3.
-fb_create	Equivalent to -fb_create_32
-fb_create_32	Instrument the code to generate profile information using 32-bit counters
-fb_create_64	Instrument the code to generate profile information using 64-bit counters
-fb_create_HW	Instrument the code to generate profile information on hardware
-fb_opt	Optimize the code using previously generated profile information
-fb_reorder	Enable function reordering when using -fb_opt
-fcheck-new	Check that the pointer returned by operator new is non-null
-fcommon	Place uninitialized global variables in common, not in bss
-fconserve-space	In C++, place uninitialized global variables in common, not in bss
-fdiagnostics-show-location=	Indicates how often source location information should be emitted [once|every-line]
-fdiagnostics-show-option	Show related command-line options in diagnostic messages
-felide-constructors	Elide constructors when this seems possible
-fexceptions	Enable support for exception handling in C++
-fextended-identifiers	Allow universal characters in identifier names
-ffast-math	Perform some optimizations that may violate ANSI or IEEE arithmetic rules
-ffor-scope	Limit the scope of variables declared in a for initialization statement to the for loop

Table A-20. Summary of Command-Line Options (continued)

Option	Description
-ffreestanding	Assert that compilation takes place in a freestanding environment
-ffunction-sections	Place each function in a separate section named .text.function_name
-fhosted	Assert that compilation takes place in a hosted environment
-finput-charset=	Set the input file character set
-fkeep-inline-functions	Do not remove inline function even if all calls are inlined
-fkeep-static-functions	Do not remove static function even if all calls are inlined
-fmemory-bank	Turn on scheduling for memory banks
-fmessage-length=	Try to format error messages so that they fit in lines of about n characters
-fms-extensions	Accept some non-standard Microsoft extensions
-fno-asm	Do not recognize `asm`, `inline` or `typeof` in C. Do not recognize `typeof` in C++.
-fno-associative-math	Disable re-association of floating-point operations.
-fno-builtin	Do not replace built-in functions with inlined code
-fno-builtin-*funcname*	Do not replace named built-in function with inlined code
-fno-default-inline	Do not assume `inline' for functions defined inside a class scope (C++)
-fno-dollars-in-identifiers	Disallow $ in identifier names
-fno-for-scope	Do not limit the scope of variables declared in a for initialization statement to the for loop
-fno-freestanding	Assert that compilation does not take place in a freestanding environment
-fno-hosted	Assert that compilation takes place in a non-hosted environment
-fno-implicit-templates	Do not implicitly instantiate templates
-fno-inline	Do not inline any functions
-fno-inline-functions	Do not inline static functions not marked as 'inline'
-fno-merge-constants	Do not merge string constants
-fno-pragma-loop-count	Ignore `loop_count` pragmas
-fno-reciprocal-math	Do not allow the reciprocal of a value to be used instead of diving by the value.
-fno-rtti	Do not generate C++ run-time type information
-fno-strict-aliasing	Equivalent to `-OPT:alias=any`

Table A-20. Summary of Command-Line Options (continued)

Option	Description
-fno-strict-overflow	Do not assume strict signed overflow rules or pointer semantics
-fno-threadsafe-statics	Do not emit extra code for thread-safe initialization of local statics
-fno-unroll-loops	Do not unroll any inner loops
-fno-unsafe-math-optimizations	At -O3, disable optimizations that don't strictly conform to C/C++ or IEEE floating point standards.t
-fno-zero-initialized-in-bss	Do not place zero-initialized variables in bss
-fopt-gen	Equivalent to -foption-file-gen=
-fopt-gen=	Equivalent to -foption-file-gen=
-fopt-use=	Equivalent to -foption-file-use=
-foption-file-gen	Specifiy the generation of a compiler options file using the output file name appended with .opt"
-foption-file-gen=	Specifiy a file to generate compiler options
-foption-file-use=	Specify a file to use for compiler options
-fpack-struct	Pack all structure members together without holes
-fpermissive	Allow some nonconforming code to compile
-fpic	Generate position-independent code
-fpragma-gen	Specifiy the output of pragmas when generating the compiler options file
-fpreprocessed	Tell preprocessor that input has already been preprocessed
-freciprocal-math	Allow the reciprocal of a value to be used instead of diving by the value. This is the default at -O3.
-fshort-enums	Allocate to an enum only as many bytes as needed
-fsigned-bitfields	Treat bitfields as signed
-fsigned-char	Make `char' signed by default
-fsingle-precision-constant	Convert floating point constant to single precision constant
-fstrict-aliasing	Eqivalent to -OPT:alias=typed
-fsyntax-only	Check the code for syntax errors only
-fthreadsafe-statics	Emit extra code for thread-safe initialization of local statics
-funsafe-math-optimizations	Enable optimizations that don't strictly conform to C/C++ or IEEE floating point standards. This is the default at -O3.
-funsigned-bitfields	Treat bitfields as unsigned
-funsigned-char	Make `char' unsigned by default
-fvisibility-inlines-hidden	Make visibility of all inline functions hidden

Table A-20. Summary of Command-Line Options (continued)

Option	Description
-fvisibility=	Set the default symbol visibility ([default|internal|hidden|protected])
-g	Generate full debugging information
-g0	Turn off generation of debugging information
-g1	Produce minimal debugging information
-g2	Generate full debugging information
-g3	Generate full debugging information
-gdwarf-2	Generate full debugging information
-help	Print this list
-hwpg	Enable hardware-based profiling with performance counters
-hwpg=	Enable hardware-based profiling with specified timer interrupt
-idirafter	Add the directory to a second include path used for files not found in the first
-imacros	Preprocess named file for macro definitions
-include	Include named file before any others
-ipa	Perform inter-procedural analysis and optimization. If any file is compiled with ipa (-c -ipa), the executable built from the .o file must be linked with the -ipa option and must be linked with xt-xcc and not xt-ld directly.
-ipaentry=	During an ipalib link, ensure that the given symbol is not optimized away
-ipalib	Link several ipa object files into a single normal object file
-iprefix	Specifiy prefix as the prefix for subsequent -iwithprefix options
-iquote	Search named directory only for quoted (not bracketed) header files
-isystem	Add a system directory to the second include path
-iwithprefix	Add a directory to the second include path using the prefix from the -iprefix option
-iwithprefixbefore	Add a directory to the main include path using the prefix from the -iprefix option
-keep	Keep intermediate files
-keep_min	Keep a minimal set of intermediate files
-l	With -lname add library libname.a to the link list
-mcbox	Do not bundle multiple loads targeting the same C-Box bank

Table A-20. Summary of Command-Line Options (continued)

Option	Description
-mcoproc	Allow use of coprocessor register files. Higher performance but perhaps not safe when compiling interrupt handlers.
-menable-non-exact-imaps	Allow use of inexact TIE imaps
-mflush-tieport	Serialize TIE port references with EXTW instructions
-mlongcalls	Translate direct calls into indirect calls
-mlsp=	Use named linker support package
-mno-enable-non-exact-imaps	Do not allow use of inexact TIE imaps
-mno-flix	Do not use or allow any FLIX formats
-mno-fused-madd	Do not generate floating-point multiply and accumulate instructions
-mno-generate-flix	Do not generate any FLIX formats, but allow them in `asm` statements
-mno-reorder-tieport	Do not reorder TIE port references
-mno-serialize-volatile	Do not generate `MEMW` instructions in between volatile memory references
-mno-target-align	Do not try to align branch targets
-mno-zero-cost-loop	Disable use of zero-overhead loop instructions
-mrename-section-	Rename a section
-mspecs=	Replace specs file with the given one
-mtext-section-literals	Place literals in the text section
-mzero-init-data	Equivalent to `-fno-zero-initialized-in-bss`
-nostdinc	Do not search predefined system header file directories
-nostdinc++	Do not search predefined C++ header file directories
-nostdlib	Do not link with predefined libraries and startup files
-o	Put output in the named file
-pedantic	Issue warnings required by strict ANSI C
-pedantic-errors	Same as `-pedantic`, with errors instead of warnings
-print-file-name=	Print the full absolute name of the library that would be used when linking
-print-libgcc-file-name	Equivalent to `-print-file-name=libgcc.a`
-print-prog-name=	Print the full absolute name of the compiler program or one of its phases
-r	Use a relocatable or partial link.
-s	Remove all symbol table and relocation information from the executable
-save-temps	Keep intermediate files

Table A-20. Summary of Command-Line Options (continued)

Option	Description
-shared	Create a shared library
-show	Print compiler phases as they are invoked
-specs=	Add spec file to the end of the list of specs files
-static	Do not link with shared libraries
-std=c++98	Support 1998 ISO C++ standard plus amendments
-std=c89	Support ISO C from 1990
-std=c99	Support revised ISO C from 1999
-std=c9x	Support revised ISO C from 1999
-std=gnu++98	The same as `-std=c++98', with GNU extensions. This is the default for C++ code.
-std=gnu89	Support ISO C from 1990, with GNU extensions
-std=gnu99	Support ISO C from 1999, with GNU extensions
-std=gnu9x	Support ISO C from 1999, with GNU extensions
-std=iso9899:1990	Support ISO C from 1990
-std=iso9899:199409	Support ISO C from 1990, with 1994 amendments
-std=iso9899:1999	Support revised ISO C from 1999
-std=iso9899:199x	Support revised ISO C from 1999
-traditional-cpp	Attempt to support some aspects of traditional C preprocessors
-trigraphs	Support ANSI C trigraphs
-u	Pretend symbol is undefined to force linking of library modules
-v	Print the compiler version and phases as they are invoked
-version	Print the compiler version
-w	Disable all warnings
-woffall	Disable all warnings
-woffoptions	Disable warnings about command-line options
-x assembler	Treat following input files as assembly language files
-x assembler-with-cpp	Treat following input files as assembly language files requiring preprocessing
-x c	Treat following input files as C language files
-x c++	Treat following input files as C++ language files
-x c++-header	Treat following input files as C++ header files
-x c-header	Treat following input files as C language files
-x none	Use input file name suffix to choose the source language

The following options are only supported in C++:

```
-Wctor-dtor-privacy, -Weffc++, -Wno-invalid-offsetof,
-Wno-non-template-friend, -Wno-pmf-conversions, -Wnon-virtual-dtor,
-Wold-style-cast, -Woverloaded-virtual, -Wreorder, -Wsign-promo,
-fcheck-new, -fconserve-space, -felide-constructors, -ffor-scope,
-fno-default-inline, -fno-for-scope, -fno-implicit-templates, -fno-rtti,
-fpermissive, -std=c++98, -std=gnu++98.
```

The following options are only supported in C:

```
--ansi, --traditional-cpp, -Wbad-function-cast,
-Wdeclaration-after-statement, -Wmissing-declarations,
-Wmissing-prototypes, -Wnested-externs, -Wno-int-to-pointer-cast,
-Wno-pointer-to-int-cast, -Wold-style-definition, -Wpointer-sign,
-Wstrict-prototypes, -Wtraditional, -ansi, -clist, -traditional-cpp.
```

B. Summary of Compiler Pragmas

Following is a list of available pragmas.Pragmas can either be used directly as below or in macros using the form

#definemacro_name _Pragma("pragma name")

Table B-21. Summary of Compiler Pragmas

Pragma	Description
`#pragma aligned (<pointer_id>, <aligment_byte_boundary>)`	Informs the compiler that arrays are correctly aligned. `alignment_byte_boundary` must be a power of two.
`#pragma concurrent`	Marks the loop that follows the pragma declaration to indicate that each iteration of the loop is independent of all other iterations.
`#pragma flush_memory`	Insures that all data is effectively flushed to or from memory at the point of the pragma.
`#pragma flush`	Same as `flush_memory`, except that it also affects the ordering of TIE ports.
`#pragma frequency_hint [NEVER | FREQUENT]`	Placed directly after the conditional test of an `if` statement; indicates how often the conditional branch is taken.
`#pragma loop_count min=<level>, max=<level>, factor=<level>, avg=<level>`	Tell the compiler information about the number of iterations in the following loop.
`#pragma no_reorder`	Prevents the compiler from reordering references.
`#pragma no_reorder_memory`	Same as `no_reorder`, except it does not affect TIE port references.
`#pragma no_simd`	Do not vectorize the following loop.
`#pragma no_unroll`	Disable loop unrolling of the following loop.
`#pragma simd_if_convert`	Instructs the compiler to perform the `if conversion` optimization.
`#pragma simd`	Equivalent to the use of both `#pragma concurrent` and `#pragma simd_if_convert`.

Table B-21. Summary of Compiler Pragmas (continued)

Pragma	Description
`#pragma super_swp ii={x}, unroll={y}`	Place immediately preceding the inner loop, this guides the compiler by telling it how much to unroll the inner loop and how many cycles (after unrolling) to try to schedule.
`#pragma swp_schedule ...`	Allows the software pipeliner to find the schedule quickly.
`#pragma ymemory`	Used to avoid stalls in configurations with two load/store units connected to single-ported local data memories using CBox.

Index

Symbols
__restrict ... 36
#pragma aligned ... 71
#pragma concurrent ... 72
#pragma flush ... 34
#pragma flush_memory ... 33, 34
#pragma frequency_hint FREQUENT ... 60
#pragma frequency_hint NEVER ... 60
#pragma no_reorder ... 35
#pragma no_reorder_memory ... 35
#pragma simd ... 73
#pragma simd_if_convert ... 72
#pragma super_swp ... 61
#pragma swp_schedule ... 62
#pragma ymemory ... 23, 84

A
Aliasing ... 52, 54
Alignment
  attribute ... 38
  for vectorization ... 69
alloca ... 32, 37
--all-warnings ... 14
Analysis
  aliasing ... 54
  messages ... 76
ANSI ... 13, 29, 36
--ansi ... 11
-ansi ... 11
-Aquestion ... 8
Arithmetic, pointer ... 36
Arrays
  variable length ... 36
  zero-length ... 37
asm ... 41
  constraint modifiers ... 43
  operand constraints ... 42
--assemble ... 7
Assembler
  notes ... 49
  options ... 25
Attributes ... 37

B
-B ... 8
Big-endian ... 6
Boolean
  registers ... 45
  types for Xtensa ... 45
bss ... 23, 38, 39
Built-in functions ... 23, 32

C
-C ... 8
-c ... 7
C compiler ... 1
C operator functions ... 46
C++
  comments in C ... 35
  compiler ... 1
  exception handling ... 2
-clist ... 18, 50, 68
Code generation options ... 22
Command-line options ... 7
  compiler output ... 7
  language dialect ... 11
  preprocessor ... 8
  summary ... 87, 101
--comments ... 8
--compile ... 7
Compiler optimizations, viewing the effects ... 49
Compiler output options ... 7

D
-D ... 9
Data types ... 30
-dD ... 8
Deadlock and TIE ports ... 34
--debug ... 18
Debugging options ... 18
Declaration attributes ... 38
--define-macro name ... 9
--dependencies ... 10
-dM ... 9
-dN ... 9
-dumpversion ... 7

E
-E ... 7
-eADDRESS ... 25
Endianness ... 6
Exception Handling ... 2
--extra-warnings ... 14

EXTW....23

F

-fb_create filename....20
-fb_create_32 filename....20
-fb_create_64 filename....20
-fb_create_HW filename....20, 59
-fb_opt filename....20
-fb_reorder....20
-fcheck-new....11
-fcommon....23
-fconserve-space....11
-felide-constructors....11
-fexceptions....2, 11
-ffast-math....20
-ffreestanding....12
-ffunction-sections....23
-fhosted....12
Files
  .d....10
  .w2c.c....18, 68
  assembler....49
  input file name suffix....5
  saving intermediate....19
-fkeep-inline-functions....22
-fkeep-static-functions....22
FLIX....20, 23
Floating point....6, 23, 53
-fmessage-length=n....18
-fms-extensions....12
-fno-asm....12
-fno-builtin....23
-fno-dollars-in-identifiers....12
-fno-for-scope....11
-fno-freestanding....12
-fno-hosted....12
-fno-inline....22
-fno-inline-functions....22
-fno-pragma-loop-count....20
-fno-signed-bitfields....12
-fno-strict-aliasing....20, 21
-fno-unsigned-char....12
-fno-zero-initialized-in-bss....23
--force-link symbol....26
-fpack-struct....12
-fpic....23
-frepo....12
-fsigned-char....12
-fstrict-aliasing....20, 21
-fsyntax-only....7
Functions
  built-in....32
  intrinsic for Xtensa instructions....29
-funsigned-bitfields....12

G

-g....18
GCC
  command-line options....2
  differences from exception handling....2
  migrating from....2
-g-dwarf-2....18
-glevel....18
Guard bits....74

H

-H....9
Header files
  intrinsic functions defined in....29
  xtensa/tie/.h....30
  xtensa/tie/xt_booleans.h....29
  xtensa/tie/xt_core.h....29
  xtensa/tie/xt_density.h....29
  xtensa/tie/xt_FP.h....29
  xtensa/tie/xt_ioports.h....29
  xtensa/tie/xt_MAC16.h....30
  xtensa/tie/xt_MISC.h....30
  xtensa/tie/xt_MUL16.h....30
  xtensa/tie/xt_vectralx.h....30
--help....7
-help....7, 87
-hwpg....21

I

-I....9
-idirafter....9
--imacros....9
-imacros....9
--include....9
-include....9
--include-directory....9
--include-directory-after....9
--include-prefix....9
--include-with-prefix....10
--include-with-prefix-before....10
-INLINE
  =off....22
  aggressive=off....22
  aggressive=on....22
  must....22
  never....22

preemptible 22
requested 22
requested_only 22
Inline assembly 41
Inlining 35, 50, 62
Inlining options 21
Input file handling 5
Intrinsic functions for Xtensa instructions 29
-ipa 21, 50, 63
ipakeep directory 63
-iprefix 9
-iquote 9
-isystem 10
-iwithprefix 10
-iwithprefixbefore 10

K

-keep 19

L

Language
dialect options 11
extensions 29
from the file name 5
-Ldir 25
--library-directory dir 25
Linker
interprocedural optimization 63
options 25
Literals 25
Little-endian 6
-lname 25
-LNO 21
simd_verbose 75
Long long variables 36
LSP, linker support package 27

M

-M 10
macros 8, 9
makefile dependencies 10
-mcbox 23
-MD 10
Memory accesses, limitation in stride 74
Memory consistency 32
MEMW 24
-mflush-tieport 23, 34, 35
-mfused-madd 23
-MG 10
-mlongcalls 25
-mlsp=lspname 27
-mno-flix 23
..... 21
-mno-fused-madd 23
-mno-l32r-flix 23
-mno-reorder-tieport 24
-mno-serialize-volatile 24
-mno-target-align 25
-mno-zero-cost-loop 24
-mrename-section-old=new 25
-mshift32 24
-mtext-section-literals 25
-mzero-init-data 24

N

--no-line-commands 10
--no-standard-includes 10
--no-standard-libraries 26
-nostdinc 10
-nostdinc++ 10
-nostdlib 26
--no-warnings 13

O

-O 19
-o 7
-O0 19
-O1 19
-O2 19
-O3 20
Operator Overloading 46
-OPT 20
OPT options 53
Optimization 54
for space 20, 50
inter-procedural 62
interprocedural 21
-OPT option group 53, 67
Options
assembler 25
code generation 22
compiler output 7
debugging 18
inlining 21
language dialect 11
linker 25
OPT 53
preprocessor 8
warning control 13
Xtensa-specific 27
-Os 20

Outer loop vectorization 74
--output....7

P

-P 10
Packed structures 40
-pedantic 13
-pedantic-errors....13
Performance counters....21
Pointer arithmetic 36
Pointers, restrict 36
Pragma memory consistency....33
Pragmas....101
  aligned....71
  concurrent 72
  flush....34
  flush memory....33, 34
  frequency....60
  memory....23, 84
  no reorder....35
  no reorder memory....35
  simd....73
  simd_if_convert 72
  software pipeliner schedule....62
  super software pipelining....61
predefined macros 9
Preprocessor
  predefined macros....6
Preprocessor options 8
-print-file-name=library 19
-print-libgcc-file-name....19
--print-missing-file-dependencies 10
-print-prog-name=program....19
Profiling 21
Profiling feedback....57

R

Registers
  asm constraints 42
  Boolean 45
Restrict pointers 36, 56

S

-S 7
-s 26
-save-temp 19
Section
  .bss....23, 38, 39
  .literal....25
  attribute 41
  renaming 25
-shared 26
-show.... 7
SIMD 68
Software pipeliner 50
-static.... 26
-std 13

T

-T scriptfile.... 26
TIE ports.... 33
  and deadlock 34
--trace-includes 9
--traditional-cpp 11
-traditional-cpp 11
-trigraphs 13

U

-U 11
-u symbol.... 26
--undefine-macro.... 11
--user-dependencies 10

V

-v 8
Variable-length arrays 36
Variables, long long.... 36
Vectorization 67
  analysis report 75
  guard bits.... 74
  outer loop.... 74
Vectra LX 45, 67
--verbose 8
--version 8
-version 8
Volatile memory consistency.... 32

W

-W.... 14
-w 13
-Waggregate-return.... 14
-Wall 14
Warning control options 13
-Wbad-function-cast 14
-Wcast-align 14
-Wcast-qual 14
-Wchar-subscripts 14
-Wcomment.... 14
-Wconversion 14
-Weffc++.... 14
-Werror 13
-Wformat 15
-Wimplicit.... 15

-Wimplicit-function-declaration 15
-Wimplicit-int 15
-Wl, option 25, 26
-Wmain 15
-Wmissing-braces 15
-Wmissing-declarations 15
-Wmissing-prototypes 15
-Wnested-externs 15
-Wno-deprecated 15, 16
-Wno-long-long 16
-Wno-non-template-friend 16
-Wnon-virtual-dtor 16
-Wno-pmf-conversions 16
-woffall 13
-Woverloaded-virtual 16
-Wp,option 11
-Wparentheses 16
-Wpointer-arith 17
-Wredundant-decls 17
-Wreturn-type 17
--write-dependencies 10
--write-user-dependencies 10
-Wshadow 17
-Wsign-compare 17
-Wsign-promo 17
-Wstrict-prototypes 17
-Wswitch 17
-Wtraditional 17
-Wtrigraphs 17
-Wundef 17
-Wunused 18
-Wunused-function 17
-Wunused-label 17
-Wunused-parameter 17
-Wunused-value 18
-Wunused-variable 18
-Wwrite-strings 18

X

-x 8
xtbool 45
Xtensa
  Boolean types 45
  intrinsic functions for instructions 29
--xtensa-core=core 27
--xtensa-params=tdk 27
Xtensa-specific options 27
--xtensa-system=registry 27
xt-xc++ 1
xt-xcc 1

Z

Zero-length arrays 37