Qualcomm Technologies, Inc.

Qualcomm® Hexagon™ LLVM C/C++ Compiler

User Guide

80-VB419-8984 Rev. B

July 9, 2020

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Figures

Tables

1 Introduction

This document is a reference for C/C++ programmers. It describes C and C++ compilers for the Qualcomm® Hexagon™ processor architecture. The compilers are based on the LLVM compiler framework and are collectively referred to as the *LLVM compilers*.

The LLVM compilers work with the Hexagon software development tools and utilities to provide a complete programming system for developing high-performance software.

The compilers run on the Windows and Linux platforms.

NOTE: The LLVM compilers are commonly referred to as *Clang*.

1.1 Conventions

`Courier` font is used for computer text and code samples, for example, `hexagon_<function_name>()`.

The following notation is used to define command syntax:

- Square brackets enclose optional items, for example, `[label]`.
- **Bold** indicates literal symbols for example, `[comment]`.
- The vertical bar character, `|`, indicates a choice of items.
- Parentheses enclose a choice of items for example, `(add|del)`.
- An ellipsis, `...`, follows items that can appear more than once.

For example:

```
#define name(parameter1[, parameter2...]) definition
logging (on|off)
```

Where:

- `#define` is a preprocessor directive
- *`name`* is the name of a defined symbol.
- *`parameter1`* and *`parameter2`* are macro parameters. The square brackets indicate that the second parameter is optional. The ellipsis indicates that the macro accepts more than parameters.
- `logging` is an interactive compiler command.
- **`on`** and **`off`** are bold to show that they are literal symbols. The vertical bar indicates that they are alternative parameters of the `logging` command.

1.2 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Functional overview

The C and C++ compilers are based on the LLVM compiler framework and are collectively referred to as the LLVM compilers.

2.1 Features

The LLVM compilers offer the following features:

- ISO C conformance

  Supports the International Standards Organization (ISO) C language standard

- Compatibility

  Supports Arm extensions and most GCC extensions to simplify porting

- System library

  Supports standard libraries for C and C++

- Processor-specific libraries

  Provides library routines that are optimized for the Hexagon architecture

- Intrinsics

  Provides a mechanism for emitting Hexagon assembly instructions in C source code

2.2 Languages

The LLVM compilers support C, C++, and many dialects of those languages:

- C language dialects:

  K&R C, ANSI C89, ISO C90, ISO C94 (C89+AMD1), ISO C99 (+TC1, TC2, TC3), ISO C11

- C++ language dialects:

  C++98, C++11, C++14, C++17

The LLVM compilers also support a broad variety of language extensions. These extensions are provided for compatibility with the GCC, Microsoft, and other popular compilers, as well as to improve functionality through the addition of extensions unique to the LLVM compilers.

All language extensions are explicitly recognized as such by the LLVM compilers. They are marked with extension diagnostics that can be mapped to warnings, errors, or simply ignored.

For compiling C sources, C11 is the default mode used by the LLVM compiler. The compiler does not support the C11 optional features of bounds checking interfaces and C11 threads. Consequently, as prescribed by the standard, the compiler does not define the `__STDC_LIB_EXT1__` macro, and it defines the `__STDC_NO_THREADS__macro`.

The following C++17 features are not supported by the compiler:

- *ISO/IEC TS 19570:2015 C++ Extensions for Parallelism* technical specification
- `constexpr std::hardware_constructive`
- `constexpr std::hardware_destructive`

2.3 GCC compatibility

The LLVM compiler driver and language features are intentionally designed to be as compatible with the GNU GCC compiler as reasonably possible, easing migration from GCC to LLVM. In most cases, code *just works*.

NOTE: The Hexagon LLVM compilers are fully compatible with the GNU toolchain.

2.4 Runtime systems

The Hexagon LLVM compilers support the development of both standalone programs and RTOS applications.

The build procedure for an RTOS application is a superset of the standard procedure for building a C/C++ application.

NOTE: For details, see the *Hexagon QuRT RTOS System Build Guide*.

2.5 Processor versions

The Hexagon LLVM compilers support all of the Hexagon processor versions listed in Table 2-1. However, not all processor versions are supported in all Hexagon tools releases. The table lists the versions supported in each release, and it provides the default version when a processor-version command option is not specified in the compiler. For more information on the Hexagon processor versions, see the appropriate *Qualcomm Hexagon Programmer's Reference Manual* (listed in Appendix B).

The compilers define command options (`-mv5`, `-mv66`, and so on) that specify the processor version for which they will generate output files. For more information on these and related command options, see Section 4.3.

Table 2-1 Supported processor versions

Tools release	Processor versions	Default version
6.2.x	V4, V5, V55, V56	V4
6.4.x		V55
7.2.x	V5, V55, V56, V60, V60 HVX, V61	V60
7.3.x		
7.4.x	V5, V55, V56, V60, V60 HVX, V61, V62, V62 HVX	
8.0.x	V5, V55, V56, V60, V60 HVX, V61, V62, V62 HVX	
8.1.x	V5, V55, V56, V60, V60 HVX, V61, V62, V62 HVX, V65, V65 HVX	
8.2.x	V5, V55, V56, V60, V60 HVX, V61, V62, V62 HVX, V65, V65 HVX, V66, V66 HVX	
8.3.x	V55, V56, V60, V60 HVX, V61, V62, V62 HVX, V65, V65 HVX, V66, V66 HVX, V67, V67 Small Core (V67t)	V60
8.4.x	V62, V62 HVX, V65, V65 HVX, V66, V66 HVX, V67, V67 Small Core (V67t), V68, V68 HVX	V68

NOTE: The LLVM compilers do not define command options specifically for V56, but they do support V56 if you use the V55 command options.

2.6 Processor-specific features

The Hexagon LLVM compilers include compiler options, language extensions, and libraries which are specific to the Hexagon processor.

2.7 C language and compiler references

This document does not describe the following:

- C or C++ languages
- Detailed descriptions of the code optimizations performed by LLVM

For suggested references, see Appendix B.

3 Get started

This chapter shows how to build and execute a simple C program using the LLVM compiler. The program is built using the Hexagon software development tools and executed on the Hexagon simulator.

The Hexagon software development tools are assumed to be already installed on your computer. The installation includes the tools required for assembling and linking a compiled program.

NOTE: The commands shown in this chapter are for illustration only. For detailed information on building programs, see Chapter 4.

3.1 Create source file

Create the following C source file:

```
#include <stdio.h>

int main()
{
     printf("Hello world\n");
     return(0);
}
```

Save the file as `hello.c`.

3.2 Compile program

Compile the program with the following command:

```
clang hello.c -o hello
```

This command translates the C source file `hello.c` into the executable file `hello`.

3.3 Execute program

To execute the program, use the following command:

```
hexagon-sim hello
```

The program outputs its message to the terminal:

```
Hello world
```

Following the program output is the simulator-generated message "`Done!`" and some execution statistics:

```
Done!
   T0: Insns=5648 Tcycles=7931
   T1: Insns=0 Tcycles=0
   T2: Insns=0 Tcycles=0
   Total: Insns=5648 Pcycles=47587
   Simulator speed=0.314599 Mips
Ratio to Real Time (600 MHz) = ~1/226 (elapsed time = 0.017953s)
```

The statistics show the number of instructions and cycles executed on each Hexagon processor thread.

You have now compiled and executed a C program using the LLVM compiler. For more information on using the compiler, see Chapter 4.

4 Use the compilers

The LLVM compilers translate C and C++ programs into Hexagon processor code.

C and C++ programs are stored in source files, which are text files created with a text editor. Hexagon processor code is stored in object files, which are executable binary files.

4.1 Start the compilers

To start the C compiler from the command line, enter:

```
clang [options...] input_files...
```

To start the C++ compiler from the command line, enter:

```
clang++ [options...] input_files...
```

The compilers accept one or more input files on the command line. Input files can be C/C++ source files or object files. For example:

```
clang hello.c mylib.c
```

Command switches are used to control various compiler options (Section 4.3). A switch consists of a dash character (`-`) followed by a switch name and optional parameter.

Switches are case-sensitive and must be separated by at least one space. For example:

```
clang hello.c -o hello
```

To list the available command options, use the `--help` option:

```
clang --help
clang++ --help
```

This option causes the compiler to display the command line syntax, followed by a list of the available command options. `clang` is the name of the front-end driver for the LLVM compiler framework.

4.2 Input and output files

The LLVM compilers preprocess and compile one or more source files into object files. The compilers then invoke the linker to combine the object files into an executable file.

Following are the input file types and the tool that processes files of each type. The compilers use the file name extension to determine how to process the file.

Table 4-1 Compiler input files

Extension	Description	Tool
.c	C source file	C compiler
.i	C preprocessed file	
.h	C header file	
.cc .cp .cxx .cpp .CPP .c++ .C	C++ source file	C++ compiler
.ii	C++ preprocessed file	
.h .hh .H	C++ header file	
.s .S	Assembly source file	Assembler
Other	Binary object file	Linker

All file name extensions are case-sensitive literal strings. Input files with unrecognized extensions are treated as object files. For more information on assembly and object files, see the *Qualcomm Hexagon Utilities User Guide* (80-N2040-15).

Following are the output file types and the tools used to generate each file type. Compiler options (Section 4.3) are used to specify the output file type.

Table 4-2 Compiler output files

File type	Default file name	Input files
Executable file	`a.out`	The specified source files are compiled and linked to a single executable file.
Object file	`file.o`	Each specified source file is compiled to a separate object file (where `file` is the source file name).
Assembly source file	`file.s`	Each specified source file is compiled to a separate assembly source file (where `file` is the source file name).
Preprocessed C/C++ source file	`stdout`	The preprocessor output is written to the standard output.

4.3 Compiler options

The LLVM compilers can be controlled by command-line options (Section 4.1). Many of the GCC options are supported, along with options that are LLVM-specific.

Many of the `-f`, `-m`, and `-W` options can be written in two ways: `-f<option>` to enable a binary option, or `-fno-<option>` to disable the option.

`-mllvm` is not a standalone option, but rather a standard prefix that appears in many LLVM-specific option names.

Following is a quick reference of the options.

Display (see Section 4.3.1)

```
-help
-v
```

Compilation (see Section 4.3.2)

```
-###
-c
-cc1
-ccc-print-phases
-E
-pipe
-o file
-S
-Wa,arg[,arg...]
-Wl,arg[,arg...]
-Wp,arg[,arg...]
-x language
-Xclang arg
```

C dialect (See Section 4.3.3)

```
-ansi
-fblocks
-fgnu-runtime
-fgnu89-inline
-fno-asm
-fsigned-bitfields
-fsigned-char
-funsigned-char
-no-integrated-cpp
-std=(c89|gnu89|c99|gnu99|c11|gnu11)
-traditional
-Wpointer-sign
```

C++ dialect (see Section 4.3.4)

```
-cxx-isystem dir
-ffor-scope  |  -fno-for-scope
-fno-gnu-keywords
-ftemplate-depth-n
```

```
-fvisibility-inlines-hidden
-fuse-cxa-atexit
-nobuiltininc
-nostdinc++
-std=(c++98|gnu++98|c++11|gnu++11|c++14|gnu++14|c++17|gnu++17)
-Wc++0x-compat
-Wno-deprecated
-Wnon-virtual-dtor
-Woverloaded-virtual
-Wreorder
```

Warning and error messages (see Section 4.3.5)

```
c++98|c++11|c++14|c++17-compat-pedantic
cast-calling-convention
-ccc-print-diag-group name | -ccc-print-diag-group-verbose name
comma
constant-conversion
expansion-to-defined
-ferror-limit=n
-ferror-warn filename
float-overflow-conversion
float-zero-conversion
-fscanban-file file
-ftemplate-backtrace-limit=n
-fsyntax-only
incompatible-sysroot
nonportable-include-path
nonportable-system-include-path
null-dereference
openmp-target
-pedantic | -Wpedantic
pedantic-core-features
-pedantic-errors
-Qunused-arguments
shadow-field-in-constructor-modified
shadow-field-in-constructor
undefined-func-template
undefined-var-template
unguarded-availability
unknown-argument
unsupported-cb
varargs
-w
-Wfoo
-Wno-foo
-Wall
-Warray-bounds
-Wcast-align
-Wchar-subscripts
-Wcomment
-Wconversion
-Wdeclaration-after-statement
-Wno-deprecated-declarations
-Wempty-body
-Wendif-labels
```

```
-Werror
-Werror=foo
-Wno-error=foo
-Werror-implicit-function-declaration
-Weverything
-Wextra
-Wfloat-equal
-Wformat
-Wformat=2
-Wformat-nonliteral
-Wformat-security
-Wignored-qualifiers
-Wimplicit
-Wimplicit-function-declaration
-Wimplicit-int
-Wno-format-extra-args
-Wno-invalid-offsetof
-Wlong-long
-Wmain
-Wmissing-braces
-Wmissing-declarations
-Wmissing-noreturn
-Wmissing-prototypes
-Wno-multichar
-Wnonnull
-Wpacked
-Wpadded
-Wparentheses
-Wpedantic
-Wpointer-arith
-Wreturn-type
-Wscanban
-Wshadow
-Wsign-compare
-Wswitch  |  -Wswitch-enum
-Wsystem-headers
-Wtautological-constant-out-of-range-compare
-Wtrigraphs
-Wundef
-Wuninitialized
-Wunknown-pragmas
-Wunreachable-code
-Wunused
-Wunused-function
-Wunused-label
-Wunused-parameter
-Wunused-value
-Wunused-variable
-Wwrite-strings
```

Debugging (see Section 4.3.6)

```
-dumpmachine
-dumpversion
-feliminate-unused-debug-symbols
-time  |  -ftime-report
```

```
-g[level]
-gline-tables-only
-print-diagnostic-categories
-print-file-name=library
-print-libgcc-file-name
-print-multi-directory
-print-multi-lib
-print-multi-os-directory
-print-prog-name=program
-print-search-dirs
-save-temps
```

Diagnostic format (see Section 4.3.7)

```
-fcaret-diagnostics  |  -fno-caret-diagnostics
-fdiagnostics-show-option  |  -fno-diagnostics-show-option
-fdiagnostics-show-category=(none|id|name)
-fdiagnostics-print-source-range-info
-fno-diagnostics-print-source-range-info
-fdiagnostics-parseable-fixits
-fdiagnostics-show-template-tree
-fmessage-length=n
```

Individual warning groups (see Section 4.3.8)

```
-Wextra-tokens
-Wambiguous-member-template
```

Compiler crash diagnostics (see Section 4.3.9)

```
-fno-crash-diagnostics
```

Preprocessor (see Section 4.3.10)

```
-A pred=ans
-A -pred=ans
-ansi
-C
-CC
-d(DMNU)
-D name  |  -D name=definition
-fexec-charset=charset
-finput-charset=charset
-fpch-deps
-fpreprocessed
-fstrict-overflow
-ftabstop=width
-fwide-exec-charset=charset
-fworking-directory
-H
--help
-I dir
-I-
-include file
-isystem prefix
```

```
-isystem-prefix prefix
-ino-system-prefix prefix
-M
-MD
-MF file
-MG
-MM
-MMD
-MP
-MQ target
-MT target
-nostdinc
-nostdinc++
-o file
-P
-remap
--target-help
-traditional-cpp
-trigraphs
-U name
-v
-version
--version
-w
-Wall
-Wcomment | -Wcomments
-Wendif-labels
-Werror
-Wimport
-Wsystem-headers
-Wtrigraphs
-Wundef
-Wunused-macros
```

Assembling (see Section 4.3.11)

```
-fintegrated-as
--fno-integrated-as
-Xassembler arg
```

Linking (see Section 4.3.12)

```
object_file_name
-c
-dynamic
-E
-l library
-moslib=library
-nodefaultlibs
-nostartfiles
-nostdlib
-pie
-s
-S
-shared
-shared-libgcc
```

```
-static
-static-libgcc
-symbolic
-u symbol
-Xlinker arg
```

Directory search (see Section 4.3.13)

```
-Bprefix
-F dir
-I dir
-I-
-Ldir
```

Processor version (see Section 4.3.14)

```
-mv62 | -mv65 | -mv66 | -mv67 | -mv67t | -mv68
-mhvx
-mhvx=version
-mno-hvx
```

Code generation (see Section 4.3.15)

```
-femit-all-data
-femit-all-decls
-ffixed-rnn
-fmerge-functions | -fno-merge-functions
-fno-exceptions
-fpic
-fPIC
-fpie | -fPIE
-fshort-enums | -fno-short-enums
-fshort-wchar | -fno-short-wchar
-ftrap-function=name
-ftrapv
-ftrapv-handler=name
-funwind-tables
-fverbose-asm
-fvisibility=[default|internal|hidden|protected]
-fwrapv
-G size
-mno-global-merge
-mno-packets
-mlong-calls
-mhvx-length=[64B,128B]
```

Optimization (see Section 4.3.16)

```
-O0
-O | -O1
-O2
-O3
-Os
```

Specific optimizations (see Section 4.3.17)

```
-fdata-sections
-ffast-math
-ffp-contract=[off|on|fast]
-ffunction-sections
-finline
-finline-functions
-fno-zero-initialized-in-bss
-fnomerge-all-constants
-fomit-frame-pointer
-foptimize-sibling-calls
-fstack-protector
-fstack-protector-all
-fstack-protector-strong
-fstrict-aliasing
-funit-at-a-time
-funroll-all-loops
-funroll-loops
-fvectorize
--param ssp-buffer-size=size
```

Link-time optimization (compiler and linker) (see Section 4.3.18)

```
-flto  |  -flto=thin
-flto-options=<>
-flto-options=codegen="list of options"
-flto-options=cache=path
-flto-options=nooptimal-preservelist
-flto-options=preserveall
-flto-options=preserve-file=file
-flto-options=preserve-sym=symbol
-flto-options=save-lto-output-file=filename
-flto-options=threads=n
-flto-use-as
-save-temps
-trace=lto
```

Compiler security (see Section 4.3.19)

```
--analyze
-analyzer-checker=checker
-analyzer-checker-help
-analyzer-disable-checker=checker
--analyzer-output html
--analyzer-Werror
--compile-and-analyze dir
--compile-and-analyze-high
--compile-and-analyze-medium
```

4.3.1 Display

-help

Displays compiler command and option summary.

-v

Displays compiler release version.

4.3.2 Compilation

-###

Prints commands used to perform the compilation.

-c

Compiles the source file, but does not link it.

-cc1

Bypasses the compiler driver and go directly to LLVM.

-ccc-print-phases

Prints the compilation stages as they occur.

-E

Preprocesses the source file only; does not compile it.

-pipe

Communicates between compiler stages using pipes not temporary files.

-o *file*

Specify the name of the compiler output file.

-S

Compiles the source file, but does not assemble it.

-Wa,*arg*[**,***arg*...]

Passes the specified arguments to the assembler.

-Wl,*arg*[**,***arg*...]

Passes the specified arguments to the linker.

-Wp,*arg*[**,***arg*...]

Passes the specified arguments to the preprocessor.

-x *language*

Specifies the language of the subsequent source files specified on the command line.

-Xclang *arg*

Passes the specified argument to the compiler.

4.3.3 C dialect

-ansi

For C, supports ISO C90.

For C++, removes conflicting GNU extensions.

-fblocks

Enables the Apple block*s* extension.

-fgnu-runtime

Generates output compatible with the standard GNU Objective-C runtime.

-fgnu89-inline

Uses the gnu89 inline semantics.

-fno-asm

Does not recognize `asm`, `inline`, or `typeof` as keywords.

-fsigned-bitfields

Defines bit fields as signed.

-fsigned-char

Defines the char type as signed.

-funsigned-char

Defines the char type as unsigned.

-no-integrated-cpp

Compiles using separate preprocessing and compilation stages.

-std=(**c89**|**gnu89**|**c99**|**gnu99**|**c11**|**gnu11**)

LLVM C language mode. The default setting is `gnu11`.

-traditional

Supports pre-standard C language.

-Wpointer-sign

Flag pointers when assigned or passed values with a differing sign.

4.3.4 C++ dialect

-cxx-isystem *dir*

Adds a specified directory to C++ SYSTEM include search path.

-ffor-scope
-fno-for-scope

Control whether the scope of a variable declared in a `for` statement is limited to the statement or to the scope enclosing the statement.

-fno-gnu-keywords

Disables recognizing `typeof` as a keyword.

-ftemplate-depth-n

Specifies the maximum instantiation depth of a template class.

-fvisibility-inlines-hidden

Specifies the default visibility for inline C++ member functions.

-fuse-cxa-atexit

Registers destructors with function `__cxa_atexit` (instead of `atexit`). This option applies only to objects that have static storage duration.

-nobuiltininc

Disables built-in `#include` directories.

-nostdinc++

Disables standard `#include` directories for the C++ standard library.

-std=(c++98|**gnu++98**|**c++11**|**gnu++11**|**c++14**|**gnu++14**|**c++17**|**gnu++17)**

LLVM C++ language mode. The default setting is `gnu++14`.

-Wc++0x-compat

Generates warnings for C++ constructs with different semantics in ISO C++ 1998 and ISO C++ 200x.

-Wno-deprecated

Does not generate warnings when deprecated features are used.

-Wnon-virtual-dtor

Generates a warning when a polymorphic class is declared with a non-virtual destructor.

-Woverloaded-virtual

Generates a warning when a function hides virtual functions from a base class.

-Wreorder

Generates a warning when member initializers do not appear in the code in the required execution order.

4.3.5 Warning and error messages

c++98|c++11|c++14|c++17-compat-pedantic

Warn if the compiled code uses experimental features added to language standards more recent than C++17.

cast-calling-convention

Cast between incompatible %0 and %1 calling conventions. Calls through this pointer might abort at runtime.

-ccc-print-diag-group *name*
-ccc-print-diag-group-verbose *name*

Display the diagnostics assigned to the diagnostic group *name*.

These options are used with `-W`*group* to control sets of diagnostics.

comma

Possible misuse of comma operator here.

constant-conversion

Implicit conversion from %2 to %3 changes the value from %0 to %1.

expansion-to-defined

Macro expansion producing defined has undefined behavior.

-ferror-limit=*n*

Stops emitting diagnostics after *n* errors have been produced. The default setting is `20`. The error limit can be disabled with `-ferror-limit=0`.

-ferror-warn *filename*

Converts the specified set of compiler warnings into errors.

The specified text file contains a list of warning names, with each warning name separated by whitespace in the file.

Warning names are based on the switch names of the corresponding compiler warning-message options. For example, to convert the warnings generated by `-Wunused-variable`, use the warning name, `unused-variable`.

This option can be specified multiple times.

NOTE: This option (and its associated file) can be integrated into a build system, and used to iteratively resolve the warning messages generated by a project.

float-overflow-conversion

Implicit conversion of out of range value from %0 to %1 changes the value from %2 to %3.

float-zero-conversion

Implicit conversion from %0 to %1 changes the non-zero value from %2 to %3.

-fscanban-file *file*

Includes functions listed in the text file into the banned functions list. The entries in the additional banned functions file are to be added on a line-by-line basis in the following format:

<banned function name>,<alternative function to be used in its place>

An alternative must be added when banning a function.

-ftemplate-backtrace-limit=*n*

Only emits up to *n* template instantiation notes within the template instantiation backtrace for a single warning or error. The default setting is `10`. The limit can be disabled with `-ftemplate-backtrace-limit=0`.

-fsyntax-only

Checks for syntax errors only.

incompatible-sysroot

Uses sysroot for %0 but targets %1.

nonportable-include-path

Non-portable path to file %0. The specified path differs in case from the file name on the disk.

nonportable-system-include-path

Non-portable path to file %0. The specified path differs in case from the file name on the disk.

null-dereference

Binding dereferenced NULL pointer to reference has undefined behavior.

openmp-target

Declaration is not declared in any declare target region.

-pedantic
-Wpedantic

Generate all warnings required by the ISO C and ISO C++ standards.

pedantic-core-features

OpenCL extension %0 is a core feature or a supported optional core feature; ignoring.

-pedantic-errors

Equivalent to `-pedantic`, but generate errors instead of warnings.

-Qunused-arguments

Does not generate warnings for unused driver arguments.

shadow-field-in-constructor-modified

Modifying constructor parameter %0 that shadows a field of %1.

shadow-field-in-constructor

Constructor parameter %0 shadows the field %1 of %2.

undefined-func-template

Instantiation of function %q0 required here, but no definition is available.

undefined-var-template

Instantiation of variable %q0 is required here, but no definition is available.

unguarded-availability

Using * case here; platform %0 is not accounted for.

unknown-argument

Unknown argument is ignored in clang-cl: %0.

unsupported-cb

Ignoring `-mcompact-branches=` option because the %0 architecture does not support it.

varargs

Passing an object that undergoes a default argument promotion to `va_start` has undefined behavior.

-w

Suppresses all warnings.

-W*foo*

Enables the diagnostic (or diagnostic group) *`foo`*.

-Wno-*foo*

Disables the diagnostic (or diagnostic group) *`foo`*.

-Wall

Enables all `-W` options.

-Warray-bounds

Generates a warning if array subscripts are out of bounds.

-Wcast-align

Generates a warning if a pointer cast increases the required alignment of the target.

-Wchar-subscripts

Generates a warning if array subscript is type `char`.

-Wcomment

Generates a warning if a comment symbol appears inside a comment.

-Wconversion

Generates a warning if an implicit conversion might alter a value.

-Wdeclaration-after-statement

Generates a warning when a declaration appears in a block after a statement.

-Wno-deprecated-declarations

Does not generate warnings for functions, variables, or types assigned the attribute `deprecated`.

-Wempty-body

Generates a warning if an `if`, `else`, or `do while` statement contains an empty body.

-Wendif-labels

Generates a warning if an `#else` or `#endif` directive is followed by text.

-Werror

Converts all warnings into errors.

-Werror=*foo*

Converts the diagnostic (or diagnostic group) *foo* into an error.

-Wno-error=*foo*

Keeps the diagnostic (or diagnostic group) *foo* as a warning, even if `-Werror` is used.

-Werror-implicit-function-declaration

Generates a warning or error if a function is used before being declared.

-Weverything

Enables all warnings.

-Wextra

Enables selected warning options, and generate warnings for selected events.

-Wfloat-equal

Generates a warning if two floating point values are compared for equality.

-Wformat

In calls to `printf`, `scanf` and other functions with format strings, ensures that the arguments are compatible with the specified format string.

-Wformat=2

This option is equivalent to specifying the following options: `-Wformat -Wformat-nonliteral -Wformat-security -Wformat-y2k`.

-Wformat-nonliteral

Generates a warning if the format string is not a string literal, except if the format arguments are passed through `va_list`.

-Wformat-security

Generates a warning for format function calls that might cause security risks.

-Wignored-qualifiers

Generates a warning if a return type has a qualifier (such as, `const`).

-Wimplicit

Equivalent to `-Wimplicit-int` and `-Wimplicit-function-declaration`.

-Wimplicit-function-declaration

Generates a warning if a function is used before it is declared.

-Wimplicit-int

Generates a warning if a declaration does not specify a type.

-Wno-format-extra-args

Does not generate a warning for passing extra arguments to `printf` or `scanf`.

-Wno-invalid-offsetof

Does not generate a warning if a non-POD type is passed to the `offsetof` macro.

-Wlong-long

Generates a warning if type`long long` is used.

-Wmain

Generates a warning if the `main()` function has any suspicious properties.

-Wmissing-braces

Generates a warning if an aggregate or union initializer is not properly bracketed.

-Wmissing-declarations

Generates a warning if a global function is defined without being first declared.

-Wmissing-noreturn

Generates a warning if a function does not include a `return` statement.

-Wmissing-prototypes

Generates a warning if a global function is defined without a prototype.

-Wno-multichar

Does not generate a warning if a multi-character constant is used.

-Wnonnull

Generates a warning if a NULL pointer is passed to an argument that is specified to require a non-NULL value (with the `nonnull` attribute).

-Wpacked

Generates a warning if the memory layout of a structure is not affected after the structure is specified with the `packed` attribute.

-Wpadded

Generates a warning if the memory layout of a structure includes padding.

-Wparentheses

Generates a warning if the parentheses are omitted in certain cases.

-Wpedantic

See `-pedantic`.

-Wpointer-arith

Generates a warning if any code depends on the size of `void` or a function type.

-Wreturn-type

Generates a warning if a function returns a type that defaults to `int`, or a value is incompatible with the defined return type.

-Wscanban

Generates a warning if a banned function is being used.

The default list of banned functions adheres to the QC-ScanBan list. Additional banned functions can be added through the `-fscanban-file` flag.

-Wshadow

Generates a warning if a local variable shadows another local variable, global variable, or parameter; or if a built-in function gets shadowed.

-Wsign-compare

Generates a warning in a signed/unsigned compare operation if the result might be inaccurate due to the signed operand being converted to unsigned.

-Wswitch
-Wswitch-enum

Generates a warning if a `switch` statement uses an enum type for the index, and does not specify a `case` for every possible enumeration value, or specifies a case with a value outside the enumeration range.

-Wsystem-headers

Generates a warning for constructs declared in system header files.

-Wtautological-constant-out-of-range-compare

Generate warning if a compare is performed using an operand type which by definition always yields a TRUE (or FALSE) compare result. For example, `i >= 0`, where `i` is an unsigned integer type.

-Wtrigraphs

Generates a warning if a trigraph forms an escaped newline in a comment.

-Wundef

Generates a warning if an undefined non-macro identifier appears in an `#if` directive.

-Wuninitialized

Generates a warning if referencing an uninitialized automatic variable.

-Wunknown-pragmas

Generates a warning if a `#pragma` directive is not recognized by the compiler.

-Wunreachable-code

Generates a warning if code will never be executed.

-Wunused

Specifies all of the `-Wunused` options.

-Wunused-function

Generates a warning if a static function is declared without being defined or used.

NOTE: No warning is generated for functions declared or defined in header files.

-Wunused-label

Generates a warning if a label is declared without being used.

-Wunused-parameter

Generates a warning if a function argument is not used in its function.

-Wunused-value

Generates a warning if the value of a statement is not subsequently used.

-Wunused-variable

Generates a warning if a local or non-constant static variable is not used in its function.

-Wwrite-strings

For C, assigns string constants the type `const char[`*`length`*`]` to ensure that a warning is generated if the string address gets copied to a non-`const char *` pointer.

For C++, generates a warning a string constant is being converted to `char *`.

4.3.6 Debugging

-dumpmachine

Displays the target machine name.

-dumpversion

Displays the compiler version.

-feliminate-unused-debug-symbols

Generates debug information only for the symbols that are used. (Debug information is generated in STABS format.)

-time
-ftime-report

Displays the elapsed time for each stage of the compilation.

-g[*level*]

Generates complete source-level debug information.

-gline-tables-only

Generates source-level debug information with line number tables only.

-print-diagnostic-categories

Displays mapping of diagnostic category names to category identifiers.

-print-file-name=*library*

Displays the full library path of the specified file.

-print-libgcc-file-name

Displays the library path for the `libgcc.a` file.

-print-multi-directory

Displays the directory names of the multi libraries specified by other compiler options in the current compilation.

-print-multi-lib

Displays the directory names of the multi libraries paired with the compiler options that specified the libraries in the current compilation.

-print-multi-os-directory

Displays the relative path that is appended to the multi-lib search paths.

-print-prog-name=*program*

Displays the absolute path of the specified program.

-print-search-dirs

Displays the search paths used to locate libraries and programs during compilation.

-save-temps

Saves the normally-temporary intermediate files generated during compilation.

4.3.7 Diagnostic format

The LLVM compilers aim to produce beautiful diagnostics by default, especially for new users just beginning to use LLVM. However, different users have different preferences, and sometimes LLVM is driven by another program that needs the diagnostic output to be simple and consistent rather than user-friendly. For these cases, LLVM provides a wide range of options to control the output format of the diagnostics that it generates.

`-fcaret-diagnostics`
`-fno-caret-diagnostics`

Print source line and ranges from source code in diagnostic format.

This option controls whether LLVM prints the source line, source ranges, and caret when emitting a diagnostic. The default setting is `enabled`. When enabled, LLVM prints information like the following example:

```
test.c:28:8: warning: extra tokens at end of #endif directive
[-Wextra-tokens]
#endif bad
       ^
       //
-fdiagnostics-format=(clang|msvc|vi)
```

Changes diagnostic output format to better match IDEs and command line tools.

This option controls the output format of the filename, line number, and column printed in diagnostic messages. The default setting is `clang`. The effect of the setting the output format is shown in the following examples.

- `clang`

  ```
  t.c:3:11: warning: conversion specifies type 'char *' but the
  argument has type 'int'
  ```

- `msvc`

  ```
  t.c(3,11) : warning: conversion specifies type 'char *' but the
  argument has type 'int'
  ```

- `vi`

  ```
  t.c +3:11: warning: conversion specifies type 'char *' but the
  argument has type 'int'
  ```

`-fdiagnostics-show-option`
`-fno-diagnostics-show-option`

Enable `[-W`*`option`*`]` information in diagnostic line.

This option controls whether LLVM prints the associated warning group option name (Section 4.3.23) when outputting a warning diagnostic. The default setting is `disabled`. For example, given the following diagnostic output:

```
test.c:28:8: warning: extra tokens at end of #endif directive
[-Wextra-tokens]
#endif bad
       ^
       //
```

In this case, specifying `-fno-diagnostics-show-option` prevents LLVM from printing the [`-Wextra-tokens`] information in the diagnostic output. This information indicates the option required to enable or disable the diagnostic, either from the command line or by using the `GCC diagnostic` pragma (Section 4.3.25).

`-fdiagnostics-show-category=(none|id|name)`

Enables printing category information in diagnostic line.

This option controls whether LLVM prints the category associated with a diagnostic when emitting it. The default setting is `none`. The effect of the setting the output format is shown in the following examples.

- `none`

  ```
  t.c:3:11: warning: conversion specifies type 'char *' but the
  argument has type 'int' [-Wformat]
  ```

- `id`

  ```
  t.c:3:11: warning: conversion specifies type 'char *' but the
  argument has type 'int' [-Wformat,1]
  ```

- `name`

  ```
  t.c:3:11: warning: conversion specifies type 'char *' but the
  argument has type 'int' [-Wformat,Format String]
  ```

Each diagnostic can have an associated category. If it has one, it is listed in the diagnostic category field of the diagnostic line (in the `[]` brackets).

This option can be used to group diagnostics by category, so it should be a high-level category; the goal is to have dozens of categories, not hundreds or thousands of them.

`-fdiagnostics-print-source-range-info`
`-fno-diagnostics-print-source-range-info`

Print machine-parseable information about source ranges.

This option controls whether LLVM prints information about source ranges in a machine-parseable format after the file/line/column number information. The default setting is `disabled`. The information is a simple sequence of brace-enclosed ranges, where each range lists the start and end line/column locations.

For example, given the following output:

```
exprs.c:47:15:{47:8-47:14}{47:17-47:24}: error: invalid operands
to binary expression ('int *' and '_Complex float')
P = (P-42) + Gamma*4;
    ~~~~~~ ^ ~~~~~~~
```

In this case, the braces, {}, are generated by `-fdiagnostics-print-source-range-info`.

The printed column numbers count bytes from the beginning of the line; take care if the source contains multi-byte characters.

-fdiagnostics-parseable-fixits

Prints Fix-Its in a machine-parseable format.

This option makes LLVM print available Fix-Its in a machine-parseable format at the end of diagnostics. The following example illustrates the format:

```
fix-it:"t.cpp":{7:25-7:29}:"Gamma"
```

In this case, the range printed is half-open, so the characters from column 25 up to (but not including) column 29 on line 7 of file `t.cpp` should be replaced with the string `Gamma`. Either the range or replacement string can be empty (representing strict insertions and strict erasures, respectively). Both the file name and insertion string escape backslash (as `\\`), tabs (as `\t`), newlines (as `\n`), double quotes (as `\"`), and non-printable characters (as octal `\xxx`).

The printed column numbers count bytes from the beginning of the line; take care if the source contains multi-byte characters.

-fdiagnostics-show-template-tree

For large templated types, this option causes LLVM to display the templates as an indented text tree, with one argument per line, and any differences marked inline.

- `default`

```
t.cc:4:5: note: candidate function not viable: no known conversion
from 'vector<map<[...], map<float, [...]>>>' to 'vector<map<[...],
map<double, [...]>>>' for 1st argument;
```

- `-fdiagnostics-show-template-tree`

```
t.cc:4:5: note: candidate function not viable: no known conversion
for 1st argument;
  vector<
    map<
      [...],
      map<
        [float != float],
        [...]>>>
```

-fmessage-length=*n*

Formats error messages to fit on lines with the specified number of characters.

4.3.8 Individual warning groups

-Wextra-tokens

Warns about excess tokens at the end of a preprocessor directive.

This option enables warnings about extra tokens at the end of preprocessor directives. The default setting is `enabled`. For example:

```
test.c:28:8: warning: extra tokens at end of #endif directive
[-Wextra-tokens]
#endif bad
       ^
```

These extra tokens are not strictly conforming, and are usually best handled by commenting them out.

-Wambiguous-member-template

Warns about unqualified uses of a member template whose name resolves to another template at the location of the use.

This option (which is enabled by default) generates a warning in the following code:

```
template<typename T> struct set{};
template<typename T> struct trait { typedef const T& type; };
struct Value {
  template<typename T> void set(typename trait<T>::type value){}
};

void foo() {
  Value v;
  v.set<double>(3.2);
}
```

C++ requires this to be an error, but because it is difficult to work around, LLVM downgrades it to a warning as an extension.

4.3.9 Compiler crash diagnostics

The LLVM compilers might crash once in a while. Generally, this only occurs when using the latest versions of LLVM.

LLVM goes to great lengths to provide assistance in filing a bug report. Specifically, after a crash, it generates preprocessed source files and associated run scripts. Attach these files to a bug report to ease reproducibility of the failure. The following compiler option is used to control the crash diagnostics.

-fno-crash-diagnostics

Disables auto-generation of preprocessed source files during an LLVM crash.

This option can be helpful for speeding up the process of generating a delta reduced test case.

4.3.10 Preprocessor

-A pred=*ans*

Asserts the predicate *pred* and answer *ans*.

-A -pred=*ans*

Cancels the specified assertion.

-ansi

Uses C89 standard.

-C

Retains comments during preprocessing.

-CC

Retains comments during preprocessing, including during macro expansion.

-d(**DMNU**)

Where:

- `D` Prints macro definitions in `-E` mode in addition to normal output.
- `M` Prints macro definitions in `-E` mode instead of normal output.
- `N` Prints macro names in `-E` mode in addition to normal output.
- `U` Prints referenced macro definitions in `-E` mode in addition to normal output. Also prints `#undef`s for macros that are undefined when referenced. Both are printed at the point they are referenced.

-D *name*
-D name=*definition*

Defines the specified macro symbol.

-fexec-charset=charset

Specifies the character set used to encode strings and character constants. The default character set is `UTF-8`.

-finput-charset=charset

Specifies the character set used to encode the input files. The default setting is `UTF-8`.

-fpch-deps

Causes the dependency-output options to additionally list the files from a precompiled header's dependencies.

-fpreprocessed

Notifies the preprocessor that the input file has already been preprocessed.

-fstrict-overflow

Enforces strict language semantics for pointer arithmetic and signed overflow.

-ftabstop=*width*

Specifies the tab stop distance.

-fwide-exec-charset=charset

Specifies the character set used to encode wide strings and character constants. The default character set is UTF-32 or UTF-16, depending on the size of `wchar_t`.

-fworking-directory

Generates line markers in the preprocessor output. The compiler uses this to determine what the current working directory was during preprocessing.

-H

Displays the header includes and nesting depth.

--help

Displays the preprocessor release version.

-I *dir*

Adds the specified directory to the list of search directories for header files.

-I-

This option is deprecated.

-include *file*

Includes the contents of the specified source file.

-isystem *prefix*

Treats an included file as a system header if it is found on the specified path (Section 4.3.26).

-isystem-prefix *prefix*

Treats an included file as a system header if it is found on the specified subpath of a defined include path (Section 4.3.26).

-ino-system-prefix *prefix*

Does not treat an included file as a system header if it is found on the specified sub-path of a defined include path (Section 4.3.26).

-M

Outputs a `make` rule describing the dependencies of the main source file.

-MD

Equivalent to `-M -MF` *file*, except `-E` is not implied.

-MF *file*

Writes dependencies to the specified file.

-MG

Adds missing headers to the dependency list.

-MM

Equivalent to `-M`, except this operation does not mention header files found in the system header directories.

-MMD

Equivalent to `-MD`, except this option only mention user header files, not system header files.

-MP

Creates artificial target for each dependency.

-MQ *target*

Specify a target to quote for dependency.

-MT *target*

Specify target for dependency.

-nostdinc

Omits searching for header files in the standard system directories.

-nostdinc++

Omits searching for header files in the C++-specific standard directories.

-o *file*

Specifies the name of the preprocessor output file.

-P

Disables line marker output when using `-E`.

-remap

Generates code for file systems that only support short file names.

--target-help

Displays all command options and exit immediately.

-traditional-cpp

Emulates pre-standard C preprocessors.

-trigraphs

Preprocesses trigraphs.

-U *name*

Cancels any previous definition of the specified macro symbol.

-v

Equivalent to `-help`.

-version

Displays the preprocessor version during preprocessing.

--version

Displays the preprocessor version and exit immediately.

-w

Suppresses all preprocessor warnings.

-Wall

Enables all warnings.

-Wcomment
-Wcomments

Generates a warning if a comment symbol appears inside a comment.

-Wendif-labels

Generates a warning if an `#else` or `#endif` directive is followed by text.

-Werror

Converts all warnings into errors.

-Wimport

Generates a warning when `#import` is used the first time.

-Wsystem-headers

Generates a warning for constructs declared in system header files.

-Wtrigraphs

Generates a warning if a trigraph forms an escaped newline in a comment.

-Wundef

Generates a warning if an undefined non-macro identifier appears in an `#if` directive.

-Wunused-macros

Generates a warning if a macro is defined without being used.

4.3.11 Assembling

-fintegrated-as

Enables the integrated assembler.

--fno-integrated-as

Disables the integrated assembler.

NOTE: The `-Wa` switches (Section 4.3.2) are accepted only if the integrated assembler is disabled with `--fno-integrated-as`.

-Xassembler *arg*

Passes the specified argument to the assembler.

4.3.12 Linking

object_file_name

Linker input file.

-c

Does not perform linking. This option is used with spec strings.

-dynamic

Links with a shared library (instead of a static library).

-E

Does not perform linking. This option is used with spec strings.

-l *library*

Searches the specified library file while linking.

-moslib=*library*

Searches the RTOS-specific library named `liblibrary.a`. The search paths for the library and include files must be explicitly specified.

-nodefaultlibs

Does not use the standard system libraries when linking.

-nostartfiles

Does not use the standard system startup files when linking.

-nostdlib

Does not use the standard system startup files or libraries when linking.

-pie

Generates a position-independent executable as the output file.

-s

Deletes all symbol table information and relocation information from the executable.

-S

Does not perform linking. This option is used with spec strings.

-shared

Generates a shared object as the output file. The resulting file can be subsequently linked with other object files to create an executable.

-shared-libgcc

Links with the shared version of the `libgcc` library.

-static

Does not link with the shared libraries. Only relevant when using dynamic libraries.

-static-libgcc

Links with the static version of the `libgcc` library.

-symbolic

Binds references to global symbols when building a shared object.

-u *symbol*

Pretends the *symbol* is undefined to force linking of library modules to define *symbol*.

-Xlinker *arg*

Passes the specified argument to the linker.

4.3.13 Directory search

-B*prefix*

Specifies the top-level directory of the compiler.

-F *dir*

Adds the specified directory to the search path for framework includes.

-I *dir*

Adds the specified directory to the include file search path.

-I-

This option is deprecated.

-L*dir*

Adds the specified directory to the list of directories searched by the `-l` option.

4.3.14 Processor version

-mv62
-mv65
-mv66
-mv67
-mv67t
-mv68

Specify the Hexagon processor version for which the compiler will generate code: V62, V65, V66, V67, V67 Small Core, V68. For more information on the processor versions, see the appropriate *Qualcomm Hexagon Programmer's Reference Manual* (listed in Appendix B).

NOTE: Not all `-mvX` options are supported in all Hexagon tools releases. The default option setting is also release-specific. For more information, see Section 2.5.

-mhvx

Enables HVX.

The default Hexagon HVX coprocessor version is the Hexagon processor version. If no `-mhvx-length` flag is specified, the vector length is based on the Hexagon processor version:

- For V62 and V65, the default vector length is 64 bytes.
- For V66 and later, the default vector length is 128 bytes.

-mhvx=version

Specifies the Hexagon HVX coprocessor version (v62, v66, v66, and so on) for which the compiler will generate code.

If no `-mhvx-length` flag is specified, the mode selected is based on the Hexagon HVX coprocessor version:

- For V62 and V65, the default vector length is 64 bytes.
- For V66 and later, the default vector length is 128 bytes.

-mno-hvx

Disables HVX.

4.3.15 Code generation

-fasynchronous-unwind-tables

Generates an unwind table. The table is stored in DWARF2 format.

-femit-all-data

Emits all data, even if unused.

-femit-all-decls

Emits all declarations, even if unused.

-ffixed-*rnn*

Reserves the `rnn` register. The compiler does not use this register in code that it generates.

-fmerge-functions
-fno-merge-functions

Attempt to merge functions that are equivalent, or differ by only a few instructions(Section 5.6). The default setting is `disabled`.

This option attempts to improve code size by merging similar functions. It uses a number of heuristics to determine whether it is worthwhile to merge a pair of functions. For instance, very small functions or functions with significant differences are usually not merged.

NOTE: Because this option might have a negative impact on program performance, it is disabled by default. It becomes enabled only when it is specified explicitly.

-fno-exceptions

Does not generate code for propagating exceptions.

-fpic

Generates position-independent code (PIC) for use in a shared library.

-fPIC

Generates position-independent code for dynamic linking, avoiding any limits on the size of the global offset table.

-fpie
-fPIE

Generates position-independent code (PIC) for linking into executables.

-fshort-enums
-fno-short-enums

Allocates to an enum type only as many bytes necessary for the declared range of possible values. The default setting is `disabled`.

-fshort-wchar
-fno-short-wchar

Forces `wchar_t` to be `short unsigned int`. The default setting is `disabled`.

-ftrap-function=*name*

Issues a call to the specified function rather than a trap instruction.

-ftrapv

Traps on integer overflow.

-ftrapv-handler=*name*

Specifies the function to be called in the case of an overflow.

-funwind-tables

Similar to `-fexceptions`, except that it only generates any necessary static data, without affecting the generated code in any other way.

-fverbose-asm

Adds commentary information to the generated assembly code to improve code readability.

-fvisibility=[**default**|**internal**|**hidden**|**protected**]

Sets the default symbol visibility for all global declarations.

-fwrapv

Treats signed integer overflow as two's complement.

-G *size*

Assign data objects that fit within the specified size to the .sdata/.sbss section. The size is in bytes, and the default value is 8.

Compiling with `-G 0` requires all other files in the application to be compiled with `-G 0`.

An application must be compiled with a `-G` value that is no larger than the smallest `-G` value that any of its libraries was compiled with.

-mno-global-merge

Disable merging of globals.

-mno-packets

Do not generate packets. This option is supported for optimization levels -O1 and -O2.

-mlong-calls

Generate long calls.

-mhvx-length=[64B,128B]

Sets the vector length to the value provided and overrides the default value set by `-mhvx` or `-mhvx=` flags.

This option is valid only for the `-mhvx` and `-mhvx=` flags.

NOTE: The 64-byte mode will soon be deprecated, and the compiler will display a warning message: `warning: The HVX vector length of 64 Bytes will soon be deprecated. Please switch to 128 Byte mode. [-Wdeprecated-hvx-length]`

4.3.16 Optimization

-O0

Does not optimize. This is the default optimization setting.

-O
-O1

Enables a small set of optimizations. QTI does not recommend this optimization level for performance or code size.

-O2

Enables optimizations for performance. Optimizations enabled at `-O2` improve performance, but they might cause a small-to-moderate increase in compiled code size.

-O3

Enables aggressive optimizations for performance. Optimizations enabled at `-O3` improve performance, but they might cause a large increase in compiled code size.

-Os

Enables optimizations for code size. Optimizations enabled at `-Os` reduce code size at the cost of a small-to-moderate decrease in compiled code performance.

Recommended options for best performance and code size

- `-O2` (for best performance)
- `-Os` (for best code size)

4.3.17 Specific optimizations

`-fdata-sections`

Assigns each data item to its own section.

`-ffast-math`

Allows aggressive, lossy floating-point optimizations. Set `-ffp-contract=fast`.

`-ffp-contract=[off|on|fast]`

Forms fused floating point operations such as fused multiply accumulates.

- OFF – Do not fuse operations. (Default)
- Fast – Fuse operations whenever possible.
- ON – Enable the FP_CONTRACT DEFAULT pragma.

`-ffunction-sections`

Assigns each function item to its own section in the output file. The section is named after the function assigned to it.

`-finline`

Specifies the `inline` keyword as active.

`-finline-functions`

Performs heuristically-selected inlining of functions.

`-fno-zero-initialized-in-bss`

Assigns all variables that are initialized to zero to the BSS section.

`-fnomerge-all-constants`

Does not merge constants.

`-fomit-frame-pointer`

Does not store the stack frame pointer in a register if it is not required in a function.

`-foptimize-sibling-calls`

Optimizes function sibling calls and tail-recursive calls.

`-fstack-protector`

Generates code which checks selected functions for buffer overflows.

`-fstack-protector-all`

Generates code which checks all functions for buffer overflows.

`-fstack-protector-strong`

Generates code which applies strong heuristic to check additional selected functions for buffer overflows. Additional functions checked include those with local array definitions or references to local frame addresses.

NOTE: `-fstack-protector-strong` is not supported in the 6.2.x tools release.

`-fstrict-aliasing`

Enforces the strictest possible aliasing rules for the language being compiled.

`-funit-at-a-time`

Parses the entire compilation unit before beginning code generation.

`-funroll-all-loops`

Unrolls all loops.

`-funroll-loops`

Unrolls selected loops.

`-fvectorize`

Enables automatic vectorization.

`--param ssp-buffer-size=`*`size`*

Specifies the minimum size (in bytes) that a buffer must be if buffer-overflow checks are to be generated for it by the `-fstack-protector` options. The default value is `8`.

4.3.18 Link-time optimization

Compiler options

`-flto`
`-flto=thin`

For future link-time optimization (LTO), change the output format to bitcode (see Section 5.8) instead of object code format:

- `-flto` enables full LTO
- `-flto=thin` enables ThinLTO (for more information, see Section 5.8.4)

The file extension remains `.o` for `-c` and `.s` for `-S`, but the file contains different code format.

Linker options

`-flto-options=<>`

LTO-specific prefix to denote a linker LTO option.

`-flto-options=codegen="`*`list of options`***`"`**

Pass the list of options to the compiler to be used for LTO global optimization. An example would be `-flto-options=codegen="-O3 -mv5"`.

This is functionally equivalent to `-flto-options=codegen=-O3 -flto-options=codegen=-mv5`.

NOTE: The optimization level specified in this manner overrides individual optimization levels specified during bitcode generation, with the exception of the `-Os` (optimize for size) option.

-flto-options=cache=*path*

Enable incremental builds with ThinLTO.

The path specifies the location where ThinLTO stores its cache. This location should be an unused directory. A good practice is to remove the contents of this directory as part of any clean operation in the build system.

-flto-options=nooptimal-preservelist

Do not use `optimal preservelist` for LTO.

-flto-options=preserveall

Preserve all symbols for LTO.

-flto-options=preserve-file=*file*

Preserve the symbols listed in *`file`*.

-flto-options=preserve-sym=*symbol*

Preserve the symbol for LTO.

-flto-options=save-lto-output-file=*filename*

Save the object file produced by LTO as a filename.

-flto-options=threads=*n*

Perform LTO in several parallel threads. This mode of operation is the default for ThinLTO, which uses as many threads as there are cores in the system. Use this option to specify a lower number of threads.

-flto-use-as

When combined with the `-save-temps` option for linker, save intermediate merged files in text assembly format, which is useful for debugging and error reporting.

-save-temps

Save the normally-temporary intermediate files generated during compilation. This option can be used for debugging.

-trace=lto

Print information about the communication between the compiler and linker during LTO. This option can be used for debugging.

4.3.19 Compiler security

`--analyze`

Invokes the static program analyzer (Section 6.11) on the specified input files.

`-analyzer-checker=`*`checker`*

Enables the specified checker or checker category in the static program analyzer.

The checker categories are `alpha`, `core`, `cplusplus`, `debug`, and `security`. Enabling a checker category enables all the checkers in that category. For a complete list of checker names use `-analyzer-checker-help`.

NOTE: `-analyzer-checker` must be prefixed with `-Xclang`

`-analyzer-checker-help`

Lists the complete set of checkers and their categories for use in `-analyzer-checker` and `-analyzer-checker-disable`.

NOTE: `-analyzer-checker-help` must be prefixed with `-cc1`.

`-analyzer-disable-checker=`*`checker`*

Disables the specified checker or checker category in the static program analyzer.

The checker categories are `alpha`, `core`, `cplusplus`, `debug`, and `security`. Disabling a checker category disables all the checkers in that category.

For a complete list of checker names use `-analyzer-checker-help`.

NOTE: `-analyzer-disable-checker` must be prefixed with `-Xclang`.

`--analyzer-output html`

Generates the static analyzer output report in HTML format.

NOTE: `--analyzer-output` and its argument must each be prefixed with `-Xclang`.

`--analyzer-Werror`

Converts all static analyzer warnings into errors.

`--compile-and-analyze` *`dir`*

Invokes the static program analyzer on an entire program.

The analysis report files are written to the specified directory.

To prevent the analyzer from generating HTML files and instead print a summarized report on the `stdout`, replace `dir` with the sub-flag, `--analyzer-perf`.

`--compile-and-analyze-high`

Invokes the static program analyzer on an entire program with only a small list of high priority checkers.

The analysis report files are written to the specified directory. If you use the sub-flag, `--analyzer-perf`, they are printed to the `stdout`.

`--compile-and-analyze-medium`

Invokes the static program analyzer on an entire program with a list of high+medium priority checkers.

The analysis report files are written to the specified directory. If you use the sub-flag, `--analyzer-perf`, they are printed to the `stdout`.

4.3.20 Warning and error messages

LLVM provides a number of ways to control which code constructs cause the compilers to emit errors and warning messages, and how the messages are displayed on the console.

4.3.21 Control how diagnostics are displayed

When LLVM emits a diagnostic message, it includes rich information in the output and provides control over which information is printed. LLVM has the ability to print this information. The following options are used to control the information:

- A file/line/column indicator that shows exactly where the diagnostic occurs in the code.
- A categorization of the diagnostic as a note, warning, error, or fatal error.
- A text string describing the problem.
- An option indicating how to control the diagnostic (for diagnostics that support it) [`-fdiagnostics-show-option`].
- A high-level category for the diagnostic for clients that want to group diagnostics by class (for diagnostics that support it) [`-fdiagnostics-show-category`].
- The line of source code that the issue occurs on, along with a caret and ranges indicating the important locations [`-fcaret-diagnostics`].
- *FixIt* information, which is a concise explanation of how to fix the problem (when LLVM is certain it knows) [`-fdiagnostics-fixit-info`].
- A machine-parseable representation of the ranges involved (disabled by default) [`-fdiagnostics-print-source-range-info`].

For more information on these options see Section 4.3.7.

4.3.22 Diagnostic mappings

All diagnostics are mapped into one of the following classes:

- Ignored
- Note
- Warning
- Error
- Fatal

4.3.23 Diagnostic categories

Though not shown by default, diagnostics can each be associated with a high-level category. This category is intended to make it possible to triage builds which generate a large number of errors or warnings in a grouped way.

Categories are not shown by default, but they can be turned on with the `-fdiagnostics-show-category` option (Section 4.3.7). When this option is set to `name`, the category is printed textually in the diagnostic output. When set to `id`, a category number is printed.

NOTE: The mapping of category names to category identifiers can be obtained by invoking LLVM with the `-print-diagnostic-categories` option.

4.3.24 Control diagnostics with compiler options

LLVM can control which diagnostics are enabled through the use of options specified on the command line.

The `-W` options are used to enable warning diagnostics for specific conditions in a program. For instance, `-Wmain` will generate a warning if the compiler detects anything unusual in the declaration of function `main()`.

`-Wall` enables all the warnings defined by LLVM. `-w` disables all of them.

Warnings for a specific condition can be disabled by specifying the corresponding `-W`*cond* option as `-Wno-`*cond*. For instance, `-Wno-main` disables the warning normally enabled by `-Wmain`.

`-Werror=`*cond* changes the specified warning to an error (Section 4.3.22). `-Werror` specified without a condition changes *all* the warnings to errors. `-ferror-warn` changes only the warnings that are listed in the specified text file.

`-pedantic` and `-pedantic-errors` enable diagnostics that are required by the ISO C and ISO C++ standards.

4.3.25 Control diagnostics with pragmas

LLVM can also control which diagnostics are enabled through the use of pragmas in the source code. This is useful for disabling specific warnings in a section of source code. LLVM supports GCC's pragma for compatibility with existing source code, as well as several extensions.

The pragma can control any warning that can be used from the command line. Warnings can be set to ignored, warning, error, or fatal. The following example instructs LLVM or GCC to ignore the `-Wall` warnings:

```
#pragma GCC diagnostic ignored "-Wall"
```

In addition to all the functionality provided by GCC’s pragma, LLVM also enables you to push and pop the current warning state. This is useful when writing a header file that will be compiled by other people, because you do not know what warning flags they build with.

In the following example, `-Wmultichar` is ignored for only a single line of code, after which the diagnostics return to whatever state had previously existed:

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wmultichar"

char b = 'df'; // no warning.

#pragma clang diagnostic pop
```

The `push` and `pop` pragmas save and restore the full diagnostic state of the compiler, regardless of how it was set. That means that it is possible to use push and pop around GCC-compatible diagnostics, and LLVM will push and pop them appropriately, while GCC will ignore the pushes and pops as unknown pragmas.

NOTE: While LLVM supports the GCC pragma, LLVM and GCC do not support the same set of warnings. Thus even when using GCC-compatible pragmas there is no guarantee that they will have identical behavior on both compilers.

4.3.26 Control diagnostics in system headers

Warnings are suppressed when they occur in system headers. By default, an included file is treated as a system header if it is found in an include path specified by `-isystem`, but this can be overridden in several ways.

The `system_header` pragma can be used to mark the current file as being a system header. No warnings will be produced from the location of the pragma onwards within the same file.

```
char a = 'xy'; // warning

#pragma clang system_header

char b = 'ab'; // no warning
```

The options `-isystem-prefix` and `-ino-system-prefix` can be used to override whether subsets of an include path are treated as system headers. When the name in a `#include` directive is found within a header search path and starts with a system prefix, the header is treated as a system header. The last prefix on the command-line which matches the specified header name takes precedence. For example:

```
$ clang -Ifoo -isystem bar -isystem-prefix x/
-ino-system-prefix x/y/
```

Here, `#include "x/a.h"` is treated as including a system header, even if the header is found in `foo`, and `#include "x/y/b.h"` is treated as not including a system header, even if the header is found in `bar`.

An `#include` directive which finds a file relative to the current directory is treated as including a system header if the including file is treated as a system header.

4.3.27 Enable all warnings

In addition to the traditional `-W` flags, *all* warnings can be enabled by specifying the `-Weverything` option.

`-Weverything` works as expected with `-Werror`, and it also includes the warnings from `-pedantic`.

NOTE: When this option is used with `-w` (which disables all warnings), `-w` takes priority.

4.4 HVX vector operand widths

HVX is a set of Hexagon instruction extensions which support the implementation of very wide vector operations.

HVX supports two execution modes:

- In 64-byte mode, the vector operands are 512 bits wide.
- In 128-byte mode, the vector operands are 1024 bits wide.

These modes are specified at compile time using the `-mhvx-length` options (Section 4.3.14). If no `-mhvx-length` is specified, a default mode is set based on either the Hexagon processor version or the Hexagon HVX coprocessor version.

A given program source file can only contain either a 64-byte or a 128-byte vector length HVX instructions (not both). However, a single executable can contain both Single- and Double-mode HVX instructions, as long as the mixing is done at link time, using object files that were compiled separately for either 64-byte or a 128-byte vector length HVX instructions.

To ensure proper alignment of vector data, HVX code should always use the generic vector data types (as defined in the `hexagon_types.h` library):

- `HVX_VectorPred`
- `HVX_Vector`
- `HVX_VectorPair`

NOTE: For more information on HVX, see the appropriate *Qualcomm Hexagon HVX Programmer's Reference Manual* (listed in Appendix B).

5 Code optimization

The LLVM compilers provide many tools and features for improving the size or speed of the generated object code.

5.1 Software pipelining

LLVM includes support for software pipelining. By default, pipelining is disabled; to enable it, use the `-O3` option (commonly used). Software pipelining attempts to increase instruction-level parallelism by dividing a loop into three distinct sections:

- The *prolog* executes the initial loop iterations that are necessary to set up the kernel section.
- The *kernel* executes the loop steady state (which is structured to execute more efficiently than the equivalent non-pipelined loop code).
- The *epilog* executes the final loop iterations that are necessary to complete any operations left undone by the kernel section.

The following example shows how software pipelining works:

```
// Vector multiply and accumulate
int foo(int *a, int *b, int n) {
  int i;
  int sum = 0;
  for (i = 0; i < n; i++) {
    sum += a[i] * b[i];
  }
  return sum;
}
```

Without software pipelining, the compiler generates the following code for the loop:

```
{
  loop0(.LBB0_2, r2)
}
.LBB0_2:
{
  r2 = memw(r0++#4)
  r4 = memw(r1++#4)
}
{
  r3 += mpyi(r4, r2)
  nop
}:endloop0
```

Each loop iteration executes two instruction packets:

- The first packet loads the array elements `a[i]` and `b[i]`.
- The second packet multiples the two array elements and accumulates the result.

The instructions in the loop cannot be stored in a single packet, because the multiple-and-accumulate instruction depends on values that are loaded in the same loop iteration.

With software pipelining enabled, the compiler generates the following code for the loop:

```
{
  loop0(.LBB0_2, r2)
}
// prolog, iteration 0
{
  r2 = memw(r0++#4)
  r4 = memw(r1++#4)
}
// kernel, iteration 1 to n-1
.LBB0_2:
{
  r3 += mpyi(r4, r2)
  r2 = memw(r0++#4)
  r4 = memw(r1++#4)
}:endloop0
// epilog, iteration n-1
{
  r3 += mpyi(r4, r2)
}
```

In this generated code example, the prolog section (which is executed only once) performs the initial load of the array elements `a[0]` and `b[0]`.

The kernel section (which executes in the loop) multiplies and accumulates the currently-loaded array element values, and then loads the next set of array element values.

The epilog section (which is executed only once) performs the final multiply-and-accumulate of the values from the last iteration of the kernel (`a[n-1]` and `b[n-1]`).

Software pipelining enables the main loop code in this example to be stored in a single instruction packet (rather than the two packets required in the non-pipelined version), because the multiple-and-accumulate instruction is able to operate on values that were already loaded in the previous loop iteration.

5.2 Merge functions

LLVM includes support for function merging. By default, this optimization is disabled; to enable it, use the `-fmerge-functions` option (Section 4.3.18).

Function merging attempts to improve code size by merging functions that are equivalent or differ in only a few instructions. The optimization uses a number of heuristics to determine whether it is worthwhile to merge a pair of functions. For instance, very small functions or functions with significant differences are usually not merged.

The following example shows how function merging works:

```
int f1(int a, int b) {
   int x;
   x = a + 4;
   return x * b;
}
```

```
int f2(int a, int b) {
   int x;
   x = a + 10;
   return x * b;
}
```

Function merging determines that functions `f1` and `f2` are similar, and replaces them with the following functions:

```
int f1__merged(int a, int b, int choice) {
      int x;
      if (choice)
         x = a + 10;
      else
         x = a + 4;
      return x * b;
}

int f1(int a, int b) {
      return f1__merged(a, b, 0);
}

int f2(int a, int b) {
      return f1__merged(a, b, 1);
}
```

This example is for illustration purposes only. In practice, the optimizer would determine that functions `f1` and `f2` are too small to be worth merging.

NOTE: Because function merging might have a negative impact on program performance, it is disabled by default, and becomes enabled only when it is specified explicitly.

5.3 Automatic vectorization

The compiler supports automatic vectorization for HVX. Auto-vectorization attempts to identify repeated operations on individual data elements (scalars) and replace them with instructions that operate on sequences of elements (vectors). A typical example is a loop that performs certain calculation on an array of input data.

Auto-vectorization is disabled by default, even when compiler optimizations are enabled with `-O`. Auto-vectorization can be turned on with the compiler option, `-fvectorize`. The code generation for HVX must also be enabled via `-mhvx`. Because the default HVX vector length depends on the specific HVX version, QTI strongly recommends using `-mhvx-length` to specify the vector length.

Example of a compiler invocation that enables auto-vectorization:

```
hexagon-clang -O2 -fvectorize -mcpu=v62 -mhvx=v62 -mhvx-length=128b -c
file.c
```

In summary, to enable auto-vectorization:

1. Enable compiler optimizations (`-O1`, `-O2`, and so on).
2. Enable code generation for HVX (`-mhvx`, `-mhvx-length`).
3. Turn on the auto-vectorizer (`-fvectorize`).

Example 1

Consider the following C code:

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

Compile it with the following options:

```
hexagon-clang -O1 -fvectorize -mcpu=v62 -mhvx=v62 -mhvx-length=128b -S
test.c
```

The following assembly code (for the loop) is produced:

```
.LBB0_5:                    // %vector.body
                            // =>This Inner Loop Header: Depth=1
        {
                v0 = vmemu(r5+#0)
        }
        {
                v1 = vmemu(r4++#1)
        }
        {
                v2.w = vadd(v0.w,v1.w)
        }
        {
                nop
                vmemu(r5++#1) = v2
        } :endloop0
```

The compiler provides options that allow you to get feedback about vectorization. More specifically:

- `-Rpass=loop-vectorize` prints a remark when a given loop was vectorized
- `-Rpass-missed=loop-vectorize` prints a remark when a loop was not vectorized
- `-Rpass-analysis=loop-vectorize` displays reasons why a given loop was not vectorized

Example 2

Consider the same C code as in Example 1, with a small modification:

```
int foo(int x);

void test(int *a, int *b, int n) {
  for (int i = 0; i != n; ++i)
    a[i] = foo(b[i]);
}
```

Because the loop contains a function call, the compiler is not expected to be able to vectorize it. Using `-Rpass-missed=loop-vectorize`:

```
hexagon-clang -O2 -fvectorize -mcpu=v62 -mhvx=v62 -mhvx-length=128b -S
test.c -Rpass-missed=loop-vectorize
```

Shows:

```
test.c:4:3: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
  for (int i = 0; i != n; ++i)
  ^
```

And using `-Rpass-analysis=loop-vectorize`:

```
hexagon-clang -O2 -fvectorize -mcpu=v62 -mhvx=v62 -mhvx-length=128b -S
test.c -Rpass-analysis=loop-vectorize
```

Shows why:

```
test.c:5:12: remark: loop not vectorized: call instruction cannot be
vectorized [-Rpass-analysis=loop-vectorize]
    a[i] = foo(b[i]);
           ^
test.c:4:3: remark: loop not vectorized: read with atomic ordering or
volatile read [-Rpass-analysis=loop-vectorize]
  for (int i = 0; i != n; ++i)
  ^
```

5.3.1 Programming considerations

The data types that can be vectorized are those that correspond to the HVX data types (8-bit, 16-bit, and 32-bit signed and unsigned integers). Operations that can be vectorized are loads, stores, arithmetic operations (except division), bitwise operations, and shifts. Floating-point types cannot be vectorized. An ideal candidate for vectorization is a loop body that only contains operations that can be vectorized and no control flow.

Again consider the C code from Example 1:

```
void test1(int *a, int *b, int n) {
  for (int i = 0; i != n; ++i)
    a[i] += b[i];
}
```

This time use `-O2` and `-Rpass=loop-vectorize`:

```
hexagon-clang -O2 -fvectorize -mcpu=v62 -mhvx=v62 -mhvx-length=128b -c
file.c -Rpass=loop-vectorize

file.c:2:3: remark: vectorized loop (vectorization width: 32, interleaved
count: 2) [-Rpass=loop-vectorize]
  for (int i = 0; i != n; ++i)
  ^
```

The remark reveals the two parameters that describe how a loop is vectorized: vectorization width and interleave count. The vectorization width is the number of scalar operations that were grouped together into a single vector operation. Interleaving is conceptually similar to unrolling, and it refers to interleaving operations from consecutive iterations.

In this example, the vectorization width is 32, which is expected because in 128-byte mode, an HVX vector register holds 32 four-byte integers. The interleave count is 2, meaning that for each statement in the loop that can be vectorized, two vectorized statements are generated. The result of loop vectorization is typically more than just the vectorized loop body:

- Function parameters `a` and `b` can refer to overlapping memory areas, and they might not be aligned to a vector alignment.
- Iteration count `n` might not be a multiple of vectorization width, or it might be too small for even a single vectorized statement.

  To ensure correctness, the compiler inserts checks for memory aliasing, uses unaligned memory accesses if required, and generates scalar epilog loops to finish the *leftover* loop iterations that were not executed by the vectorized code. However, this can reduce performance or increase code size. There are several ways you can mitigate these issues:

- Use aligned data and inform the compiler about the alignment. For example, the previous code in test1 can be modified as follows:

  ```
  void test2(int *a __attribute__((align_value(128))),
             int *b __attribute__((align_value(128))),
             int n) {
    for (int i = 0; i != n; ++i)
      a[i] += b[i];
  }
  ```

  The `align_value` attribute indicates that the pointer parameter value is aligned to a 128-byte boundary.

- Use the `restrict` keyword on pointer parameters. The code example can be further modified as follows:

```
void test3(int *restrict a __attribute__((align_value(128))),
           int *restrict b __attribute__((align_value(128))),
           int n) {
  for (int i = 0; i != n; ++i)
    a[i] += b[i];
}
```

  This modification will eliminate the need for memory aliasing check.

  NOTE: The `restrict` keyword only exists in C. Clang supports the equivalent keyword, `__restrict`, in C++ (also in C) as an extension.

Clang allows you to specify loop optimization hints via a special #pragma clang loop construct. For example, to reduce the code size in test3, you can suggest that neither interleaving nor unrolling should be performed:

```
void test4(int *restrict a __attribute__((align_value(128))),
           int *restrict b __attribute__((align_value(128))),
           int n) {
  #pragma clang loop interleave(disable)
  #pragma clang loop unroll(disable)
  for (int i = 0; i != n; ++i)
    a[i] += b[i];
}
```

The full list of such #pragmas is at:

clang.llvm.org/docs/LanguageExtensions.html#extensions-for-loop-hint-optimizations

5.4 Link-time optimization

Link-time optimization (LTO) comprises a set of powerful optimizations that are performed during the linking stage of compilation across multiple files or libraries. There are two variants of LTO: full LTO and ThinLTO.

NOTE: LTO is applicable to the 8.0 tools release and later. Support for ThinLTO is available with the 8.2 tools release and later.

LTO expands the scope of optimizations from an individual source file to the entire program (or at least to all the files and libraries visible at link time and marked for LTO processing). This enables deeper compiler analysis (such as better alias analysis) and more effective code transformations (such as function inlining and dead code elimination), which can result in improved performance and code size.

To mark a source file or a library for later full LTO or ThinLTO processing, it must be compiled with the `-flto` or `-flto=thin` option respectively. The compiler then produces a file containing the LLVM compiler's intermediate representation (also known as *bitcode*) instead of the usual object code format. This file can be subsequently used in a final link step that then performs inter-module code optimizations on the file contents.

Most of the optimization work is done at the link time. The linker can recognize that some of the input files or libraries contain bitcode and invoke the compiler on the combined scope for global processing.

NOTE: A mixture of object and bitcode inputs is possible, and the optimization scope will be limited to those inputs containing bitcode.

You can choose an optimization level at the individual file stage, such as `-Os`, `-O1`, and `-O3`. This optimization level is used for bitcode optimization. At the LTO stage, the same optimizations are always be applied, but when necessary, `-Os` is respected. You can specify an optimization level for code generation (back end) via `-flto-options=codegen=-Ox`, where the following choices are available for *x*:

- `0 = none`
- `1 = less`
- `2 = default`
- `3 = aggressive`

These values will apply regardless of the optimization level selected for the individual file.

Typically, LTO results in an increase in build time due to its larger optimization scope. full LTO, in particular, does not support incremental builds. If one of the bitcode input files is changed, the linker triggers recompilation of all bitcode input files. ThinLTO, described in the Section 5.4.1, does not have this limitation.

5.4.1 ThinLTO

ThinLTO is a variant of LTO that significantly reduces build time compared to full LTO. It achieves this at the expense of performing fewer cross-module optimizations than full LTO. The main benefit of ThinLTO is the ability to inline functions across file boundaries, which can improve runtime performance in many applications. Enable ThinLTO by passing the `-flto=thin` option to the compiler instead of `-flto`.

Unlike full LTO, ThinLTO also supports incremental builds. To enable support for incremental builds, use the `-flto-options=cache=`*`path`* linker option to create a cache directory in the build directory. This cache directory must be unused; as part of the cleanup operation for the build system, remove the contents of this directory.

5.4.2 Simple LTO usage example

file_01.c:

```
int foo (int val) {
  return val << 2;
}
```

file_02.c:

```
int foo(int);
int bar (int val) {
  return foo(val);
}
```

file_03.c:

```
#include <stdio.h>
int bar(int);
int main(int argc) {
  return printf("result %d\n", bar(argc));
}
```

To perform LTO on these files, enter the following sequence of commands.

```
hexagon-clang -flto -c file_01.c file_02.c file_03.c
hexagon-link file_01.o file_02.o file_03.o
```

NOTE: This sequence is one of several possible option combinations; architecture options are not shown for simplicity.

This sequence allows the compiler to perform cross-file optimizations that result in an executable equivalent to compiling a single file with the following content:

```
file_lto.c
#include <stdio.h>
int main(int argc) {
  return printf("result %d\n", (argc<<2));
}
```

5.5 Profile-driven section assignment

Automated section assignment injects a user-specified infix (such as hot) into the section names of symbols that exceed a certain profile count threshold. For example, you can define a rule that causes a function `foo` in section `.text.foo`, with a profile count of `1000`, to be placed in `.text.hot.foo`. You can add a rule to the linker script that matches `.text.hot.*` and places these input sections into a different output section.

5.5.1 Configuration

Two additional input files for the compiler, or, in the case of LTO, the linker, are used to configure section assignment: a placement rules file and a placement profile.

5.5.1.1 Placement rules file

The placement rules determine which symbols are tagged with an infix, what this infix should be, and above which profile count threshold the tagging should occur.

```
1|FUNCS:quitehot:1
2|FUNCS:prettyhot:100
3|FUNCS:veryhot:1000
4|VARS:hotvars:1
```

Explanation:

- Line 1: Add the infix, `quitehot`, to the linker input section of all functions with a profile entry count of `[1..100)`, where 100 is the starting point of the next rule.

  For example, a function `bla` with entry count 5 that would have been in the `.text.bla` section is now placed in the `.text.quitehot.bla` section. In the linker script, this can then be matched by a rule in the `.text.resident` section for `.text.quitehot.*`.
- Line 2: Add the infix, `prettyhot`, for functions with an entry count of `[100...1000)`.
- Line 3: Add the infix, `veryhot`, for functions with an entry count of `[1000..infinity)`.
- Line 4: Add the infix, `hotvars`, to any variables with access counts of `[1..infinity)`.

  For example, a variable that would have gone to `.bss.somevar` will now go to `.bss.hotvars.somevar`. A variable that would have gone to `.rodata.somevar` will now go to `.rodata.hotvars.somevar`.

Functions and variables that are not matched by any rule or that have explicit sections set (using `__attribute__((section(.somesection))`) are not touched.

5.5.1.2 Placement profile file

This file provides profile counts for variables and functions.

- For functions, the profile counts can also come from the SamplePGO profile that is used for other profile-guided optimizations in the compiler. If a symbol has an entry in both the sample profile and placement profile, the placement profile takes precedence for section assignment.
- For variables, the placement profile is the only way of providing profile counts because variables are not represented in the sample profile format. The format is straightforward (`SymbolName:ProfileCount`):

```
1|myfunction:1000
2|somevariable:42
```

Also, a section assignment can include the profile count in the section name of every symbol being placed. This feature is required for section budgeting (see the *Qualcomm Hexagon Utilities User Guide* (80-N2040-1583).

The profile count is encoded in HEX format and is inserted immediately after the infix. In the example, the `bla` function in the `.text.bla` section with count 5 is assigned to the `.text.quitehot._F5.bla` section, where `._Fxxx` indicates the profile count.

5.5.2 Command line options

The following command line options specify the configuration files and enable PGO counts in section names.

- For non-LTO builds, add the following command line options to every invocation of hexagon-clang (for example, via CFLAGS):

```
-mllvm -pgo-section-rules=FILENAME -mllvm-pgo-placement-profile= FILENAME
-mllvm -enable-pgo-count-infix
```

- For LTO builds, add the following command line options to every invocation of the linker (for example, via LFLAGS):

```
-flto-options=codegen="-pgo-section-rules=FILENAME -pgo-placement-
profile=FILENAME -enable-pgo-count-infix"
```

5.5.3 Verify the section assignment

To verify the correct operation of section assignment, inspect the map file of the linker. If no assignment takes place, check that the options actually reach the hexagon-link invocation. With SCons, use the `--verbose=2` build option.

6 Compiler security tools

The LLVM compiler release includes the following tools for performing static analysis on a program:

- Static analyzer

  A tool that analyzes source code to find potential bugs in C and C++ programs.

- Postprocessor

  A tool that creates a summary of the reports that are generated by performing static analysis while compiling a program.

6.1 Static analyzer

The static analyzer is a source code analysis tool which is integrated into the LLVM compiler. It analyzes a program for potential bugs—including security threats, memory corruption, and garbage values—and generates a diagnostic report describing the potential bugs it detected.

The static analyzer has the following features:

- It supports more than 100 distinct *checkers* which are organized into the `alpha`, `core`, `cplusplus`, `debug`, and `security` categories
- Checkers can be selectively enabled or disabled from the command line
- Disabling a checker category disables all the checkers in that category
- Selected parts of the program code can be excluded from checking

6.1.1 Analyze programs

To use the static analyzer on an entire program, invoke the LLVM compiler on the program using the `--compile-and-analyze` option (Section 4.3.28). For example:

```
clang --compile-and-analyze dir input_files...
```

- `--compile-and-analyze <dir>` specifies the directory where the static analyzer report will be stored. (If the directory does not exist, the compiler automatically creates it.). The report is automatically generated in HTML format. The files are named `report*.html`.
- *`input_files`* specifies the program source files.

QTI recommends statically analyzing an entire program at once (as opposed to selected source files) for the following reasons:

- The generated analysis report files are all stored in a single location.
- The command option can be passed from the build system, which helps perform the static analysis and compilation every time the program is built.
- Because build systems are good at tracking files that have changed, and compiling only the minimal set of required files, the overall turnaround time for static analysis is relatively small, making it reasonable to run the static analyzer with every build.

When using a build system, specifying the same directory name throughout the build generates all the HTML report files in the specified directory. The filenames generated for a report are based on hashing functions, so the report files is not overwritten.

6.1.2 Analyze programs using default flags

To use the static analyzer on specific program source files, invoke the LLVM compiler on the files using the static analyzer options (Section 4.3.28). For example:

`clang --analyze -Xclang --analyzer-output -Xclang html -o` *`dir files`*

Where:

- `--analyze` causes the compiler to generate a static analyzer report instead of a program object file.
- `--analyzer-output html` specifies that the report is generated in HTML format.
- `-Xclang` must be used (twice) to pass the `--analyzer-output html` option to the compiler. For details, see Section 4.3.2.
- `-o` specifies the directory where the report files will be stored. (If the directory does not exist, the compiler automatically creates it.). The files are named `report*.html`.
- *`files`* specifies the program source files to be analyzed.

Example of a diagnostic report entry:

```
// @file: test.cpp
int main() {
  int* p = new int();
  return* p;
}
warning: Potential leak of memory pointed to by ‘p’
```

Each potential bug flagged in a report includes the path (control and data) required for locating the bug in the program.

To convert static analyzer warnings to errors, use the `--analyzer-Werror` option.

6.1.3 Analyze programs with priority modes

In addition to the default `--compile-and-analyze` flag, the static analyzer also offers a list of specific directives for more targeted analysis.

- High Priority mode

  This mode turns on the highest priority checkers that catch critical issues and have low false positive rates.

  This mode can be enabled by the `--compile-and-analyze-high` flag.

- Medium Priority mode

  This mode is slightly more inclusive compared to the high priority mode, but it still avoids some noisy checkers that are enabled by default.

  This mode can be enabled by the `--compile-and-analyze-medium` flag.

- KW mode

  This mode reduces issues that are typically caught by commercial static analysis tools. It is especially helpful for users who want to catch those issues early in the development process.

  This mode can be enabled by the `--compile-and-analyze-kw` flag.

6.1.4 Manage checkers

The static analyzer supports more than 100 individual checkers which can analyze programs for various types of potential bugs. By default, only a subset of these checkers is enabled, to minimize both the compile time and the generation of false positives.

To enable additional checkers, invoke the static analyzer using the option, `-analyzer-checker` (Section 4.3.28). This option specifies the checker to be enabled (in the following example, `NewDelete`).

```
clang++ --compile-and-analyze <dir> -Xclang -analyzer-checker=NewDelete
test.cpp
```

To disable individual checkers, invoke the static analyzer using the option, `-analyzer-disable-checker` (Section 4.3.28). For example:

```
clang++ --compile-and-analyze <dir> -Xclang -analyzer-disable-checker=
deadcode.deadstore test.cpp
```

NOTE: `-Xclang` must be used to pass the `-analyzer-output`, `-analyzer-checker`, and `-analyzer-disable-checker` options to the compiler. For details, see Section 4.3.28.

To list all the supported checkers, use the following command:

```
clang -cc1 -analyzer-checker-help
```

To list only the default checkers, use the `-v` option while running the analyzer on a test file (Section 4.3.2). For example:

```
clang++ -v --compile-and-analyze --analyzer-perf test.cpp
```

6.1.4.1 Packages

The individual checkers are organized into the following categories:

- alpha
- core
- cplusplus (only for analyzing C++ programs)
- debug
- security
- unix
- optin

Each category (or `package`) is defined to include a number of checkers. For example, the checker `NullDereference` is a `core` checker, while `NewDelete` is a `cplusplus` checker. Organizing checkers into packages (and sub-packages) makes it easier to enable/disable specific sets of checkers.

Example of using the static analyzer with all `alpha` checkers enabled:

```
clang --compile-and-analyze <dir> -Xclang -analyzer-checker=alpha
test.cpp
```

6.1.4.2 Lists

To enable or disable multiple individual checkers, multiple checker and package names can be specified as a single comma-separated list. For example:

```
clang --compile-and-analyze <dir> -Xclang -analyzer-checker=alpha,core
test.cpp
```

6.1.5 Cross-file analysis

A limitation of the clang static analyzer is its inability to analyze code that is called across file boundaries. This is due to the nature of the compiler, which compiles files individually. This feature enables the clang static analyzer to perform its analysis on code that is called across file boundaries.

The cross-translation unit (CTU) feature allows the analysis of called functions even if the definition of the function is external to the currently analyzed file. This feature allows detection of bugs in library functions stemming from incorrect usage. If an external function is invoked, this feature also allows for more precise analysis of the caller in general.

Enabling CTU analysis is a three-step process:

1. Generate the `compile_commands.json` file.

   The makefile is read by the intercept-build script. This script is responsible for creating a `compile_commands.json` file that contains a list of every build command that is run from this build system.

   ```
   $share/scan-build-py/bin/intercept-build make
   ```

2. Generate the AST database.

   Generate the ASTs of every function in the build tree by issuing the following command:

   ```
   $share/scan-build-py/bin/analyze-build --cdb <location of the
   compile_commands json file> -o
   <location of the intended report directory> --ctu-collect-only --ctudir
   <location where all the AST's are to be stored>
   ```

3. Run the cross-file analysis.

   Kick off the static analyzer in the usual way (with the `--compile-and-analyze` flag) but with following added flags:

   ```
   -Xclang -analyzer-checker=debug.ExprInspection -Xclang -analyzer-config
   -Xclang -ctu-dir=<location of the intended report directory>
   ```

6.1.6 Handle false positives

While checking a program for potential bugs, the static analyzer might report false positives, which are sections of code that the analyzer incorrectly flags as bugs.

To minimize false positives, the static analyzer, by default, enables a set of checkers that has been tested to identify a high percentage of actual program bugs (Section 6.1.4). And if necessary, additional checkers can be individually enabled.

However, despite the overall accuracy of the checkers, several cases still exist where false positives can be generated. For instance, if you enable the checker used to analyze dead code, the static analyzer will flag as a false positive any code that has been conditionally enabled for debugging purposes.

To handle such cases, the static analyzer supports several features for handling false positives:

- Compile-time blacklist file
- Special comment
- Preprocessor symbol
- Function attribute
- Postprocess blacklist file

QTI does not recommend using comments or symbols to handle false positives because they make the code inaccessible to the analyzer. Instead, report any false positives so the existing checkers can be improved to eliminate them. For more information on false positives, go to:

clang-analyzer.llvm.org/faq.html

6.1.6.1 Compile-time blacklist file

To silence warnings before they are reported by the analyzer, use the external file-based suppression mechanism. Enable this mechanism by adding the compiler flag, `-analyzer-suppression-file <file location>`.

Suppression data in this text file must be entered line-by-line with the following format:

<filename.extension> <function identifier> <checker name>

The analyzer reads the contents of this file during the analysis and does not report any warnings that are mentioned in it.

6.1.6.2 Blacklist file

The blacklist file consists of row-wise indications of warnings that must silenced. It uniquely identifies each warning by filename, function name, and bug description. Place this file in a network location that is accessible to all developer machines using the analyzer.

During postprocessing, identify the blacklist file through the flag `--blacklist-file <file>`. The `post-process` script does not report the bugs marked in this file. Besides silencing warnings across developer machines, this mechanism also allows silence warnings across build variants.

A blacklist file contains row-by-row warnings in the following format:

<filename.cpp> <function identifier/Global> <full warning description>

For example:

```
test.cpp foo Address of stack memory associated with local variable buff
returned to caller [core.StackAddressEscape]
```

6.1.6.3 Special comment

Individual lines of code can be excluded from checking by adding the following comment to the line:

`// clang_sa_ignore [`*`checker`*`] [`*`user_comment_text`*`]`

`checker` specifies a checker, package, or list (Section 6.1.4) that is excluded from being applied to the line of code. It must be enclosed in square brackets. For example:

```
g_ptr = new int(0); // clang_sa_ignore [deadcode.DeadStores]
g_ptr = new int(0); // clang_sa_ignore [alpha] my comment text
g_ptr = new int(0); // clang_sa_ignore [alpha,deadcode.DeadStores]
```

6.1.6.4 Preprocessor symbol

One or more lines of code can be conditionally excluded from all checking by using the preprocessor symbol `__clang_analyzer__`, which is automatically defined by the static analyzer. For example:

```
#ifndef __clang_analyzer__
   // Code excluded from checking
#endif
```

When using the preprocessor symbol with the static analyzer, the code must remain compilable, even though it does not need to be linkable or executable. For example, to exclude the body of a function from being analyzed, use the following conditional code:

```
#ifdef __clang_analyzer__
void noisyFunction(); // this version is for analysis only

#else // __clang_analyzer__
static void noisyFunction() {
// function body is generating too many false positives
}
#endif // __clang_analyzer__
```

6.1.6.5 Postprocess blacklist file

This blacklist file consists of row-wise indications of warnings that must be silenced. It uniquely identifies each warning by filename, function name, and bug description. Place this file in a network location that is accessible to all developer machines using the analyzer.

During postprocessing, identify the blacklist file through the `--blacklist-file <file>` flag. The postprocess script will not report the bugs marked in this file. Besides silencing warnings across developer machines, this mechanism also allows you to silence warnings across build variants.

The blacklist file contains row-by-row warnings in the following format:

```
<filename.cpp> <function identifier/Global> <full warning description>
```

For example:

```
test.cpp foo Address of stack memory associated with local variable buff
returned to caller [core.StackAddressEscape]
```

6.1.6.6 Function attribute

A common source of false positives is non-returning functions such as assert functions.

Although the static analyzer is aware of the standard library non-returning functions, if (for example) a program has its own implementation of asserts, it helps to mark them with the following function attribute:

```
__attribute__((__noreturn__))
```

Using this attribute greatly improves the static analysis diagnostics and lessens the number of false positives. For example:

```
void my_abort(const char* msg) __attribute__((__noreturn__)) {
printf("%s", msg);
exit(1);
}
```

6.1.7 Create whitelist directories

Often, unwieldy build systems make it impossible to turn on the static analyzer for only some subdirectories. For example, given the following hierarchy, assume you want to see results for only the `Qcomm_code` directories and not the `Open_source` directory:

ANDROID

Open_source Qcomm_hardware_code Qcomm_software_code

The build system employed here allows only `cflags` to be set at a global level. In this case, create a whitelist file containing row-wise entries of the folders to be scanned. For example:

```
ANDROID/Qcomm_hardware_code
ANDROID/Qcomm_software_code
```

The `post-process` script must be notified of the whitelist file through the flag, `--whitelist-file <whitelist text file>`. The script then identifies the row-wise entries and checks them against any static analysis warnings found. If none of the entries are a substring of the file location of a specific bug, that bug is silenced. Thus, having only `Qcomm_hardware_code` and `Qcomm_software_code` in the whitelist file will suffice because they will both be part of the location of the warnings you are interested in.

6.1.8 Treat warnings as errors

Static analysis warnings can be treated as errors by appending the `--analyzer-Werror` flag to the build flags. For example:

```
clang --compile-and-analyze <dir> -Xclang --analyzer-Werror test.c
```

6.1.9 Checker categorization by priority

Because checkers fall under two major priority categories (high and medium), the following flags are available in addition to the regular `--compile-and-analyze` flag:

`--compile-and-analyze-high`

The high flag allows only a small subset of checkers to be turned on. These checkers are security critical and have an extremely low false positive rate. Teams adopting this tool should start with the high flag.

`--compile-and-analyze-medium`

The medium flag is slightly more permissive than the high flag. The medium flag contains a few more checkers that are deemed security critical but have a slightly higher false positive rate.

6.1.10 Performance mode

The static analyzer can be run in Performance mode. This mode prevents the creation of all the raw HTML files, and instead it displays a summary of the warnings in the console output.

To enable this mode, append the `--analyzer-perf` flag to the `--compile-and-analyze` flag in place of the `<dir>` option. For example:

```
clang --compile-and-analyze --analyzer-perf test.c
clang --compile-and-analyze-high --analyzer-perf test.c
```

6.1.11 YAML configuration file

The static analyzer can also be configured using a single configuration file (`--qc-config-file`) that combines all above-mentioned options in a single location. For example:

```
clang --qc-config-file qc_sa_config.txt test.c
```

The configuration file uses YAML format and is subject to all syntax restrictions. Following is a sample configuration file with embedded comments.

```
##############################################
# Static Analyzer configuration file        #
##############################################

##############################################
# Global SA options                          #
##############################################
---
entry-type:      global_settings
# Verbosity suppression level. Most useful is "high" i.e. "high suppression" == fewer false
positives. The "high" setting here is equivalent to the command line compile-and-analyze-
high flag.
# Available options: zero|default|high|medium|low
verbosity:       high
# Location for individual reports. Absolute or relative path.
analyzer-output-dir: sa_report_dir
# Format for output reports. Most useful html.
analyzer-output-format: html
```

```
# Same as -analyzer-config stable-report-filename=true reduces the number of duplicate
reports
stable-report-filename: true
# Do not output duplicate report for different path to the same location.
# The same header file can be included from multiple .c/cpp locations - report issues only
once.
no-duplicate-reports: true

##############################################
# Per-checker individual options            #
# It is applied _after_ the GLOBAL settings #
##############################################
---
entry-type:      checker_config
config:
  - package:         core
    # "default" - use verbosity setting from global_settings
    # "on" - enable checker
    # "off" - disable checker
    checkers:
      # Check for dereferences of null pointers
      - checker-name:    NullDereference
        state:           off
      # Check for logical errors for function calls and Objective-C message expressions
(e.g., uninitialized arguments, null function pointers)
      - checker-name:    CallAndMessage
        state:           off
  - package:         alpha.core
    checkers:
     # Check when casting a malloc'ed type T, whether the size is a multiple of the size of T
      - checker-name:    CastSize
        state:           default
      # Here follows the full list of all available packages and checkers with individual
settings for each.

##############################################
# Individual files to skip during SA        #
##############################################
---
entry-type:      suppression_list
# Blacklist file. See -analyzer-suppression-file option for details
analyzer-suppression-file: suppression_file
# Per path report suppression. If any part of the file path matches this list - no report is
generated. Currently there is no wildcard support.
suppress-path:
  - modem_proc/audio_avs/main/voice/algos/mmecns/mmecns_lib/src
  - modem_proc/audio_avs
...
```

6.2 Postprocessor

The postprocessor is a report generator which is implemented as a standalone script. This `post-process` script creates a summary of the report that is generated by using the `-compile-and-analyze` option (Section 6.1). To invoke the postprocessor, enter the following command:

`post-process --report-dir` *dir*

The postprocessor reads all the files from the directory specified by the `--report-dir` option and writes in the same directory a summary report file named `index.html`. The report title can be specified with the `--html-title` option. For more information on the postprocessor, enter the **`post-process --help`** command.

NOTE: The `post-process` script is stored in the `$INSTALL_PREFIX/bin` directory.

NOTE: In some cases the static analyzer might generate multiple report files for the same bug. The postprocessor cleans up after multiple report files. For this reason it should be run regularly to keep the report directory clean.

7 Port code from GCC

This chapter describes issues commonly encountered while porting to LLVM an application that was previously built only with GCC.

NOTE: For more information on GCC compatibility, see Chapter 9.

7.1 Command options

LLVM supports many but not all of the GCC command options. Unsupported options are either ignored or flagged with a warning or error message; most receive warning messages. For more information, see Section 4.3.5.

7.2 Errors and warnings

LLVM enforces strict conformance to the C11 language standard. As a result, you might encounter new errors and warnings when compiling GCC code.

To handle these messages when porting to LLVM, consider the following steps:

1. Remove the `-Werror` command option if it is being used (because it converts all warnings into errors).
2. Update the code to eliminate the remaining errors and warnings.

7.3 Function declarations

LLVM enforces the C11 rules for function declarations. In particular:

- A function declared with a non-`void` return type must return a value of that type.
- A function referenced before being declared is assumed to return a value of type `int`. If the function is subsequently declared to return some other type, it is flagged as an error.
- A function declaration with the `inline` attribute assumes the existence of a separate definition for the function, which does not include the `inline` attribute. If no such definition appears in the program, a link-time error will occur.

To satisfy these restrictions when porting to LLVM, consider the following steps:

1. Use `-Wreturn-type` to generate a warning whenever a function definition does not return a value of its declared type.
2. Use `-Wimplicit-function-declaration` to generate a warning whenever a function is used before being declared.
3. Update the code to eliminate the remaining errors and warnings.

For more information on inlining, go to:

clang.llvm.org/compatibility.html#inline

For a discussion of different inlining approaches, go to:

http://www.greenend.org.uk/rjk/tech/inline.html

7.4 Casting to incompatible types

LLVM enforces the C11 rules for *strict aliasing*.

In the C language, two pointers that reference the same memory location are said to *alias* one another. Because any store through an aliased pointer can potentially modify the data referenced by one of its pointer aliases, pointer aliases can limit the compiler's ability to generate optimized code.

In strict aliasing, pointers to different types are prevented from being aliased with one another. The compiler flags pointer aliases with an error message.

Strict aliasing has a few exceptions:

- Any pointer type can be cast to `char*` or `void*`.
- A `char*` or `void*` can be cast to any pointer type.
- Pointers to types that differ only by whether they are signed or unsigned (for example, `int` versus `unsigned int`) can be aliased.

To satisfy strict aliasing when porting to LLVM, consider the following steps:

1. Use `-Wcast-align` to generate a warning whenever a pointer alias is detected.
2. Update the code to eliminate the resulting warnings.

NOTE: Dereferencing a pointer that is cast from a less strictly aligned type has undefined behavior.

7.5 aligned attribute

LLVM does not allow the `aligned` attribute to appear inside the `__alignof__` operator.

To satisfy this restriction when porting to LLVM, create a typedef with the `aligned` attribute. For example:

```
typedef unsigned char u8;
#ifdef __llvm__
    typedef u8 __attribute((aligned)) aligned_u8;
#endif
unsigned int foo()
{
#ifndef __llvm__
    return __alignof__(u8 __attribute__ ((aligned)));
#else
    return __alignof__(aligned_u8);
#endif
}
```

7.6 Reserved registers

LLVM does not support the GCC extension to place global variables in specific registers.

To satisfy this restriction when porting to LLVM, use the equivalent LLVM intrinsics whenever possible. For example:

```
#ifndef __llvm__
    register unsigned long current_frame_pointer asm("r11");
#endif
...
#ifndef __llvm__
    fp = current_frame_pointer;
#else
    fp = (unsigned long)__builtin_frame_address(0);
#endif
```

7.7 Inline versus extern inline

LLVM conforms to the C11 language standard, which defines different semantics for the `inline` keyword than GCC. For example, consider the following code:

```
inline int add(int i, int j) { return i + j; }

int main() {
  int i = add(4, 5);
  return i;
}
```

In C11 the function attribute `inline` specifies that a function's definition is provided only for inlining, and that another definition (without the `inline` attribute) is specified elsewhere in the program.

This implies that this example is incomplete, because if `add()` is not inlined (for example, when compiling without optimization), `main()` will include an unresolved reference to that other function definition. This will result in the following link-time error:

```
Undefined symbols:
  "_add", referenced from:   _main in cc-y1jXIr.o
```

By contrast, GCC's default behavior follows the GNU89 dialect, which is based on the C89 language standard. C89 does not support the `inline` keyword; however, GCC recognizes it as a language extension, and treats it as a hint to the optimizer.

There are several ways to fix this problem:

- Change `add()` to a static inline function. This is usually the right solution if only one translation unit needs to use the function. Static inline functions are always resolved within the translation unit, so it will not be necessary to add a non-inline definition of the function elsewhere in the program.
- Remove the `inline` keyword from this definition of `add()`. The `inline` keyword is not required for a function to be inlined, nor does it guarantee that it will be. Some compilers ignore it completely. LLVM treats it as a mild suggestion from the programmer.
- Provide an external (non-inline) definition of `add()` somewhere else in the program. The two definitions *must* be equivalent.
- Compile with the GNU89 dialect by adding `-std=gnu89` to the set of LLVM options. QTI does not recommend this approach.

8 Coding practices

This chapter describes recommended coding practices for users of the LLVM compilers. These practices typically result in the compiler generating more optimized code.

8.1 Use int types for loop counters

QTI strongly recommends using an `int` type for loop counters, which results in the compiler generating more efficient code. If the code uses a non-`int` type, the compiler must insert zero and sign-extensions to abide by C rules. For example, QTI does not recommend the following code:

```
extern int A[30], B[30];
for (short int ctr = 0; ctr < 30; ++ctr) {
    A[ctr] = B[ctr] + 55;
}
```

Use this code instead:

```
extern int A[30], B[30];
for (int ctr = 0; ctr < 30; ++ctr) {
    A[ctr] = B[ctr] + 55;
}
```

8.2 Mark function arguments as restrict (if possible)

LLVM supports the `restrict` keyword for function arguments. Using `restrict` on a pointer passed in as a function argument indicates to the compiler that the pointer will be used exclusively to dereference the address it points at. This allows the compiler to enable more aggressive optimizations on memory accesses.

NOTE: When using the `restrict` keyword, you must ensure that the restrict condition holds for all calls made to that function. If an argument is erroneously marked as `restrict`, the compiler might generate incorrect code.

8.3 Do not pass or return structs by value

QTI strongly recommends that structs are passed to (and returned from) functions by reference and not by value.

If a struct is passed to a function by value, the compiler must generate code which makes a copy of the struct during application runtime. This can be extremely inefficient, and will reduce the performance of the compiled code. For this reason, QTI recommends that structs be passed by pointer.

For example, the following code is inefficient:

```
struct S {
   int z;
   int y[50];
   char *x;
   long int w[40];
};

int bar(struct S arg1) {
   ...
}

int baz() {
  struct S s;
   ...
   bar(s);
}
While this code is much more efficient:
struct S {
   int z;
   int y[50];
   char *x;
   long int w[40];
};

int bar(struct *S arg1/* Access z here using ‘arg1->z’ (instead of
‘arg1.z’) */
  ...
}

int baz() {
  struct S s;
   ...
   bar(&s);
}
```

Alternatively, in C++, the efficient code can be simplified by using reference parameters:

```
struct S {
   int z;
   int y[50];
   char *x;
   long int w[40];
};

int bar(struct &S arg1) {
   ...
}

int baz() {
  struct S;
   ... populate elements of S ...
   bar(S);
}
```

8.4 Avoid using inline assembly

Using inline assembly snippets in C files is strongly discouraged for two reasons:

- Inline assembly snippets are extremely difficult to write correctly. For instance, omitting the input, output, or clobber parameters frequently leads to incorrect code. The resulting failure can be extremely difficult to debug.
- Inline assembly is not portable across processor versions. If you need to emit a specific assembly instruction, QTI recommends using a compiler intrinsic instead of inline assembly.

Intrinsics are easy to insert in a C file, and they are portable across processor versions. If intrinsics are insufficient, add a new function written in assembly. The assembly function should be called from C code.

9 Language compatibility

LLVM strives to both conform to current language standards, and to implement many widely-used extensions available in other compilers, so that most correct code will *just work* when compiled with LLVM. However, LLVM is more strict than other popular compilers, and it might reject incorrect code that other compilers allow.

This chapter describes common compatibility and portability issues with LLVM to help you understand and fix the problem in your code when LLVM emits an error message.

9.1 C compatibility

This section describes common compatibility and portability issues with LLVM C.

9.1.1 Differences between various standard modes

LLVM supports the `-std` option, which changes what language mode LLVM uses. The supported modes for C are c89, gnu89, c94, c99, gnu99, c11, gnu11, and various aliases for those modes. If no `-std` option is specified, LLVM defaults to gnu99 mode.

The c* and gnu* modes have the following differences:

- c* modes define `__STRICT_ANSI__`.
- Target-specific defines not prefixed by underscores (such as `linux`) are defined in gnu* modes.
- The default for trigraphs in gnu* modes is Off. Trigraphs can be enabled by the `-trigraphs` option.
- The parser recognizes `asm` and `typeof` as keywords in gnu* modes. The variants `__asm__` and `__typeof__` are recognized in all modes.
- Arrays that are VLAs according to the standard, but which can be constant-folded by the compiler front end, are treated as fixed size arrays.

  For example, `int X[(1, 2)];` is technically a VLA. c* modes are strictly compliant and treat these as VLAs.
- The Apple *blocks* extension is recognized by default in gnu* modes on some platforms. It can be enabled in any mode with the `-fblocks` option.

The *89 and *99 modes have the following differences:

- The default for *99 modes is to implement `inline` as specified in C11, while the *89 modes implement the GNU version.

  The `__gnu_inline__` attribute can be used to override the default for individual functions.
- Digraphs are not recognized in c89 mode.
- The scope of names defined in a `for`, `if`, `switch`, `while`, or `do` statement is different. For example, `if ((struct x {int x;}*)0) {}`.
- `__STDC_VERSION__` is not defined in *89 modes.
- `inline` is not recognized as a keyword in c89 mode.
- `restrict` is not recognized as a keyword in *89 modes.
- Commas are allowed in integer constant expressions in *99 modes.
- Arrays that are not lvalues are not implicitly promoted to pointers in *89 modes.
- Some warnings are different.

c94 mode is identical to c89 mode except that digraphs are enabled in c94 mode.

9.1.2 GCC extensions not yet implemented

LLVM tries to be compatible with GCC as much as possible, but the following GCC extensions are not yet implemented in LLVM:

- `#pragma weak`

  Will probably be implemented at some point in the future, at least partially.
- Decimal floating (`_Decimal32`, and so on) and fixed-point types (`_Fract`, and so on)

  No one has expressed interest in this extension yet; currently, it is not clear when they will be implemented.
- Nested functions

  A complex feature that is infrequently used, so it is unlikely to be implemented soon.
- Global register variables

  Requires additional LLVM back-end support, so this extension is unlikely to be implemented soon.
- Static initialization of flexible array members

  A rarely used extension, but it could be implemented pending user demand.
- `__builtin_va_arg_pack` and `__builtin_va_arg_pack_len`

  Rarely used, but it might be implemented in potentially interesting places such as the glibc headers, pending user demand.

  Because LLVM pretends to be like GCC 4.2 and this extension was introduced in GCC 4.3, the glibc headers currently do not try to use this extension with LLVM.

- Forward-declaring function parameters

  This extension has not shown up in any real-world code yet, though, so it might never be implemented.

9.1.3 Intentionally unsupported GCC extensions

LLVM intentionally does not implement the following GCC extensions:

- Variable-length arrays in structures

  This extension is not implemented for several reasons:

  - It is tricky to implement
  - The extension is completely undocumented
  - The extension appears to be rarely used

  LLVM supports flexible array members (arrays with a zero or unspecified size at the end of a structure).

- An equivalent to GCC's `fold`

  This extension implies that LLVM does not accept some constructs GCC might accept in contexts where a constant expression is required, such as `x-x`, where `x` is a variable.

- `__builtin_apply` and related attributes

  This extension is extremely obscure and difficult to reliably implement.

9.1.4 Lvalue casts

Old versions of GCC permit casting the left-hand side of an assignment to a different type. LLVM produces an error for code like this:

```
lvalue.c:2:3: error: assignment to cast is illegal, lvalue casts are not
supported
(int*)addr = val;
^~~~~~~~~~ ~
```

To fix this problem, move the cast to the right-hand side. For example:

```
addr = (float *)val;
```

9.1.5 Jumps to within __block variable scope

LLVM disallows jumps into the scope of a `__block` variable. Variables marked with `__block` require special runtime initialization. A jump into the scope of a `__block` variable bypasses this initialization, leaving the variable's metadata in an invalid state. Consider the following code fragment:

```
int fetch_object_state(struct MyObject *c) {
  if (!c->active) goto error;

  __block int result;
  run_specially_somehow(^{ result = c->state; });
  return result;

 error:
  fprintf(stderr, "error while fetching object state");
  return -1;
}
```

GCC accepts this code, but it produces code that typically crashes when the result goes out of scope if the jump is taken. (It is possible for this bug to go undetected, because it often does not crash if the stack is fresh; that is, it is still zeroed.) Therefore, LLVM rejects this code with a hard error:

```
t.c:3:5: error: goto into protected scope
    goto error;
    ^
t.c:5:15: note: jump bypasses setup of __block variable
  __block int result;
              ^
```

The fix is to rewrite the code so it does not require jumping into the scope of a `__block` variable; for example, by limiting that scope:

```
{
  __block int result;
  run_specially_somehow(^{ result = c->state; });
  return result;
}
```

9.1.6 Non-initialization of __block variables

In the following example, the variable `x` is used before it is defined:

```
int f0() {
  __block int x;
  return ^(){ return x; }();
}
```

By an accident of implementation, GCC and `llvm-gcc` unintentionally always zero any initialized `__block` variables. However, any programs that depend on this behavior relies on unspecified compiler behavior. Programs must explicitly initialize all local block variables before they are used, as with other local variables.

LLVM does not zero-initialize local block variables—thus any programs that rely on such behavior will most likely break when built with LLVM.

9.2 C++ compatibility

This section describes common compatibility and portability issues with LLVM C++.

The types in C++ code compiled with `-std=c++11` are not inter-operable with the types in C++ code compiled with `-std=c++03`, or with code compiled with no `-std=` specification. For example, the `std::string` from a C++03 compilation is unrelated to a `std::string` from a C++11 compilation, and the two `std::string` types cannot be easily converted to each other.

9.2.1 Variable-length arrays

GCC and C11 allow the size of an array to be determined at runtime. This extension is not permitted in standard C++. However, LLVM supports such variable length arrays in very limited circumstances for compatibility with GNU C and C11 programs.

- The element type of a variable length array must be a *plain old data* (POD) type. It cannot have any user-declared constructors or destructors, any base classes, or any members of non-POD type. All C types are POD types.
- Variable length arrays cannot be used as the type of a non-type template parameter.

If the code uses variable length arrays in a manner that LLVM does not support, several methods are available to fix the code:

- Replace the variable length array with a fixed-size array if you can determine a reasonable upper bound at compile time. Sometimes this is as simple as changing `int size = ...;` to `const int size = ...;` (if the initializer is a compile-time constant).
- Use `std::vector` or some other suitable container type.
- Allocate the array on the heap instead using `new Type[]`; remember to delete it using `delete[]`.

9.2.2 Unqualified lookup in templates

Some versions of GCC accept the following invalid code:

```
template <typename T> T Squared(T x) {
  return Multiply(x, x);
}

int Multiply(int x, int y) {
  return x * y;
}

int main() {
  Squared(5);
}
```

LLVM flags this code with the following messages:

```
my_file.cpp:2:10: error: call to function 'Multiply' that is neither
visible in the template definition nor found by argument-dependent lookup


   return Multiply(x, x);
          ^


my_file.cpp:10:3: note: in instantiation of function template
specialization 'Squared<int>' requested here

  Squared(5);
  ^


my_file.cpp:5:5: note: 'Multiply' should be declared prior to the call
site

  int Multiply(int x, int y) {

      ^
```

The C++ standard states that unqualified names such as `Multiply` are looked up in two ways:

- First, the compiler performs an *unqualified lookup* in the scope where the name was written.

  For a template, this means the lookup is done at the point where the template is defined, not where it is instantiated. Because `Multiply` has not been declared yet at this point, unqualified lookup will not find it.

- Second, if the name is called like a function, the compiler also does an *argument-dependent lookup* (ADL).

  In an ADL, the compiler looks at the types of all the arguments to the call. When it finds a class type, it looks up the name in that class's namespace. The result is all the declarations it finds in those namespaces plus the declarations from unqualified lookup. However, the compiler does not do an ADL until it knows all the argument types.

In the example code, `Multiply` is called with dependent arguments, so an ADL is not done until the template is instantiated. At that point, the arguments both have type `int`, which does not contain any class types, and so the ADL does not look in any namespaces. Because neither form of lookup found the declaration of `Multiply`, the code does not compile.

Following is another example that uses overloaded operators, which obey very similar rules.

```
#include <iostream>

template<typename T>

void Dump(const T& value) {
  std::cout << value << "\n";
}

namespace ns {  struct Data {};
}
```

```
std::ostream& operator<<(std::ostream& out, ns::Data data) {    return
out << "Some data";
}

void Use() {
  Dump(ns::Data());
}
```

Again, LLVM flags this code with the following messages:

```
my_file2.cpp:5:13: error: call to function 'operator<<' that is neither
visible in the template definition nor found by argument-dependent lookup

  std::cout << value << "\n";
            ^


my_file2.cpp:17:3: note: in instantiation of function template
specialization 'Dump<ns::Data>' requested here

  Dump(ns::Data());
  ^


my_file2.cpp:12:15: note: 'operator<<' should be declared prior to the
call site or in namespace 'ns'

  std::ostream& operator<<(std::ostream& out, ns::Data data) {
                ^
```

As before, unqualified lookup did not find any declarations with the name, `operator<<`. Unlike before, the argument types both contain class types:

- One type is an instance of the class template type, `std::basic_ostream`
- The other is the type `ns::Data` that is declared in the example

Therefore, the ADL looks in the `std` and `ns` namespaces for an `operator<<`. Because one of the argument types was still dependent during the template definition, the ADL is not done until the template is instantiated during `Use`, which means that the `operator<<` it finds has already been declared. Unfortunately, it was declared in the global namespace, not in either of the namespaces that the ADL look in.

Two methods exist to fix this problem:

- Make sure the function you want to call is declared before the template that might call it.

  This is the only option if none of its argument types contain classes. Either move the template definition or function definition, or add a forward declaration of the function before the template.
- Move the function into the same namespace as one of its arguments so that the ADL applies.

9.2.3 Unqualified lookup into dependent bases of class templates

Some versions of GCC accept the following invalid code:

```
template <typename T> struct Base {
  void DoThis(T x) {}
  static void DoThat(T x) {}
};

template <typename T> struct Derived : public Base<T> {
  void Work(T x) {
    DoThis(x); // Invalid!
    DoThat(x); // Invalid!
  }
};
```

LLVM correctly rejects this code with the following errors (when `Derived` is eventually instantiated):

```
my_file.cpp:8:5: error: use of undeclared identifier 'DoThis'

  DoThis(x);
  ^
  this->

my_file.cpp:2:8: note: must qualify identifier to find this declaration
in dependent base class

void DoThis(T x) {}
     ^


my_file.cpp:9:5: error: use of undeclared identifier 'DoThat'

DoThat(x);
^
this->

my_file.cpp:3:15: note: must qualify identifier to find this declaration
in dependent base class

static void DoThat(T x) {}
```

As noted in Section 9.2.2, unqualified names such as `DoThis` and `DoThat` are looked up when the `Derived` template is defined, not when it is instantiated. When looking up a name used in a class, you typically look into the base classes. However, you cannot look into the `Base<T>` base class because its type depends on the template argument `T`, so the standard says to ignore it.

As LLVM indicates, the fix is to tell the compiler that you want a class member by prefixing the calls with `this->`:

```
void Work(T x) {
  this->DoThis(x);
  this->DoThat(x);
}
```

Alternatively, you can tell the compiler exactly where to look:

```
void Work(T x) {
  Base<T>::DoThis(x);
  Base<T>::DoThat(x);
}
```

This approach works whether the methods are static or not, but be careful: if `DoThis` is virtual, calling it this way will bypass virtual dispatch.

9.2.4 Incomplete types in templates

The following code is invalid, but compilers are allowed to accept it:

```
class IOOptions;

template <class T> bool read(T &value) {
  IOOptions opts;
  return read(opts, value);
}

class IOOptions { bool ForceReads; };
bool read(const IOOptions &opts, int &x);
template bool read<>(int &);
```

According to the standard, types that do not depend on template parameters must be complete when a template is defined if they affect the program's behavior. However, the standard also says that compilers are free to not enforce this rule. Most compilers enforce it to some extent; for example, it would be an error in GCC to write `opts.ForceReads` in the code example.

In LLVM, the decision to enforce the rule consistently provides a better experience, but unfortunately, it also results in some code that is rejected when other compilers will accept it.

9.2.5 Templates with no valid instantiations

The following code contains a typo: `innit()` was written instead of `init()`.

```
template <class T> class Processor {
  ...
  void init();
  ...
};

...

template <class T> void process() {
  Processor<T> processor;
  processor.innit();      // <-- should be 'init()'
  ...
}
```

Unfortunately, the compiler cannot flag this mistake as soon as it detects it. Inside a template, assumptions cannot be made about dependent types such as `Processor<T>`. Suppose that later in this file you add an explicit specialization of `Processor`, like the following example:

```
template <> class Processor<char*> {
  void innit();
};
```

The program will work, but only if you instantiate `process()` with `T = char*`. Therefore, it is hard, and sometimes impossible, to diagnose mistakes in a template definition before it is instantiated.

The standard states that a template with no valid instantiations is ill-formed. LLVM tries to do as much checking as possible at definition time instead of instantiation time; not only does this produce clearer diagnostics, but it also substantially improves compile times when using precompiled headers. The downside to this philosophy is that LLVM sometimes fails to process files because they contain broken templates that are no longer used. The solution is simple: because the code is unused, remove it.

9.2.6 Default initialization of const variable of class type

The default initialization of a `const` variable of a class type requires a user-defined default constructor.

If a `class` or `struct` has no user-defined default constructor, C++ does not allow you to default-construct a `const` instance of it. For example:

```
class Foo {
public:
  // The compiler-supplied default constructor works fine, so we    //
don't bother with defining one.
  ...
}
void Bar() {
  const Foo foo; // Error!
  ...
}
```

To fix this issue, define a default constructor for the class:

```
class Foo {
public:
  Foo() {}
  ...
};

void Bar() {
  const Foo foo; // Now the compiler is happy.
  ...
}
```

9.2.7 Parameter name lookup

Due to a bug in its implementation, GCC allows the redeclaration of function parameter names within a function prototype in C++ code. For example:

```
void f(int a, int a);
```

LLVM diagnoses this error (where the parameter name has been redeclared). To fix this problem, rename one of the parameters.

10 Language extensions

The Hexagon LLVM compilers support language features that are not defined in the standard LLVM compilers. These features are specific to the Hexagon processor and are known as *language extensions*.

The compilers predefine several symbols to simplify the development of portable software. For example:

```
#if __hexagon__
...                     // Hexagon processor-specific code
#endif
```

10.1 Predefined symbols for processor versions

Following are the symbols defined when compiling code for specific versions of the Hexagon processor (Section 4.3.16).

Table 10-1 Predefined symbols (processor version)

Symbol	V62	V65	V66	V67	V67 Small Core	V68
__hexagon__	1	1	1	1	1	1
__HEXAGON_V62__	1	Symbol not defined	Symbol not defined	Symbol not defined	Symbol not defined	Symbol not defined
__HEXAGON_V65__	Symbol not defined	1	Symbol not defined	1	Symbol not defined	Symbol not defined
__HEXAGON_V66__	Symbol not defined	Symbol not defined	1	Symbol not defined	Symbol not defined	Symbol not defined
__HEXAGON_V67__	Symbol not defined	Symbol not defined	Symbol not defined	1	Symbol not defined	Symbol not defined
__HEXAGON_V67T__	Symbol not defined	Symbol not defined	Symbol not defined	Symbol not defined	1	Symbol not defined
__HEXAGON_V68	Symbol not defined	Symbol not defined	Symbol not defined	Symbol not defined	Symbol not defined	1
__HEXAGON_ARCH__	62	65	66	67	67	68

10.2 Predefined symbols for HVX coprocessor versions

Following are the symbols defined when compiling code for specific versions of the HVX coprocessor. Refer to Section 4.3.16 on how to enable HVX and select an HVX coprocessor version.

Table 10-2 Predefined HVX symbols (coprocessor version)

Symbol	V62	V65	V66	V68
`__HVX__`	1	1	1	1
`__HVX_ARCH__`	62	65	66	68

The `__HVX_LENGTH__` predefined macro is set to either 64 or 128. The value is based on the vector length that is selected for compilation.

11 Inline assembly language

LLVM supports inline assembly language (assembly instructions in a C function). Although this chapter provides an overview of inline assembly usage, Qualcomm discourages the use of inline assembly because it is error-prone and difficult to maintain. If assembly language is required, use a separate file containing the assembly code.

11.1 Asm statement

LLVM uses the `asm` statement to support the inline assembly language:

```
asm ("nop");
```

In this example, the processor instruction specified in the string is inserted directly into the generated assembly code at its current position in the C program.

LLVM also allows the operands of inline assembly instructions to be specified as C expressions, which allows you write assembly code that is more tightly integrated with the surrounding C code. For example:

```
asm ("%0=add(%1,#%2)" : "=r" (result) : "r" (myvar), "i" (4));
```

This example is equivalent to the C statement, `result=myvar+4`, where `result` and `myvar` are C variables declared in the program. The strings preceding each operand specify the operand type, and whether the operand is an input or output.

The `asm` instruction has the following syntax:

```
asm ( instruction_template
 [: output_operands ]
 [: input_operands ]
 [: clobber_registers ]
 );
```

- The instruction template consists of a character string containing the text of the inline assembly code, with embedded parameters specifying where the operands are to be substituted (for example, `%0=add(%1,#%2)`). The parameter numbers (`%n`) correspond to the declaration order of the operands following the template.
- Each input or output operand consists of an operand constraint string followed by a C expression in parentheses (for example, `"=r" (result)`). Typical operand constraint characters are `r` (for register operands), `i` (for integer constant operands), and `m` (for memory operands).

- Each clobber register consists of a string containing the name of a register.
- Colons are used to separate the instruction template, output operand list, input operand list, and clobber register list. The lists are optional; however, if a list (such as output operands) is omitted while a following list (such as input operands) is not, two consecutive colons must appear to indicate the omitted list.
- Commas are used to separate the operands (or registers) within each list.

Branches from one asm statement to another asm statement are not supported. The compiler code optimizer is not aware of such branches, and therefore it cannot take them into account when deciding how to optimize.

Input and output operands can be specified using symbolic names that can be referenced within the assembly code. These names are specified inside square brackets preceding the constraint string, and they can be referenced inside the assembly code using `%[name]` instead of a percentage sign followed by the operand number. Using named operands, the example can be specified as:

```
asm ("%[dest]=add(%[src1],#%[src2])"
 : [dest] "=r" (result)
 : [src1] "r" (myvar), [src2] "i" (4));
```

NOTE: Symbolic operand names have no relation to other C identifiers. They can be assigned any name (including those of existing C symbols), but no two operands within the same asm statement can use the same symbolic name.

11.2 Operand constraints

Operand constraints are specified with the input and output operands of an asm statement. Constraints specify the following properties of an operand:

- Whether an operand can be in a register, and what kind of register
- Whether the operand can be a memory reference, and what kind of address
- Whether the operand can be an immediate constant, and what possible values

Constraints can also require two operands to match.

11.2.1 Simple constraints

Following are the most common constraints supported by inline assembly code:

`m` Memory address operand

`r` Register in the general purpose register class of the target: r0-r31

`i` Integer constant (of target-specific width); allows either a simple immediate or a relocatable value

`l` Low integer register: r0-r7

`a` Modifier register: m0-m1

`v` Vector register: v0-v31

`q` Vector predicate register: q0-q3

11.2.2 Constraint modifiers

In addition to the constraints described in Section 11.2.1, constraint modifiers can be added to specify the behavior for operands. If constraint modifiers are specified, they should be the first character of the constraint string. Following are commonly used modifiers:

`=` Operand is write-only for this instruction; the output data discards and replaces the previous value

`+` Instruction reads and writes the operand

In processing operands to satisfy the constraints, the compiler identifies which operands are inputs and outputs for the inline assembly code. = identifies an output; + identifies an operand that is both input and output; all other operands are assumed to be input only.

12 Attributes

Attributes are annotations that programmers add to the code. Attributes specify characteristics of program entities such as functions or variables. These characteristics influence the code generated by the compiler.

Add an attribute by using the `__attribute__` keyword. For example:

```
int add(int a, int b) __attribute__((always_inline));
```

For functions, add the attributes to the function declaration.

Commonly used attributes

`always_inline`

Specifies that the compiler must inline this function.

`noinline`

Specifies that the compiler must not inline this function.

`section`

Specifies the section where a function or a variable is placed. In the following example, the code snippet places the `foo` function in the section named `.text2`:

```
int foo() __attribute__((section(".text2")));
```

`aligned`

Specifies the minimum alignment of a variable or function in bytes. In the following example, the code snippet aligns the variable to 32 bytes:

```
int var __attribute__((aligned(32)));
```

`packed`

Specifies that no padding is introduced between members of a structure. For example, in the following data structure:

```
struct __attribute__((packed)){
  char x;
  int y;
} N;
```

No padding is introduced between elements x and y of struct N. Without the packed attribute, the compiler generates three bytes of padding between x and y to align x to its natural boundary.

NOTE: Dereferencing the address of a packed structure member might result in unaligned memory accesses.

`used`

Marks a function or variable as `used` so the compiler does not delete it, even if it is not used anywhere in the application or if all uses are optimized out.

This attribute is relevant for LTO, where the compiler can remove a larger number of symbols due to its global view of the code.

In conjunction with the linker script KEEP directive, this attribute can be used to preserve variables that are not used anywhere in the source code, but that are required to be present in the final binary.

`optnone`

Changes the optimization level for a function. If this attribute is added to a function, the compiler generates code for the function with the lowest optimization level.

13 Libraries

The Hexagon LLVM compilers include the following libraries:

- C standard library
- C++ standard library
- Dynamic loading library
- Hexagon-specific libraries
- HVX-specific libraries
- Intrinsics emulation library

The C and C++ standard libraries are based on the industry-standard Dinkumware libraries.

The dynamic loading library allows programs to load and unload libraries at runtime. It is derived from the standard UNIX dynamic loading library (`libdl`).

The Hexagon-specific libraries support the following features:

- Low-level intrinsics (instruction access, vector access, circular addressing)
- An API for accessing timer information in the Hexagon processor simulator

The HVX-specific library supports low-level intrinsics for HVX operations (instruction access, vector access, vector initialization, vector assignment).

The intrinsics emulation library allows developers to compile a C or C++ program that contains Hexagon instruction intrinsics into a native executable file for the host development system.

13.1 C standard library and C++ libraries

The C standard library and one of the C++ libraries are derived from the corresponding libraries developed by Dinkumware.

The Dinkumware C and C++ libraries are conforming implementations of the standard C and C++ libraries. For more information, see the following documents:

- *Hexagon C Library User Guide*
- *Hexagon C++ Library User Guide*

Users have two C++ libraries from which to choose, Dinkumware or libc++. See Section 13.1.2 for details.

13.1.1 C library

The C library is always the Dinkumware C library. It supports C11.

13.1.2 C++ libraries

You can choose from Dinkumware or libc++. Dinkumware is the default library when no standard is specified or when `-std=c++03` is specified. Libc++ is the default library when one of the following is specified: `-std=c++11`, `-std=c++14`, `std=c++17`.

The user can override the default by specifying `-stdlib=libstdc++` for Dinkumware and `-std=libc++` for libc++.

Dinkumware's C++ library supports C++03. It does not support C++11 or newer C++ standards. Dinkumware is supported on the standalone environment and QuRT™ software environment.

Libc++ supports C++03, C++11, C++14, and C++17. Libc++ was designed for QuRT, but it requires pending support from the QuRT OS to be fully operational.

13.2 Hexagon-specific libraries

The Hexagon processor-specific libraries support the following features:

- Instruction access
- Vector access (32- and 64-bit; C and C++)
- Circular addressing
- Bit-reversed addressing
- Simulation timer

Vector access is supported in the `hexagon_types.h` library.

Instruction access and circular/bit-reversed addressing are supported in the `hexagon_protos.h` library.

Simulator timer access is supported in the `hexagon_sim_timer.h` library.

13.2.1 Instruction access

To support efficient coding of the time-intensive sections of a program (without resorting to inline assembly language), LLVM provides intrinsics which are used to directly access Hexagon processor instructions.

The instruction intrinsics are accessed by including the `hexagon_protos.h` library file.

The following example shows how an instruction intrinsic is used to directly access the ALU64 instruction `Rdd=vminh(Rtt,Rss)`:

```
#include <hexagon_protos.h>

int main()
{
    long long v1=0xFFFF0000FFFF0000;
    long long v2=0x0000FFFF0000FFFF;
    long long result;

    // find the minimum for each half-word in 64-bit vector
    result = Q6_P_vminh_PP(v1,v2);
}
```

Intrinsics are defined for most of the Hexagon processor instructions. For more information, see the appropriate *Qualcomm Hexagon Programmer's Reference Manual* (listed in Appendix B).

13.2.2 Vector access (32-bit)

To support efficient coding of 32-bit vector operations, LLVM provides intrinsics which are used to perform the following tasks:

- Read or write individual words, half-words, or bytes in 32-bit vectors
- Construct 32-bit vectors from a sequence of words, half-words, or bytes

The vector intrinsics are accessed by including the header file `hexagon_types.h`. 32-bit vectors are defined as variables of type `Q6Vect32`.

The following example shows how the 32-bit vector intrinsics are used:

```
#include <hexagon_types.h>
extern int foo(unsigned char, Q6Vect32, Q6Vect32);

int bar()
{
    short w=1, x=3;
    int t=1;
    unsigned char b;
    Q6Vect32 v, rslt;

    // construct 32-bit vector from two half-words
    v = Q6V32_CREATE_H(w, x);

    // read low-order byte of 32-bit vector
    b = Q6V32_GET_UB0(v);

    // write word as 32-bit vector
    // return modified vector, v remains unchanged
    rslt = Q6V32_PUT_W(v, t);

    return foo(b, v, rslt);
}
```

Figure 13-1 shows how words, half-words, and bytes are accessed in a 32-bit vector.

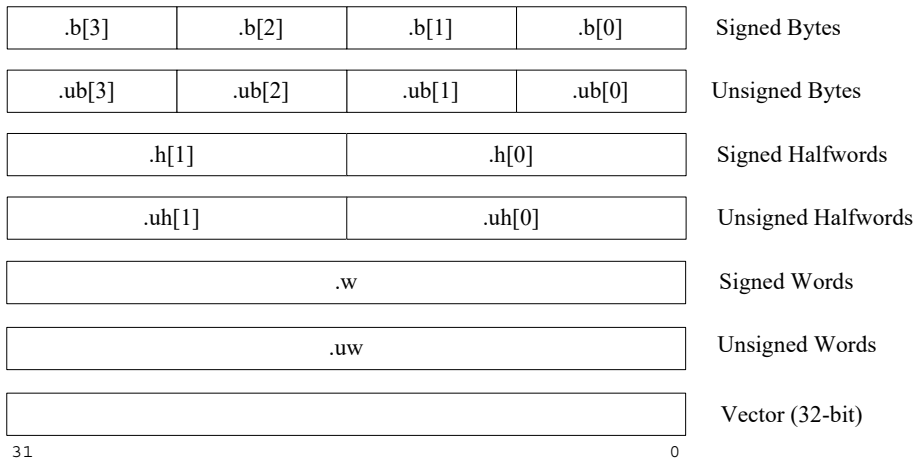

Figure 13-1 Vector access (32-bit)

13.2.2.1 Construct vector

Construct a 32-bit vector from the specified values and return it as a result value.

```
Q6Vect32 Q6V32_CREATE_W(int w);
Q6Vect32 Q6V32_CREATE_H(short h1, short h0);
Q6Vect32 Q6V32_CREATE_B(char b3, char b2, char b1, char b0);
```

13.2.2.2 Extract word

Extract word (either signed or unsigned) from a 32-bit vector and return it as a result value.

```
int Q6V32_GET_W(Q6Vect32 v);

unsigned int Q6V32_GET_UW(Q6Vect32 v);
```

13.2.2.3 Extract half-word

Extract the specified half-word (either signed or unsigned) from a 32-bit vector and return it as a result value.

```
short Q6V32_GET_H0(Q6Vect32 v);
short Q6V32_GET_H1(Q6Vect32 v);

unsigned short Q6V32_GET_UH0(Q6Vect32 v);
unsigned short Q6V32_GET_UH1(Q6Vect32 v);
```

13.2.2.4 Extract byte

Extract the specified byte (either signed or unsigned) from a 32-bit vector and return it as a result value.

```
signed char Q6V32_GET_B0(Q6Vect32 v);
signed char Q6V32_GET_B1(Q6Vect32 v);
signed char Q6V32_GET_B2(Q6Vect32 v);
signed char Q6V32_GET_B3(Q6Vect32 v);

unsigned char Q6V32_GET_UB0(Q6Vect32 v);
unsigned char Q6V32_GET_UB1(Q6Vect32 v);
unsigned char Q6V32_GET_UB2(Q6Vect32 v);
unsigned char Q6V32_GET_UB3(Q6Vect32 v);
```

13.2.2.5 Set word

Assign word value to a 32-bit vector and return the modified vector as a result value.

```
Q6Vect32 Q6V32_PUT_W(Q6Vect32 v, int val);
```

13.2.2.6 Set half-word

Assign a value to the specified half-word in a 32-bit vector and return the modified vector as a result value.

```
Q6Vect32 Q6V32_PUT_H0(Q6Vect32 v, short val);
Q6Vect32 Q6V32_PUT_H1(Q6Vect32 v, short val);
```

13.2.2.7 Set byte

Assign a value to the specified byte in a 32-bit vector and return the modified vector as a result value.

```
Q6Vect32 Q6V32_PUT_B0(Q6Vect32 v, char val);
Q6Vect32 Q6V32_PUT_B1(Q6Vect32 v, char val);
Q6Vect32 Q6V32_PUT_B2(Q6Vect32 v, char val);
Q6Vect32 Q6V32_PUT_B3(Q6Vect32 v, char val);
```

13.2.3 Vector access (32-bit; C++)

For C++ programs the vector access library also implements the 32-bit vector access operations (Section 13.2.2) as member functions of the class `Q6Vect32C`.

13.2.3.1 Constructors

Construct a 32-bit vector from the specified values.

```
Q6Vect32C::Q6Vect32C(int w);
Q6Vect32C::Q6Vect32C(short h1, short h0);
Q6Vect32C::Q6Vect32C(signed char b3, signed char b2,
                     signed char b1, signed char b0);

Q6Vect32C::Q6Vect32C(const Q6Vect32C & v);
Q6Vect32C::Q6Vect32C& operator = (const Q6Vect32C & v);
```

13.2.3.2 Extract word

Extract word (either signed or unsigned) from a 32-bit vector and return it as a result value.

```
int Q6Vect32C::W(void);

unsigned int Q6Vect32C::UW(void);
```

13.2.3.3 Extract half-word

Extract the specified half-word (either signed or unsigned) from a 32-bit vector and return it as a result value.

```
short Q6Vect32C::H0(void);
short Q6Vect32C::H1(void);

unsigned short Q6Vect32C::UH0(void);
unsigned short Q6Vect32C::UH1(void);
```

13.2.3.4 Extract byte

Extract the specified byte (either signed or unsigned) from a 32-bit vector and return it as a result value.

```
signed char Q6Vect32C::B0(void);
signed char Q6Vect32C::B1(void);
signed char Q6Vect32C::B2(void);
signed char Q6Vect32C::B3(void);

unsigned char Q6Vect32C::UB0(void);
unsigned char Q6Vect32C::UB1(void);
unsigned char Q6Vect32C::UB2(void);
unsigned char Q6Vect32C::UB3(void);
```

13.2.3.5 Set word

Assign word value to a 32-bit vector and return the modified vector as a result value.

```
Q6Vect32C Q6Vect32C::W(int w);
```

13.2.3.6 Set half-word

Assign a value to the specified half-word in a 32-bit vector and return the modified vector as a result value.

```
Q6Vect32C Q6Vect32C::H0(short h);
Q6Vect32C Q6Vect32C::H1(short h);
```

13.2.3.7 Set byte

Assign a value to the specified byte in a 32-bit vector and return the modified vector as a result value.

```
Q6Vect32C Q6Vect32C::B0(signed char b);
Q6Vect32C Q6Vect32C::B1(signed char b);
Q6Vect32C Q6Vect32C::B2(signed char b);
Q6Vect32C Q6Vect32C::B3(signed char b);
```

13.2.4 Vector access (64-bit)

To support efficient coding of 64-bit vector operations, LLVM provides intrinsics which are used to perform the following tasks:

- Read or write double-words, words, half-words, or bytes in 64-bit vectors
- Construct 64-bit vectors from double-words, words, half-words, or bytes

The vector intrinsics are accessed by including the header file `hexagon_types.h`. 64-bit vectors are defined as variables of type `Q6Vect64`.

The following example shows how the 64-bit vector intrinsics are used:

```
#include <hexagon_types.h>
extern int foo(unsigned char, Q6Vect64, Q6Vect64);

int bar()
{
    short w=1, x=3, y=2, z=1;
    int t=1;
    unsigned char b;
    Q6Vect64 v, rslt;

    // construct 64-bit vector from four half-words
    v = Q6V64_CREATE_H(w, x, y, z);

    // read low-order byte of 64-bit vector
    b = Q6V64_GET_UB0(v);

    // write high-order word of 64-bit vector
    // return modified vector, v remains unchanged
    rslt = Q6V64_PUT_W1(v, t);

    return foo(b, v, rslt);
}
```

Figure 13-2 shows how double-words, words, half-words, and bytes are accessed.

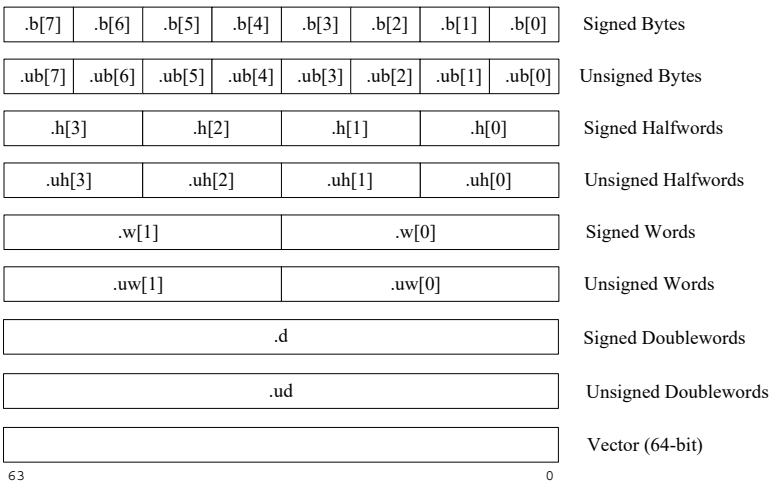

Figure 13-2 Vector access (64-bit)

13.2.4.1 Construct vector

Construct a 64-bit vector from the specified values and return it as a result value.

```
Q6Vect64 Q6V64_CREATE_D(long long d);
Q6Vect64 Q6V64_CREATE_W(int w1, int w0);
Q6Vect64 Q6V64_CREATE_H(short h3, short h2,
                        short h1, short h0);
Q6Vect64 Q6V64_CREATE_B(char b7, char b6, char b5, char b4,
                        char b3, char b2, char b1, char b0);
```

13.2.4.2 Extract double-word

Extract double-word (either signed or unsigned) from a 64-bit vector and return it as a result value.

```
long long Q6V64_GET_D(Q6Vect64 v);

unsigned long long Q6V64_GET_UD(Q6Vect64 v);
```

13.2.4.3 Extract word

Extract the specified word (either signed or unsigned) from a 64-bit vector and return it as a result value.

```
int Q6V64_GET_W0(Q6Vect64 v);
int Q6V64_GET_W1(Q6Vect64 v);

unsigned int Q6V64_GET_UW0(Q6Vect64 v);
unsigned int Q6V64_GET_UW1(Q6Vect64 v);
```

13.2.4.4 Extract half-word

Extract the specified half-word (either signed or unsigned) from a 64-bit vector and return it as a result value.

```
short Q6V64_GET_H0(Q6Vect64 v);
short Q6V64_GET_H1(Q6Vect64 v);
short Q6V64_GET_H2(Q6Vect64 v);
short Q6V64_GET_H3(Q6Vect64 v);

unsigned short Q6V64_GET_UH0(Q6Vect64 v);
unsigned short Q6V64_GET_UH1(Q6Vect64 v);
unsigned short Q6V64_GET_UH2(Q6Vect64 v);
unsigned short Q6V64_GET_UH3(Q6Vect64 v);
```

13.2.4.5 Extract byte

Extract the specified byte (either signed or unsigned) from a 64-bit vector and return it as a result value.

```
signed char Q6V64_GET_B0(Q6Vect64 v);
signed char Q6V64_GET_B1(Q6Vect64 v);
signed char Q6V64_GET_B2(Q6Vect64 v);
signed char Q6V64_GET_B3(Q6Vect64 v);
signed char Q6V64_GET_B4(Q6Vect64 v);
signed char Q6V64_GET_B5(Q6Vect64 v);
signed char Q6V64_GET_B6(Q6Vect64 v);
signed char Q6V64_GET_B7(Q6Vect64 v);

unsigned char Q6V64_GET_UB0(Q6Vect64 v);
unsigned char Q6V64_GET_UB1(Q6Vect64 v);
unsigned char Q6V64_GET_UB2(Q6Vect64 v);
unsigned char Q6V64_GET_UB3(Q6Vect64 v);
unsigned char Q6V64_GET_UB4(Q6Vect64 v);
unsigned char Q6V64_GET_UB5(Q6Vect64 v);
unsigned char Q6V64_GET_UB6(Q6Vect64 v);
unsigned char Q6V64_GET_UB7(Q6Vect64 v);
```

13.2.4.6 Set double-word

Assign double-word value to a 64-bit vector and return the modified vector as a result value.

```
Q6Vect64 Q6V64_PUT_D(Q6Vect64 v, long long val);
```

13.2.4.7 Set word

Assign a value to the specified word in a 64-bit vector and return the modified vector as a result value.

```
Q6Vect64 Q6V64_PUT_W0(Q6Vect64 v, int val);
Q6Vect64 Q6V64_PUT_W1(Q6Vect64 v, int val);
```

13.2.4.8 Set half-word

Assign a value to the specified half-word in a 64-bit vector and return the modified vector as a result value.

```
Q6Vect64 Q6V64_PUT_H0(Q6Vect64 v, short val);
Q6Vect64 Q6V64_PUT_H1(Q6Vect64 v, short val);
Q6Vect64 Q6V64_PUT_H2(Q6Vect64 v, short val);
Q6Vect64 Q6V64_PUT_H3(Q6Vect64 v, short val);
```

13.2.4.9 Set byte

Assign a value to the specified byte in a 64-bit vector and return the modified vector as a result value.

```
Q6Vect64 Q6V64_PUT_B0(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B1(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B2(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B3(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B4(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B5(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B6(Q6Vect64 v, char val);
Q6Vect64 Q6V64_PUT_B7(Q6Vect64 v, char val);
```

13.2.5 Vector access (64-bit; C++)

For C++ programs the vector access library also implements the 64-bit vector access operations (Section 13.2.4) as member functions of the class `Q6Vect64C`.

13.2.5.1 Constructors

Construct a 64-bit vector from the specified values.

```
Q6Vect64C::Q6Vect64C(long long d = 0);
Q6Vect64C::Q6Vect64C(int w1, int w0);
Q6Vect64C::Q6Vect64C(short h3, short h2, short h1, short h0);
Q6Vect64C::Q6Vect64C(signed char b7, signed char b6,
                     signed char b5, signed char b4,
                     signed char b3, signed char b2,
                     signed char b1, signed char b0);

Q6Vect64C::Q6Vect64C(const Q6Vect64C & v);
Q6Vect64C::Q6Vect64C& operator = (const Q6Vect64C & v);
```

13.2.5.2 Extract double-word

Extract double-word (either signed or unsigned) from a 64-bit vector and return it as a result value.

```
long long Q6Vect64C::D(void);

unsigned long long Q6Vect64C::UD(void);
```

13.2.5.3 Extract word

Extract the specified word (either signed or unsigned) from a 64-bit vector and return it as a result value.

```
int Q6Vect64C::W0(void);
int Q6Vect64C::W1(void);

unsigned int Q6Vect64C::UW0(void);
unsigned int Q6Vect64C::UW1(void);
```

13.2.5.4 Extract half-word

Extract the specified half-word (either signed or unsigned) from a 64-bit vector and return it as a result value.

```
short Q6Vect64C::H0(void);
short Q6Vect64C::H1(void);
short Q6Vect64C::H2(void);
short Q6Vect64C::H3(void);

unsigned short Q6Vect64C::UH0(void);
unsigned short Q6Vect64C::UH1(void);
unsigned short Q6Vect64C::UH2(void);
unsigned short Q6Vect64C::UH3(void);
```

13.2.5.5 Extract byte

Extract the specified byte (either signed or unsigned) from a 64-bit vector and return it as a result value.

```
signed char Q6Vect64C::B0(void);
signed char Q6Vect64C::B1(void);
signed char Q6Vect64C::B2(void);
signed char Q6Vect64C::B3(void);
signed char Q6Vect64C::B4(void);
signed char Q6Vect64C::B5(void);
signed char Q6Vect64C::B6(void);
signed char Q6Vect64C::B7(void);

unsigned char Q6Vect64C::UB0(void);
unsigned char Q6Vect64C::UB1(void);
unsigned char Q6Vect64C::UB2(void);
unsigned char Q6Vect64C::UB3(void);
unsigned char Q6Vect64C::UB4(void);
unsigned char Q6Vect64C::UB5(void);
unsigned char Q6Vect64C::UB6(void);
unsigned char Q6Vect64C::UB7(void);
```

13.2.5.6 Set double-word

Assign double-word value to a 64-bit vector and return the modified vector as a result value.

```
Q6Vect64C Q6Vect64C::D(long long d);
```

13.2.5.7 Set word

Assign a value to the specified word in a 64-bit vector and return the modified vector as a result value.

```
Q6Vect64C Q6Vect64C::W0(int w);
Q6Vect64C Q6Vect64C::W1(int w);
```

13.2.5.8 Set half-word

Assign a value to the specified half-word in a 64-bit vector and return the modified vector as a result value.

```
Q6Vect64C Q6Vect64C::H0(short h);
Q6Vect64C Q6Vect64C::H1(short h);
Q6Vect64C Q6Vect64C::H2(short h);
Q6Vect64C Q6Vect64C::H3(short h);
```

13.2.5.9 Set byte

Assign a value to the specified byte in a 64-bit vector and return the modified vector as a result value.

```
Q6Vect64C Q6Vect64C::B0(signed char b);
Q6Vect64C Q6Vect64C::B1(signed char b);
Q6Vect64C Q6Vect64C::B2(signed char b);
Q6Vect64C Q6Vect64C::B3(signed char b);
Q6Vect64C Q6Vect64C::B4(signed char b);
Q6Vect64C Q6Vect64C::B5(signed char b);
Q6Vect64C Q6Vect64C::B6(signed char b);
Q6Vect64C Q6Vect64C::B7(signed char b);
```

13.2.6 Circular addressing

To support data stream processing the Hexagon processor supports circular buffer addressing. Because it is difficult for compilers to generate efficient code for this address mode, the following intrinsics have been defined to efficiently support circular addressing:

```
Q6_circ_load_update_XX
Q6_circ_store_update_XX
```

These intrinsics perform both the circular buffer access (load or store) and the updating of the circular buffer pointer. The pointer is updated with a constant increment/decrement value.

NOTE: The `_XX` suffix specifies the type of data accessed by the intrinsic (such as byte).

The circular addressing intrinsics are accessed by including the `hexagon_protos.h` library file.

For more information on circular addressing, see the appropriate *Qualcomm Hexagon Programmer's Reference Manual* (listed in Appendix B).

Example 1

The following example shows how the circular addressing intrinsics are used.

```
#include <stdio.h>
#include <hexagon_protos.h>

void circular(int input_array[],
              int output_array[],
              int element_load[])
{
   int *p0 = &input_array[20];
```

```
   int *p1 = output_array;
   int i;

   for (i = 0; i < 150; i++)
   {
      Q6_circ_load_update_W(element_load[i], p0, 1, 150, 8);
   }

   for (i = 0; i < 160; i++)
   {
      Q6_circ_store_update_W(i, p1, 1, 150, 8);
   }
}

#define N 150
int g_input_array[N] __attribute__((aligned(1024)));
int g_output_array[N] __attribute__((aligned(1024)));
//circular buffers should be global or static

int main()
{
   int i;
   int element_load[150];
   for (i = 0; i < N; i++)
   {
      g_input_array[i] = i;
   }

   circular(g_input_array, g_output_array, element_load);

   printf("\n-----------element_load------------\n");
   for (i = 0; i < 150; i++)
   {
      printf("%d; ", element_load[i]);
      if((i%15)==14) printf("\n");
   }
   printf("\n-----------g_output_array------------\n");
   for (i = 0; i < 150; i++)
   {
      printf("%d; ", g_output_array[i]);
      if((i%15)==14)printf("\n");
   }
   printf("\n-----------end------------\n");
}
```

- In the first for-loop of function `circular`, pointer `p0` is used to load array `element_load` with the contents of circular buffer `input_array`.
- `input_array` is initialized with the values 0 to 149, and 150 circular loads are performed on it; however, because the circular loads start from the 21st element in `input_array`, the values loaded into `element_load` initially range from 20 to 149, then (wrapping around to the front of `input_array`) continue from 0 to 19, as shown in Example 2.

- In the second for-loop of function `circular`, pointer `p1` is used to store the current value of the loop variable into circular buffer `output_array`.
- The circular stores write 160 elements, which is 10 more than the circular buffer length. Because of this, the circular stores first write the values 0 to 149 to `output_array` (starting at index 0), then wrap around to the front of `output_array` and overwrite its first 10 elements (index 0 to 9) with the values 150 to 159, as shown in Example 2.

Example 2

Following is an example of the program output generated by the program in Example 1.

```
----------element_load-----------
20; 21; 22; 23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33; 34;
35; 36; 37; 38; 39; 40; 41; 42; 43; 44; 45; 46; 47; 48; 49;
50; 51; 52; 53; 54; 55; 56; 57; 58; 59; 60; 61; 62; 63; 64;
65; 66; 67; 68; 69; 70; 71; 72; 73; 74; 75; 76; 77; 78; 79;
80; 81; 82; 83; 84; 85; 86; 87; 88; 89; 90; 91; 92; 93; 94;
95; 96; 97; 98; 99; 100; 101; 102; 103; 104; 105; 106; 107; 108; 109;
110; 111; 112; 113; 114; 115; 116; 117; 118; 119; 120; 121; 122; 123; 124;
125; 126; 127; 128; 129; 130; 131; 132; 133; 134; 135; 136; 137; 138; 139;
140; 141; 142; 143; 144; 145; 146; 147; 148; 149; 0; 1; 2; 3; 4;
5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19;

----------g_output_array-----------
150; 151; 152; 153; 154; 155; 156; 157; 158; 159; 10; 11; 12; 13; 14;
15; 16; 17; 18; 19; 20; 21; 22; 23; 24; 25; 26; 27; 28; 29;
30; 31; 32; 33; 34; 35; 36; 37; 38; 39; 40; 41; 42; 43; 44;
45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55; 56; 57; 58; 59;
60; 61; 62; 63; 64; 65; 66; 67; 68; 69; 70; 71; 72; 73; 74;
75; 76; 77; 78; 79; 80; 81; 82; 83; 84; 85; 86; 87; 88; 89;
90; 91; 92; 93; 94; 95; 96; 97; 98; 99; 100; 101; 102; 103; 104;
105; 106; 107; 108; 109; 110; 111; 112; 113; 114; 115; 116; 117; 118; 119;
120; 121; 122; 123; 124; 125; 126; 127; 128; 129; 130; 131; 132; 133; 134;
135; 136; 137; 138; 139; 140; 141; 142; 143; 144; 145; 146; 147; 148; 149;

----------end-----------
```

13.2.6.1 Circular load

Load values from the circular buffer and update the circular buffer pointer.

```
Q6_circ_load_update_XX( <destination variable>,
                        <buffer location pointer>,
                        <increment/decrement value>,
                        <buffer length>,
                        <K1 value> );
```

- The `_XX` suffix specifies the data type of the intrinsic (Section 13.2.6.3).
- The destination and buffer location pointer must be L-values.
- The increment/decrement value must be specified as a signed constant value, with one array element considered as unity (in conformance with ANSI C pointer increments). The value must be in the range -8 ... 7, and must have an absolute value less than the buffer length.

- The buffer length specifies the number of data elements in the buffer, and can be specified by either a constant or a variable. The length can be between 3 and (128K-1) bytes.
- The K1 value determines the maximum possible buffer length (that is, 2^{K1}-1 elements, where 2^{K1-1} <= buffer length <= 2^{K1}-1).
- The intrinsic behavior is undefined if the specified K1 value is incompatible with the circular buffer length.
- If the circular load intrinsic is used only to increment the circular buffer pointer (that is, a *dummy* load), the destination variable must be declared `volatile` to prevent the compiler from optimizing away the intrinsic call.

Example

For an example of a circular load, consider the variable `input_array`, which is a circular buffer of 150 integer elements. Access from the first element (`input_array[0]`) to the 150th element (`input_array[149]`) is to be circular, with access initially starting from the 21st element (`input_array[20]`).

The pointer initialization statement would appear as follows:

```
int *p0 = &input_array[20];
```

This C statement initializes the buffer location pointer (which will be used to perform circular access on the buffer) to the address of the 21st element of `input_array`.

The circular load statement would appear as follows:

```
Q6_circ_load_update_W(x, p0, 1, 150, 8);
```

This C statement loads the data currently pointed by the pointer variable `p0` into the integer variable `x`. The load intrinsic circularly post-increments the pointer variable by 1 (that is, by one unit of the `input_array` base type (`int`)).

The circular buffer in the example has the following properties:

- buffer start address = `&input_array[0]`, because the current location that `p0` points to (that is, element `input_array[20]`) has element `input_array[0]` as the closest memory location that is aligned to 1024 bytes (the alignment requirement for circular buffer `input_array`. See Section 13.2.6.4), and that appears at or before it.
- buffer length =150 integer words = 150 × 4 bytes

If the increment/decrement value is a positive number, a circular post-increment is performed. If it is a negative number, a circular post-decrement is performed. If the circular pointer increment/decrement is not required at the access point, the increment/decrement value can be specified as zero:

```
Q6_circ_load_update_W(x, p0, 0, 150, 8)
```

The buffer length can be a variable, but must have a known upper bound. In this case, the K1 value is based on the upper bound of the buffer length.

For example, `Q6_circ_load_update_W(x, p0, 1, y, 8)` specifies that the buffer length variable `y` can specify a maximum length of 255 elements (since the K1 value is 8).

The K1 value field is redundant if the buffer length is constant, but it still must be assigned a valid K1 value.

NOTE: Substituting `element_load[i]` for the `x` parameter makes this example identical to part of the program in Example 1.

13.2.6.2 Circular store

Store value to circular buffer and update circular buffer pointer.

```
Q6_circ_store_update_XX( <source expression>,
                         <buffer location pointer>,
                         <increment/decrement value>,
                         <buffer length>,
                         <K1 value> );
```

- The `_XX` suffix specifies the data type of the intrinsic (Section 13.2.6.3).
- The source expression specifies the value to be stored.
- The buffer location pointer must be an L-value.
- The increment/decrement value must be specified as a signed constant value, with one array element considered as unity (in conformance with ANSI C pointer increments). The value must be in the range -8 ... 7, and must have an absolute value less than the buffer length.
- The buffer length specifies the number of data elements in the buffer, and can be specified by either a constant or a variable. The length can be between 3 and (128K-1) bytes.
- The K1 value determines the maximum possible buffer length (that is, 2^{K1}-1 elements, where 2^{K1-1} <= buffer length <= 2^{K1}-1).

NOTE: The intrinsic behavior is undefined if the specified K1 value is incompatible with the circular buffer length.

Example

For an example of a circular store, consider the variable `output_array`, which is a circular buffer of 150 integer elements. Access from the first element (`output_array[0]`) to the 150th element (`output_array[149]`) is to be circular, with access initially starting from the first element (`output_array[0]`).

The pointer initialization statement would appear as follows:

```
int *p1 = output_array[0];
```

This C statement initializes the buffer location pointer (that will be used to perform circular access on the buffer) to the address of the first element of `output_array`.

The circular store statement would appear as follows:

```
Q6_circ_store_update_W(x, p1, 1, 150, 8);
```

This C statement loads the data currently pointed by the pointer variable `p1` into the integer variable `x`. The store intrinsic circularly post-increments the pointer variable by 1 (that is, by one unit of the `output_array` base type (`int`)).

The circular buffer in the example has the following properties:

- Buffer start address = `&output_array[0]`, because the current location that `p1` points to (element `output_array[0]`) has element `output_array[0]` as the closest memory location that is aligned to 1024 bytes (the alignment requirement for circular buffer `output_array`; see Section 13.2.6.4), and that appears at or before it.
- Buffer length =150 integer words = 150 × 4 bytes

If the increment/decrement value is a positive number a circular post-increment is performed. If it is a negative number a circular post-decrement is performed.

If the circular pointer increment/decrement is not needed at the access point, then the increment/decrement value can be specified as 0:

```
Q6_circ_store_update_W(x, p1, 0, 150, 8)
```

The buffer length can be a variable, but must have a known upper bound. In this case, the K1 value is based on the upper bound of the buffer length.

For example, `Q6_circ_store_update_W(x, p1, 1, y, 8)` specifies that the buffer length variable `y` can specify a maximum length of 255 elements (since the K1 value is 8).

The K1 value field is redundant if the buffer length is constant, but still must be assigned a valid K1 value.

NOTE: Substituting `i` for the `x` parameter makes the preceding example identical to part of the program in Example 1.

13.2.6.3 Data type specification

The Hexagon processor can perform memory accesses with different data types.

The data type (size, signed/unsigned, and byte positions) is an important aspect in determining the correct assembly instruction for circular access. Therefore a data type suffix is appended to the name of the circular load/store intrinsics:

- The suffix B, H, UB, UBH, UH, W, or D is appended to the name of the circular load intrinsic.
- The suffix B, HL, HH, W, or D is appended to the name of the circular store intrinsic.

Section 13-1 and Section 13-2 list the names of the specific circular load/store intrinsics.

Table 13-1 Circular load intrinsics

Intrinsic name	Memory size
`Q6_circ_load_update_UB`	8-bit, with zero extension to 32 bits
`Q6_circ_load_update_B`	8-bit, with sign extension to 32 bits
`Q6_circ_load_update_UH`	16-bit, with zero extension to 32 bits
`Q6_circ_load_update_H`	16-bit, with sign extension to 32 bits
`Q6_circ_load_update_UBH`	Currently unused
`Q6_circ_load_update_W`	32-bit, all bits loaded
`Q6_circ_load_update_D`	64-bit, all bits loaded

Two versions of the UBH data type exist: one returns a 32-bit result, the other a 64-bit result. This would normally present a problem for C. However, because the UBH types specify vector operations and circular/bit-reversed addressing cannot be vectorized, the UBH data type is ignored in this version.

Table 13-2 Circular store intrinsics

Intrinsic name	Memory size
`Q6_circ_store_update_B`	8-bit
`Q6_circ_store_update_HL`	16-bit, store lower half (bits 15:0)
`Q6_circ_store_update_HH`	16-bit, store upper half (bits 31:16)
`Q6_circ_store_update_W`	32-bit
`Q6_circ_store_update_D`	64-bit

13.2.6.4 Circular buffer alignment

To support circular addressing circular buffers must be properly aligned in memory. Alignment is performed with the `aligned` attribute. For example:

```
int circ_buffer[150] __attribute__((aligned(1024)));
```

A circular buffer is properly aligned when its starting byte address is an integral multiple of the closest power-of-2 value larger than the buffer size (in bytes). For example, the circular buffer declared in the example is aligned to 1024 bytes because the buffer size is 600 bytes (150 integer words $\times$ 4 bytes), and 1024 is the next power of 2 above 600.

The alignment can be expressed in terms of K1 values (Section 13.2.6.1):

- 2^{K1+3} for 8-byte data types (such as `long long`)
- 2^{K1+2} for 4-byte data types (such as `integer`)
- 2^{K1+1} for 2-byte data types (such as `short`)
- 2^{K1} for 1-byte data types (such as `char`)

Circular buffers can be set up on portions (or *slices*) of an array as long as the buffers themselves are properly aligned. For example, consider the following array:

```
int big_array[1600] __attribute__((aligned(1024)));
```

Given the alignment of `big_array`, it is possible to allocate circular buffers of length 150 integer words (600 bytes) at each location in `big_array` that has an alignment equal to 1024 bytes, as shown in Figure 13-3.

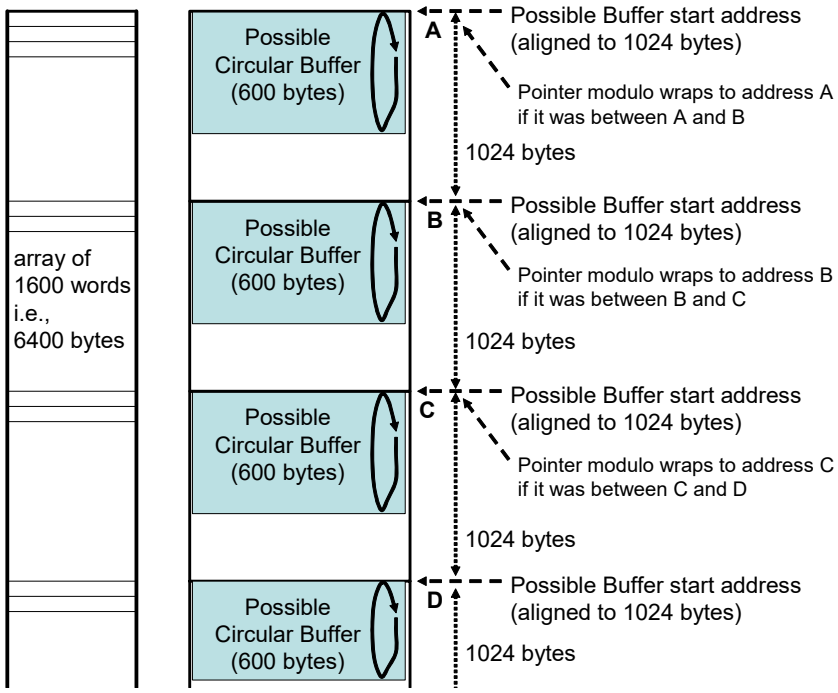

Figure 13-3 Buffer alignment

In this case, when multiple circular buffer start addresses potentially exist within an array, an arbitrary pointer variable used for circular buffer access will always wrap around to the most recently preceding circular buffer start address, as shown in Figure 13-3. Therefore circular buffers are feasible within an array only under the following conditions:

- The buffer start address is one of the possible buffer start addresses
- The buffer size (in bytes) is less than the buffer alignment value

NOTE: Because variables allocated in the stack should have an alignment of 8 bytes or less, circular buffers (in most cases) must be declared either as global variables or (if declared locally within a function) as static.

13.2.6.5 Special issues

Exceptional cases may arise if ordinary pointer arithmetic is performed on the C pointer variable used for circular accesses. Whenever ordinary pointer arithmetic is performed on the pointer variable, the pointer update no longer occurs in the modulo-wrap around fashion.

If the pointer arithmetic (such as p = p + 256, where p is a pointer to `int` on a buffer of 600 bytes, as shown in Figure 13-4) adds a value that makes the pointer point to a location outside the intended circular buffer, the subsequent circular load/store would access data from a new invalid circular buffer.

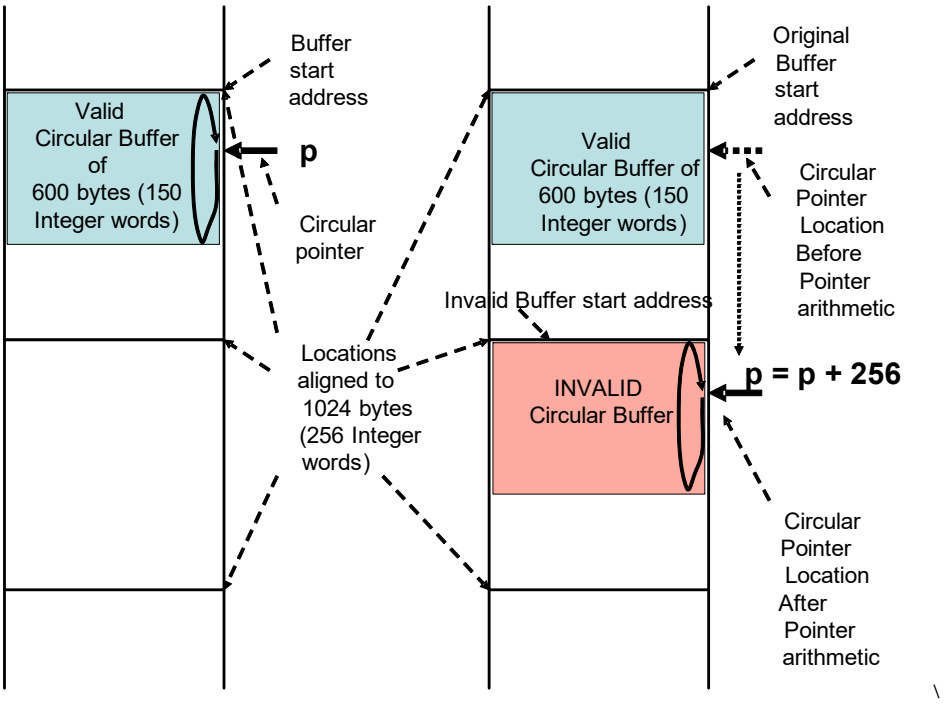

Figure 13-4 Invalid circular buffer access

13.2.6.6 Mapping intrinsics to assembly code

The intrinsic `Q6_circ_load_update_W(x,p,8,24,5)` translates to the following assembly code sequence:

```
R1.H = #0x5000;
R1.L = #96;
M1 = R1;
R0 = MEMW(R2++#32):circ(M1); // Word load with post-update of 32
```

This example sets up a 24 × 4 byte circular buffer. The length must be set to 96. The closest power-of-2 size larger than 96 is 128. The K1 value is set to 5.

The Modifier register used is M1. The base address register used is R2. However, any other allowed modify registers and address registers can be used, respectively.

The circular buffer must be aligned on a 128-byte boundary.

13.2.7 Bit-reversed addressing

To support Viterbi encoding and fast Fourier transforms (FFT) the Hexagon processor supports bit-reversed addressing. Because it is difficult for compilers to generate efficient code for this address mode, the following intrinsics have been defined to efficiently support bit-reversed addressing:

- `Q6_bitrev_load_update_XX`
- `Q6_bitrev_store_update_XX`

These intrinsics perform both the bit-reversed buffer access (load or store) and the updating of the bit-reversed buffer pointer.

The bit-reversed addressing intrinsics are accessed by including the `hexagon_protos.h` library file.

Example 1

This example shows how the intrinsics are used.

```
#include <stdio.h>
#include <hexagon_protos.h>

void bitrev(int input_array[],
            int output_array[],
            int element_load[])
{
   int *p0 = input_array;
   int *p1 = output_array;
   int i;

   for (i = 0; i < 256; i++)
   {
      Q6_bitrev_load_update_W(element_load[i], p0, 8);
   }


   for (i = 0; i < 256; i++)
   {
      Q6_bitrev_store_update_W(element_load[i], p1, 8);
   }
}

#define N 256
int g_input_array[N] __attribute__((aligned(1<<16)));
int g_output_array[N] __attribute__((aligned(1<<16)));
//bit-reversed buffers should be global or static

int main()
{
   int i;
   int element_load[256];
   for (i = 0; i < N; i++)
   {
      g_input_array[i] = i;
   }
```

```
    bitreverse(g_input_array, g_output_array, element_load);

    printf("\n-----------element_load------------\n");
    for (i = 0; i < 256; i++)
    {
       printf("%d; ", element_load[i]);
       if((i%16)==15) printf("\n");
    }
    printf("\n-----------output_array------------\n");
    for (i = 0; i < 256; i++)
    {
       printf("%d; ", g_output_array[i]);
       if((i%16)==15)printf("\n");
    }
    printf("\n-----------end------------\n");
}
```

In the first for-loop of function `bitreverse` (shown in Example 1), pointer `p0` is used to load array `element_load` with the bit-reversed contents of array `input_array`.

`input_array` was initialized with the values 0 to 255, so `element_load` contains the bit-reversed values of the sequence 0 to 255, as shown in Example 2.

In the second for-loop of `bitreverse`, pointer `p1` is used to store the bit-reversed contents of `element_load` into array `output_array`.

Because the contents of `element_load` are already bit-reversed, `output_array` ends up containing the original sequence 0 to 255, as shown in Example 2.

Example 2

This example shows the program output generated by the program in Example 1.

```
-----------element_load------------
0; 128; 64; 192; 32; 160; 96; 224; 16; 144; 80; 208; 48; 176; 112; 240;
8; 136; 72; 200; 40; 168; 104; 232; 24; 152; 88; 216; 56; 184; 120; 248;
4; 132; 68; 196; 36; 164; 100; 228; 20; 148; 84; 212; 52; 180; 116; 244;
12; 140; 76; 204; 44; 172; 108; 236; 28; 156; 92; 220; 60; 188; 124; 252;
2; 130; 66; 194; 34; 162; 98; 226; 18; 146; 82; 210; 50; 178; 114; 242;
10; 138; 74; 202; 42; 170; 106; 234; 26; 154; 90; 218; 58; 186; 122; 250;
6; 134; 70; 198; 38; 166; 102; 230; 22; 150; 86; 214; 54; 182; 118; 246;
14; 142; 78; 206; 46; 174; 110; 238; 30; 158; 94; 222; 62; 190; 126; 254;
1; 129; 65; 193; 33; 161; 97; 225; 17; 145; 81; 209; 49; 177; 113; 241;
9; 137; 73; 201; 41; 169; 105; 233; 25; 153; 89; 217; 57; 185; 121; 249;
5; 133; 69; 197; 37; 165; 101; 229; 21; 149; 85; 213; 53; 181; 117; 245;
13; 141; 77; 205; 45; 173; 109; 237; 29; 157; 93; 221; 61; 189; 125; 253;
3; 131; 67; 195; 35; 163; 99; 227; 19; 147; 83; 211; 51; 179; 115; 243;
11; 139; 75; 203; 43; 171; 107; 235; 27; 155; 91; 219; 59; 187; 123; 251;
7; 135; 71; 199; 39; 167; 103; 231; 23; 151; 87; 215; 55; 183; 119; 247;
15; 143; 79; 207; 47; 175; 111; 239; 31; 159; 95; 223; 63; 191; 127; 255;
```

```
----------output_array-----------
0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15;
16; 17; 18; 19; 20; 21; 22; 23; 24; 25; 26; 27; 28; 29; 30; 31;
32; 33; 34; 35; 36; 37; 38; 39; 40; 41; 42; 43; 44; 45; 46; 47;
48; 49; 50; 51; 52; 53; 54; 55; 56; 57; 58; 59; 60; 61; 62; 63;
64; 65; 66; 67; 68; 69; 70; 71; 72; 73; 74; 75; 76; 77; 78; 79;
80; 81; 82; 83; 84; 85; 86; 87; 88; 89; 90; 91; 92; 93; 94; 95;
96; 97; 98; 99; 100; 101; 102; 103; 104; 105; 106; 107; 108; 109; 110; 111;
112; 113; 114; 115; 116; 117; 118; 119; 120; 121; 122; 123; 124; 125; 126; 127;
128; 129; 130; 131; 132; 133; 134; 135; 136; 137; 138; 139; 140; 141; 142; 143;
144; 145; 146; 147; 148; 149; 150; 151; 152; 153; 154; 155; 156; 157; 158; 159;
160; 161; 162; 163; 164; 165; 166; 167; 168; 169; 170; 171; 172; 173; 174; 175;
176; 177; 178; 179; 180; 181; 182; 183; 184; 185; 186; 187; 188; 189; 190; 191;
192; 193; 194; 195; 196; 197; 198; 199; 200; 201; 202; 203; 204; 205; 206; 207;
208; 209; 210; 211; 212; 213; 214; 215; 216; 217; 218; 219; 220; 221; 222; 223;
224; 225; 226; 227; 228; 229; 230; 231; 232; 233; 234; 235; 236; 237; 238; 239;
240; 241; 242; 243; 244; 245; 246; 247; 248; 249; 250; 251; 252; 253; 254; 255;

----------end------------
```

13.2.7.1 Bit-reversed load

Load value from bit-reversed buffer and update bit-reversed buffer pointer.

```
Q6_bitrev_load_update_XX( <destination variable>,
                          <buffer location pointer,
                          <log2(buffer length> );
```

- The `_XX` suffix specifies the data type of the intrinsic (Section 13.2.7.3).
- The destination and buffer location pointer must be L-values.
- The buffer length specifies the number of data elements in the buffer, and can be specified by either a constant or a variable. The length must be an integral power of 2, with a maximum length of 64K bytes.

Example

For example, consider the following bit-reversed buffer:

```
int input_array[256] __attribute__((aligned(1<<16)));
```

Access from the first element (`input_array[0]`) to the 256th element (`input_array[255]`) is to be bit-reversed, with access initially starting from the first element (`input_array[0]`).

The pointer initialization statement would appear as follows:

```
int *p = &input_array[0];
```

This C statement initializes the buffer location pointer (which will be used to perform bit-reversed access on the buffer) to the address of the first element of `input_array`, with the least-significant 16 bits of the address value bit-reversed.

NOTE: Because `input_array` is aligned on a 64K byte boundary (Section 13.2.7.4) the least-significant 16 bits of the buffer location pointer's initial value are all set to zero. Bit-reversing all zeros yields all zeros; therefore the C statement shown does in fact generate the value described in the preceding paragraph.

The bit-reversed load statement would appear as follows:

```
Q6_bitrev_load_update_W(x,p,8);
```

This C statement loads the data from an address created by bit-reversing the least-significant 16 bits of the buffer location pointer, and then updates the buffer location pointer by post-incrementing it by the fixed value Mt.

The auto-increment value Mt is determined from the size (in bytes) of the bit-reversed buffer. In the example, the load intrinsic post-increments by the following amount:

$$Mt = 1 << (16 - \log_2(256 \times 4)) = 1 << (16 - 10) = 1 << 6 = 64 \text{ bytes}$$

NOTE: It is not possible to perform bit-reversed access without changing the buffer location pointer.

Normal loads and stores cannot be performed using the buffer location pointer when the buffer length is large enough that the value of Mt is less than the byte alignment required for a normal load or store. Otherwise the buffer location pointer would be incremented by the value in Mt, which in turn could lead to address values that work for the bit-reversed loads and stores but not for normal loads and stores (which are subject to the alignment restrictions).

For example, a normal load or store on a `long long` data element requires an address alignment of 8 bytes. To ensure that this alignment always exists in a buffer location pointer, the bit-reversed buffer length must not be greater than 1024 `long long` data elements:

$$Mt = 1 << (16 - \log_2(1024 \times 8)) = 1 << (16 - 13) = 1 << 3 = 8 \text{ bytes}$$

13.2.7.2 Bit-reversed store

Store value to bit-reversed buffer and update bit-reversed buffer pointer.

```
Q6_bitrev_store_update_XX( <source expression>,
                           <buffer location pointer>,
                           <log2(buffer length> );
```

- The `_XX` suffix specifies the data type of the intrinsic (Section 13.2.7.3).
- The source expression specifies the value to be stored.
- The buffer location pointer must be an L-value.
- The buffer length specifies the number of data elements in the buffer, and can be specified by either a constant or a variable. The length must be an integral power of 2, with a maximum length of 64K bytes.

Example

For example, consider the following bit-reversed buffer:

```
int output_array[256] __attribute__((aligned(1<<16)));
```

Access from the first element (`output_array[0]`) to the 256th element (`output_array[255]`) is to be bit-reversed, with access initially starting from the first element (`output_array[0]`).

The pointer initialization statement would appear as follows:

```
int *p = &output_array[0];
```

This C statement initializes the buffer location pointer (which will be used to perform bit-reversed access on the buffer) to the address of the first element of `output_array`, with the least-significant 16 bits of the address value bit-reversed.

NOTE: Because `output_array` is aligned on a 64K byte boundary (Section 13.2.7.4) the least-significant 16 bits of the buffer location pointer's initial value are all set to zero. Bit-reversing all zeros yields all zeros; therefore the C statement shown does in fact generate the value described in the preceding paragraph.

The bit-reversed store statement would appear as follows:

```
Q6_bitrev_store_update_W(x,p,8);
```

This C statement stores the data to an address created by bit-reversing the least-significant 16 bits of the buffer location pointer, and then updates the buffer location pointer by post-incrementing it by the fixed value Mt.

The auto-increment value Mt is determined from the size (in bytes) of the bit-reversed buffer. In the example, the store intrinsic post-increments by the following amount:

```
Mt = 1 << (16 - log2(256 x 4)) = 1 << (16 -10) = 1 << 6 = 64 bytes
```

NOTE: It is not possible to perform bit-reversed access without changing the buffer location pointer.

Normal loads and stores cannot be performed using the buffer location pointer when the buffer length is large enough that the value of Mt is less than the byte alignment required for a normal load or store. For more information see Section 13.2.7.1.

13.2.7.3 Data type specification

The Hexagon processor can perform memory accesses of different data types.

The data type (size, signed/unsigned, and byte positions) is an important aspect in determining the correct assembly instruction for bit-reversed access. Therefore a data type suffix is appended to the name of the bit-reversed load/store intrinsics:

- The suffix B, H, UB, UBH, UH, W, or D is appended to the name of the bit-reversed load intrinsic.
- The suffix B, HL, HH, W, or D is appended to the name of the bit-reversed store intrinsic.

Table 13-3 and Table 13-4 list the names of the specific bit-reversed load/store intrinsics.

Table 13-3 Bit-reversed load intrinsics

Intrinsic name	Memory size
`Q6_bitrev_load_update_UB`	8-bit, with zero extension to 32 bits
`Q6_bitrev_load_update_B`	8-bit, with sign extension to 32 bits
`Q6_bitrev_load_update_UH`	16-bit, with zero extension to 32 bits
`Q6_bitrev_load_update_H`	16-bit, with sign extension to 32 bits
`Q6_bitrev_load_update_UBH`	Currently unused
`Q6_bitrev_load_update_W`	32-bit, all bits loaded
`Q6_bitrev_load_update_D`	64-bit, all bits loaded

Two versions of the UBH data type exist: one returns a 32-bit result and the other returns a 64-bit result. This would typically present a problem for C. However, because the UBH types specify vector operations and circular/bit-reversed addressing cannot be vectorized, the UBH data type is ignored in this version.

Table 13-4 Bit-reversed store intrinsics

Intrinsic name	Memory size
`Q6_bitrev_store_update_B`	8-bit
`Q6_bitrev_store_update_HL`	16-bit, store in lower half (bits 15:0)
`Q6_bitrev_store_update_HH`	16-bit, store in upper half (bits 31:16)
`Q6_bitrev_store_update_W`	32-bit
`Q6_bitrev_store_update_D`	64-bit

13.2.7.4 Bit-reversed buffer alignment

To support bit-reversed addressing bit-reversed buffers must be properly aligned in memory. Alignment is performed with the `aligned` attribute. For example:

```
int input_array[256] __attribute__((aligned(1024)));
```

A bit-reversed buffer is properly aligned when its starting byte address is aligned to a power of 2 greater than or equal to the buffer size (in bytes). For example, the bit-reversed buffer declared in the example is aligned to 1024 bytes because the buffer size is 1024 bytes (256 integer words $\times$ 4 bytes), and 1024 is an integral power of 2.

The buffer location pointer for a bit-reversed buffer must be initialized so the least-significant 16 bits of the address value are bit-reversed. However, the C language offers no simple way to express a pointer value with its least-significant 16 bits bit-reversed.

NOTE: To simplify the initialization of the bit-reversed pointer, bit-reversed buffers can be aligned to a 64K-byte boundary. This has the advantage of allowing the bit-reversed pointer to be initialized to the base address of the bit-reversed buffer, with no bit-reversing required for the least-significant 16 bits of the pointer value (which are all set to 0 by the 64K alignment).

Because variables allocated in the stack should have an alignment of 8 bytes or less, in most cases bit-reversed buffers need to be declared either as global variables or (if declared locally within a function) as static.

13.2.7.5 Special issues

Exceptional cases might arise if ordinary pointer arithmetic is performed on the C pointer variable used for bit-reversed accesses. If the pointer arithmetic adds a value that makes the pointer point to a location outside the intended bit-reversed buffer, the subsequent bit-reversed load/store accesses data from a new invalid buffer. For more information, see the appropriate *Qualcomm Hexagon Programmer's Reference Manual* (listed in Appendix B).

NOTE: A pointer used for bit-reversed addressing is not a valid pointer for any other purpose. Therefore, performing any pointer arithmetic on it outside of the bit-reversed intrinsics (except for adding or subtracting a multiple of Mt) is inherently not useful and should therefore be avoided.

13.2.7.6 Mapping intrinsics to assembly code

The intrinsic `Q6_bitrev_load_update_W(x,p,4)` translates to the following assembly code sequence:

```
R1 = 512;
M0 = R1;
R0 = MEMW(R2++M0:brev);
```

This sequence example sets up a 32 (2^4) $\times$ 4 byte buffer where bit-reversed access will be performed. The Modifier Register used is M0. The Base Address Register used is R2. The Modifier setting is:

```
M0 = 1 << (16 - log2(32 ´ 4)) = 1 << (16 -7) = 1 << 9 = 512
```

However, any other allowed modify registers and address registers can be used respectively.

Use the alignment language extensions to perform any buffer alignment requirements.

13.2.8 Simulation timer (hexagon_sim_timer.h)

The Hexagon processor simulator supports a timer interface to enable users to collect timing information on the execution of target applications.

The timer interface functions are accessed by including the `hexagon_sim_timer.h` library file. For more information, see the *Qualcomm Hexagon Simulator User Guide* (80-N2040-17).

13.3 HVX-specific library

The HVX-specific library supports the following operations:

- Instruction access
- Vector access
- Vector initialization
- Vector assignment

These operations are supported in the `hvx_hexagon_protos.h` library.

13.3.1 Instruction access

To support efficient coding of the time-intensive sections of a program (without resorting to inline assembly language), LLVM provides intrinsics which are used to directly access HVX instructions.

The instruction intrinsics are accessed by including the `hvx_hexagon_protos.h` library file.

The following example shows how intrinsics are used to initialize two HVX vectors and directly access the ALU-DOUBLE-RESOURCE instruction, `Vdd=vcombine(Vu,Vv)`:

```
#include <hvx_hexagon_protos.h>

int main()
{
    HVX_Vector v1, v2;
    HVX_VectorPair result;

    // initialize vectors v1 and v2
    v1 = Q6_V_vzero();
    v2 = Q6_V_vzero();

    // combine v1 and v2 into vector pair
    result = Q6_W_vcombine_VV(v1,v2);
}
```

Intrinsics are defined for most of the HVX processor instructions. For more information see the appropriate *Qualcomm Hexagon HVX Programmer's Reference Manual* (listed in Appendix B).

13.3.2 Vector access

To support efficient coding of vector operations, LLVM provides intrinsics which are used to read the individual vectors from a vector pair.

The vector access intrinsics are accessed by including the header file `hvx_hexagon_protos.h`.

The following example shows how intrinsics are used to read the low and high vectors from a vector pair:

```
#include <hvx_hexagon_protos.h>

int main()
{
    HVX_Vector v1, v2;
    HVX_VectorPair vp;

    // read low-order vector from vector pair
    v1 = Q6_V_lo_W(vp);

    // read high-order vector from vector pair
    v2 = Q6_V_hi_W(vp);
}
```

Figure 13-5 shows how vectors are accessed in a vector pair.

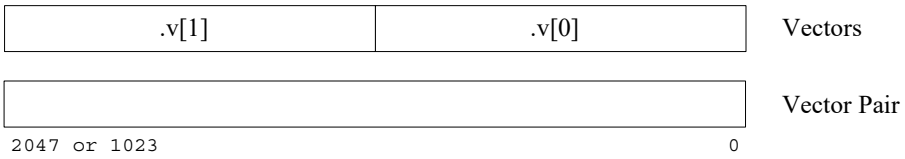

Figure 13-5 HVX vector access

13.3.3 Vector initialization

To support efficient initialization of vectors, LLVM provides a dedicated intrinsic for setting a vector to zero.

The vector initialization intrinsic is accessed by including the header file `hvx_hexagon_protos.h`.

The following example shows how the vector initialization intrinsic is used:

```
#include <hvx_hexagon_protos.h>

int main()
{
    HVX_Vector v1;

    // initialize vector v1 to zero
    v1 = Q6_V_vzero();
}
```

13.3.4 Vector assignment

To support efficient assignment of vectors, LLVM provides a dedicated intrinsic for assigning values between two vector pairs.

The following example shows how the vector assignment intrinsic is used:

```
#include <hvx_hexagon_protos.h>

int main()
{
    HVX_VectorPair vp1, vp2;

    // assign vp1 = vp2
    vp1 = Q6_W_equals_W(vp2);
}
```

13.4 Hexagon intrinsics emulation library

The intrinsics emulation library emulates Hexagon processor instruction intrinsics (Section 13.2) on non-Hexagon processor target platforms. The library (named `libnative`) allows developers to compile a C or C++ program that contains intrinsics into a native Linux or Windows executable. This enables code portability, easier debugging, and a more efficient development work flow for Hexagon processor applications.

The library is included in the tools release in the `Tools/libnative/lib` directory.

13.4.1 Application build procedure

The intrinsics emulation library enables Hexagon processor C or C++ applications to be compiled into Linux or Windows binaries. Therefore, to use the library a native compiler (such as GCC) must be used to build the applications.

To successfully compile the intrinsics, the compiler must include the header file `hexagon_protos.h` from directory `Tools/libnative/include`, and the linker must include `libnative` from `Tools/libnative/lib`.

This is done by passing the following three command line options to the native compiler:

```
-Itools_dir/Tools/libnative/include
   -Ltools_dir//Tools/libnative/lib
   -lnative
   -lm
```

For example, the following command uses GCC to compile an application file `foo.c` which includes calls to Hexagon instruction intrinsics:

```
gcc -Itools_dir/Tools/libnative/include foo.c \
    -Ltools_dir/Tools/libnative/lib -lnative -lm
```

The linker options in this example (`-L`, `-lnative`, `-lm`) must follow any source files specified on the compiler command line.

The `-lm` option is required for floating point support on Linux only.

13.4.2 Restrictions

The intrinsics emulation library has the following restrictions on its use:

- It is not intended for performance modeling, because the emulation does not accurately represent how the Hexagon processor core actually operates. It is thus best suited for tasks such as unit testing, or to facilitate porting of code from a PC platform to the Hexagon processor.
- It supports only the intrinsics that specify instructions directly supported by the Hexagon processor. In particular, it does not support instructions that the assembler maps to an equivalent assembly instruction; for instance, the comparison instruction (`x >= 8`) which is mapped by the assembler to (`x > 7`).

  NOTE: Most assembler-mapped instructions are provided as a convenience for the compiler or assembly language programmer. Some of them enable newer Hexagon processor versions to maintain backwards compatibility with previous versions.

A Acknowledgments

QTI would like to thank the LLVM community for their many contributions to the LLVM Project.

This document includes content derived from the LLVM Project documentation under the terms of the LLVM Release License:

llvm.org/releases/3.8.0/LICENSE.TXT

B References

Title	Number
Qualcomm Technologies, Inc.	
Qualcomm Hexagon Simulator User Guide	80-N2040-17
Qualcomm Hexagon Utilities User Guide	80-N2040-15
Applicable Qualcomm Hexagon Programmer's Reference Manuals:	
■ *Hexagon V62 Programmer's Reference Manual*	80-N2040-36
■ *Qualcomm Hexagon V65 Programmer's Reference Manual*	80-N2040-39
■ *Qualcomm Hexagon V66 Programmer's Reference Manual*	80-N2040-42
■ *Qualcomm Hexagon V67 Programmer's Reference Manual*	80-N2040-45
■ *Qualcomm Hexagon V68 Programmer's Reference Manual*	80-N2040-46
Applicable Hexagon HVX Programmer's Reference Manuals:	
■ *Hexagon V62 HVX Programmer's Reference Manual*	80-N2040-37
■ *Qualcomm Hexagon V65 HVX Programmer's Reference Manual*	80-N2040-41
■ *Qualcomm Hexagon V66 HVX Programmer's Reference Manual*	80-N2040-44
■ *Qualcomm Hexagon V68 HVX Programmer's Reference Manual*	80-N2040-47
C language references	
The C Programming Language (2nd Edition), Brian Kernighan and Dennis Ritchie, Prentice Hall, 1988	ISBN 0-13-110370-9
The C++ Programming Language (3rd Edition), Bjarne Stroustrup, Addison-Wesley, 1997	ISBN 0201889544
Compiler references	
Compilers: Principles, Techniques, and Tools (2nd Edition), Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, Prentice Hall, 2006	ISBN 0-321-48681-1
Engineering a Compiler (2nd Edition), Keith Cooper and Linda Torczon, Morgan Kaufmann, 2011	ISBN-13: 978-0120884780