cādence®

Xtensa® C Application Programmer's Guide

For Xtensa® Tools Version 12

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

Issue Date:10/2015
Build 332171

RG-2015.2
PD-14--0316-10-02

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

Contents

List of Tables

List of Figures

Frontmatter

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- *$* represents your shell prompt, in user-session examples.
- ***literal_input*** indicates literal command-line input.
- *variable* indicates a user parameter.
- *literal_keyword (* in text paragraphs) indicates a literal command keyword.
- *literal_output* indicates literal program output.
- *... output ...* indicates unspecified program output.
- *[optional- variable]* indicates an optional parameter.
- *[variable]* indicates a parameter within literal square-braces.
- *{ variable }* indicates a parameter within literal curly-braces.
- *(variable)* indicates a parameter within literal parentheses.
- | means *OR* .
- *(var1* | *var2)* indicates a required choice between one of multiple parameters.
- *[var1* | *var2]* indicates an optional choice between one of multiple parameters.
- *var1 [, varn]** indicates a list of 1 or more parameters (0 or more repetitions).
- *4* 'b *0010* is a 4-bit value specified in binary.
- *12* ' *o7016* is a 12-bit value specified in octal.
- *10* ' *d4839* is a 10-bit value specified in decimal.
- *32* ' *hff2a* or *32* ' *HFF2A* is a 32-bit value specified in hexadecimal

Terms

- 0x at the beginning of a value indicates a hexadecimal value.
- b means bit.
- B means byte.
- flush is deprecated due to potential ambiguity (it may mean write-back or discard).
- Mb means megabit.
- MB means megabyte.
- PC means program counter.
- word means 4 bytes.

1. Introduction to C/C++ Level Xtensa Processor Programming

Topics:

The Xtensa processor architecture is the first major architecture designed expressly for embedded system-on-a-chip (SOC) applications, instead of traditional desktop computer system applications. It was explicitly designed to allow easy customizing and extension for unparalleled levels of application specific performance, code size and power, while maintaining the ability to efficiently program in C or C++.

This guide describes how to effectively program applications on Xtensa processors at the C and C++ level in the following areas:

- The basics of the tool chain and choice of software configuration options
- Performance tuning methodologies
- How to effectively tune code for high performance and smaller code size
- Choosing hardware configuration options
- How to program using TIE
- How to control memory layout from the C level
- Devices and synchronization

Scope

This guide does not cover OS level programming or assembly level programming typically used in OS level code. See the *Xtensa System Software Reference Manual* and the *Xtensa Microprocessor Programmer's Guide* for more information on those topics. Cadence provides and supports XCC, the Xtensa C Compiler for compiling C and C++ applications on Cadence processors. A more detailed description of XCC is provided in the *Xtensa C and C++ Compiler User's Guide*.

This guide does not describe how to compile and run Xtensa target code using cstub files on an x86 platform. For more information, refer to the *Tensilica Instruction Extension (TIE) Language User' Guide*, chapter "Simulating TIE Instructions in a Native Environment".

1.1 Basic Tool Usage

XCC uses the GCC compiler as a front end, the portion of the compiler responsible for parsing and analyzing the C or C++ input language; XCC and GCC share the same preprocessor, assembler, and linker. Therefore, usage of XCC is very similar to GCC. They share most of the same options and XCC supports most, but not all GCC extensions. More details are in the *Xtensa C and C++ Compiler User's Guide*.

XCC supports, as a beta, using CLANG rather than GCC as a front end. Differences between the two are not described in this document. Please see the *Xtensa C and C++ Compiler User's Guide*.

1.1.1 Differences Between XCC and GCC

C++ Exception Handling

One noticeable difference between XCC and GCC is in the use of C++ exception handling. Exception handling adds substantially to code size, whether or not the exceptions are ever used. Therefore, `xt-xc++` has exception handling turned off by default to avoid penalizing programs that do not use exceptions. This is the opposite of standard GCC, which has exception handling enabled. If your program uses exceptions, you can enable exception handling in `xt-xc++` by supplying the `-fexceptions` option on the command line. Otherwise, `xt-xc++` will produce an error for a source file that contains exception constructs. If you need exception handling in any of your source files, Cadence recommends that you compile all your source files with the `-fexceptions` option.

Common Block

Another difference between XCC and GCC is the default use of the common block. XCC by default allocates uninitialized, non static, global variables in the `.bss` section. This provides slightly better performance, and enforces the ISO C standard requirement that non static global variables be defined (without the `extern` specifier) only once in a program. GCC, on the other hand, by default allocates uninitialized, non static, global variables in the common block. Using the common block allows the linker to merge multiple such variables of the same name (possibly with one that is initialized), and thus allows building code that relies on the pre-ISO/ANSI ability to define the same variable multiple times. Use the standard GNU option `-fcommon` to get the same behavior as GCC.

Unsigned Char

While not a difference between XCC and GCC, the following is sometimes encountered when compiling code that was only tested on a limited set of processor architectures.

The C and C++ language standards do not define whether the data type char is a signed or unsigned quantity. Most implementations choose to make it signed. However, the base Cadence architecture does not have instructions to directly load signed, 8-bit quantities. Instead, an unsigned load followed by a sign extension must be used to implement a signed, 8-bit load. To avoid the additional overhead, in the common case, Cadence defines a char to be unsigned. Most applications work either way. However, some applications assume that char is signed. For such applications, the standard GNU option `-fsigned-char` can be used to change the compiler's default.

1.1.2 Compiler Options

To compile source files into .o object files, use:

```
xt-xcc -c source_files.c
```

and to link the object files into the final executable use:

```
xt-xcc object_files.o -o executable
```

While you can you use the linker, `xt-ld`, directly to link your object files, Cadence recommends that you use the XCC driver to invoke the linker with the appropriate libraries.

Files suffixed with `.c` are assumed to be C language files. Files suffixed with `.C`, `.cc`, `.c++`, `.cpp` or `.cxx` are assumed to be C++ files. You can also use `xt-xc++` to tell the compiler that a file is C++ rather than C. If you use any C++ files, you must link with `xt-xc++`.

By default, XCC performs no optimization and does not generate debug information needed by debuggers to allow you to symbolically debug your application. Therefore by default, your resultant application will be large, slow, and undebuggable.

To symbolically debug your application, you must use the `-g` option. This option is applicable with or without optimization. However, it is much easier to debug your application if you do not optimize. With optimization, a debugger is not able to print the values of local variables, is not able to accurately print the value of any variables except on function entry, and will jump around as you single step because the optimizer will aggressively interleave code from multiple lines or even multiple functions.

Optimization flags are discussed in more detail in Chapter 3. Briefly, compiler optimization is mainly controlled through the use of the `-Ox` flags. Production code should always be compiled using at least `-O`, which is equivalent to `-O2`. Unoptimized code will on average be about three times slower and 50% bigger than code compiled with `-O`.

The `-O3` option will cause the compiler to optimize more aggressively, particularly for DSP type code. Performance on average will be a little faster. However, a small number of programs will speed up significantly, up to two times, while others will slow down a little. Most programs will be larger.

The XCC compiler is able to automatically infer the use of coprocessor instructions from standard C/C++ applications. As one example, standard C multiplication can be emulated efficiently on ConnX D2 or HiFi using their coprocessor multipliers. In addition, the ConnX D2 engine supports VLIW execution of many standard C operations. However, some of these coprocessor instructions use coprocessor register files, and it may not always be safe to use those register files, particularly inside of interrupt handlers. Coprocessor register files may be lazily saved and restored using a coprocessor exception mechanism, and some operating systems might not support the use of coprocessor exceptions inside of interrupt handlers.

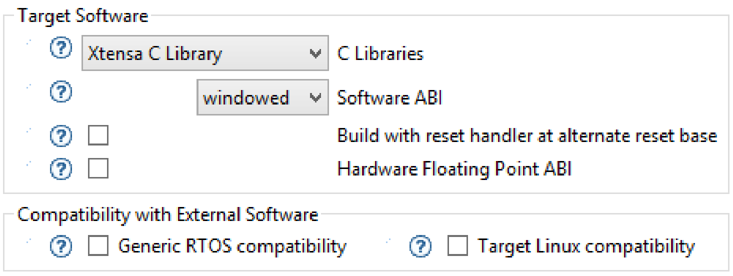
To insure safety inside of handlers, XCC does not by default infer the use of any instruction that uses any coprocessor state. Nonetheless, application code can benefit from this inference, and this inference can be enabled by compiling with the `-mcoproc` option. Note that the compiler option for vectorization, `-LNO:simd`, also implies `-mcoproc` in addition to enabling vectorization.

The `-Os` flag can be used together with either `-O2` or `-O3`. By itself it implies `-O2`. It instructs the compiler to try to optimize for space rather than for speed. In comparison to `-O2`, code will on average be 10% slower but 15% smaller.

There are many more advanced optimization options, in particular `-ipa` for interprocedural analysis and `-fb_opt` for feedback-directed optimization. These are covered in more detail in Chapter 3 as well as in the *Xtensa C and C++ Compiler User's Guide*.

1.2 Software Configuration Options

Configuration options which only affect the target software


Target Software

Xtensa C Library — C Libraries

windowed — Software ABI

☐ Build with reset handler at alternate reset base

☐ Hardware Floating Point ABI

Compatibility with External Software

☐ Generic RTOS compatibility   ☐ Target Linux compatibility

Software configuration options require a new processor configuration, but do not affect the generated hardware. Therefore, options can be selected after a hardware design has been completed. The initial choices set in the software pane of the processor generator when creating a configuration will used to generate matching diagnostics with the HW package.

You can easily explore alternatives by building the HW + SW one way, and then building "software upgrades" of the original configuration with different combinations of target software options. "Upgrades" can be variants built with the same XPG release, or they can be built with newer XPG releases.

1.2.1 C and Math Libraries

Cadence offers the choice of three C and math libraries: `newlib` from Red Hat, Inc., the Xtensa C library and `uClibc`

You choose between the libraries when building your software configuration through the Xtensa Processor Generator. The libraries cannot be mixed.

- The `newlib` library is more complete, fully documented, higher performance and supports reentrancy for multi-threaded environments.
- The Xtensa C library has similar performance to newlib and is smaller. It strictly implements the C library as defined by the C standard and hence may not implement all the extensions supported by `newlib`. The philosophy of the library is standards compliance and simplicity. So, for example, the malloc routine is simple and hence fast but might cause more memory fragmentation on programs that extensively malloc and free. The Xtensa C library places no open source restrictions *on the C user (there are minor restrictions for the C++ user)*.
- `uClibc` is significantly smaller. `uClibc` can be configured with or without support for floating point. Without floating point support, it is not possible, for example, to print floating point numbers and it is not possible to use C++ I/O streams, but the resultant library is significantly smaller still. Note that `uClibc` comes with more restrictive open source licensing requirements than even `newlib`.

Review your contract or the files in the `XtensaTools/misc` directory for details about the various licensing requirements.

1.2.2 Application Binary Interfaces

Cadence offers the choice of two Application Binary Interfaces (ABIs) for Xtensa X and LX processors: the windowed ABI and the CALL0 ABI. Xtensa TX processors only support the CALL0 ABI because they only have 16 AR registers. With the windowed ABI, each function call is implemented using a CALL4, CALL8 or CALL12 instruction that rotates the Xtensa register windows and thereby immediately gives the called function a set of extra scratch registers. Without the windowed ABI, each function call is implemented using a CALL0 instruction and the compiler must typically save and restore to memory scratch variables used by the callee. Application code compiled using CALL0 is typically 5-10% larger than application compiled using the windowed ABI. Performance of loop intensive code is marginally slower with CALL0 while more call intensive code is up to 10% slower. At time of writing, CALL0 is only supported with the ThreadX RTOS from Express Logic or with the XTOS runtime from Cadence.

Given these characteristics, most will use the windowed ABI. However, there are also advantages to the CALL0 ABI. The CALL0 ABI enables hardware configurations with only 16 AR registers, thereby allowing significantly smaller hardware configurations. Interrupt and context switching latency is lower with CALL0 than with the windowed ABI. Using the CALL0 ABI, you can manually rotate the register files in a single cycle in special code or interrupt handlers for very fast specialized context switching.

An application cannot mix the two ABIs. However, it is possible to use the windowed ABI for an application and CALL0 for certain high priority interrupts. The use of CALL0 in this context enables interrupt handlers to be written in C without the higher overhead of saving and restoring all the AR registers.

Related Links

AR Registers Count on page 124
Number of physical AR registers. Setting to 16 registers means windowed calls are not supported

1.2.3 Hardware Floating Point ABI

Cadence offers two familiar floating point options - namely single and double precision, and also several coprocessors which optionally include floating point hardware capabilities including HiFi3, HiFi4, and Fusion.

The coprocessors which include floating point capabilities implicitly use the Floating Point ABI which makes use of floating point registers for function calls.

The familiar single and double precision options by default use the standard ABI which makes use of AR registers for FP value passing. The Hardware Floating Point ABI is available as an option for single and double precision FP. It will lead to more efficient passing of floating point variables, especially for double precision, but exercise caution when integrating with object code compiled for other configurations because the ABI is not compatible.

Related Links

Single Precision FP Option on page 118
IEEE single precision floating point instruction family. Uses coprocessor ID 0

Single+Double Precision FP option on page 119
IEEE single + double precision floating point instruction family. Uses coprocessor ID 0

HiFi3 Vector FP on page 95
Vector Floating Point extensions to HiFi 3 Audio Engine

HiFi4 Vector FP on page 96
Vector Floating Point extensions to HiFi 4 Audio Engine

Fusion FP on page 114
Support for IEEE 754 single precision floating point

1.2.4 Use Extended L32R Instruction (for Legacy Hardware)

All Cadence processors from Xtensa LX and Xtensa6 through LX3 and Xtensa 8 include hardware to support two different kinds of L32R instruction: normal (PC relative) and "based". This software selection chooses how the target software is built (and how the compiler generates code) - whether it makes use of the standard PC relative address mode, or whether it uses the extended L32R HW support for a larger address range.

A processor needs a mechanism for accessing literals; numbers with values known at compile or link time. Literals are used for constants directly written in C and for the address of global variables. For example, in order to implement the C statement `x=123456`, when x is a global variable, the compiler must put the literal `123456` into a register, put the address of the variable `x` into another register, and then store the value in the first register into the address from the second register. For small literals, the literal can be encoded directly inside a `movi` instruction that moves the value of a literal into a register. For larger literals, the compiler places the literal value in memory and uses the `l32r` instruction to load the literal from memory into a register. Normally `l32r` is a PC-relative instruction which can load a literal from any memory location within 256 Kilobytes of the instruction invoking the `l32r`. The compiler, assembler and linker work together to ensure that literals are placed in range of their invoking `l32r` instruction, and even combine multiple literals with the same value to save memory. If however, you are using a local IRAM that is 256 KBytes or larger, the only place to put a literal that is in range might be inside the IRAM itself. Prior to the RD release, loading memory from an IRAM was slow and therefore undesirable. Therefore, Cadence offered the extended `l32r` option for users of large IRAMs. With this option, the `l32r` instruction loads from an offset of a global base register. This global base register can point to any memory allowing literals to be placed anywhere. However, there are disadvantages to the use of extended L32R instruction. In particular, this option is not supported by third-party RTOSs. Furthermore, since the base register is global, all the literals used by the program must be contiguous and therefore a program is limited to a total of 256 KBytes of literals, or equivalently 64K literals. Additionally, Cadence does not support dynamically loading and unloading code on software configurations that uses the extended L32R. Starting with the RD release, the hardware is able to load literals quickly even when they are in IRAM. Therefore, the extended L32R option is only supported for older hardware.

1.2.5 Build with Reset Handler at Alternate Reset Base

For processor configurations that support relocatable vectors, at configuration time, a primary and alternate "static vector group base address" can be configured. This address is a base from which the reset vector and memory error vector (if configured) are offset. Which address (primary / alternate) is used at processor reset is controlled by an input pin which can be asserted to select the alternate base. By default, the software configuration build will assume the primary static base is used, and will generate reset code for those addresses. This option chooses whether to build software with the reset code at the alternate address.

1.2.6 RTOS Compatibility Option

Generic RTOS Compatibility - ensures selection of a set of features required by many RTOSes

This option does not have any direct effect on processor software or hardware; it is a compatibility checking option to help avoid configuration omissions that might have later impact on what software can run on the processor.

1.2.7 Target Linux Compatibility Option

Ensures the minimum requirements for running Linux on Xtensa are met

This option does not have any direct effect on processor software or hardware; it is a compatibility checking option to help avoid configuration omissions that might have later impact on what software can run on the processor.

Xtensa processors support very varied memory maps with flexible combinations of local and system memories and caches. Running Linux on an Xtensa processor, however, places several significant restrictions on the memory subsystem - for example the Full MMU with TLBs and a memory layout that is conducive to an appropriate virtual memory layout.

If you plan to run Linux on an Xtensa processor, it is strongly recommended that you start with the Diamond 233L processor template which sets up an approrpiate memory configuration.

Related Links

Load a Configuration Template on page 91
Initialize the configuration from a template - replacing all current selections

Memory Management Selection on page 115
Region protection options provide coarse protection at a low gate count The Full MMU provides resource management capabilities sufficient for running Linux

2. Performance Tuning Methodologies

Topics:

This section details the tools and methodologies for maximizing application performance and minimizing code size on Cadence processors.

Subsections describe the methodologies in more detail.

2.1 Diagnosing Performance Problems

Without tools, it can be difficult to know where your performance problems are or what you can do to solve them. For example, a floating-point multiplication using software emulation might take approximately 100 cycles. If 1% of the operations in your application are floating-point multiplies, the multiplication routine represents significant computational overhead. If 0.01% of the operations are floating-point multiplies, the overhead might be negligible.

There are multiple types of tools that can help you diagnose software-performance problems. Profiling, available in simulation or on real hardware through Xtensa Xplorer, as well as through `xt-gprof`, shows you where your application code is spending the most time, either at the function level or at the source-line or assembly-line level. You should concentrate your efforts on regions that consume a significant portion of the processor's or task's time. In addition, the tools are able to break down the time spent executing an application event-by-event. For example, they can tell you how much time is spent globally servicing cache misses or accessing uncached memory, and they can tell you which regions of code suffer the most from those events. This information helps you know whether to concentrate your efforts on generated code sequences or on your memory subsystem. Similarly, the ISA profiler can tell you how frequently every instruction or operation is executed, allowing you to discover if a selected hardware configuration or TIE instruction is being effectively used.

Software development tools also give static information useful in tuning applications. Measuring code size on a function by function basis is useful when trying to minimize memory usage. Looking at the compiler generated assembly file, the `.s` file, the compiler inserts comments to aid in tuning. When vectorizing to try to take advantage of a ConnX, IVP or HiFi3 processor, the compiler can also generate analysis messages explaining what issues were encountered in the code. The messages are enabled on the command line with the `-LNO:simd_v` option or in Xplorer's *Vectorization Assistant* tool.

2.2 Types of Performance Problems

When you know where to start, you can begin to tune your code. There are several aspects to performance including algorithmic, configuration, memory system, microarchitectural, and compiler code quality, as discussed in this section.

2.2.1 Choosing Algorithms

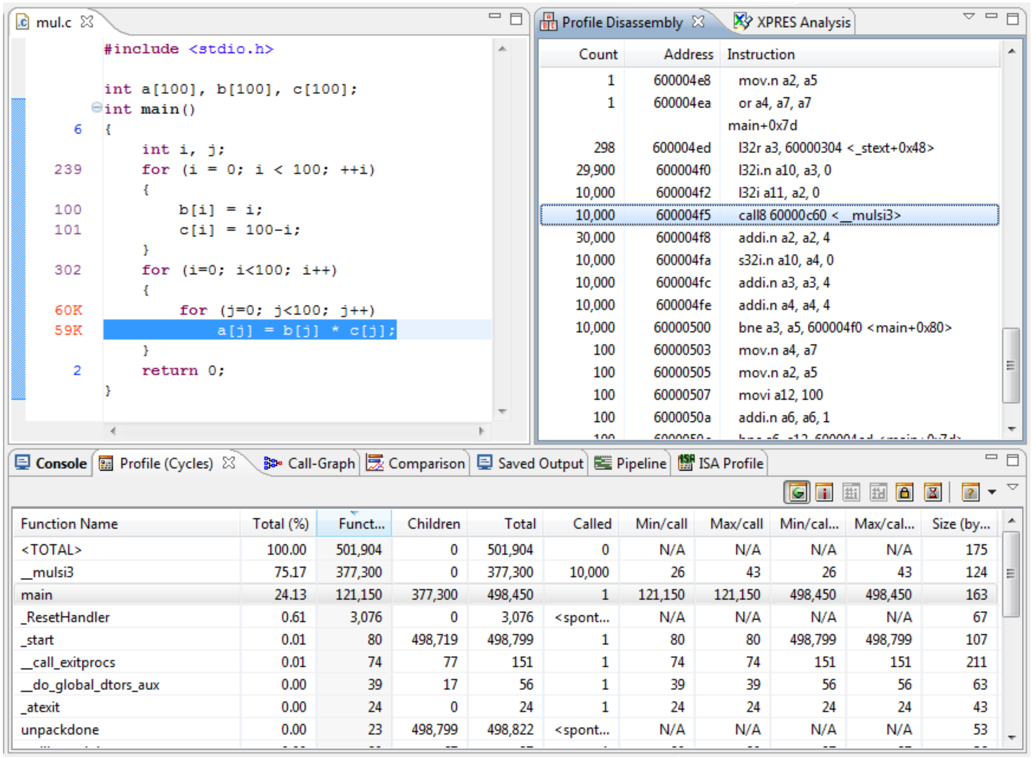
Choice of algorithm has a first-order effect on performance, code size and power. A radix-4 FFT fundamentally has fewer multiplies than a radix-2 FFT. A heap sort fundamentally requires fewer compares than an insertion sort. Algorithm choice is mostly independent of processor architecture, and there are few tools to help select among alternative algorithms. However, there is some connection between algorithm and processor or system architecture.

For example, a radix-4 FFT requires fewer multiplies than a radix-2, and a radix-8 FFT requires fewer multiplies than either of the other two types. However, as you increase the FFT radix, you increase the operation complexity and the number of registers required. At sufficiently high radix, performance becomes dominated by overhead and data shuffling. A typical Xtensa processor configuration tends to handle radix-4 better than either radix-2 or radix-8. Be mindful of your architecture when choosing algorithms and, for the Xtensa processor, be mindful of your algorithm when tuning the architecture.

2.2.2 Configuration

The choice of processor configuration can greatly affect performance. The use of TIE can increase performance by many factors, but configuration parameters and choice of memory subsystem can also make a substantial difference. Specific details of different configuration options are discussed in **Chapter 4**.

There are two basic techniques that can be used to improve your choice of core processor configuration. To use the first technique, profile your application and take a closer look at the hot spots. Consider the example in *Figure 1: Profiling in Xplorer* on page 25. Profiling shows that much time is spent in *__mulsi3*, the integer multiplication emulation routine. Adding a multiplier to your configuration will probably help.



mul.c

```
      #include <stdio.h>

      int a[100], b[100], c[100];
      int main()
  6   {
          int i, j;
239       for (i = 0; i < 100; ++i)
          {
100           b[i] = i;
101           c[i] = 100-i;
          }
302       for (i=0; i<100; i++)
          {
60K           for (j=0; j<100; j++)
59K               a[j] = b[j] * c[j];
          }
  2       return 0;
      }
```

Profile Disassembly | XPRES Analysis

```
 Count    Address   Instruction
     1    600004e8    mov.n a2, a5
     1    600004ea    or a4, a7, a7
                    main+0x7d
   298    600004ed    l32r a3, 60000304 <_stext+0x48>
29,900    600004f0    l32i.n a10, a3, 0
10,000    600004f2    l32i a11, a2, 0
10,000    600004f5    call8 60000c60 <__mulsi3>
30,000    600004f8    addi.n a2, a2, 4
10,000    600004fa    s32i.n a10, a4, 0
10,000    600004fc    addi.n a3, a3, 4
10,000    600004fe    addi.n a4, a4, 4
10,000    60000500    bne a3, a5, 600004f0 <main+0x80>
   100    60000503    mov.n a4, a7
   100    60000505    mov.n a2, a5
   100    60000507    movi a12, 100
   100    6000050a    addi.n a6, a6, 1
```

Console | Profile (Cycles) | Call-Graph | Comparison | Saved Output | Pipeline | ISA Profile

Function Name	Total (%)	Funct...	Children	Total	Called	Min/call	Max/call	Min/cal...	Max/cal...	Size (by...
<TOTAL>	100.00	501,904	0	501,904	0	N/A	N/A	N/A	N/A	175
__mulsi3	75.17	377,300	0	377,300	10,000	26	43	26	43	124
main	24.13	121,150	377,300	498,450	1	121,150	121,150	498,450	498,450	163
_ResetHandler	0.61	3,076	0	3,076	<spont...	N/A	N/A	N/A	N/A	67
_start	0.01	80	498,719	498,799	1	80	80	498,799	498,799	107
__call_exitprocs	0.01	74	77	151	1	74	74	151	151	211
__do_global_dtors_aux	0.00	39	17	56	1	39	39	56	56	63
_atexit	0.00	24	0	24	1	24	24	24	24	43
unpackdone	0.00	23	498,799	498,822	<spont...	N/A	N/A	N/A	N/A	53

Figure 1: Profiling in Xplorer

Using the second technique, build multiple configurations and see how performance of your application varies across configurations. Xtensa Xplorer allows you to manage multiple configurations and graphically compare the performance of your application using different configurations.

In *Figure 2: Adding a MUL16 to a Configuration* on page 26, Xtensa Xplorer is used to compare the integer multiply example above using two configurations; one with no hardware multipliers, and one with a MUL16.

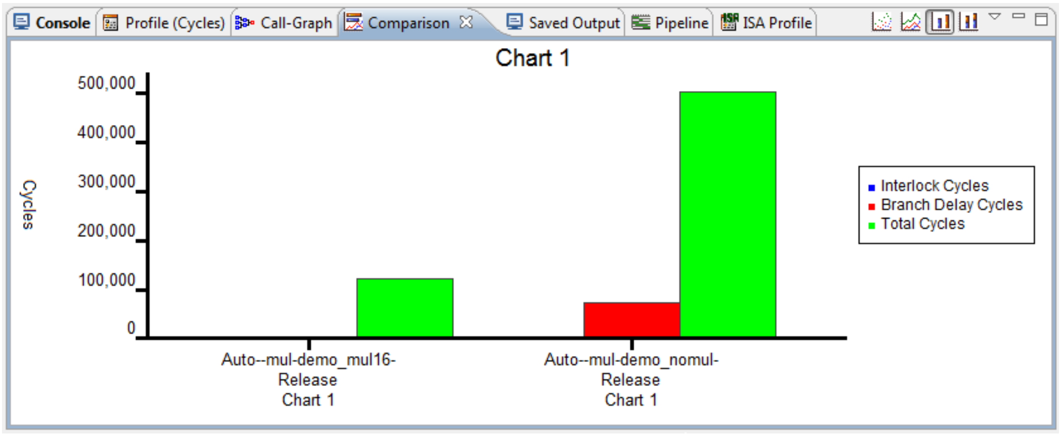


Figure 2: Adding a MUL16 to a Configuration

In *Figure 3: Adding a Floating-Point Unit to a Configuration* on page 27, Xtensa Xplorer is used to compare the performance of an application performing floating-point multiplies using two configurations; one without the Xtensa floating-point unit (FPU) and one with the FPU. Xplorer is used to profile the application using both configurations. Then, from the Benchmark Perspective, in the Benchmark Results tab, Select Profile Data Sets was used to select the two runs and then **Create New Custom Chart** was selected to create a chart to compare the two runs. The figure shows two sets of bars. If your comparison window looks different, select the correct display format. Several graph display modes are available, selected by the buttons at the upper-right of the comparison window. The right-hand set of three bars (two have a value of (close to) zero and are therefore not visible) shows software cycle counts for a processor configuration with an FPU. The left-hand set of bars shows a processor configuration without an FPU. Within each set of three bars, the first bar shows interlock cycles, the second bar shows the branch delays and the third bar shows the total cycle count for the software routine.

As shown in *Figure 3: Adding a Floating-Point Unit to a Configuration* on page 27, the floating-point unit greatly improves performance. In both cases, the interlock bar is not visible as the amount of time where the generated hardware is stalled due to interlocks is negligible.

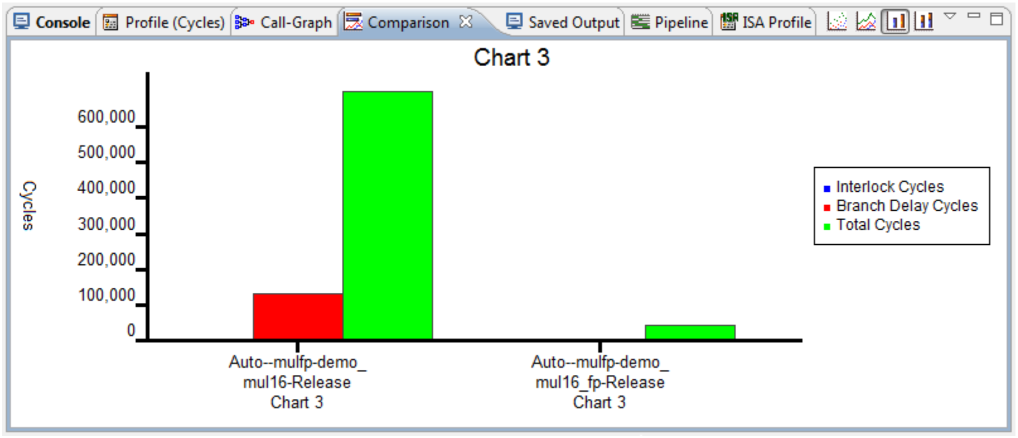
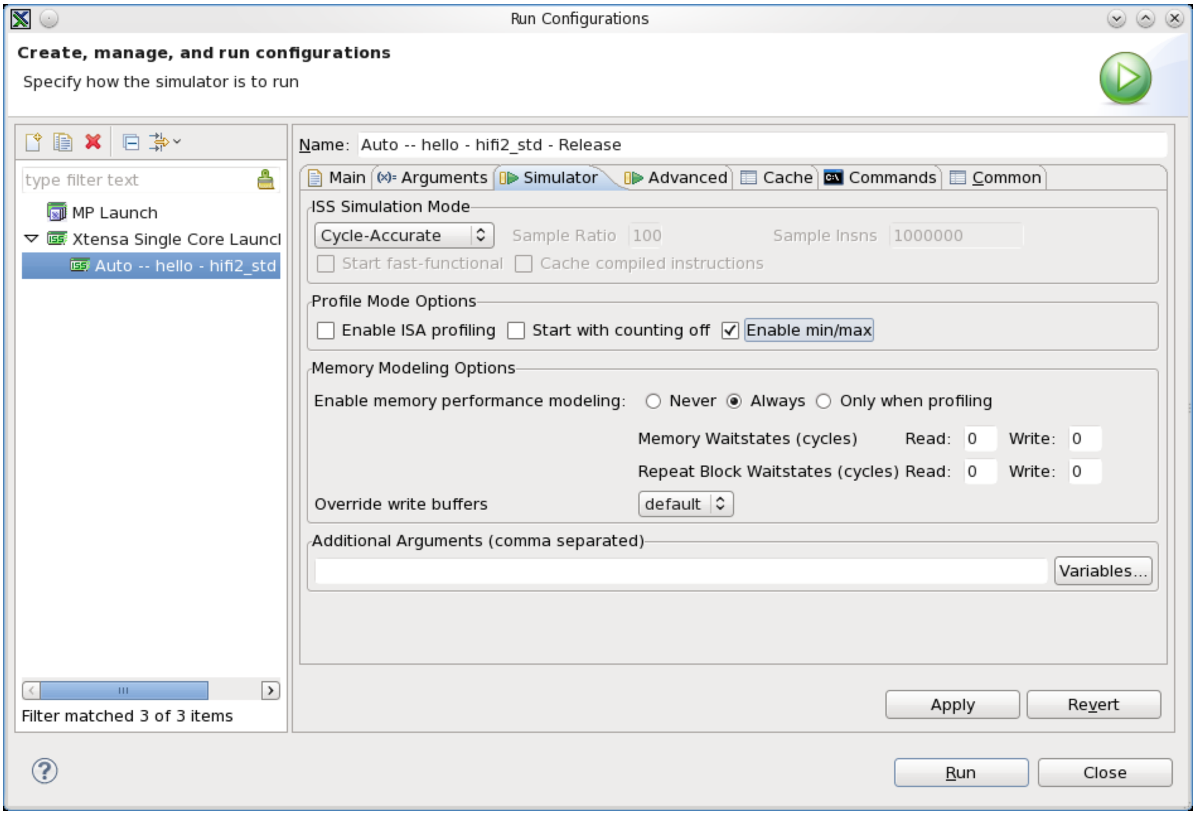


Figure 3: Adding a Floating-Point Unit to a Configuration

2.2.3 Memory System

In modern embedded systems, how fast you get data into and out of the processor can often be the most critical aspect of performance. Xtensa processors give you a lot of freedom in designing external interfaces. Communication can be through local memory ports, the Xtensa Local Memory Interface (XLMI), the Xtensa Processor Interface (PIF), AHB, AXI, TIE ports, TIE queue interfaces, TIE lookups, or interrupt lines. Each of these interface types can be tailored for the application. You can, for example, choose the width of the PIF, the size of caches (if any), and the number of memory units.

Cadence provides several tools to measure and analyze memory-system performance in simulation. The standalone ISS simulator accurately models caches and the interactions between the memory system and the pipeline for a fixed-latency/fixed-bandwidth memory system. The standalone ISS allows you to independently set read and write latencies as well as block repeat rates using the flags `--read_delay`, `--read_repeat`, `--write_delay` and `--write_repeat`. On a block read transaction, a cache miss for example, the first PIF sized chunk of data is assumed to arrive `--read_delay+1` cycles after the address is presented to the PIF. Each subsequent PIF sized chunk of data arrives `--read_repeat+1` cycles later. These flags are also settable from the Run dialog box in Xplorer as shown in the figure below.


Run Configurations

Create, manage, and run configurations

Specify how the simulator is to run

type filter text

MP Launch

Xtensa Single Core Launcl

Auto -- hello - hifi2_std

Filter matched 3 of 3 items

Name: Auto -- hello - hifi2_std - Release

Main | Arguments | Simulator | Advanced | Cache | Commands | Common

ISS Simulation Mode

Cycle-Accurate | Sample Ratio 100 | Sample Insns 1000000

[ ] Start fast-functional [ ] Cache compiled instructions

Profile Mode Options

[ ] Enable ISA profiling [ ] Start with counting off [✓] Enable min/max

Memory Modeling Options

Enable memory performance modeling: Never [✓ Always] Only when profiling

Memory Waitstates (cycles) Read: 0 Write: 0

Repeat Block Waitstates (cycles) Read: 0 Write: 0

Override write buffers: default

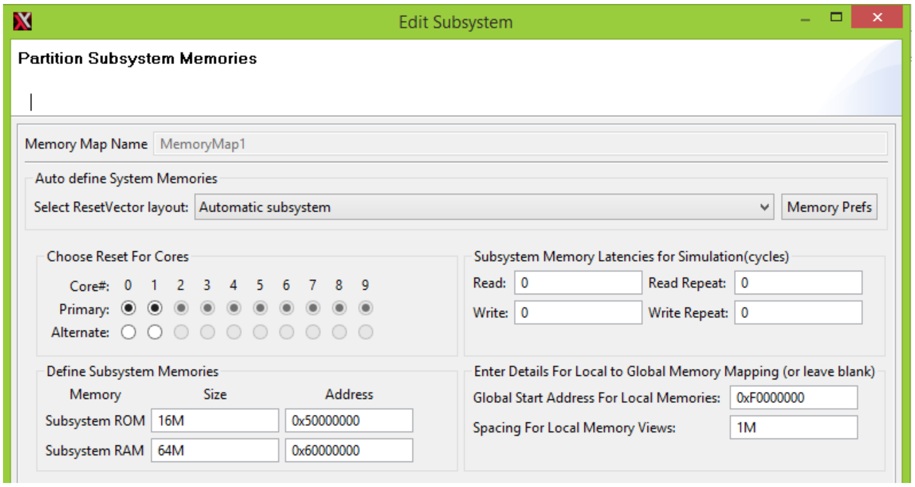
Additional Arguments (comma separated)

Figure 4: Setting Memory Latencies in Xplorer

When creating shared memory subsystems in Xplorer, the memory delays can also be set when partitioning the subsystem memories as shown in *Figure 5: Subsystem Memory Configuration* on page 29.

For more complicated memory modeling, Cadence provides two simulation libraries, XTSC (SystemC based) and XTMP (C based), that allow you to build accurate models of more complicated memory systems including DMA, TIE queue interfaces, ports and lookups, and custom devices and interconnects.

By default, the stand-alone uniprocessor ISS does not simulate the memory subsystem; it assumes zero delay cycles on all loads and stores. This policy allows you to tune the other aspects of performance independently of your memory subsystem. Particularly with caches, small changes that improve microarchitectural performance might randomly degrade memory-system performance as they change data or instruction layout. Nonetheless, all these small changes in aggregate will probably improve performance. Simulations with an ideal memory system allow you to ignore these effects until you are ready to tune your memory system. After architectural tuning of the processor is finished, you can select memory modeling in the stand-alone ISS simulation, or use the XTSC or XTMP simulation environment to add memory-delay characteristics to the simulation.



Memory Map Name: MemoryMap1

Auto define System Memories

Select ResetVector layout: Automatic subsystem

Memory Prefs

Choose Reset For Cores

Core#:	0	1	2	3	4	5	6	7	8	9
Primary:	●	●	●	●	●	●	●	●	●	●
Alternate:	○	○	○	○	○	○	○	○	○	○

Subsystem Memory Latencies for Simulation(cycles)

Read: 0 Read Repeat: 0

Write: 0 Write Repeat: 0

Define Subsystem Memories

Memory	Size	Address
Subsystem ROM	16M	0x50000000
Subsystem RAM	64M	0x60000000

Enter Details For Local to Global Memory Mapping (or leave blank)

Global Start Address For Local Memories: 0xF0000000

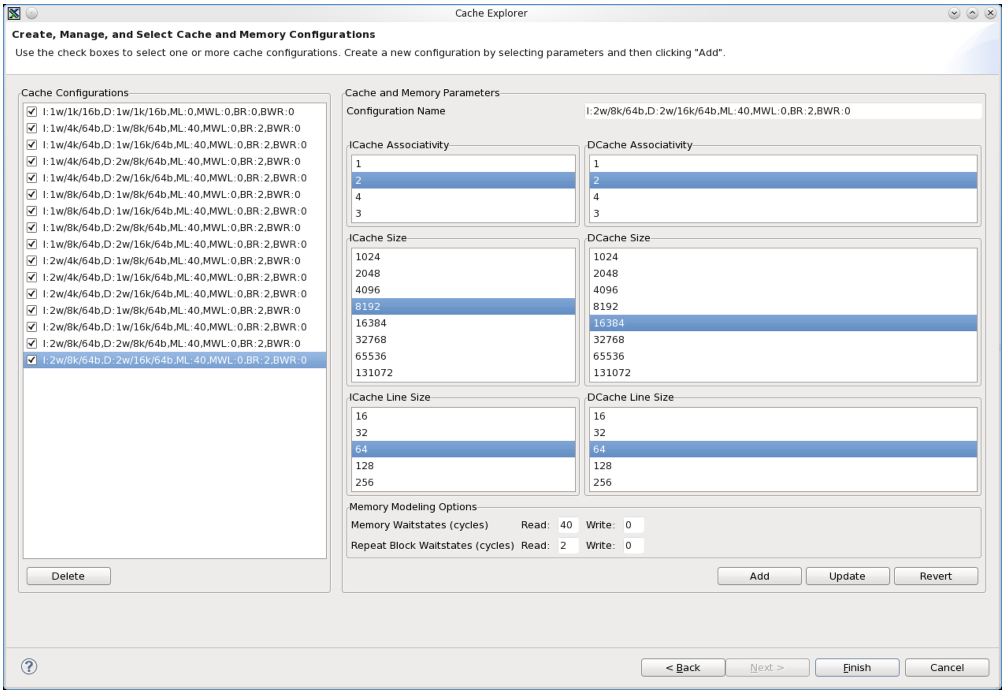
Spacing For Local Memory Views: 1M

Figure 5: Subsystem Memory Configuration

MP subsystems, either created by Xplorer or manually using XTSC or XTMP always simulate the memory subsystem unless run in the TurboXim fast functional simulation mode.

As with other aspects of configuration, Xtensa Xplorer allows you to easily compare performance of an application using multiple memory systems. The Cache Explorer feature of Xtensa Xplorer allows you to graphically select a range of cache parameters and have the system automatically simulate and compare performance for all selected configurations. To illustrate the use of this tool, an MP3 decoder running on a series of Cadence HiFi 2 configurations was tested.

Figure 6: Cache Explorer Control Panel on page 30 shows the control panel of the Cache Explorer. This control panel is accessible via the Tools/Cache and Memory Explorer menu in Xplorer. In this example, all combinations of direct mapped and 2-way set associative 4K and 8K Icache sizes together with all combinations of direct mapped and 2-way set associative 8K and 16K Dcache sizes are to be evaluated.



Cache Explorer

Create, Manage, and Select Cache and Memory Configurations

Use the check boxes to select one or more cache configurations. Create a new configuration by selecting parameters and then clicking "Add".

Cache Configurations

- ✓ I:1w/1k/16b,D:1w/1k/16b,ML:0,MWL:0,BR:0,BWR:0
- ✓ I:1w/4k/64b,D:1w/8k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:1w/4k/64b,D:1w/16k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:1w/4k/64b,D:2w/8k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:1w/4k/64b,D:2w/16k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:1w/8k/64b,D:1w/8k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:1w/8k/64b,D:1w/16k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:1w/8k/64b,D:2w/8k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:1w/8k/64b,D:2w/16k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:2w/4k/64b,D:1w/8k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:2w/4k/64b,D:1w/16k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:2w/4k/64b,D:2w/16k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:2w/8k/64b,D:1w/8k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:2w/8k/64b,D:1w/16k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:2w/8k/64b,D:2w/8k/64b,ML:40,MWL:0,BR:2,BWR:0
- ✓ I:2w/8k/64b,D:2w/16k/64b,ML:40,MWL:0,BR:2,BWR:0

Cache and Memory Parameters

Configuration Name: I:2w/8k/64b,D:2w/16k/64b,ML:40,MWL:0,BR:2,BWR:0

ICache Associativity: 1 [✓ 2] 4 3

DCache Associativity: 1 [✓ 2] 4 3

ICache Size: 1024 2048 4096 [✓ 8192] 16384 32768 65536 131072

DCache Size: 1024 2048 4096 8192 [✓ 16384] 32768 65536 131072

ICache Line Size: 16 32 [✓ 64] 128 256

DCache Line Size: 16 32 [✓ 64] 128 256

Memory Modeling Options

Memory Waitstates (cycles) Read: 40 Write: 0

Repeat Block Waitstates (cycles) Read: 2 Write: 0

Figure 6: Cache Explorer Control Panel

The results are shown in the figure below. The miss penalities are significant, and the data cache misses are larger than the instruction cache misses. For the data cache misses, every fourth bar is significantly larger than the others. This corresponds to an 8K, direct mapped data cache. It is likely necessary for this application to either increase the data cache size to 16K or use a set-associative cache. The first four results correspond to a 4K direct-mapped instruction cache. The instruction cache penalties are significantly higher for these configurations.

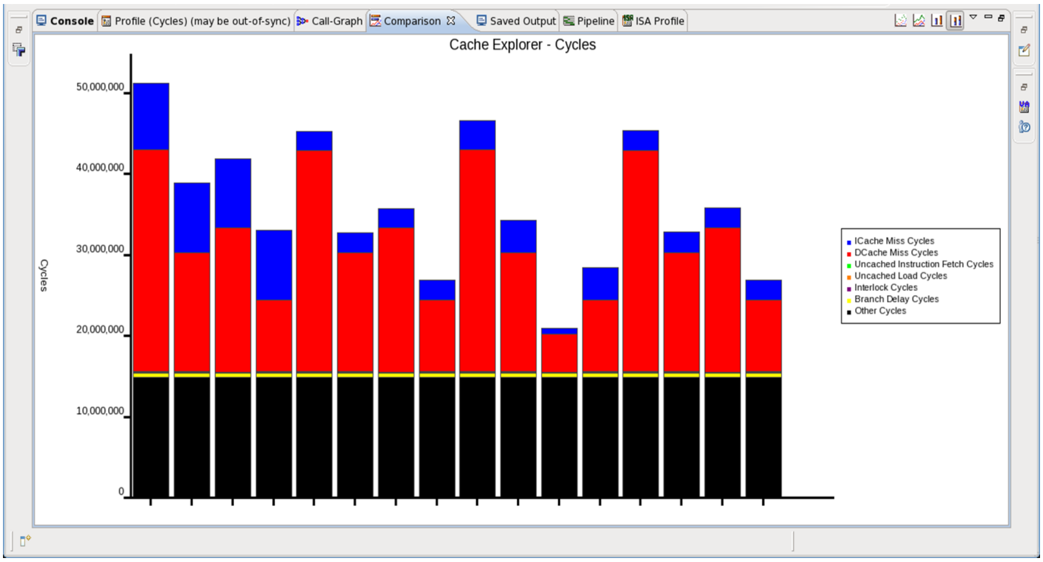


Figure 7: Cache Explorer Cycles

The Cache Explorer is a good tool for helping you decide which cache configurations to select, and it is also useful for software tuning. For example, we see from the tool that I cache misses are a problem by default when the I-cache is direct mapped and only 4 KB. Xplorer provides a software tool to rearrange the order of functions in an executable to minimize I cache misses.To invoke the tool, first profile the application. Then right-click the project and select *Update Link Order*. Choose one or more profile files to guide the reording. These profile files should be representative of typical execution runs. Typically one should either use a profile file that measures cycles or one that measures cycles due to instruction cache misses. Once the link order file has been generated, change the project build properties to use the link order file as part of the link step. The results of using the tool on the MP3 application are shown in the figure below. The penalty for instruction cache misses has been significantly reduced, allowing the use of the direct-mapped 4K I cache.

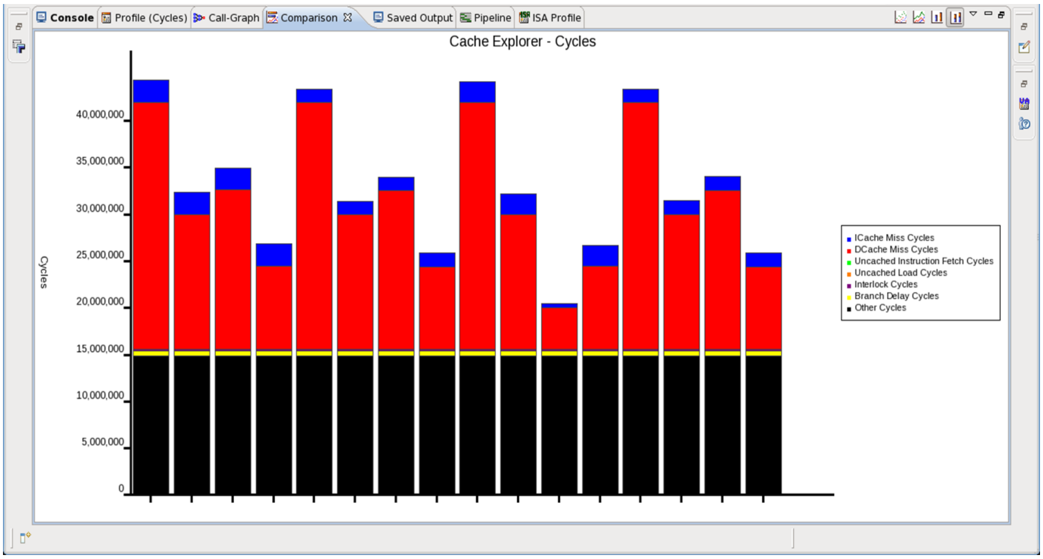
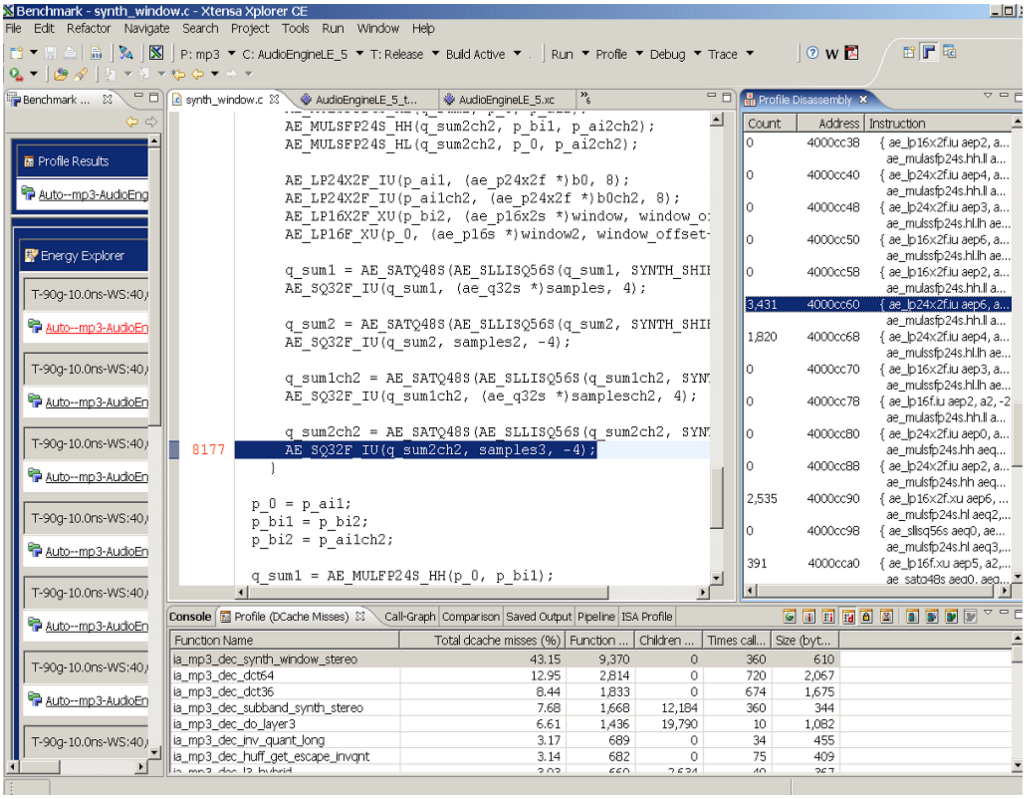


Figure 8: Cache Explorer Cycles Using the Link Order Tool

The Cache Explorer is closely tie together with the profiling tools, it allows you to easily focus in on problematic portions of your application. For each evaluated configuration, you can easily see the corresponding profile of your application. In addition to the traditional time based profiling, you can also profile based on events such as cache misses. Consider again the MP3 example. The performance of the direct-mapped 8K data cache is poor. *Figure 9: Data Cache Profile* on page 33 shows a data cache profile of that configuration. From the profile, you can see that over 40% of the data cache misses are in a single C function, and the vast majority of those misses are in one loop. Direct-mapped caches perform poorly relative to set associative caches when multiple memory references conflict with each other. Looking at the single loop, you see that it is streaming through two different arrays. Perhaps those two arrays line up in the same cache lines. As a quick experiment, some padding was randomly added to the data memory. The number of data cache misses in the function was reduced by 45% and overall performance improved by 5%.


Benchmark - synth_window.c - Xtensa Xplorer CE
File Edit Refactor Navigate Search Project Tools Run Window Help
P: mp3 C: AudioEngineLE_5 T: Release Build Active Run Profile Debug Trace
Benchmark ...
Profile Results
Auto--mp3-AudioEng
Energy Explorer
T-90g-10.0ns-WS:40,
Auto--mp3-AudioEn
synth_window.c AudioEngineLE_5_t... AudioEngineLE_5.xc
Profile Disassembly
Console Profile (DCache Misses) Call-Graph Comparison Saved Output Pipeline ISA Profile

```
        AE_MULSFP24S_HH(q_sum2ch2, p_bi1, p_ai2ch2);
        AE_MULSFP24S_HL(q_sum2ch2, p_0, p_ai2ch2);

        AE_LP24X2F_IU(p_ai1, (ae_p24x2f *)b0, 8);
        AE_LP24X2F_IU(p_ai1ch2, (ae_p24x2f *)b0ch2, 8);
        AE_LP16X2F_XU(p_bi2, (ae_p16x2s *)window, window_o
        AE_LP16F_XU(p_0, (ae_p16s *)window2, window_offset-

        q_sum1 = AE_SATQ48S(AE_SLLISQ56S(q_sum1, SYNTH_SHI
        AE_SQ32F_IU(q_sum1, (ae_q32s *)samples, 4);

        q_sum2 = AE_SATQ48S(AE_SLLISQ56S(q_sum2, SYNTH_SHI
        AE_SQ32F_IU(q_sum2, samples2, -4);

        q_sum1ch2 = AE_SATQ48S(AE_SLLISQ56S(q_sum1ch2, SYN
        AE_SQ32F_IU(q_sum1ch2, (ae_q32s *)samplesch2, 4);

        q_sum2ch2 = AE_SATQ48S(AE_SLLISQ56S(q_sum2ch2, SYN
8177    AE_SQ32F_IU(q_sum2ch2, samples3, -4);
      }

    p_0 = p_ai1;
    p_bi1 = p_bi2;
    p_bi2 = p_ai1ch2;

    q_sum1 = AE_MULFP24S_HH(p_0, p_bi1);
```

Count	Address	Instruction
0	4000cc38	{ ae_lp16x2f.iu aep2, a... ae_mulasfp24s.hh.ll a...
0	4000cc40	{ ae_lp24x2f.iu aep4, a... ae_mulasfp24s.hh.ll a...
0	4000cc48	{ ae_lp24x2f.iu aep3, a... ae_mulssfp24s.hl.lh ae...
0	4000cc50	{ ae_lp16x2f.iu aep6, a... ae_mulssfp24s.hl.lh ae...
0	4000cc58	{ ae_lp16x2f.iu aep2, a... ae_mulasfp24s.hh.ll a...
3,431	4000cc60	{ ae_lp24x2f.iu aep6, a... ae_mulasfp24s.hh.ll a...
1,820	4000cc68	{ ae_lp24x2f.iu aep4, a... ae_mulssfp24s.hl.lh ae...
0	4000cc70	{ ae_lp16x2f.iu aep3, a... ae_mulssfp24s.hl.lh ae...
0	4000cc78	{ ae_lp16f.iu aep2, a2, -2 ae_mulasfp24s.hh.ll a...
0	4000cc80	{ ae_lp24x2f.iu aep0, a... ae_mulsfp24s.hh aeq...
0	4000cc88	{ ae_lp24x2f.iu aep2, a... ae_mulsfp24s.hh aeq...
2,535	4000cc90	{ ae_lp16x2f.xu aep6, ... ae_mulsfp24s.hl aeq2,...
0	4000cc98	{ ae_slisq56s aeq0, ae... ae_mulsfp24s.hl aeq3,...
391	4000cca0	{ ae_lp16f.xu aep5, a2,... ae_satq48s aeq0, aeq...

Function Name	Total dcache misses (%)	Function ...	Children ...	Times call...	Size (byt...
ia_mp3_dec_synth_window_stereo	43.15	9,370	0	360	610
ia_mp3_dec_dct64	12.95	2,814	0	720	2,067
ia_mp3_dec_dct36	8.44	1,833	0	674	1,675
ia_mp3_dec_subband_synth_stereo	7.68	1,668	12,184	360	344
ia_mp3_dec_do_layer3	6.61	1,436	19,790	10	1,082
ia_mp3_dec_inv_quant_long	3.17	689	0	34	455
ia_mp3_dec_huff_get_escape_invqnt	3.14	682	0	75	409

Figure 9: Data Cache Profile

Finally, when tuning your memory subsystem, it is important to model a system and application similar to the one you actually will use. For example, on a system with local memories, DMA might significantly improve performance for some applications. Simulating such applications without modeling DMA might lead one to incorrect conclusions about the required cache parameters.

2.2.4 Microarchitectural Description

At full speed, the Xtensa processor's pipeline can execute one instruction every cycle. However, various situations can stall the pipeline. The most common reasons for stalling the processor are memory delays, branch delays, and pipeline interlocks. Other reasons are described more fully in the Xtensa Instruction Set Simulator (ISS) User's Guide. The ISS summarizes the number of cycles stalled for different reasons. Below shows sample output from an ISS summary:

```
Cycles: total = 467284
                                            Summed |           Summed
                                     CPI      CPI    |% Cycle   % Cycle
```

```
 Committed instructions        447814 ( 1.0000   1.0000 |  95.83    95.83 )
 Taken branches                  3140 ( 0.0070   1.0070 |   0.67    96.51 )
 Pipeline interlocks               47 ( 0.0001   1.0071 |   0.01    96.52 )
 ICache misses                    455 ( 0.0010   1.0081 |   0.10    96.61 )
 DCache misses                   1999 ( 0.0045   1.0126 |   0.43    97.04 )
 Exceptions                        15 ( 0.0000   1.0126 |   0.00    97.04 )
 Uncached ifetches               9252 ( 0.0207   1.0333 |   1.98    99.02 )
 Uncached loads                    61 ( 0.0001   1.0334 |   0.01    99.04 )
 Sync replays                    1985 ( 0.0044   1.0379 |   0.42    99.46 )
 Special instructions             510 ( 0.0011   1.0390 |   0.11    99.57 )
 Loop overhead                   2001 ( 0.0045   1.0435 |   0.43   100.00 )
 Reset                              5 ( 0.0000   1.0435 |   0.00   100.00 )
```

In addition, the profiler allows you to create profiles based on events, such as branch penalties. This feature allows you, for example, to identify functions, lines, or instructions in your program that suffer the most from branch penalties.

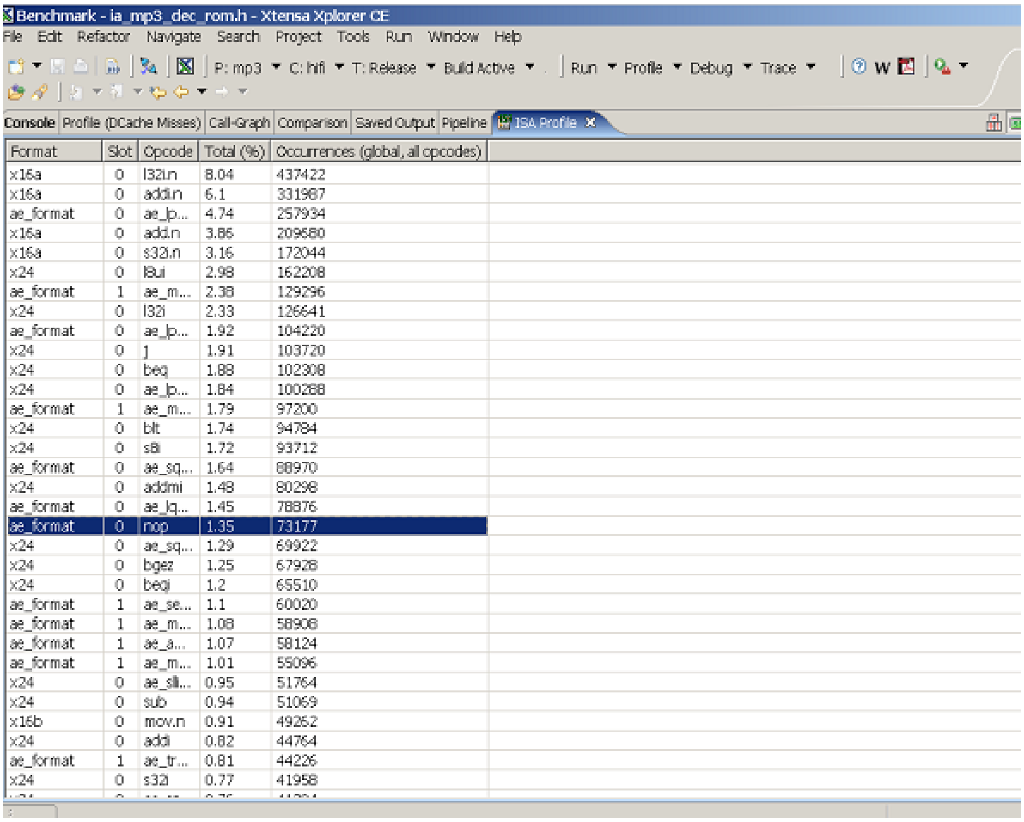
In many situations, branch penalties are a bigger problem than interlocks. Except for zero-overhead loops, every aligned branch taken in the 5-stage version of the processor's pipeline suffers a 2-cycle branch penalty. One additional penalty cycle is required for unaligned branch targets (the instruction at the branch target address is not aligned so that the entire instruction fits in a single, 32-bit-aligned fetch boundary for a 32-bit-wide instruction fetch or a single, 64-/128-bit aligned fetch boundary for a 64-/128-bit instruction fetch). A 7-stage version of the Xtensa pipeline adds one additional branch-penalty cycle.

Note: If the branch is not taken, there are no branch-penalty cycles.

Branch penalties can be mitigated or reduced by judicious use of compiler optimizations such as the use of profiling feedback. This is covered in more detail in Section 3.1.4. When compiling for speed rather than space, the tool-chain might replace a 16-bit instruction with an equivalent 24-bit instruction or might add padding to unexecuted regions of code in order to better align branch targets. Penalties can also be mitigated through processor configuration choices. A 5-stage configuration will suffer significantly fewer branch delays than a 7-stage one. A configuration with a 64- or 128-bit fetch will suffer a little less from branch delays. Additionally, the range of branch immediates in the base 24-bit Cadence architecture is limited. Branch delays can be significantly reduced by creating wider branch instructions in TIE that utilize larger immediates. Such branches are standard with the Diamond 570T core and can be easily added to any FLIX configuration by the Xtensa TIE developer. See the Cadence Instruction Extension (TIE) Language Reference Manual for details.

The output of the ISS also shows you the percentage of cycles spent in source interlocks. The pipeline interlocks whenever a result takes multiple cycles to compute, and the compiler is not able to schedule meaningful other instructions between the instruction that defines a result and the one that consumes it. Interlocks might be fundamental; some code does not have meaningful parallelism. Some might be a function of the configuration; a 7-stage pipe may have significantly more load-use interlocks. They might also be caused by non-optimal scheduling from the compiler. The latter will be covered in more detail in Chapter 3.

With FLIX (VLIW), the summary figure for interlocks might only show part of the problem. Non-optimal scheduling might not cause the processor to stall, but might prevent the compiler from filling all the VLIW slots. In such scenarios, some slots might be filled by NOP operations. The ISA profiler is a good tool for seeing how many NOP operations are dynamically executed by the processor. The ISA profiler can be used as a standalone command line tool or as a tab in the Profile tool in Xplorer. The tool can be used to give total counts for the entire application or can be broken down by application source function. The results of the ISA profiler for the MP3 example are show below. The MP3 example is highly tuned and the number of NOP operations is very small, less than 2% of the total dynamic count.


Benchmark - ia_mp3_dec_rom.h - Xtensa Xplorer CE
File Edit Refactor Navigate Search Project Tools Run Window Help
P: mp3 C: hifi T: Release Build Active Run Profile Debug Trace
Console Profile (DCache Misses) Call-Graph Comparison Saved Output Pipeline ISA Profile

Format	Slot	Opcode	Total (%)	Occurrences (global, all opcodes)
x16a	0	l32i.n	8.04	437422
x16a	0	addi.n	6.1	331987
ae_format	0	ae_lp...	4.74	257934
x16a	0	add.n	3.86	209680
x16a	0	s32i.n	3.16	172044
x24	0	l8ui	2.98	162208
ae_format	1	ae_m...	2.38	129296
x24	0	l32i	2.33	126641
ae_format	0	ae_lp...	1.92	104220
x24	0	j	1.91	103720
x24	0	beq	1.88	102308
x24	0	ae_lp...	1.84	100288
ae_format	1	ae_m...	1.79	97200
x24	0	blt	1.74	94784
x24	0	s8i	1.72	93712
ae_format	0	ae_sq...	1.64	88970
x24	0	addmi	1.48	80298
ae_format	0	ae_lq...	1.45	78876
ae_format	0	nop	1.35	73177
x24	0	ae_sq...	1.29	69922
x24	0	bgez	1.25	67928
x24	0	beqi	1.2	65510
ae_format	1	ae_se...	1.1	60020
ae_format	1	ae_m...	1.08	58908
ae_format	1	ae_a...	1.07	58124
ae_format	1	ae_m...	1.01	55096
x24	0	ae_sll...	0.95	51764
x24	0	sub	0.94	51069
x16b	0	mov.n	0.91	49262
x24	0	addi	0.82	44764
ae_format	1	ae_tr...	0.81	44226
x24	0	s32i	0.77	41958

Figure 10: ISA Profiling

2.2.5 Compiled Code Quality

While the ISA profiler and the ISS summary are good tools for obtaining data, nothing can substitute for raw human intelligence. For important loops, look at the code generated by the compiler and reason about its efficiency. You can either look at the disassembly of the actual object code generated or you can ask the compiler to save the assembly file that it generates before passing it to the assembler. The former is easier to access; for example the Profile view in Xplorer shows the disassembly along with profile counts. However, the assembly file contains human readable comments that make it easier to understand the generated code.

To see the generated code, compile with either `--save-temps` or `-keep`, or select Keep intermediate compilation files in the Xplorer's Compiler Optimization tab of the Build Properties dialog. As part of the compilation process, the compiler generates a file name.s where name.c is the original source file name[1].

Every loop in the generated .s file is annotated with information to help you understand how the compiler optimized the loop. Most inner loops are software pipelined. Software pipelining is a process where the compiler schedules together operations from multiple loop iterations of a loop. Every software pipelined loop is annotated with a message prefixed with <swps>. The following figure is an example from the MP3 application. This example was chosen because it is performance critical, yet simple enough to understand easily.

```
loopgtz a3,.LBB32_ia_mp3_dec_dct64      # [18]

.LBB30_ia_mp3_dec_dct64:        # 0x4a
#<loop> Loop body line 105, nesting depth: 1, iterations: 16
#<swps>
#<swps>   2 cycles per pipeline stage in steady state with unroll=1
#<swps>   2 pipeline stages
#<swps>   3 real ops (excluding nop)
#<swps>
#<swps>      min 2 cycles required by resources
#<swps>      min 1 cycles required by recurrences
#<swps>      min 2 cycles required by resources/recurrence
#<swps>      min 3 cycles required for critical path
#<swps>          3 cycles non-loop schedule length
#<swps>
#<freq> BB:30 => BB:30 probability = 0.93750
#<freq> BB:30 => BB:32 probability = 0.06250
        .frequency 1.000 15.938
 {      # format ae_format
        ae_lp24x2f.iu   aep0,a2,8               # [0*II+0]  id:785
        ae_orp48        aep2,aep2,aep1          # [1*II+0]
 }
 {      # format ae_format
        nop                             #
        ae_abssp24s     aep1,aep0               # [0*II+1]
 }
```

The first line in the message gives the source position of the loop (making it easy to locate) as well as an estimate of the trip count of the loop. If the trip count is a literal, the compiler will know it exactly. If the count is a variable, the compiler will make an arbitrary guess unless the program has been compiled using the feedback optimization option. With feedback, the compiler will use an average value over all runs.

[1] For compilations using -ipa, or interprocedural analysis, the compiler will instead generate a series of files in the directory binary.ipakeep/ called 1.s, 2.s, ... where binary is the name of the generated executable.

The next line gives the achieved schedule as well as the unrolling count. When software pipelining, the compiler will try to successively unroll the loop with higher counts until there is no significant performance benefit to unrolling further. When a loop is unrolled, the schedule achieved is the schedule for each unrolled iteration, so divide by the unroll factor to get a schedule in terms of the original loop. Note that for loops with small, literal, trip counts, the compiler might bias towards fully unrolling the loop. In such cases a message might refer to an outer loop in the source that has become an inner loop in the generated code.

For our example, each iteration is scheduled in two cycles. Looking at the code, one of the two instructions contains a NOP operation. At first glance, this appears to be a scheduling problem. Since the HiFi 2 architecture can issue two operations in every VLIW bundle, one might expect this loop to execute in 1.5 cycles per iteration or equivalently 3 cycles per unrolled iteration.

Looking further, you can see that the compiler believes that this loop requires at least two cycles given the machine resources. That means that if you ignore all dependences and just pack all the operations as tightly as possible, the loop will still require two cycles. As it happens, in the HiFi 2 architecture, ae_orp48 and ae_abssp24s can only be issued in the second slot of an instruction. Therefore, it is not possible to schedule this loop more efficiently. If the achieved schedule is equal to the resource limit, it is not possible to improve performance by scheduling, that is, by rearranging operations in the loop. Note however, that the limit is a lower bound. Cadence allows irregular VLIW bundling constraints. It is therefore possible that due to an irregular constraint, the actual resource limit of the loop is higher than that computed by the compiler.

The other interesting information is the line labelled cycles required by recurrences. This is an estimate of how many cycles the loop could be scheduled looking only at dependences, assuming infinite resources. If each iteration of the loop can safely be executed in parallel, the recurrence limit will always be 1. Many times a loop that appears to be parallel has a high recurrence limit. This is often caused by potential aliases that the compiler cannot disambiguate. Aliasing is discussed in more detail in Section 3.1.5.

Not every inner loop is software pipelined. Some reasons why a loop might not be software pipelined are:

- The loop contains function calls.
- The loop contains branches.
- The loop has a provably small trip count.
- The function is being compiled for space rather than speed[2].
- The compiler was not able to find a software pipelined schedule.

[2] A function will be compiled for space if you use the compiler option -Os or if you compile with feedback optimization and the function is not frequently executed or if you specify -Os in the -fopt-use file. For all functions compiled for space, the compiler inserts into the .s file a comment # Optimized for space

Inner loops that are not software pipelined yet contain no calls or branches are commented with a different formatted message as shown in the following example.

```
#<loop> Loop body line 87, nesting depth: 2, iterations: 2
#<swpf>  swp not attempted
#<loop> unrolled 2 times
#<sched>
#<sched> Loop schedule length: 16 cycles (ignoring nested loops)
#<sched>
#<sched>    16 mem refs     (100% of peak)
#<sched>    12 integer ops  ( 75% of peak)
#<sched>    28 instructions (175% of peak)
{      # format ae_format
        ae_lp24x2f.iu   aep7,a5,8               # [0]  id:114
        ae_mulzasfp24s.hh.ll    aeq3,aep2,aep3  # [0]
 }
 {      # format ae_format
        ae_lp24x2f.iu   aep5,a3,-8              # [1]  id:115
        ae_mulzaafp24s.hl.lh    aeq2,aep2,aep3  # [1]
 }
...
```

The schedule length gives the total number of cycles to execute one iteration of the loop. Any delay between the end of an iteration and the start of the next is not noted. Inside the square brackets to the right of each instruction is a comment giving the current cycle count of the instruction, assuming a start of 0 on every iteration. Gaps in the numbering shows bubbles in the schedule. Just before the actual instructions is a set of comments giving the number of memory references, ALU operations, and total instructions. Often, you can bound the potential schedule using one of these fields. For example, the HiFi 2 architecture can only issue one vector load or store per cycle. Because the example schedule issues 16 memory references in 16 cycles, it is not possible to schedule this loop any better. Note that unlike the software pipeliner, this scheduler is much less sophisticated with regards to loop unrolling. Typically, the scheduler will unroll all sufficiently small loops by a factor of two. If you want more significant unrolling, you must unroll in the source code.

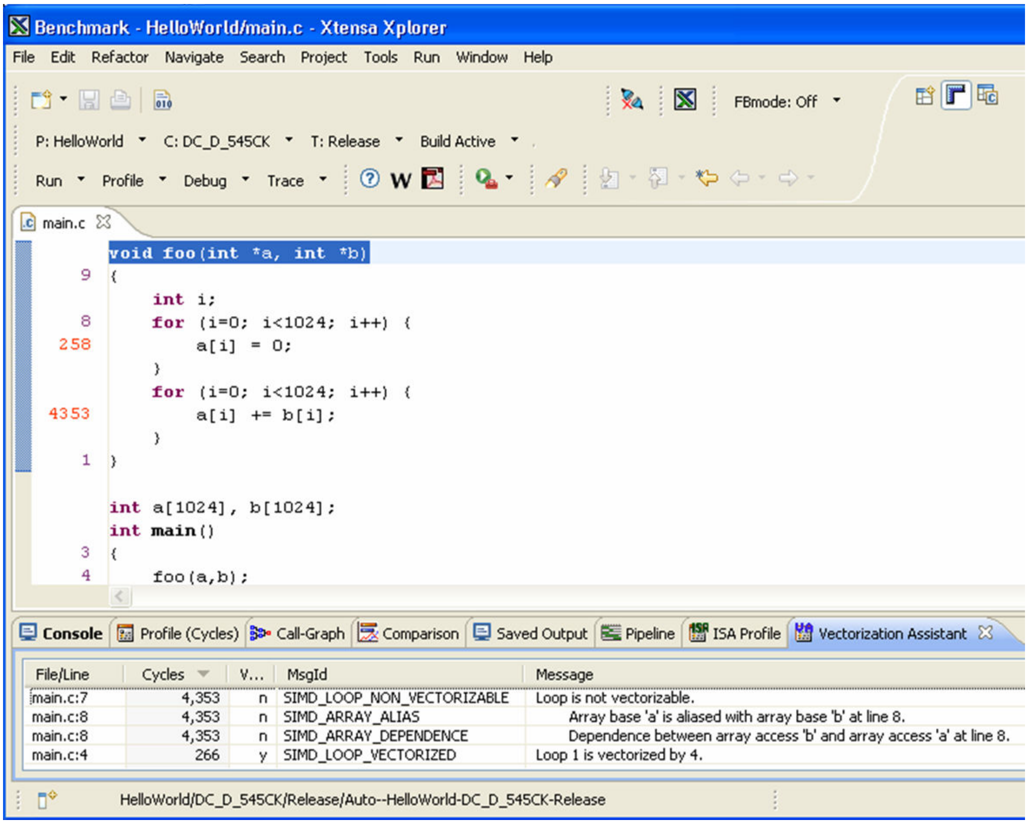
To see the effects of higher level optimizations such as vectorization or inlining, compile with the `-clist` option. For a given file foo.c, the use of this option will generate a file named foo.w2c.c in addition to the normal output files. This file is a C representation of the original file after inlining and loop optimizations. It is meant to be looked at to gain insight into what the compiler does, not to be compiled itself. For compilations using -ipa, or interprocedural analysis, the compiler will instead generate a series of files in the directory binary.ipakeep called 1.w2c.c, 2.w2c.c, ... where binary is the name of the generated executable. Note that `-clist` is not supported with C++.

When trying to vectorize your code, the vectorization assistant allows you to graphically browse through your code and see problems that prevent vectorization. To use the assistant, compile your code with `-O3 -LNO:simd`, profile your code and then enable the assistant from the Tools menu.

Consider the following example.

```
void foo(int *a, int *b)
{
    int i;
    for (i=0; i<1024; i++) {
        a[i] = 0;
    }
    for (i=0; i<1024; i++) {
        a[i] += b[i];
    }
}
```

The vectorization assistant can be seen in *Figure 11: Vectorization Assistant* on page 39. The first lines, correspond to the second loop which is not vectorizable because the two array references potentially alias each other. The vectorization assistant prints the message "Array base 'a' is aliased with array based 'b' at line 8". The first loop, on line 4, is successfully vectorized.



```
       void foo(int *a, int *b)
   9   {
           int i;
   8       for (i=0; i<1024; i++) {
 258           a[i] = 0;
           }
           for (i=0; i<1024; i++) {
4353           a[i] += b[i];
           }
   1   }

       int a[1024], b[1024];
       int main()
   3   {
   4       foo(a,b);
```

File/Line	Cycles	V...	MsgId	Message
main.c:7	4,353	n	SIMD_LOOP_NON_VECTORIZABLE	Loop is not vectorizable.
main.c:8	4,353	n	SIMD_ARRAY_ALIAS	Array base 'a' is aliased with array base 'b' at line 8.
main.c:8	4,353	n	SIMD_ARRAY_DEPENDENCE	Dependence between array access 'b' and array access 'a' at line 8.
main.c:4	266	y	SIMD_LOOP_VECTORIZED	Loop 1 is vectorized by 4.

Figure 11: Vectorization Assistant

3. Performance

Topics:

This section looks at specific uses of compiler flags and coding guidelines.

Related Links

Performance Tuning Methodologies on page 23
This section details the tools and methodologies for maximizing application performance and minimizing code size on Cadence processors.

Processor Configuration Options on page 89
Descriptions of processor configuration options

3.1 Controlling The Compiler

The compiler has many options to control the amount and types of optimizations it performs. Choosing the right options can significantly increase performance. Compiler options can be controlled with both command line tools and with Xplorer. This section describes multiple compiler options by name. These options can directly be given to the command line XCC. Inside Xplorer, you can control the compiler options via the Project/Xtensa Project Build Properties menu. Use this menu to set options globally, for a project or for a file inside of a project. The more common optimization flags are check box items in the Optimization tab of the menu. All other options can be entered directly into the Addl tab of the same menu.

By default, the compiler compiles a file with `-O0` (no optimization). The compiler makes almost no effort to optimize the code either for space or performance. The main goals for unoptimized code generation are faster compilation and ease of debugging.

The main optimization flag is the `-On` flag, where n can be set between 0 and 3. The flag `-O` by itself is equivalent to `-O2`. For production code, the lowest useful optimization level is `-O2`. The `-O1` level is only rarely useful, for example if you want better debugability than is possible with `-O2` but cannot use `-O0` because of insufficient target memory.

The `-O3` compiler flag enables additional optimizations that are mostly (but not exclusively) useful for DSP-like code. While the `-O2` flag will almost always result in better code than `-O1`, the `-O3` will usually result in faster code than `-O2` but not always. Some code will speed up by a factor of two while other will slow down a little. It is not possible to predict in advance. Most programs will be larger with `-O3` than `-O2`.

Orthogonal to the use of `-O2` or `-O3`, the `-Os` flag tells the compiler to optimize for space rather than speed. The flag `-Os` can be used with either `-O2` or `-O3`; if neither is given, the compiler assumes `-O2`. In comparison to `-O2`, `-Os` generated code will on average be 10% slower, but 15% smaller. In many scenarios, you might want to compile your important functions for speed and the rest for space. When using feedback directed compilation without `-Os`, the compiler automatically compiles less important functions for space.

The `-g` option instructs the compiler to generate symbolic debugging information when compiling. It can be used together with other optimization flags without adversely affecting performance. This flag is required for symbolic debugging and for line by line profiling.

3.1.1 Optimizing Functions Individually

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

The *-fopt-use flag* may used to specify function optimization levels by placing a list of one line entries in a text file and supplying that file name through the *-fopt-use* flag.The flag usage is as follows: *-fopt-use=<optimization_file>*, where the *<optimization_file>* is a text file with a list of lines, each with the general format:

[<directory_path>]<filename>:<procedure_name>: <optimization_string>

The optional *<directory_path>* may be needed to disambiguate file name collisions. Lines beginning with # are ignored as comments. The longest line allowed is 1024 characters. The format for the *<optimization_string>* for function-level optimization is as follows: *[-O{0,1,2,3}] [-Os]* to indicate compiling a certain function at levels 0 through 3, and/or to optimize for space. The optimization levels specified in the file will override any command line flags except for the -O0 flag. The command line -O0 flag overrides any file specifications, allowing easier debugging.

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

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

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

Using the compiler flag -fopt-gen at both compile and link time will cause the compiler to create an output file named executable.opt mentioning every function in the application in a format that is suitable for use by -fopt-use. For most functions, the optimization will be set to whatever was given in the command line. However, if feedback compilation is used, the compiler will automatically compile certain functions for space. By generating and then keeping the generated file, the user can take advantage of the decisions made by the feedback optimization without having to keep compiling the application using feedback. The use of the -fopt-use flag is meant to be robust to changes in the application. If a function is later deleted, the compiler will ignore its entry in the file. If a function is added and no entry is added to the file, the function will be compiled with whatever flags are given on the command line.

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

Rather than use a separate file, one can also use the attribute feature in the source file to set the optimization level for the attributed function to opt_level, where opt_level is one of *-O0*, *-O1*, *-O2* or *-O3*, possibly paired with *-Os*. This attribute instructs the compiler to compile the function at the specified optimization level regardless of the command-line optimization level.

For example, the following attribute instructs the compiler to compile the function func0() for space rather than speed.

```
__attribute__((optimize ("-Os"))) void func0() {
    int i;
    for(i = 0; i < 100; i++) {
      a[i] = b[i] + c[i];
    }
}
```

3.1.2 Attributes

The GCC compiler supports attributes for giving the compiler information beyond what is expressible in standard C or C++. Attributes in gcc are used similarly to pragmas in other compilers and are often used to control memory placement, alias analysis and inlining, for example. The XCC compiler supports most of the common GCC attributes and gives warning messages when encountering unsupported ones. See the *Xtensa C and C++ Compiler User's Guide* for more details.

3.1.3 Interprocedural Analysis

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

XCC implements the following interprocedural optimizations:

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

To enable interprocedural analysis and optimization, you must use the `-ipa` (or `-IPA`) command-line option both when generating object (`.o`) files and when linking them. When this option is in effect, the `.o` files produced by the compiler will not be the normal object files; instead, they will contain information about the original code, summarized in a form that is suitable for interprocedural analysis. During the link step, rather than invoking the standard linker (*xt-ld*), the XCC driver invokes the interprocedural module, which digests the summary `.o` files and performs various analyses and transformations.

This module then generates intermediate files and writes them into a temporary directory, `*binary*.ipakeep`, whose contents can be saved with `-keep`. In the final step, it invokes the compiler to transform the intermediate files into real object files, and then the standard linker to create the final output.

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

- The `-ipa` option should be used for both compiling and linking. Although the interprocedural module will accept normal `.o` object files (produced without `-ipa`) and generate correct code, it cannot derive information useful for its analyses from such files. An attempt to link interprocedural summary `.o` files without the `-ipa` option on the link line will result in an error message produced by the linker.
- You must use the XCC compiler driver (*xt-xcc* or *xt-xc++*) during the link step with IPA. Using the standard linker (*xt-ld*) will result in an error message.
- The linking step with the interprocedural module hides the entire back-end compilation and optimization, in addition to the processing of summary `.o` files. Therefore, re-linking even after changing just a single `.o` file takes much more time than without IPA.
- The interprocedural module can inline functions across file boundaries. If different command line options are used for different files compiled with IPA, which options are used for any particular function is not defined. Therefore, it is recommended (although not required) that you use the same command-line options (especially the optimization level) for all the files compiled with IPA.
- Because of the dramatic extent of code changes performed during interprocedural optimization, using `-g` in conjunction with `-ipa` will produce extremely limited debugging information. This information will allow you to do line profiling (allow a profiler to show how much time was spent in different source lines) as well as set breakpoints and step through the code in a debugger. However, examining source-level data in a debugger will not be possible.
- The `-ipa` option has no effect if it is used in conjunction with a command-line option that terminates the compilation process before producing object files (`-E` or `-S`).

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

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

Interprocedural analysis can be used in conjunction with the other optimization flags. For example, you can use `-ipa` together with `-O2` or `-O3` or `-Os`. When compiled for speed, IPA tends to inline heavily and thus increase code size. When compiled for space, IPA is capable of deleting unused functions and variables and thus is sometimes able to significantly reduce memory requirements. When compiled with feedback, IPA will automate the trade-off between speed and space and will tend to give both better speed and space characteristics.

Building Libraries with IPA

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

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

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

To build a normal, compiled, library with IPA, compile the source files with `-ipa` added to the compilation flags. Linking the library requires two flags: `-ipalib` and `-ipaentry=symbol_name` repeated for each symbol (function or variable) that is externally accessible. For optimization purposes, IPA will assume that all the symbols marked with an `-ipaentry=symbol_name` may be accessed arbitrarily outside the library and will also assume that no additional global symbols are accessed.

3.1.4 Inlining Functions

The XCC compiler will inline functions; that is, replace a function call with the actual body of the function. Particularly for small functions, inlining can significantly improve performance. XCC will inline both with and without IPA. Without IPA, XCC can only inline functions that are defined in the same file as the caller (either directly or via a header file that is included in the calling file). Even then, by default without IPA XCC will only inline static or inline functions. This is done to improve code size because non-static files must still be emitted even if they are inlined because they might also get called from other files. If you want to inline non-static functions, compile with the *-INLINE:preemptible* flag. With IPA, by default the compiler will potentially inline any function no matter how it is defined and no matter from where it is called.

The compiler uses its own heuristics to decide which functions to inline. Even if a function is marked as inline, the compiler might still decide not to inline the function. Deciding on the benefits of inlining is very complicated, and the compiler does not always make the right decision. The compiler gives you a lot of freedom if you want to control its inlining heuristics. You can request that a specific function always or never be inlined using the flags `-INLINE:must=function` or `-INLINE:never=function`. A function marked never to be inlined will not be inlined. A function marked must to be inlined will be inlined if possible. Certain function calls, such as recursive calls, can never be inlined. Similar behavior can be achieved via attributes in the code rather than via compiler options. As shown in the following two examples, a function attributed as `always_inline` will be inlined if possible while a function attributed as `noinline` will never be inlined.

```
int foo () __attribute__ ((always_inline));
static int foo2 () __attribute__ ((noinline));
```

Inlining can also be controlled more globally. The option *-INLINE:requested* causes the compiler to inline all functions marked as inline, if possible. The option `-INLINE:requested_only` inlines all functions marked as `inline` if possible and inlines no other functions.

3.1.5 Using Profiling Feedback

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

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

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

1. In the first step, set the top level **FBmode** switch to either **SW** or **HW** depending on whether you intend to generate feedback data in simulation or on a hardware target.

Select a target whose name ends in **_Feedback**. Alternatively, from the command line invoke the compiler with one of the following options:

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

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

```
-fb_create_64 filename
```

Note that for this first step, you must both compile and link your application with this flag. Also note that the compiler instrumentation uses standard library functions to do memory allocation and file I/O. If you redefine any of the library functions used by the instrumentation library, you cannot compile any file containing those redefined functions using `-fb_create`. XCC gives an error message in such situations. If you redefine any library function in a way that redefines its externally visible behavior, even if you do not compile the function with `-fb_create`, you may get unpredictable results in the instrumented version.

2. In the second step, run your application with a representative input set. In the first step, Xplorer automatically created a target named *target_Feedback* where target was the name or the target used when setting the optimization flags. Use that `target`. You may run the application multiple times with different representative sets. Each run creates a new profiling file prefixed by *filename*. The program can be run on any system that supports basic file I/O, including the simulator. This run will take significantly longer than a normal run, often several factors longer. The input sets do not have to be the same as the ones used in production (often, shorter running input sets are used). However, the closer the run is to the production run, the more accurate the information. If a particular run only invokes some subset of the common modes expected in production, it is possible and desirable to execute multiple runs.

   If your target hardware does not support a file system, in the first step, set the mode to **HW**. From the command line, use the following option:

```
-fb_create_HW filename
```

   The compiler instruments the code as in the `-fb_create_64` case, but also includes software to allow the data to be retrieved on real hardware automatically through the debugger. Code compiled and linked with this option must be executed on the hardware with the debugger attached, either through Xplorer or through xt-gdb using an OCD target.

Once the file is retrieved, it can be used identically as files produced with the other options. See the *Xtensa Software Development Toolkit User's Guide* for more information.

3. In the third step, switch the target back to the original target (making sure that FBmode is still set) or from the command line reinvoke the compiler including the following flag:

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

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

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

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

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

Using this flag, XCC optimizes a function for space whenever that function takes less than (n/10)% of the total execution time. The default value of n is 10, meaning that XCC optimizes for space any function that takes less than 1% of the total execution time.

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

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

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

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

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

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

```
    blta2,a3,.LABEL
    movi.na2,0
    retw.n
.LABEL:
    movi.na2,1
    retw.n
```

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

```
    bgea2,a3,.LABEL
    movi.na2,1
    retw.n
.LABEL:
    movi.na2,0
    retw.n
```

3.1.6 Aliasing

Consider the code in the following example. You might assume that the compiler will generate code that loads `*a` into a register before the loop and contains an inner loop that loads `b[i]` into a register and adds it into the register containing `*a` in every iteration.

```
void foo(int *a, int *b)
{
    int i;
    for (i=0; i<100; i++) {
        *a += b[i];
    }
}
```

In fact, you will find that the compiler generates code that stores `*a` into memory on every iteration. The reason is that `a` and `b` might be aliased; `*a` might be one element in the `b` array.

While it is very unlikely in this example that the variables will be aliased, the compiler cannot be sure.

There are several techniques to help the compiler optimize better in the presence of aliasing. One technique is to compile using IPA. With IPA, the compiler sees the entire program and might be able to figure out in the above example that `*a` cannot be an element of the `b` array. Using IPA has some nice advantages; it is easy and it is safe. However, even with IPA the compiler is not always able to figure out what is obvious to the programmer. Consider the next example. Sometimes the first parameter to foo is the global `a` and sometimes it is `b`, similarly for the second parameter. Unless the function happens to be inlined, the compiler is not smart enough to realize that the first parameter is only `a` when the second parameter is `b`, and therefore the compiler conservatively assumes the two parameters are aliased.

```
int a[100], b[100];
void foo(int *a, int *b)
{
    int i;
    for (i=0; i<100; i++) {
        *a += b[i];
    }
}
int bar()
{
    foo(a,b);
    foo(b,a);
}
```

Alternatively, you can use the global variables directly instead of parameters. The compiler knows that two globals cannot alias with each other. Of course, globals cannot be used inside of functions that access different data in different invocations and the overuse of globals does not promote good software engineering practices.

Parameters or local pointers scoped at the function level can have their type qualified with the `__restrict` type qualifier as show in the next figure. Pointers with the `restrict` type qualifier are assumed by the compiler to not alias any other pointers or globals for the entire dynamic duration of the function. The type qualifier is easy to use and is effective. However, you must be very careful. If two restrict pointers do in fact refer to overlapping memory locations, the compiler may generate incorrect code. Users frequently use this feature when they are not sure if pointers are aliased, and then just check if the program still generates correct results. Unfortunately, incorrect usage of restrict may or may not lead to incorrect results. So, the program may generate correct results when the qualifier is used but might start generating incorrect results after random changes are made to the code.

```
void foo(int * __restrict a, int * __restrict b)
{
    int i;
    for (i=0; i<100; i++) {
        *a += b[i];
    }
}
```

The restrict type qualifier also interacts with inlining. Consider the next example, where the compiler has been told that the portion of array a accessed in functions barney or wilma does not overlap with the portion of array b accessed in the same function. The user has said nothing about whether the portion of array a accessed in barney overlaps with the portion of array b accessed in wilma. After inlining, the compiler cannot distinguish between array references that originally came from barney from those that originally came from wilma. Therefore, the compiler cannot inline the two functions and still keep the restrict qualifier. Given the choice between preventing the inlining and dropping the restrict qualifier, the compiler chooses to prevent inlining.

```
void barney(int * __restrict a, int * __restrict b);
void wilma (int * __restrict a, int * __restrict b);
void fred(...)
{
    barney(a, b);
    wilma(a, b);
}
```

Finally, the compiler has an option, -OPT:alias=value, to globally control how the compiler handles alias analysis. Value can be one of any, typed, restrict and disjoint. The option -OPT:alias=any (or equivalently -fno-strict-aliasing) tells the compiler to make no assumption about aliasing. Unless the compiler can prove that two variables or pointers do not overlap, the compiler will assume that they do. The option -OPT:alias=typed (or equivalently -fstrict-aliasing) tells the compiler to assume that pointers to different types do not overlap. So for example, a float * pointer will not point to the same location as an int * pointer. Similarly, a pointer to a particular field of a structure will not overlap to a different field of any structure even if the underlying types of the fields are the same. With this rule, there is a special exception for variables of type char *. The compiler will conservatively assume that such pointers can alias with any other. These type based rules are requirements of the C and C++ language. A program that casts an int * pointer to a float * pointer, and then uses both pointers, is not a legal C or C++ program. Therefore, -OPT:alias=typed is the default behavior of the compiler and you must explicitly set -OPT:alias=any if your code violates the standard.

The option -OPT:alias=restrict instructs the compiler to treat every pointer as if it had the type qualifier __restrict. This option should only be used in very limited circumstances where you are sure that there are no aliases. Frequently, this option is useful as a quick testing mechanism to see if the compiler is being limited by aliasing in a particular loop. If using this flag does not improve performance, there is no point in trying any of the more conservative methods.

Even the option -OPT:alias=restrict does not eliminate all aliases since it tells the compiler that two pointers are not aliased but says nothing about the indirection of two pointers. Consider the following example. Using the option -OPT:alias=restrict tells the compiler that *a cannot equal *b any place in the function but it does not say that **a cannot equal **b. The flag -OPT:alias=disjoint is a stronger flag; it tells the compiler that arbitrary indirections off of named pointers do not alias with other pointers or with other named variables.

```
void foo(int **a, int **b)
{
    int i;
    for (i=0; i<100; i++) {
        **a += **b++;
    }
}
```

3.1.7 Loop Pragmas

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

```
#pragma no_unroll
```

This pragma disables unrolling of the loop it immediately precedes.

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

This pragma asserts the minimum trip count [3] (min), the maximum trip count (max), and the trip count's even divisibility (factor) of the loop it immediately precedes. The minimum and factor values must be exactly correct as the compiler may use this information for optimizations such as omitting unrolling remainder iterations. Incorrect values for these parameters could result in incorrect code.

The use of the min value allows the compiler to improve performance when software pipelining loops. When software pipelining, the first few and the last few iterations are split up into loop prologues and epilogues. In order to handle the situation where the trip count of the loop is smaller than the number of prologues or epilogues, the compiler must schedule each one separately and test for an early exit after each one. The use of the pragma avoids this overhead. The use of the factor value allows the compiler to avoid generating cleanup loops when vectorizing or unrolling loops. It will not typically have a major performance impact but might substantially reduce code size.

You can disable the use of this pragma with the -fno-pragma-loop-count command-line option. Additionally, the average value (avg) provides a way to tell the compiler an estimate of the average trip count for this loop. The compiler will try to use this in heuristics that require an estimate of the amount of work done each time the loop is entered.

[3] The trip count is the number of times the loop body is executed.

3.1.8 SIMD Vectorization

Many coprocessors, including ConnX, HiFi3, HiFi4, Fusion and Imaging contain SIMD (or vector) instructions, meaning, instructions that perform the same operation on multiple pieces of data, each piece occupying a slice of a register file or memory location. SIMD offers a great potential for performance improvement. The user can manually take advantage of SIMD using explicitly SIMD intrinsics or operator overloading on explicitly SIMD data types. The compiler is also able to vectorize code, automatically convert scalar code into vector code. This subsection is mainly concerned with automatic vectorization. Vectorization can be difficult to utilize effectively. To utilize vector techniques, three general conditions need to be met:

- First, fundamentally the algorithm being implemented must be performing the same independent operations on multiple pieces of data. If the algorithm is not fundamentally amenable to SIMD, it will not vectorize.
- Second, the multiple pieces of data must be contiguous in memory. Consider a simple image processing example operating on RGB data. Imagine that a particular algorithm is manipulating only the red data. RGB data is typically stored in one of two ways; separate arrays for each component or a single, interleaved array. If the algorithm is only manipulating the red data, the red data must be stored in its own array. The compiler is not going to deinterleave the storage for you.
- Third, the compiler must be able analyze the code and prove that the first two conditions hold. The remainder of this subsection discusses this third condition.

Vectorization is invoked in one of two ways:

1. Vectorization can be invoked as a check box in the Optimization tab of Xplorer's Build Properties menu when developing software for a configuration that supports vector instructions.
2. Vectorization can be invoked on the command line by using the `-O3 -LNO:simd` command-line option.

Table 1: Vectorization Options on page 54 summarizes the options that control the vectorizer. The remainder of this subsection describes the use of the vectorizer in more detail.

Table 1: Vectorization Options

Option	Description
`-LNO:simd`	Enables vectorization. This option is only valid in conjunction with the `-O3` option.
`-LNO:simd_v`	Prints a summary report of compiler vectorization efforts to standard output.
`-TENV:X=4`	Vectorizes a loop containing an *IF* statement even if vectorizing the statement might cause the system to load a value that otherwise would not have been loaded.

Option	Description
`-LNO:aligned_pointers=on`	Treats all pointers used as array bases as if they are aligned correctly for use with the vectorizer.
`-LNO:aligned_formal_pointers=on`	Treats all pointers passed as formal parameters to functions as if they are aligned correctly for use in the vectorizer.

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

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

1. Do not modify your code at all, and simply use the automatic vectorization feature in XCC. This is the easiest method, but it may not take full advantage of all hardware capabilities.
2. Modify your code to meet one of the constrained memory models described in *Aliasing* on page 50. Depending on the source code, these modifications may be relatively easy or very difficult. This task is more difficult than not modifying your code at all, but can result in dramatically improved performance.
3. Rewrite the code partially or completely using explicit vector C types along with operator overloading on those types.
4. Rewrite the code partially or completely using Xtensa processor's TIE intrinsic functions. This method is the most time-consuming (although it is not very difficult), but allows you total control and has the most potential for improving the performance of time-critical portions of your application.

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

3.1.8.1 Viewing the Results of Vectorizing Transformations

When writing and debugging SIMD code, it is often useful to see how the compiler has optimized the code you originally wrote. As described in *Compiled Code Quality* on page 35, you can look at the `.s` assembly file that the compiler generates if the `-S` or `-keep` command-line option is in effect. But a higher, C-level view may be much easier to understand.

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

Consider the following example:

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

Compile the example targeting the ConnX Vectra LX coprocessor using

```
xt-xcc -O3 -LNO:simd -clist main.c
```

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

```
#include "main.w2c.h"
_INT32 main()
{
    vec4x40 V_00;
    vec4x40 V_;
    vec4x40 V_0;
    vec4x40 V_4;
    _INT32 reg2;
    _INT32 i;

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

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

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

3.1.8.2 Aligning Data for Vectorization

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

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

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

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

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

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

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

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

Assuming that you relax the memory model by using a proper aliasing option (see *Aliasing* on page 50), the vectorizer will be able to transform the loop. However, as it does not have enough information about the alignment of the arrays, it will assume that they are not aligned. An extract from the `.w2c.c` file generated by using the `-clist` command on this example follows. While the inner loop is as efficient as the provably aligned example, there is a significant outer loop overhead in using alignment registers.

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1.8.3 Controlling Vectorization Through Pragmas

#pragma concurrent

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

```
void copy (int *a, int *b, int n)
{
    int i;
#pragma concurrent
    for (i = 0; i < n; i++)
        a[i] = b[i];
}
```

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

#pragma simd_if_convert

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

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

XCC can vectorize the code using if-conversion as can be seen in the following pseudo assembly code:

```
loop
    lvv2, a[i]
    lvv3, b[i]
    addvv4, v3, 1
    lvv5, cond[i]
    eqvb0, v5, 3
    movtv2, v4, b0
    svv2, a[i]
```

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

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

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

Some processors, such as the ConnX BBE 32 and 64 families, have support for predicated load and store instructions, for aligned loads and stores. These processors do not need to unconditionally load or store data in order to vectorize a loop. Therefore, for aligned loads and stores, the use of the pragma or the compiler switch should not make a difference. The pragma and compiler switch will still make a difference when the loop contains potentially unaligned loads or stores.

#pragma simd

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

3.1.8.4 Speculation

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

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

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

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

n=0: No speculation is allowed.

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

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

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

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

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

3.1.8.5 Features and Limitations of the Vectorizer

To use XCC's vectorization feature effectively, it is helpful to know and understand some of its limitations. This feature works best when the program is written in a vectorizable form. Some programs may need to be rewritten to use a different algorithm or different data organization to take full advantage of the vectorizer. The most common limitations relate to aliasing as described in the earlier *Aliasing* on page 50 section. Other limitations and features follow.

Limitation in the Stride of Memory Accesses

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

Choosing which Loop to Vectorize

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

Guard Bits

The XCC vectorizing feature does not obey the standard C/C++ integer overflow semantics when targeting several coprocessors including the ConnX, Vectra LX and Imaging coprocessors. The ConnX BBE16 register files, for example, contain additional guard bits to allow for increased precision on intermediate arithmetic: 20 bits for `short` data types and 40 bits for `int` data types. Results are saturated to 16 or 32 bits respectively when stored back to memory. The vectorizer automatically uses the extra precision, thereby changing the behavior of programs that would otherwise suffer from overflow.

Related Links

Aliasing on page 50

3.1.8.6 Vectorization Analysis Report

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

Xplorer automatically adds `-LNO:simd_v` to the compile line of any file if vectorization is selected and presents the information in the *Vectorization Assistant* view.

Analysis information is reported using the following format:

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

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

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

The following analysis file is generated by the compiler:

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

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

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

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

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

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

Vectorization Messages on page 63 lists the analysis messages currently produced by the vectorizer.

3.1.8.7 Vectorization Messages

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

SIMD_ACCESS_GAPS	Gaps in memory access sequence (base %s1). The compiler is unable to vectorize array accesses with gaps. A loop can only be vectorized if it can be transformed so that it accesses sequential array elements on each loop iteration. The example loop below can't be vectorized because it accesses only the even elements of the arrays.
	`for (i = 0; i < N; i += 2)`
	`    a[i] = b[i] + c[i];`

SIMD_ANALYSIS_OP	Unable to analyze %s. The current version of the compiler does not support vectorization of this type of operation.
SIMD_ANALYSIS_REDUCTION	Unable to analyze reduction of %s. The current version of the compiler does not support this type of vector reduction.
SIMD_ARRAY_ALIAS	Array base %s is aliased with array base %s at line %d. The compiler conservatively assumes that the two array bases may point to the same memory location. See *Aliasing* on page 50

SIMD_ARRAY_DEPENDENCE	Dependence between array access '%s' and array access '%s' at line %d.
	Vectorization requires that loop iterations can be executed in parallel. This message indicates that data dependences make parallel execution illegal. The example loop below can't be vectorized because the data computed in each iteration (i) depends on the data computed in the previous iteration (i-1).
	`for (i = 1; i < N; i++)`
	`    b[i] = b[i - 1] + a[i];`

SIMD_BAD_ACCESS	%s is not a simple array access. The compiler is unable to vectorize loops that contain complex indirect memory accesses.
SIMD_BAD_ACCESS_STRIDE	Bad array access stride.

	The compiler is unable to vectorize array accesses with negative or non-small strides.
SIMD_BAD_CALL	Loop contains call to function. The compiler is unable to vectorize loops that contains function calls.
SIMD_BAD_LOOP	Non-array indirect memory accesses, calls, or unhandled control-flow. Loops with non-array indirect memory accesses, function calls, non-vectorizable input TIE instructions or unsupported control-flow such as `GOTO` statements and loops with non-computable trip counts cannot be vectorized.
SIMD_BAD_LOOP_UPPER_BOUND	Unsupported loop upper bound expression. The compiler tries to standardize all `for` loop upper bounds to `index <= bound` expressions. Vectorization is not possible if the compiler is unable to perform this transformation.
SIMD_BAD_TIE_OP	Non-vectorizable input TIE instruction %s. The compiler is unable to vectorize loops containing TIE instructions that access states, memory or queues.
SIMD_BOOLEAN	Boolean coprocessor required.

SIMD_DATA_DEPENDENCE	Data dependences prevent vectorization.
	Vectorization requires that loop iterations can be executed in parallel. This message indicates that data dependences make parallel execution illegal. The `i` loop in the sample loop nest below cannot be vectorized because the data computed in each iteration `i` depends on the data computed in the previous iteration `i-1`.
	`for (i = 0; i < N; i++)`
	`    a[i] += a[i-1];`

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

	The verbosity option was set (`-LNO:simd_v`) without setting the appropriate optimization options (`-O3 -LNO:simd`).
SIMD_IF_DIFF_VL	Mismatched if-statement vector lengths. The then and else branch of an if statement must be vectorized by the same amount.
SIMD_IF_HAS_LOOP	Loop in if-statement. The current version of the compiler is unable to vectorize a loop containing an `if` statement if the `if` statement in turn contains another loop.
SIMD_IF_UNSAFE_ACCESS	Unsafe accesses in an if-statement. Vectorization vectorizes an `if` statement using a technique called if-conversion that executes both sides of the conditional and then conditionally merges the results. Since if-conversion executes the `then` and the `else` of the `if` statement before testing the if-condition, safe transformation requires that each operation or memory access is present on both branches of the `if` statement. If the operation or access is not on both sides of the `if` statement but it can be speculated safely, the `-TENV:X=?` or the `-LNO:simd_agg_if_conv` compiler option, or the `simd_if_convert` pragma can be used to force transformation.
SIMD_LOOP_BEGIN	Vectorization analysis begins with a new loop. Before vectorizing a loop, the compiler performs complete operator and data dependence analysis on the loop to check the legality of the transformation.
SIMD_LOOP_COST_MODEL	Vectorization of this loop is not beneficial. The compiler may choose not to vectorize a loop because the estimated performance of the vectorized loop is worse than the original, non-vector loop performance or than the performance achieved by vectorizing another loop in the same loop nest.

SIMD_LOOP_NON_VECTORIZABLE	Loop is not vectorizable. The compiler is unable to vectorize this loop.
SIMD_LOOP_STEP	Non-unit loop step %d. Vectorization is supported only if the loop step is equal to 1 or can be normalized by the compiler.
SIMD_LOOP_TOO_DEEP	Loop nest is too deep. Vectorization analysis is not performed on outer loops that contain inner loops that are too deeply nested.
SIMD_LOOP_VECTORIZABLE	Loop is vectorizable.
SIMD_LOOP_VECTORIZATION_TRY	XCC is trying to vectorize the loop.
SIMD_LOOP_VECTORIZATION_RETRY	XCC is trying again to vectorize the loop. The vectorizer may try to vectorize the same loop using different vectorization factors. This message will be output for each tried vectorization factor.

SIMD_NEGATIVE_ACCESS_STRIDE	Negative memory access stride (base %s).
	The compiler is unable to vectorize array accesses with negative stride. For example, the loop below can't be vectorized because array `b` is accessed with a negative stride.
	`for (i = 0; i < N; i ++)`
	`    a[i] = b[N - i - 1];`

SIMD_NO_COMMON_VL	No common vectorization length is available. All operations in a loop must be vectorized by the same amount.
SIMD_NO_COND_MOVE	No conditional move instruction for vector type %s. Vector if-conversion requires a conditional move instruction for merging a result of the specified vector type, but no such instruction is available in the current processor configuration.
SIMD_NO_COPROC	Vectorization turned off by `-mno-use-coproc`.

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

SIMD_NO_GUARDS

Insufficient guard bits.

SIMD_NO_LOAD_CONV

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

SIMD_NO_LOOP

No well-formed loops in function.

The compiler can only vectorize loops with computable bounds.

SIMD_NO_REDUCTION

Processor configuration does not support vector reduction of %s.

SIMD_NO_SCALAR_CONV

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

SIMD_NO_SELECT

No vector select instruction available to transform %s accesses.

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

SIMD_NO_SNL

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

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

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

Loop j and k are at the same nesting depth, so loop i is not simply nested. The compiler may transform the loop nest by fusing the j and k loop into one or by fissioning the i loop into two simply nested loops -- one for i and j,

	and another one for i and k. If an outer loop cannot be transformed to simply-nested loops, it will not be vectorized.
SIMD_NO_STORE_CONV	Processor configuration does not support store conversion from %s to %s.
SIMD_NO_TYPE_CONV	Processor configuration does not support type conversion from %s to %s.
SIMD_NO_VECTOR_OP	Processor configuration does not support vector %s.
SIMD_NO_VECTOR_TYPE	Processor configuration does not support vector %s.
SIMD_NO_VECTOR_TYPE_SIZE	Processor configuration does not support %d-way vector %s.
SIMD_NON_COUNTABLE_LOOP	The loop is not a countable for-loop. The compiler can vectorize only single induction variable countable for-loops, or loops that can be transformed automatically into this form. Use of non-int induction variables or pointer accesses may prevent the compiler from recognizing countable for-loops.
SIMD_PRAGMA_IGNORED	Unable to apply the #pragma at line %d to a for-loop. #pragma 'concurrent', 'simd' or 'simd_if_convert' must be placed immediately before the countable for-loop to which it applies. For example,

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

SIMD_PROC_BEGIN	Vectorization analysis for function %s.
SIMD_SCALAR_ARRAY_STORE	Scalar array store %s prevents vectorization. A non-vector store to an array element prevents vectorization. For example, the loop

below can't be vectorized because of the use of array 'temp'.

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

SIMD_SCALAR_DEPENDENCE

Bad scalar dependences (variable %s).

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

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

SIMD_SIGNED_POW_TWO_DIV

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

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

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

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

```
unsigned int ui;
signed int si;
```

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

The equivalent shift expressions for the two variables are:

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

SIMD_SMALL_TRIP_COUNT	Loop trip count is too small. Vectorization is not supported if the smallest available vectorization factor is less than the loop iteration count.
SIMD_TRAPEZOIDAL_INNER_LOOP	Inner loop bounds depend on outer loop index. Loop nests where the loop bounds of an inner loop depend on an outer loop index are called trapezoidal. The compiler may vectorize trapezoidal loops of depth 2. In this example, loops j and k can be vectorized but loop i cannot because it contains multiple trapezoidal inner loops.

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

SIMD_UNALIGNED_LOAD	Processor configuration does not support unaligned loads.
SIMD_UNALIGNED_STORE	Processor configuration does not support unaligned stores.
SIMD_UNSIGNED_LOOP_UPPER_BOUND	Unsigned < upper bound expression may prevent loop vectorization. The compiler tries to standardize all `for` loop upper bounds to `index <= bound` expressions. When an unsigned < loop bound expression is transformed to <=, the extra code required to guarantee correctness cannot be vectorized. Therefore, unsigned <

	upper bounds may prevent vectorization of enclosing loops.
SIMD_VARIANT_SHIFT	Shift by loop-variant shift amount not supported (%s). The compiler is unable to vectorize expressions that shift by an amount that varies across loop iterations. For example, if `i` is the loop index, the left-shift `a[i] << b[i]` has a loop-variant shift-amount `b[i]`.
SIMD_VAR_STRIDE	Processor configuration does not support variable stride accesses. In certain cases, variable stride accesses can be vectorized using indexed updating scalar-to-vector loads (LS.XU) and vector select instructions. However, these instructions are not available in the current processor configuration.

3.1.9 Super Software Pipelining

For many applications, particularly DSP code, performance can be dominated by the performance of inner loops. As discussed in *Compiled Code Quality* on page 35, the compiler will try to software pipeline every inner loop. In order to software pipeline a loop, the compiler needs to come up with an ordering, or schedule, for all the operations in the loop. Finding the optimal ordering is an NP-complete problem, meaning that it is impossible to always find the best schedule in a tractable amount of time. Instead, the compiler uses heuristics to try many but not all potentially good orderings. Looking at an inner loop, you may feel that the compiler should have scheduled it better. Sometimes you are wrong, sometimes the problem is unrelated to scheduling, but sometimes the problem is simply that the compiler did not find the best schedule.

With super software pipelining, you can ask the compiler to exhaustively, but intelligently, search all possible schedules. To use this feature, add the following `#pragma super_swp` immediately preceding the inner loop. In many cases, the compiler will complete quickly, but in some cases the compiler will never complete. The process can be made somewhat faster by guiding the compiler and telling it how much to unroll the inner loop and how many cycles (after unrolling) to try to schedule. The following code is an example where the software pipeliner is asked to not unroll an inner loop and schedule it in 28 cycles.

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

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

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

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

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

3.2 Prefetching

The Xtensa processor includes a hardware prefetch option geared for systems with long memory latency.

When the processor detects a stream of cache misses (either data or instruction), it can speculatively prefetch ahead up to four cache lines and place them in a buffer close to the processor or in some configurations into the L1 data cache (there is no support for prefetching directly into the L1 instruction cache). In addition, the user can manually issue prefetch instructions.

For configurations that support prefetching, hardware prefetching is enabled by default in the reset code provided by Cadence with a low setting. By default, on configurations that support it, data prefetches are placed into the L1 data cache. You can use the following HAL calls to explicitly disable prefetching or to change its aggressiveness in different sections of your code. With more aggressive prefetching the hardware will prefetch earlier when detecting a stream and will prefetch more lines ahead. Assuming sufficient bus bandwidth, performance will improve with more aggressive prefetch but the system will require more bandwidth. Prefetching instructions and data can be controlled separately.

```
#include <xtensa/hal.h>

int xthal_set_cache_prefetch(unsigned long long mode);
```

The value returned is not meant for direct use or interpretation; it is, however suitable for passing to a subsequent call to `xthal_set_cache_prefetch`.

The mode parameter can be one of the following:

- The value returned from a previous call to `xthal_set_cache_prefetch` or `xthal_get_cache_prefetch`.
- One of the following constants, which apply to both instruction and data caches:

    - XTHAL_PREFETCH_ENABLE (enable cache prefetch)
    - XTHAL_PREFETCH_DISABLE (disable cache prefetch)
- A bit-wise OR of two cache prefetch mode constants, one for the instruction cache:
    - XTHAL_ICACHE_PREFETCH_OFF (disable instruction cache prefetch)
    - XTHAL_ICACHE_PREFETCH_LOW (enable, less aggressive prefetch)
    - XTHAL_ICACHE_PREFETCH_MEDIUM (enable, midway aggressive prefetch)
    - XTHAL_ICACHE_PREFETCH_HIGH (enable, more aggressive prefetch)
    - XTHAL_ICACHE_PREFETCH(n) (explicitly set the InstCtl field of the PREFCTL register to 0..15. See the Prefetch Architectural Additions section of the Prefetch Unit Option chapter in the Xtensa Microprocessor Data Book for details.)
- and one for the data cache:
    - XTHAL_DCACHE_PREFETCH_OFF(disable data cache prefetch)
    - XTHAL_DCACHE_PREFETCH_LOW(enable, less aggressive prefetch)
    - XTHAL_DCACHE_PREFETCH_MEDIUM(enable, midway aggressive prefetch)
    - XTHAL_DCACHE_PREFETCH_HIGH(enable, more aggressive prefetch)
    - XTHAL_DCACHE_PREFETCH(n) (explicitly set the DataCtl field of the PREFCTL register to 0..15. See the Prefetch Architectural Additions section of the Prefetch Unit Option chapter in the Xtensa Microprocessor Data Book for details.)
    - XTHAL_DCACHE_PREFETCH_L1_OFF (prefetch data to prefetch buffers only)
    - XTHAL_DCACHE_PREFETCH_L1 (on configurations that support it, prefetch directly to L1 data cache)

For easier simulation, prefetching can also be disabled in the simulator using the `xt-run --prefetch=0` flag. Disabling prefetching from the simulation command line will override any HAL calls.

3.2.1 Software Prefetching

Prefetching can also be individually controlled via software

Prefetching can also be individually controlled via software using the following GCC extension.

```
__builtin_prefetch(addr);
```

Software prefetches can be used for either data or instructions. They can be used in addition to or instead of hardware prefetching. If hardware prefetching is disabled, the software prefetches are still enabled.

Note that some implementations of the hardware do not prefetch into the caches-they prefetch into a small, 8- or 16-entry buffer outside of the cache. This means that when you use software prefetching, you must be careful not to prefetch too far ahead. Otherwise, the data will be overwritten before it's needed by the processor.

Some configurations support block prefetches. These can be invoked using the following macros.

```
#include <xtensa/core-macros.h>

xthal_dcache_block_prefetch_for_read(void *addr,unsigned size);

xthal_dcache_block_prefetch_for_write(void *addr, unsigned size);

xthal_dcache_block_prefetch_modify(void *addr, size);

xthal_dcache_block_prefetch_read_write(void *addr, unsigned size);

xthal_dcache_block_prefetch_for_read_grp(void *addr, unsigned size);

xthal_dcache_block_prefetch_for_write_grp(void *addr, unsigned size);

xthal_dcache_block_prefetch_modify_grp(void *addr, unsigned size);

xthal_dcache_block_prefetch_read_write_grp(void *addr, unsigned size);
```

These macros prefetch a block of data (potentially many cache lines) in the background. The _for_read and for_write macros are hints to the hardware. Prefetching undesired data might hurt performance but will not affect the correctness of the program. The _modify macros will mark a set of lines as being present in the cache without actually fetching the data from memory. They are equivalent to writing random values into memory. They are meant for data that will be stored before being read. Using these macros on memory that will be read before being written, will result in undefined behavior. Note that the _modify macros need not align their requests to cache line boundaries. The hardware will use normal prefetches for any partial lines. The hardware block prefetcher will issue prefetch requests in a round-robin fashion among all incomplete blocks unless one uses the _grp macros. These macros signify the start of a new group. All prefetches for an older group will complete before any prefetches for a new group are issued. Note that additional macros are also avaiable but are less useful for application level software.

3.3 General Coding Guidelines

Previous sections discuss utilizing the compiler to more effectively improve the performance of your application. This section contains a series of general coding guidelines.

3.3.1 Avoid Short Scalar Datatypes

Most Xtensa instructions operate on 32-bit data. There are no 16-bit addition instructions. The system emulates 16-bit addition using 32-bit arithmetic instructions and this sometimes forces the system to convert a 32-bit value into a 16-bit one by extending the sign of a 16-bit result to the upper bits of a register. Consider this example:

```
int foo(short a, short b)
{
    short c;
    c = a+b;
    return c;
}
```

You might expect that the routine would use a single add instruction to add a and b and place the result in the return register. However, C semantics say that if the result of a+b overflows 16 bits, the compiler must set c to the sign-extended value of the result's lower 16 bits, leading to this code: [4]

```
entry a1,32
add.n a2, a2, a3
slli a2, a2, 16
srai a2, a2, 16
retw.n
```

Much more efficient code would be generated if `c` was declared to be an `int`. In general, avoid using `short` and `char` for temporaries and loop variables. Try to limit their use to memory-resident data. An exception to this rule is multiplication. If you have 16-bit multipliers and no 32-bit multipliers, make sure that the multiplicands' data types are 8-bit or 16-bit types.

3.3.2 Use Locals Instead of Globals

Global variables carry their values throughout the life of a program. The compiler must assume that the value of a global might be used by calls or by pointer dereferences. Consider this example:

```
int g;
void foo()
{
    int i;
    for (i=0; i<100; i++){
        fred(i,g);
    }
}
```

Ideally, `g` would be loaded once outside of the loop, and its value would be passed in a register into the function fred. However, the compiler does not know that `fred` does not

[4] The code below assumes the configuration does not have the sext instruction. If the configuration had the sext instruction, the two shift instructions would be replaced with a single *sext* instruction.

modify the value of g. If fred does not modify g, you should rewrite the code using a local variable as in *Figure 12: Replacing Locals With Globals* on page 77. Doing so saves a load of g into a register on every loop iteration.

```
int g;
void foo()
{
    int i, local_g=g;
    for (i=0; i<100; i++){
    fred(i,local_g);
    }
}
```

Figure 12: Replacing Locals With Globals

Alternatively, if the function fred does not read or write any global variables other than its function arguments, you can mark the function with the pure attribute as show in *Figure 13: Pure Attribute* on page 77. If the function fred reads but does not write global variables, you can instead using the const attribute. For this example, both const and pure will eliminate the load. In other examples where the variable is written in the calling function, the use of pure will eliminate a store but const will not.

```
int g;
void __attribute__ ((pure)) fred(int, int)
void foo()
{
    int i;
    for (i=0; i<100; i++){
        fred(i,g);
    }
}
```

Figure 13: Pure Attribute

3.3.3 Use Arrays Instead of Pointers

Consider a piece of code that accesses an array through a pointer such as in this example:

```
for (i=0; i<100; i++)
    *p++ = ...
```

In every iteration of the loop, *p is being assigned, but so is the pointer p. Depending on circumstances, the assignment to the pointer can hinder optimization. In some cases it is possible that the assignment to *p changes the value of the pointer itself, forcing the compiler to generate code to reload and increment the pointer during each iteration. In other cases, the compiler cannot prove that the pointer is not used outside the loop, and the compiler must therefore generate code after the loop to update the pointer with its incremented value. To avoid these types of problems, it is better to use arrays rather than pointers as shown below.

```
for (i=0; i<100; i++)
    p[i] = ...
```

3.3.4 Minimizing Conditionals

As mentioned in *Using Profiling Feedback* on page 47, taken branches impose at least a two cycle penalty. The problem is even worse for branches inside of loops because the compiler is typically unable to schedule operations across branches. For performance-critical regions of code, there are several techniques that can be used to mitigate the effects of branches.

Avoid Conditionals

Keep your code simple and avoid conditionals that are not needed. Consider the example in *Figure 14: Conditional Multiplication* on page 78. A conditional is used to avoid doing a multiplication in the case that a multiplicand is zero. Unless your configuration has no hardware multiplier, checking for zero will be more expensive than actually doing the multiplication. The simple code, shown in *Figure 15: Unconditional Multiplication* on page 78, will perform better.

```
for (i=0; i<n; i++)
    if (a[i] != 0)
        c[i] += a[i] * b[i];
```

Figure 14: Conditional Multiplication

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

Figure 15: Unconditional Multiplication

Consider the example in *Figure 16: Conditionals that Can Be Replaced by Lookups* on page 78. The variable result is set to different values depending on the value of a. Instead of a series of conditionals, you can create a 16-element lookup array to hold the value of result for each possible value of a as shown in *Figure 17: Lookups Instead of Conditionals* on page 79. The use of the lookup array is a trade-off, requiring more memory, but less time.

```
if (a == 1) result = 2;
else if (a == 7) result = 5;
else if (a == 15) result = 20;
else result = 0;
```

Figure 16: Conditionals that Can Be Replaced by Lookups

```
const char look[16] = {    0, 2, 0, 0,
                           0, 0, 0, 5,
                           0, 0, 0, 0,
                           0, 0, 0, 20};
...
result = look[a];
```

Figure 17: Lookups Instead of Conditionals

Order Conditionals

Given a series of conditional statements such as in *Figure 16: Conditionals that Can Be Replaced by Lookups* on page 78, the compiler will order them in the order given. If the variable a is usually 15, for example, check for the value 15 first. That way the checks against the other values will not be executed.

Switch Statements

Given a `switch` statement, the compiler has a complicated set of rules for best optimizing the switch. If the number of cases is small, the compiler will implement the switch using a series of `if` statements to check for every value in series. If the number is large and dense (most values within a range are used), the compiler will use a jump table. Otherwise, the compiler will do a binary search using a series of `if` statements. The actual code generated will differ when compiling for space versus compiling for speed. Therefore, for a series of conditionals based on a single value, it is usually better to use a `switch` statement than to write the code manually using `if` statements. However, when using a `switch` statement, it helps to tell the compiler which conditions are more common. This can be done by compiling with feedback optimization as described in *Using Profiling Feedback* on page 47. If this is not possible, for a `switch` with a small number of cases, it might be better to order the cases so that the more frequent ones occur first.

Use #pragma frequency_hint

Every taken branch incurs at least a two cycle penalty, but branches that are not taken do not incur any penalty. If the compiler knows whether a conditional is usually true or usually false, the compiler can try to rearrange the code to minimize the number of taken branches. As described in *Using Profiling Feedback* on page 47, you can use pragmas or profiling feedback techniques to provide the compiler with information about branches.

Related Links

Using Profiling Feedback on page 47

3.3.5 Passing Function Parameters

Consider a situation where you want to write a function that computes the value of some variable, `x`, in the caller. You can either have the function return a value and assign the result of the function to `x`, or alternatively, you can pass the address of `x` into the function and have the function assign `*x` directly inside the function. It is better to have the function return a value. By passing the address of a variable into the function, the compiler must assume that the address is saved away by the function and any pointer dereference anywhere in the program might actually change the value of `x`.

Similarly, scalar variables should always be passed by value. Passing the address of a scalar variable forces the compiler to conservatively assume that the address of the variable is saved by the function.

In contrast, if a structure or class is large enough, it should be passed by reference rather than by value. A structure passed by value must be completely copied on entry to a function. How large is large enough? Unfortunately, it depends. You must trade-off the overhead from the copying versus the limitations caused by poorer compiler analysis when passing pointers. Typically structures of three or fewer fields should always be passed by value in performance-critical code.

Avoid variable arguments. While potentially convenient, variable argument facilities are not efficient.

3.3.6 Use Direct Calls

Avoid indirect calls. These are calls via function pointers. Particularly with IPA, the compiler is not able to analyze indirect calls and must assume that an indirect function might cause unknown side effects like modifying global or pointer variables. Even without IPA, every indirect call requires that the address of the call be loaded, leading to additional overhead.

3.4 Floating Point

Cadence offers multiple options for implementing floating point calculations. In base processor configurations, floating point computations are emulated using the integer functional units. Typical emulation times for base operations can be from 35 to 100 cycles if you have an integer multiplier to approximately 1,000 cycles if you do not.

Cadence offers a double precision floating point acceleration option. For a modest number of gates, 4,000-7,000, performance for the base operations is improved by a factor of 2 to 3. Divide is improved by almost a factor of 7. This option will also improve the emulation time of integer division by a factor of 2 to 3 when compiling code with the `-mcoproc` option.

Cadence also offers a hardware single-precision or double-precision floating point coprocessor. With this coprocessor, most base single (or double) precision floating point operations are fully pipelined, meaning that you can issue one every cycle. Note that the C and C++ language require that a floating point computation be promoted to double precision if any of the operands or results are of type `double`, and if you only have the single-precision coprocessor, double precision arithmetic is emulated using integer operations whether or not you have the single-precision coprocessor. This is frequently an issue with the use of literals, which are by default double precision. Adding the value `1.0` to a single precision variable gets translated by the compiler into a 64-bit emulation routine rather than a single, single-precision floating point add instruction. To avoid this problem either add the suffix `F` to all single precision literals or compile with the flag `-fsingle-precision-constant`.

Note that with the traditional ABI, floating point function arguments are never passed in floating point registers. If you pass a floating point variable to a function, the compiler will move the variable into the integral AR registers before the call and move them back inside the function. With the optional Hardware Floating Point ABI, arguments are passed directly in floating point registers.

Cadence offers several DSP coprocessors, such as Fusion and HiFi 3/4, with optional floating point units. These are compatible with but not identical to the core hardware floating point options. For example, each one uses a DSP coprocessor register file rather than a dedicated register file for floating point variables. All such coprocessors only support the Hardware Floating Point ABI.

The XCC compiler, when compiling at optimization level -O3, will by default transform floating point code in ways that might not be bit-exact with the original code. Consider the following example:

```
for (i=0; i<100; i++)
    p[i] = p[i] / n;
```

Rather than issue a divide every iteration of the loop, the compiler might compute the reciprocal of n outside of the loop and then just multiply that reciprocal by p[i] inside of the loop. The resultant code will be much faster but less accurate. If you need bit-exact code, compile with *-fno-unsafe-math-optimizations*. Similarly, if you are using -O2 but don't care about bit-exactness, you can compile with *-funsafe-math-optimizations*.

3.5 C++ Language

C++ is gaining greater acceptance as a language suitable for embedded systems. When used effectively, C++ offers many software engineering advantages with no performance penalties. When used ineffectively, C++ can cause severe bloat, performance penalties and can even result in unreadable and unmaintainable code. It is well beyond the scope of this guide to teach effective programming in C++. However, this section will briefly describe the performance implications of some basic C++ features that might cause you problems.

3.5.1 Streams

C++ encourages using streams for I/O rather than the traditional `printf` from the C library, although the traditional `printf` is also supported. The stream library is very large. A traditional hello program using `printf` requires about 35KB using the `newlib` library, 20KB using Xclib, 11KB using the `uclibc` library and only 7KB using the version of `uclibc` that does not support floating point. In contrast, the stream version shown below requires 286KB with the `newlib` library, 143KB with the `uclibc` library, 56KB using Xclib and is not supported with the `uclibc` library without floating point.

```
using namespace std;
#include <iostream>
int main()
{
    cout << "Hello \n";
    return 0;
}
```

If your application is large, the overhead of a library might not matter. However, for small applications, the use of the stream library can dominate the code size of the application.

3.5.2 Exception Handling

C++ exceptions allow a program to throw an exception which is caught by a try-block higher in the call chain, thus allowing a jump across function boundaries. C++ exceptions can replace the traditional technique of checking every function call for errors and returning the error further up to one's own caller. The use of exception handling entails overhead even if exceptions are not used. Consider the example shown below. In the leftmost box, the C++ function `fred` calls the function `wilma`, which may or may not contain code that might throw an exception. In the right most box, is a C variant of the code. The call to `wilma` checks for the return of an error value and if it finds one, passes the error value (-1) back to its caller and also cleans up by deleting the locally allocated variable `f`.

```
class foo;
void fred()
{
    foo f;
    wilma();
    ...
```

```
class foo;
int fred()
{
    foo f;
    if (wilma()==-1) {
        delete f;
        return -1;
    }
    ...
```

When looking at the call to `wilma`, the C++ compiler does not know whether or not the call contains code to `throw()` an exception. If the call does throw an exception, the compiler must be prepared to clean up the local variable `f`, by calling its destructor during the processing of the exception. The compiler does this by creating special tables in the binary containing code to call the appropriate destructors relevant for each call site. The optimizer must conservatively hinder optimization to safely handle a potential throw.

In the case that an exception is actually thrown, the C++ code is somewhat slower than the C variant, but that should not matter much since thrown exceptions should not be common. In the case that an exception is not actually thrown, the C++ code might even be faster than the C code since no comparison is required. However, the C++ code pays some overhead for every function call while the C code only pays the overhead for code that is explicitly checking the return value. Similarly, the C++ code will be larger, particularly in the common case that many C applications do not check every function call.

Because of the overheads even when not being used, XCC does not enable exception handling by default. If you use exception handling, you will by default get an error message from XCC, and you must then explicitly enable exception handling with `-fexceptions`.

3.5.3 Templates

C++ templates allow the writing of generic functions, functions that are parameterized by type. A much simplified example of a templatized traditional linked list data structure is shown below. The template defines a list of arbitrary type `T`. Normally, generic functions would be defined to manipulate the list; for example, adding or deleting an element. The variable declaration for `my_int_list`, instantiates a list of integers. If the list template contains a function, `Append`, and that function is invoked on the variable, `my_int_list`, the compiler will create a version of the function `Append` for the `int` datatype. Templates allow you to write one piece of code that is used for multiple datatypes. Since instantiation happens at compile time rather than at run time, template functions can be more efficient than manually written code that attempts to handle multiple types. However, because the preprocessor creates a version of an instantiated function for every type that is instantiated, the use of templates often leads to greatly increased code size.

```
template <class T>
class LIST
{
    T data;
    class LIST *next;
};

LIST<int> my_int_list;
```

3.5.4 Virtual Functions

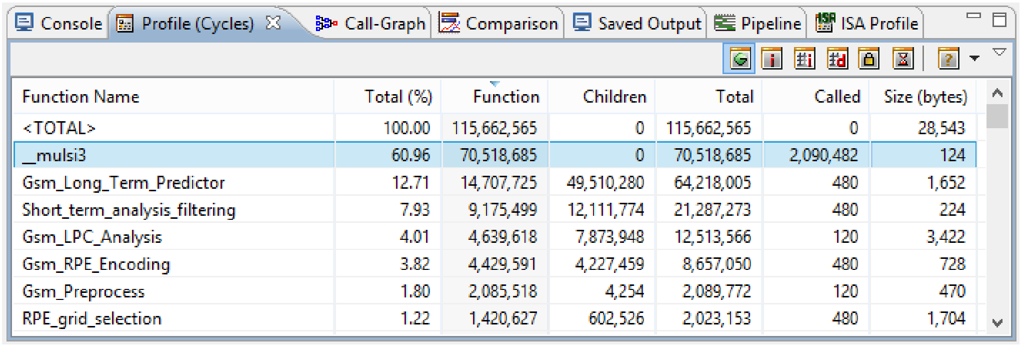
Virtual functions can be used together with C++ inheritance. Given a base class and a set of other classes derived off of the base class, a different implementation of a virtual function can be defined for every derived class. Given a variable declared but not defined as the base type, when a virtual function is applied to that variable, the compiler will invoke the version of the virtual function for the actual derived type. The compiler implements virtual functions by creating a table of function pointers. At run time, the actual function address for the derived type is obtained by indexing into the table. With virtual functions, every function call is an indirect call and suffers the overhead of indirect calls.

3.6 GSM

A more realistic example of some of the techniques discussed

In this section, some of the techniques that we discussed will be applied to a more realistic example GSM Toast from the Mediabench benchmarks. See http://euler.slu.edu/~fritts/mediabench for details about these benchmarks.

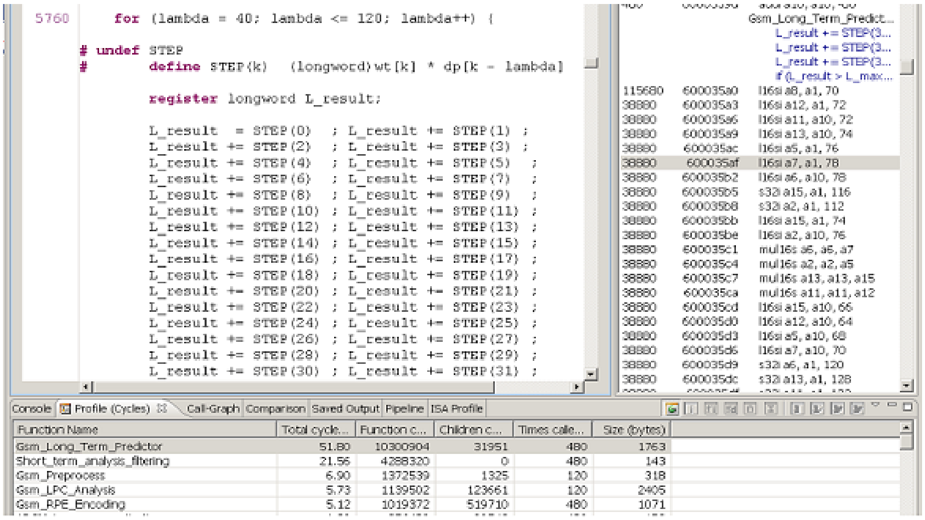
We started with a base configuration and compiled the application using the -O2 optimization. The profile results are shown in *Figure 18: Base GSM Profile* on page 84. The results are similar to the results we saw for the multiplication kernel in *Figure 1: Profiling in Xplorer* on page 25. The large majority of the time is spent in emulating multiplications.



Function Name	Total (%)	Function	Children	Total	Called	Size (bytes)
<TOTAL>	100.00	115,662,565	0	115,662,565	0	28,543
__mulsi3	60.96	70,518,685	0	70,518,685	2,090,482	124
Gsm_Long_Term_Predictor	12.71	14,707,725	49,510,280	64,218,005	480	1,652
Short_term_analysis_filtering	7.93	9,175,499	12,111,774	21,287,273	480	224
Gsm_LPC_Analysis	4.01	4,639,618	7,873,948	12,513,566	120	3,422
Gsm_RPE_Encoding	3.82	4,429,591	4,227,459	8,657,050	480	728
Gsm_Preprocess	1.80	2,085,518	4,254	2,089,772	120	470
RPE_grid_selection	1.22	1,420,627	602,526	2,023,153	480	1,704

Figure 18: Base GSM Profile

We added a 16-bit multiplier to our configuration and reran the application. In *Figure 19: Adding a Multiplier* on page 85 we see the results.



```
5760     for (lambda = 40; lambda <= 120; lambda++) {

# undef STEP
#         define STEP(k)   (longword)wt[k] * dp[k - lambda]

          register longword L_result;

          L_result  = STEP(0)   ; L_result += STEP(1)  ;
          L_result += STEP(2)   ; L_result += STEP(3)  ;
          L_result += STEP(4)   ; L_result += STEP(5)   ;
          L_result += STEP(6)   ; L_result += STEP(7)   ;
          L_result += STEP(8)   ; L_result += STEP(9)   ;
          L_result += STEP(10)  ; L_result += STEP(11)  ;
          L_result += STEP(12)  ; L_result += STEP(13)  ;
          L_result += STEP(14)  ; L_result += STEP(15)  ;
          L_result += STEP(16)  ; L_result += STEP(17)  ;
          L_result += STEP(18)  ; L_result += STEP(19)  ;
          L_result += STEP(20)  ; L_result += STEP(21)  ;
          L_result += STEP(22)  ; L_result += STEP(23)  ;
          L_result += STEP(24)  ; L_result += STEP(25)  ;
          L_result += STEP(26)  ; L_result += STEP(27)  ;
          L_result += STEP(28)  ; L_result += STEP(29)  ;
          L_result += STEP(30)  ; L_result += STEP(31)  ;
```

```
                       Gsm_Long_Term_Predict...
                            L_result += STEP(3...
                            L_result += STEP(3...
                            L_result += STEP(3...
                            if (L_result > L_max...
115680    600035a0    l16si a8, a1, 70
38880     600035a3    l16si a12, a1, 72
38880     600035a6    l16si a11, a10, 72
38880     600035a9    l16si a13, a10, 74
38880     600035ac    l16si a5, a1, 76
38880     600035af    l16si a7, a1, 78
38880     600035b2    l16si a6, a10, 78
38880     600035b5    s32i a15, a1, 116
38880     600035b8    s32i a2, a1, 112
38880     600035bb    l16si a15, a1, 74
38880     600035be    l16si a2, a10, 76
38880     600035c1    mul16s a6, a6, a7
38880     600035c4    mul16s a2, a2, a5
38880     600035c7    mul16s a13, a13, a15
38880     600035ca    mul16s a11, a11, a12
38880     600035cd    l16si a15, a10, 66
38880     600035d0    l16si a12, a10, 64
38880     600035d3    l16si a5, a10, 68
38880     600035d6    l16si a7, a10, 70
38880     600035d9    s32i a6, a1, 120
38880     600035dc    s32i a13, a1, 128
```

Console | Profile (Cycles) | Call-Graph | Comparison | Saved Output | Pipeline | ISA Profile

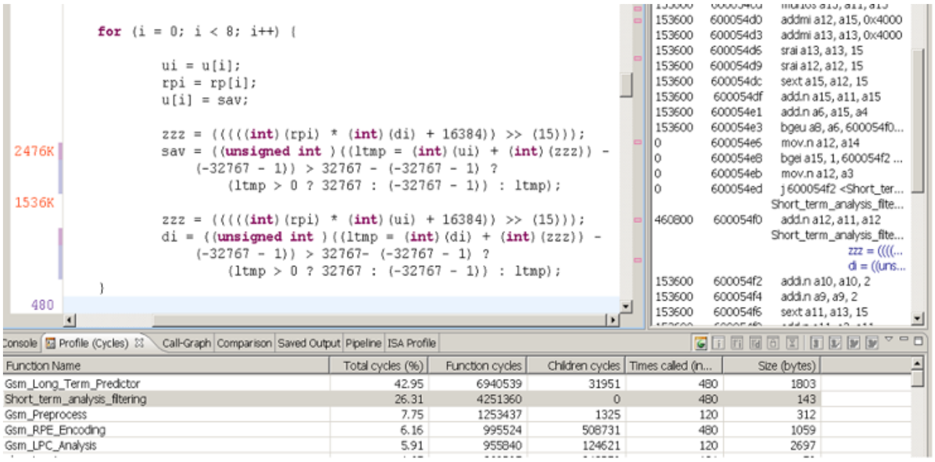
Function Name	Total cycle...	Function c...	Children c...	Times calle...	Size (bytes)
Gsm_Long_Term_Predictor	51.80	10300904	31951	480	1763
Short_term_analysis_filtering	21.56	4288320	0	480	143
Gsm_Preprocess	6.90	1372539	1325	120	318
Gsm_LPC_Analysis	5.73	1139502	123661	120	2405
Gsm_RPE_Encoding	5.12	1019372	519710	480	1071

Figure 19: Adding a Multiplier

Performance has significantly improved. The top routine in the profile is now Gsm_Long_Term_Predictor. Looking at the profile in the source code, almost all the time is spent in a single loop that computes an FIR filter with the inner, dot product loop, fully unrolled. Each iteration of the loop needs to use the same 40, loop-invariant values, wt[0] through wt[39]. The Cadence processor does not have 40 registers that can be used to store all the values. Therefore, the compiler must reload at least some of the values on every iteration, leading to suboptimal performance. A better solution is to fission the loop into multiple loops as shown below. Each fissioned loop requires a single load and a single store to hold the temporary summation for each iteration of lambda, but by carefully selecting how many loops to create, you can avoid any loading of the wt array in every iteration.

```
for (lambda = 40; lambda <= 120; lambda++) {
     tmp[lambda] =  STEP(0) ;
     tmp[lambda] += STEP(1) ;
     tmp[lambda] += STEP(2) ;
     tmp[lambda] += STEP(3) ;
     tmp[lambda] += STEP(4) ;
     tmp[lambda] += STEP(5) ;
}
for (lambda = 40; lambda <= 120; lambda++) {
     tmp[lambda] += STEP(6) ;
     tmp[lambda] += STEP(7) ...
```

The compiler fissions automatically when compiling with -O3. Therefore, performance can be improved in this case simply by recompiling with -O3. Results are shown in *Figure 20: Compiling Using -O3* on page 86. Performance of Gsm_Long_Term_Predictor improved 30% and overall performance improved almost 20% compared to the original C code on the same configuration. Gsm_Long_Term_Predictor is still the most time consuming function, but the generated code looks reasonable. We next turn our attention to the next routine on the profile, Short_term_analysis_filtering. As can be see in *Figure 20: Compiling Using -O3* on page 86, the code is computing some multiplications that are being clamped to fit into 16 bits. The set of conditionals that is computing the clamping is fairly convoluted, and the compiler is not able to recognize the clamping. Therefore, the compiler leaves branches inside the inner loop and the scheduler is not able to optimize the loop well. In *Figure 21: Optimizing Short_term_analysis_filtering* on page 86. , we cleaned up the loop to make the clamping clearer. The compiler is able to optimize the loop much better and performance of the function improves approximately 30%.



```
          for (i = 0; i < 8; i++) {

                  ui = u[i];
                  rpi = rp[i];
                  u[i] = sav;

                  zzz = (((((int)(rpi) * (int)(di) + 16384)) >> (15)));
2476K             sav = ((unsigned int )((ltmp = (int)(ui) + (int)(zzz)) -
                      (-32767 - 1)) > 32767 - (-32767 - 1) ?
                          (ltmp > 0 ? 32767 : (-32767 - 1)) : ltmp);
1536K
                  zzz = (((((int)(rpi) * (int)(ui) + 16384)) >> (15)));
                  di = ((unsigned int )((ltmp = (int)(di) + (int)(zzz)) -
                      (-32767 - 1)) > 32767- (-32767 - 1) ?
                          (ltmp > 0 ? 32767 : (-32767 - 1)) : ltmp);
          }
  480
```

```
153600   600054d0   addmi a12, a15, 0x4000
153600   600054d3   addmi a13, a13, 0x4000
153600   600054d6   srai a13, a13, 15
153600   600054d9   srai a12, a12, 15
153600   600054dc   sext a15, a12, 15
153600   600054df   add.n a15, a11, a15
153600   600054e1   add.n a6, a15, a4
153600   600054e3   bgeu a8, a6, 600054f0...
0        600054e6   mov.n a12, a14
0        600054e8   bgei a15, 1, 600054f2 ...
0        600054eb   mov.n a12, a3
0        600054ed   j 600054f2 <Short_ter...
                  Short_term_analysis_filte...
460800   600054f0   add.n a12, a11, a12
                  Short_term_analysis_filte...
                                  zzz = ((((...
                                  di = ((uns...
153600   600054f2   add.n a10, a10, 2
153600   600054f4   add.n a9, a9, 2
153600   600054f6   sext a11, a13, 15
```

Function Name	Total cycles (%)	Function cycles	Children cycles	Times called (in...	Size (bytes)
Gsm_Long_Term_Predictor	42.95	6940539	31951	480	1803
Short_term_analysis_filtering	26.31	4251360	0	480	143
Gsm_Preprocess	7.75	1253437	1325	120	312
Gsm_RPE_Encoding	6.16	995524	508731	480	1059
Gsm_LPC_Analysis	5.91	955840	124621	120	2697

Figure 20: Compiling Using -O3

```
for (i = 0; i < 8; i++) {/* YYY */
    ui = u[i];
    rpi = rp[i];
    u[i] = sav;

    zzz = (rpi*(short) di + 16384) >> 15;
    sav = ui+zzz;
    if (ui+zzz > (32*1024-1)) {
        sav = 32*1024-1;
    } else {
        sav = ui+zzz;
    }
    if (sav < (-32*1024)) {
        sav=-32*1024;
    }

    zzz = (rpi*ui + 16384) >> 15;
    di = di+zzz;
    if (di > (32*1024-1)) {
        di = 32*1024-1;
    }
    if (di < (-32*1024)) {
        di=-32*1024;
        }
}
```

Figure 21: Optimizing Short_term_analysis_filtering

Given the configuration, the generated code now looks good for both key loops. Taking a closer look at the generated assembly, however, we see that in both loops there are many opportunities for issues of multiple operations in parallel. Taking advantage of FLIX, or VLIW, is potentially promising. The processor generator allows the selection of a generic 3-way VLIW coprocessor compatible with the Diamond DC-570T core. Selecting the option and rerunning the example improved the performance of the application by an additional 45%.

While improving performance, FLIX also has the potential to increase code size, and in fact the use of FLIX did increase the size of the GSM benchmark from 17.3 KBytes to 20.4 KBytes. As mentioned in *Using Profiling Feedback* on page 47, feedback has the potential to improve code size by automatically compiling important functions for speed and unimportant ones for space. Using feedback together with GSM brought the code size back down to 19.0 KBytes while further improving performance by 7%.

4. Processor Configuration Options

Topics:

Descriptions of processor configuration options

The power of the Xtensa processor is the power to tailor your processor (or choose a pre-tailored processor) to match your application. Using TIE allows the greatest freedom in customizing your processor. However, even without TIE, there is much you can do to configure your processor for your application or application domain. This section concentrates on instruction set configuration and memory system options that are directly related to application development. Aspects of configuration such as implementation, interrupts and vectors and memory management are beyond the scope of this book. Software configuration options are described in *Software Configuration Options* on page 18.

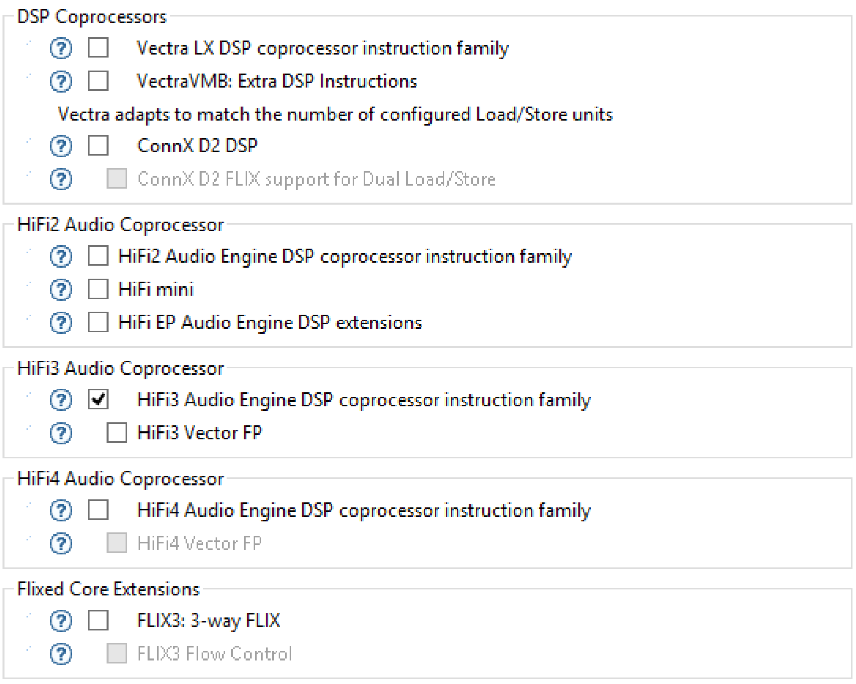
Note that most of these options change the generated hardware. However, using the instruction set simulator, it is easy to do quick, software-only experiments before settling on a final configuration.

Subsections describe the processor configuration options organized in the same page structure as the Xplorer Configuration Editor.

This includes all available options for processors of type Xtensa LX without knowledge of the options available to a particular XPG account. Not all option combinations are compatible; the Xplorer Configuration Editor advises when conflicts are detected.

4.1 Processor Selections

Processor and Coprocessor Options



DSP Coprocessors

- [ ] Vectra LX DSP coprocessor instruction family
- [ ] VectraVMB: Extra DSP Instructions

Vectra adapts to match the number of configured Load/Store units

- [ ] ConnX D2 DSP
- [ ] ConnX D2 FLIX support for Dual Load/Store

HiFi2 Audio Coprocessor

- [ ] HiFi2 Audio Engine DSP coprocessor instruction family
- [ ] HiFi mini
- [ ] HiFi EP Audio Engine DSP extensions

HiFi3 Audio Coprocessor

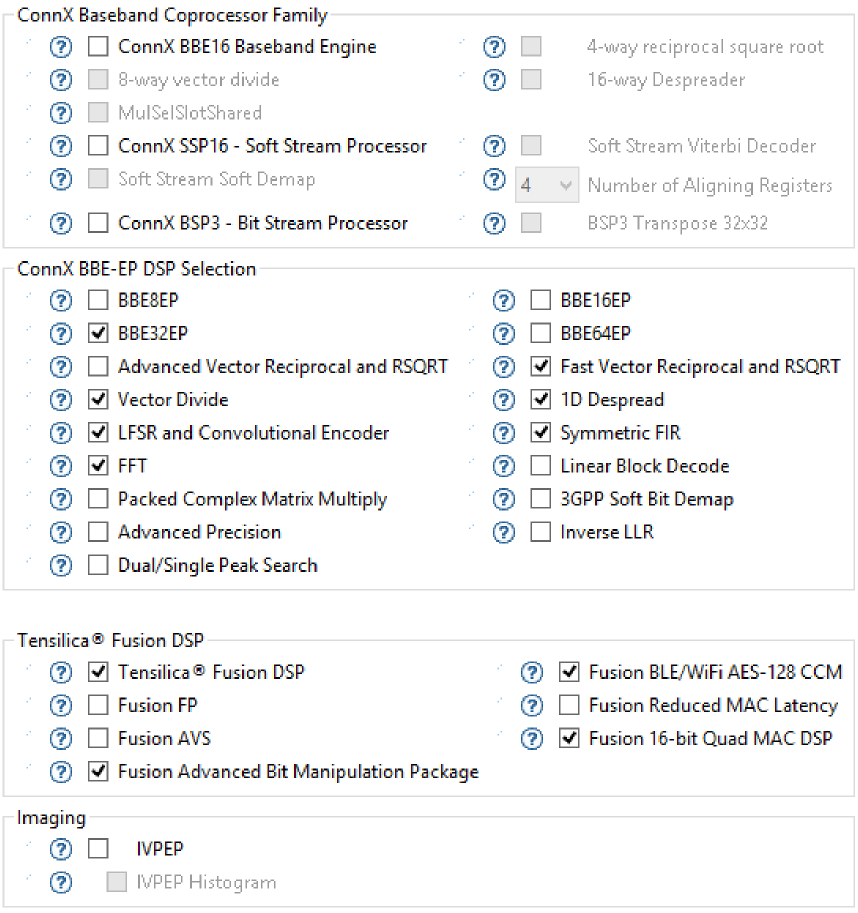
- [✓] HiFi3 Audio Engine DSP coprocessor instruction family
- [ ] HiFi3 Vector FP

HiFi4 Audio Coprocessor

- [ ] HiFi4 Audio Engine DSP coprocessor instruction family
- [ ] HiFi4 Vector FP

Flixed Core Extensions

- [ ] FLIX3: 3-way FLIX
- [ ] FLIX3 Flow Control



ConnX Baseband Coprocessor Family

☐ ConnX BBE16 Baseband Engine	☐ 4-way reciprocal square root
☐ 8-way vector divide	☐ 16-way Despreader
☐ MulSelSlotShared	
☐ ConnX SSP16 - Soft Stream Processor	☐ Soft Stream Viterbi Decoder
☐ Soft Stream Soft Demap	4 Number of Aligning Registers
☐ ConnX BSP3 - Bit Stream Processor	☐ BSP3 Transpose 32x32

ConnX BBE-EP DSP Selection

☐ BBE8EP	☐ BBE16EP
✓ BBE32EP	☐ BBE64EP
☐ Advanced Vector Reciprocal and RSQRT	✓ Fast Vector Reciprocal and RSQRT
✓ Vector Divide	✓ 1D Despread
✓ LFSR and Convolutional Encoder	✓ Symmetric FIR
✓ FFT	☐ Linear Block Decode
☐ Packed Complex Matrix Multiply	☐ 3GPP Soft Bit Demap
☐ Advanced Precision	☐ Inverse LLR
☐ Dual/Single Peak Search	

Tensilica® Fusion DSP

✓ Tensilica® Fusion DSP	✓ Fusion BLE/WiFi AES-128 CCM
☐ Fusion FP	☐ Fusion Reduced MAC Latency
☐ Fusion AVS	✓ Fusion 16-bit Quad MAC DSP
✓ Fusion Advanced Bit Manipulation Package	

Imaging

☐ IVPEP

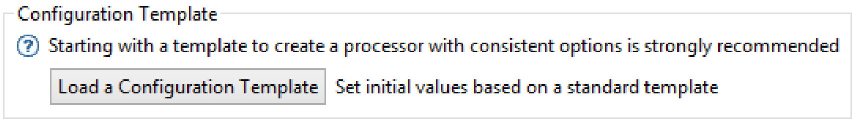
☐ IVPEP Histogram

The main set of processor and coprocessor options.

Note that most of these are distinct products, and will only be available if enabled for your XPG account.

4.1.1 Load a Configuration Template

Initialize the configuration from a template - replacing all current selections



Configuration Template

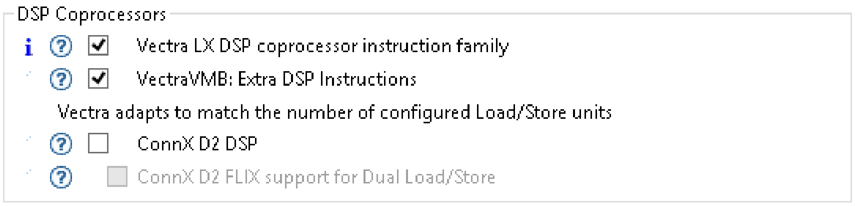
Starting with a template to create a processor with consistent options is strongly recommended

Set initial values based on a standard template

When configuring a processor, it is recommended that you start from a processor template. Most DSP and DPU options depend on multiple consistent configuration selections (e.g. requirements for instruction and data widths), and a template can help by satisfying the appropriate dependencies. You can then add other compatible options to suit your needs.

Note that the list of templates offered depends on the XPG login's access to the referenced processor and configuration options.

4.1.2 ConnX D2 and Vectra



DSP Coprocessors

- [✓] Vectra LX DSP coprocessor instruction family
- [✓] VectraVMB: Extra DSP Instructions

Vectra adapts to match the number of configured Load/Store units

- [ ] ConnX D2 DSP
- [ ] ConnX D2 FLIX support for Dual Load/Store

The following subsections describe the different ConnX D2 and Vectra coprocessors with 2, 4 and 8 MAC capabilites.

4.1.2.1 ConnX D2 DSP Engine

A dual-MAC, 16-bit communications DSP engine with ITU-T Intrinsic and TI C6x Intrinsic programming support

The ConnX D2 configuration option, which is described fully in the ConnX D2 DSP Engine User's Guide, contains a 16-bit Dual MAC SIMD (single-instruction / multiple-data) unit, coupled with a 2-way FLIX (VLIW) processor used to execute multiple instructions in parallel. The ConnX D2 option also includes 275 DSP instructions, which deliver optimal performance for a wide array of algorithms written in C.

One of the main strengths of the ConnX D2 option is the ease of programmability. Due to the advanced compiler technology and combined SIMD / FLIX, the compiler has many options for mapping algorithms written in C to high performance hardware of ConnX D2. One example of this is the case of when vectorization fails (the SIMD engine cannot be used). The compiler may still speed up loop execution using the ConnX D2 option's dual-issue FLIX abilities, possibly with the two MAC units allocated to each of the two FLIX slots. This is all possible from algorithms written in C, without the need for hand-coded assembly code.

An Xtensa processor with the ConnX D2 configuration option can be programmed either with ANSI standard C, TI C6x fixed-point intrinsics, ITU-T fixed-point intrinsics, or ConnX D2 C intrinsics for fixed-point data. The majority of TI C6x and ITU-T intrinsics map 1-to-1 into ConnX D2 instructions. After mapping, the compiler further optimizes performance. For example, ITU reference code is scalar code base. The compiler can take mapped ITU code and optimize it to execute on the SIMD engine and utilize parallel FLIX execution to give increased performance.

The ConnX D2 option adds an 8-entry, 40-bit data register file (XD_DR) that is used to hold 2x16-bit vectors, 32-bit scalars, or 16-bit scalars. The extensions also add a 4-entry, 32-bit register file (XD_UR) that is used as a buffer by the aligning updating load/store operations.

4.1.2.1.1 ConnX D2 Dual Load/Store FLIX

Optional additions to the ConnX D2 engine to support dual load/store FLIX

This ConnX D2 option allows an additional FLIX format to be added when the basic D2 configuration has two load-store units, under the user's control. This new FLIX format is a three-slot 64-bit FLIX format which adds many basic AR (address register) and D2 DSP instructions to the three slots in the format, as well as taking advantage of the second load-store unit by adding load-store operations to slot 2. There is a hardware performance, power and area cost to adding this new option to ConnX D2; hence, it is a user selectable option. The out of the box performance of many DSP kernels, especially those which are load/store bound, will improve with this option.

Restriction: LX6++ only

4.1.2.2 ConnX Vectra LX option

ConnX Vectra LX DSP coprocessor instruction family

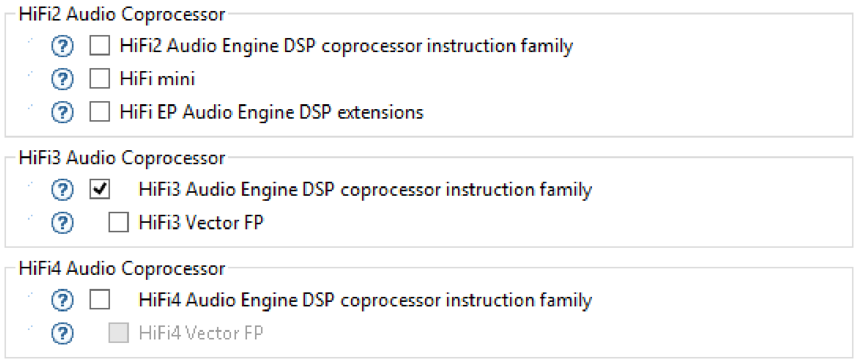
The ConnX Vectra LX DSP option is a high performance, four MAC, 16-bit SIMD and VLIW DSP coprocessor suitable for general purpose DSP processing needs. The XCC compiler is able to vectorize and schedule C/C++ applications to automatically take advantage of the coprocessor. The ConnX Vectra LX DSP coprocessor supports either one or two load/store units, each capable of loading or storing 128-bits per cycle. The variant is determined automatically from the number of load/store units in the processor.

4.1.2.2.1 VectraVMB option

The VectraVMB option augments the Vectra LX with additional specialized DSP functional units This includes four additional multipliers as well as specialized instructions for Viterbi processing and bit unpacking.

4.1.3 HiFi Audio Engine Options

HiFi Audio Engine DSP coprocessors


HiFi2 Audio Coprocessor
- ☐ HiFi2 Audio Engine DSP coprocessor instruction family
- ☐ HiFi mini
- ☐ HiFi EP Audio Engine DSP extensions

HiFi3 Audio Coprocessor
- ✓ HiFi3 Audio Engine DSP coprocessor instruction family
- ☐ HiFi3 Vector FP

HiFi4 Audio Coprocessor
- ☐ HiFi4 Audio Engine DSP coprocessor instruction family
- ☐ HiFi4 Vector FP

The following subsections describe the different HiFi Audio Engine DSP coprocessors.

4.1.3.1 HiFi 2 Audio Engine option

HiFi 2 Audio Engine DSP coprocessor instruction family

Cadence HiFi 2/EP Audio Engine is a highly optimized audio processor geared for efficient execution of audio and voice codecs and pre- and post-processing modules. The HiFi 2/EP Audio Engine is a configuration option that can be included with the Xtensa LX processor.

HiFi 2, a SIMD (single-instruction/multiple-data) processor, has the ability to work in parallel on two data items at the same time. For example, it allows for one operation to perform two 24-bit additions in parallel, with each addition occupying half of a 48-bit AE_PR register. The favored data size of the architecture is 24 bits for operands and 56 bits for accumulators, providing eight extra guard bits. HiFi 2 also supports 16-bit loads and stores which expand into the 24-bits available in the AE_PR registers. HiFi 2 supports dual 32x16-bit multipliers, in addition to the dual 24x24-bit, but because the 32-bit operand must come from the accumulator register, 32x16-bit multipliers are often less efficient than the 24x24-bit. Support for a single 16x16-bit multiplier using a 32-bit accumulator enables better support for voice algorithms require bit exact 32-bit saturation.

HiFi 2 is a VLIW architecture, supporting the execution of two operations in parallel. DSP loads and stores, bit-stream and Huffman operations and core operations are available in slot 0 of a VLIW instruction. DSP MAC and ALU operations are available in slot 1.

4.1.3.2 HiFi Mini option

Encode HiFi2/EP using narrow (40 bit) encoding. Requires Max Instruction size of 5 bytes

HiFi Mini is an enhanced version of the HiFi 2 architecture especially well suited for extremely low power voice recognition and voice processing. The use of a 40-bit VLIW instruction word (compared to the 64-bits of HiFi 2) as well as support for additional 16-bit instructions and bidirectional shifts result in significantly smaller code size. Additional instructions have been added for voice recognition applications.

Restriction: LX4++ only

4.1.3.3 HiFi EP option

HiFi EP Audio Engine DSP extensions to HiFi 2

The HiFi EP Audio Engine DSP extension configuration option extends the HiFi 2 Audio Engine ISA with 32x24-bit multiply/accumulate operations, circular buffer loads and stores, a slot 0 select unit and bidirectional shifts.

HiFi EP by default includes a prefetch option geared for systems with long memory latency. When the HiFi EP processor detects a stream of cache misses (either data or instruction), it can speculatively prefetch ahead up to four cache lines and place them either in a buffer close to the processor or in the data cache itself. In addition, the user can manually issue prefetch instructions.

4.1.4 HiFi 3 Audio Engine option

HiFi 3 Audio Engine DSP coprocessor instruction family

Cadence HiFi 3 Audio Engine is a highly optimized audio processor geared for efficient execution of audio and voice codecs and pre- and post-processing modules. It goes beyond the two MAC, two issue, HiFi 2/EP architecture with four multipliers, three VLIW slots, good support for 32x16-bit and 32x32-bit multiplication, a true 64-bit data path and native support for ITU-T/ETSI intrinsics. The extra resources provide for significant performance improvements compared to HiFi 2/EP, particularly on pre/post-processing algorithms as well as voice codecs. The support for 32-bit audio as well as ITU-T/ETSI intrinsics, including automatic vectorization, provides much better performance on out-of-the-box C programs and voice algorithms.

HiFi 3 is backward compatible at the C/C++ source level with HiFi 2/EP. Any algorithm written in C/C++, including all HiFi 2/EP packages from Cadence, can simply be recompiled on HiFi 3 and will get modest performance improvements. For maximum performance, key kernels may need to be retuned for the HiFi 3 architecture.

All HiFi 3 Audio Engine operations can be used as intrinsics in standard C/C++ applications. In addition, when compiling with automatic vectorization or with the -mcoproc option, the compiler will automatically use HiFi 3 operations when compiling standard C code.

Cadence HiFi 3 Audio Engine consists of two main components: a DSP subsystem and a subsystem to assist with bit stream access and variable-length (Huffman) encoding and decoding.

Restriction: LX4++ only

4.1.4.1 HiFi3 Vector FP

Vector Floating Point extensions to HiFi 3 Audio Engine

HiFi 3 supports an optional 2-way SIMD IEEE floating point unit. The floating point unit shares the AE_DR register file with the rest of HiFi 3. Therefore, standard loads, stores and selects can all be used together with floating point compute operations. Every floating point operation, other than the conversion to integral types, is replicated two ways. The same operation is performed on each half of the AE_DR register file. Floating point operations that are controlled by integral AR arguments use a single AR argument that controls each half of the AE_DR register.

Floating point operations typically have four cycles of latency, but are fully pipelined. Divide and sqrt operations are implemented using instruction sequences.

The standard C float type is supported using SIMD operations. Since loads replicate their value into each half, each scalar floating point ALU operation will perform the same operation on both halves. Each SIMD instruction supports two intrinsics: one with the same name as the operation to be used with scalar float arguments and one where _S is replaced with _SX2, to be used with xtfloatx2 arguments.

4.1.5 HiFi 4 Audio Engine option

HiFi 4 Audio Engine DSP coprocessor instruction family

Cadence HiFi 4 Audio Engine is a highly optimized audio processor geared for efficient execution of audio and voice codecs and pre- and post-processing modules. It goes beyond HiFi 3 with support for four 32x32-bit MACs, some support for 72-bit accumulators, limited ability to support eight 32x16-bit MACs, a fourth VLIW slot and the ability to issue two 64-bit loads per cycle. There is an optional floating point unit available, providing up to four single-precision IEEE floating point MACs per cycle. The extra resources provide for significant performance improvements compared to HiFi 3, particularly on pre/post-processing algorithms.

HiFi 4 is backward compatible at the C/C++ source level with HiFi 3 and HiFi 2/EP. Any algorithm written in C/C++, including all HiFi packages from Cadence, can simply be recompiled on HiFi 4 to get modest performance improvements. For additional performance, key kernels may need to be retuned for the HiFi 4 architecture.

The HiFi 4 Audio Engine is a configuration option that can be included with the Xtensa LX6 (and later versions) processor. All HiFi 4 Audio Engine operations can be used as intrinsics in standard C/C++ applications. In addition, when compiling with automatic vectorization or with the –mcoproc option, the compiler will automatically use HiFi 4 operations when compiling standard C code.

Cadence HiFi 4 Audio Engine consists of three main components: a DSP subsystem, an optional floating point unit, and a subsystem to assist with bit stream access and variable-length (Huffman) encoding and decoding. These are covered in detail the separate HiFi 4 documentation.

Restriction: LX6++ only

4.1.5.1 HiFi4 Vector FP

Vector Floating Point extensions to HiFi 4 Audio Engine

HiFi 4 supports an optional pair of 2-way SIMD IEEE floating point units. The floating point units share the AE_DR register file with the rest of HiFi 4. Therefore, standard loads, stores, and selects can all be used together with floating point compute operations. Every floating point operation, other than the conversion to integral types, is replicated two ways. The same operation is performed on each half of the AE_DR register file. Floating point operations that are controlled by integral AR arguments use a single AR argument that controls each half of the AE_DR register.

Floating point operations typically have four cycles of latency but are fully pipelined. Divide and sqrt are implemented using instruction sequences.

The standard C float type is supported using SIMD operations. Since loads replicate their value into each half, each scalar floating point ALU operation will perform the same operation on both halves. Each SIMD instruction supports two intrinsics: one with the same name as the operation to be used with scalar float arguments and one where _S is replaced with _SX2, to be used with xtfloatx2 arguments.

4.1.6 FLIX3 option

Configuration for 3-way FLIX, with a 64-bit instruction length. Enables execution of 2 or 3 instructions per cycle

The FLIX3 option selects a pre-configured 3-way FLIX Xtensa processor core, along with Xtensa instructions allocation per FLIX slot. It has a 32-bit data path with the base 24/16-bit Xtensa ISA and 64-bit FLIX instructions.The FLIX instructions allow multiple core instructions to be executed at the same time. Certain additional instruction options are selected as well, and can be executed as part of a FLIX instruction. Wide branches are defined, which increase the range of branches.

Note: The FLIX3 option uses the same FLIX structure as the Diamond DC_570T. Because of this, it does not combine with other processor selections and is intended to be used alone. If you wish to achieve the same goal (FLIXing core instructions) with a different processor selection, this can be achieved within the FLIX structure of that processor by adding user TIE as shown below. This TIE assumes a wide variety of optional instructions have been selected. If your configuration does not have them, simply remove them from the user TIE.

```
format xt_format1 64 {xt_flix64_slot0, xt_flix64_slot1, xt_flix64_slot2}
format xt_format2 64 {xt_flix64_slot0, xt_flix64_slot3}
slot_opcodes xt_flix64_slot0 {ABS, ADD, ADDI, ADDMI, ADDX2, ADDX4, ADDX8, AND, CLAMPS,
EXTUI,
     L16SI, L16UI, L32I, L32R, L8UI, MAX, MAXU, MIN, MINU, MOVEQZ, MOVGEZ, MOV.N,
MOVI, MOVLTZ,
     MOVNEZ, MUL16S, MUL16U, MULL, NEG, NOP, NSA, NSAU, OR, S16I, S32I, S8I, SEXT,
SLL, SLLI,
     SRA, SRAI, SRC, SRL, SRLI, SSA8B, SSA8L, SSAI, SSL, SSR, SUB, SUBX2, SUBX4,
SUBX8, XOR}
slot_opcodes xt_flix64_slot1 {ADD, ADDI, ADDMI, ADDX2, ADDX4, ADDX8, AND, EXTUI, J,
JX, MOVEQZ,
     MOVGEZ, MOVLTZ, MOVNEZ, MOV.N, MOVI, MUL16S, MUL16U, MULL, NEG, NOP, OR, SEXT,
SLL, SLLI,
     SRA, SRAI, SRC, SRL, SRLI, SSL, SUB, XOR}
slot_opcodes xt_flix64_slot2 {ABS, ADD, ADDI.N, ADDX2, ADDX4, AND, MOV.N, MOVI.N, NEG,
NOP, OR,
     SEXT, SRA, SRAI, SRL, SRLI, SUB, XOR}
slot_opcodes xt_flix64_slot3 {xt_widebranch18}
```

4.1.6.1 FLIX3 Flow Control option

Optional addition of flow control instructions to FLIX3 slots

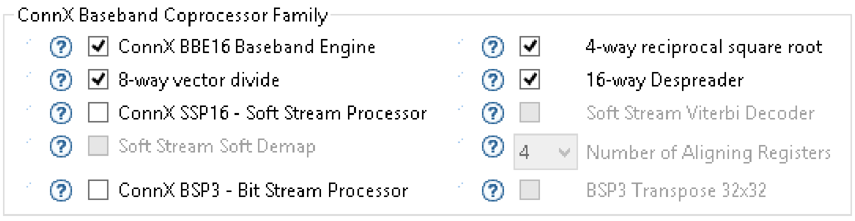
The FLIX3 flow control option allows the following operations to issue in FLIX slots, in parallel with other core operations, increasing performance modestly at the also modest the expense of code size:

- Calls: CALL0, CALLX0, CALL4, CALLX4, CALL8, CALLX8, CALL12, CALLX12
- Returns: RET, RET.N, RETW, RETW.N
- Zero-overhead loops: LOOP, LOOPGTZ, LOOPNEZ

Restriction: LX6++ only

4.1.7 ConnX BBE16 Baseband Engine option

High performance DSP designed for use in next generation communication baseband processors, such as those found in LTE and 4G cellular radios and multi-standard broadcast receivers



ConnX Baseband Coprocessor Family

- [✓] ConnX BBE16 Baseband Engine
- [✓] 8-way vector divide
- [ ] ConnX SSP16 - Soft Stream Processor
- [ ] Soft Stream Soft Demap
- [ ] ConnX BSP3 - Bit Stream Processor
- [✓] 4-way reciprocal square root
- [✓] 16-way Despreader
- [ ] Soft Stream Viterbi Decoder
- 4 Number of Aligning Registers
- [ ] BSP3 Transpose 32x32

The high computation requirements in such applications require new and innovative architectures with a high degree of parallelism and efficient I/Os. The ConnX BBE16 DSP Engine meets these needs by combining a 8-way SIMD, 3-slot VLIW processing pipeline with a rich and extensible set of interfaces.

The ConnX BBE16 DSP Engine (often referred to as "ConnX BBE16") is built around a core vector pipeline made of 16 18bx18b MACs. These multipliers and associated adder and multiplexor trees enable operations such as FFT butterflies, parallel complex multiply operations and signal filter structures. The results of these operations can be full precision or truncated/rounded/saturated and shifted to meet the needs of different algorithms and implementations. The instruction set has been optimized for DSP kernel operations such as FFT and FIR as well as matrix multiplies. Acceleration has been added for a wide range of key wireless functions.

ConnX BBE16 supports programming in C with a vectorizing compiler. Automatic vectorization of scalar C and full support for vector data types allows the development of algorithms without the need to program at the assembly level. Native C operator overloading is supported for natural programming with standard C operators on real and complex vector data types.

4.1.7.1 4-Way Reciprocal Square Root

Optional 4-way SIMD reciprocal square root

The reciprocal square root operation is important for algorithms that involve accurate normalization. The input is a 4x40 bit vector and the output is the same type. The 4-way SIMD instruction assumes its inputs are 32-bit integers and computes 16-bit fractional outputs. Alternatively, the input can be interpreted as a 32-bit fractional value and the output as 16-bit integers. The reciprocal square root is accurate to approximately one bit in the least significant position. This operation is undefined on negative or zero inputs, returning 0x7FFFF. This is a pipelined 4-cycle operation.

4.1.7.2 8-Way Vector Divide

Optional 8-way SIMD integer and fractional divide

Eight fixed point divides are performed per instruction, with a 16-bit dividend divided by a 16-bit divisor to create a 16-bit quotient. Inputs are in two 8x20 bit vectors and the output is an 8x20 bit vector.

Forms of divide are:

- Eight signed, 16-bit dividends and eight signed, 16-bit divisors in a 6-cycle, 8-way SIMD signed divide which returns eight, 16-bit signed integer results.
- Eight unsigned, 16-bit dividends and eight unsigned, 16-bit divisors in a 6-cycle, 8-way SIMD unsigned divide which returns eight, 16-bit unsigned integer results.
- An unsigned fractional divide which shifts its 16-bit dividend input left by 16 bits before performing unsigned divide. Eight unsigned, 16-bit dividends and eight unsigned, 16-bit divisors; the higher-order bits are ignored and each dividend must be less than its divisor. This instruction computes an 8-way SIMD unsigned divide in a 6-cycle operation, which results in eight 16-bit positive fractional results.
- Signed 32-bit by 16-bit divide, dividing eight signed 32-bit numbers in two vector registers by eight signed 16-bit numbers in one vector register resulting in exact integer quotients. The 6-cycle, 8-way SIMD signed divide returns eight, 16-bit signed integer results.

4.1.7.3 16-Way Despreader

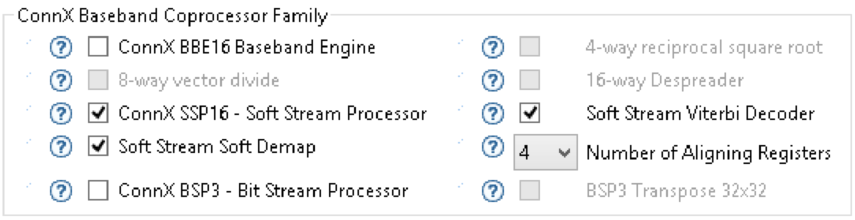
Optional 16-way despreader

The optional despread operation, designed for CDMA code (Code Division Multiple Access) uses 2-bit input codes, which represent complex phasors. This operation uses 16 effective complex multiply operations per cycle and is implemented with adders. Four independent code streams (four codes) each apply to four complex inputs and the result is added to the accumulator.

The despread operation also contains instructions used for the computation of Hadamard transforms.

4.1.8 ConnX SSP16 option

16-way SIMD engine operating on 8-bit and 10-bit soft bits used in wireless communication level 1 (PHY) systems


ConnX Baseband Coprocessor Family

☐ ConnX BBE16 Baseband Engine	☐ 4-way reciprocal square root
☐ 8-way vector divide	☐ 16-way Despreader
☑ ConnX SSP16 - Soft Stream Processor	☑ Soft Stream Viterbi Decoder
☑ Soft Stream Soft Demap	4 Number of Aligning Registers
☐ ConnX BSP3 - Bit Stream Processor	☐ BSP3 Transpose 32x32

The ConnX SSP16 Soft Stream Processor is an extension to the Xtensa LX Dataplane Processor, which addresses the processing requirements for applications with extensive use of 8-bit data types. It is oriented to 16-way operations on these 8-bit data types with a 160 bit vector register file in which 2 guard bits are used to reduce the chance of overflow in the 8 bit types. This gives optimized computation and performance for the operation of Soft bit data algorithms in Wireless Communication modem systems. The ConnX SSP16 DPU combined with the ConnX BSP3 DPU and the ConnX BBE16 DSP core offer an industry leading small size and low power implementation for the next generation Wireless Communication modem systems.

The ConnX SSP16 architecture has been designed for ease of programming. The best performance will be attained with vectorized algorithms where 16 operations can be issued in parallel with a vector load or vector store.

The programming model of the ConnX SSP16 is designed for ease of use and performance from C-level algorithms on 8-bit data types. Most C-level algorithms will automatically vectorize to SSP16 by Cadence XCC advanced compiler. In the cases where automatic vectorization is not possible, C-intrinsics can be used to achieve vector performance.

4.1.8.1 Soft Demap option

Optional soft demap for ConnX SSP16 cores

This option adds three optional operations that have been designed to speed up Soft Bit Demapping. The operations BBE_SDMAP64_2XC20, BBE_SDMAP16_2XC20 and BBE_SDMAP4_2XC20 demap complex 2 complex values that have been mapped to 64QAM, 16QAM and 4QAM constellations. An immediate value cxsel identifies whether the least significant 2 complex values or the most significant ones are demapped.

The operations compute log2(P1/P0) -- the log base 2 of the probability that each input bit to the generated constellation was a 1 divided by the probability that input bit was a 0. If the immediate value pllr0 is set to 1, then log2(P0/P1) is computed instead.

4.1.8.2 Viterbi Decoder option

Optional Viterbi decoder for ConnX SSP16 cores

The optional Viterbi operations support 1/2 and 1/3 rate with arbitrary polynomials of constraint length 7 through 9. Four-bit soft-bit values from interleaved streams of soft bit data are loaded from memory. Internal state values are stored in 10-bit unsigned elements of the vector register files. The operations are designed to perform a forward pass through input soft bits, updating the states and buffering swap decisions. These 1-bit decisions are packed and stored to memory. After all decisions have been stored, the maximal state is identified, then a backwards traceback pass through the decision bits computes the hard-bit outputs.

The `BBE_VTADDSUBQUAD` operation pre-conditions the 4-bit signed input data.

The Add-Compare-Select `BBE_VTACS` operation performs 32 forward state updates each cycle from the preconditioned input data and programmable polynomials. Constraint length 7 operations must update 64 states for each new set of input bits and take 2 cycles per set of input bit. After the forward phase, the backward traceback operation `BBE_VTTB2X64` operation for constraint length 7 collects 2 traceback bits per cycle.

The state metrics are maintained in 10 bit unsigned vector elements. The metric updates add a bias value to the signed inputs to maintain them as unsigned data. To support overflow detection and re-normalization, the `BBE_VTACS` operation has 2 1-bit immediates reset_enable and normalize_enable. The reset_enable is set on the first operation of a set of operations that perform a state update for one pair (1/2 rate) or triplet (1/3 rate) of soft bits. The normalize_enable should be set for all of the state update operations involved in the state updates for following soft bit.

At the end of the viterbi computation on the input data streams, before backtrace, the maximal metric must be identified. This can be performed with a reduction index operation. In the case of tail-biting convolutional encoding, it can also be performed by continuing the computations on bits copied from the beginning of the input stream to force the maximal state to be state 0. Before the soft bit data is passed to the viterbi operations, the initial 2 or 3 streams of data must be reordered to place one element of each stream in every 4 bytes. There are two mechanisms that can be used for finding the minimal state at the end of the viterbi computations.

The examples `ssp16_LTE_viterbi` and `ssp16_IEEE_viterbi` demonstrate the use of the viterbi operations with LTE and Wifi standard rates and polynomials. Each uses extra iterations to force the state to 0, then backtrace without recording through the extra bits before performing the backtrace that generates the output hard bit stream.

4.1.8.3 Number of Aligning Registers option

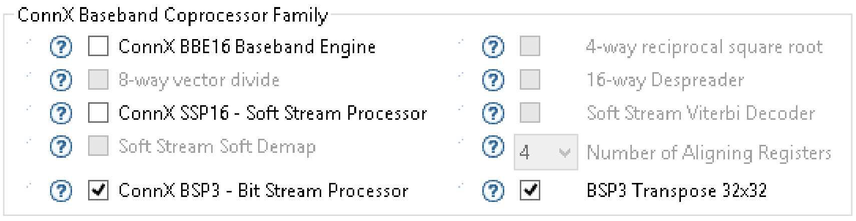
ConnX SSP16 number of aligning vector registers

Number of aligning vector registers for SSP16. The default is 4; the other choices are 8 and 16.

Restriction: Available from RE-2014.5. Defaulted to 4 in prior releases.

4.1.9 ConnX BSP3 option

3-issue VLIW machine with advanced Bit manipulation capabilities engine used in wireless communication level 1 (PHY) systems for Bit stream processing



ConnX Baseband Coprocessor Family

☐ ConnX BBE16 Baseband Engine	☐ 4-way reciprocal square root
☐ 8-way vector divide	☐ 16-way Despreader
☐ ConnX SSP16 - Soft Stream Processor	☐ Soft Stream Viterbi Decoder
☐ Soft Stream Soft Demap	4 Number of Aligning Registers
✓ ConnX BSP3 - Bit Stream Processor	✓ BSP3 Transpose 32x32

The ConnX Bit Stream Processor 3 (BSP3) is an extension to the Xtensa LX4 Dataplane Processor, which addresses the processing requirements for applications with extensive use of bit-level manipulation operations.

The ConnX BSP3 Engine has been designed for ease of programming. High performance can be achieved using standard C code with the Xtensa optimizing compiler. The best performance will be attained with manual operation selection (using "intrinsics") in C code.

The programming model of the ConnX BSP3 Engine is designed for ease of use and performance from C-level algorithms. The base architecture compiles standard C code well. However, bit-manipulation C intrinsics are provided for optimizing bit-manipulation algorithms like interleavers, CRCs, and turbo encoders.

4.1.9.1 Transpose 32x32 option

This option adds special operations to handle the cases where the number of rows and the number of columns are multiples of 32. Specifically, the following is included if the transpose 32x32 option is selected:

- A special 1024-bit state register.
- Operations BSP_XCTP32 and BSP_XRTP32
- RUR and WUR operations to read/write contiguous 32 bits in the 1024-bit state register

Following is a brief description of these additional optional operations. BSP_XCTP32 and BSP_XRTP32 treat the 1024-bit state as a 32-bit x 32-bit matrix.

- The BSP_XRTP32 operation exchanges 32 bits from user with 32 bits in the state matrix at a specified row index.
- The BSP_XCTP32 operation exchanges 32 bits from user with 32 bits in the state matrix at a specified column index.
- The RUR and WUR operations read or write 32 contiguous bits of the 1024-bit state register.

These optional transpose32 operations can be used to efficiently implement a 32 x 32n bit matrix transpose, or a 32n x 32-bit matrix transpose. This is demonstrated in the project transpose_32nx32. These specific bit matrix transposes are useful in the bit manipulations required in the LTE and HSPA standards.

Note that the transpose_32nx32 example also includes code to do the same transpose function using the other BSP Bit Interleave operations. Using the optional BSP_XRTP32 and BSPXCTP32 operations are more efficient for 32nx32 and 32x32n transpose sizes.

When an element contains more than one bit that is a power of two, the transpose can be performed more efficiently. When an element contains a non-power of two number of bits, the elements can be expanded before transposition.

Restriction: LX4++ only

4.1.10 ConnX BBE-EP Family

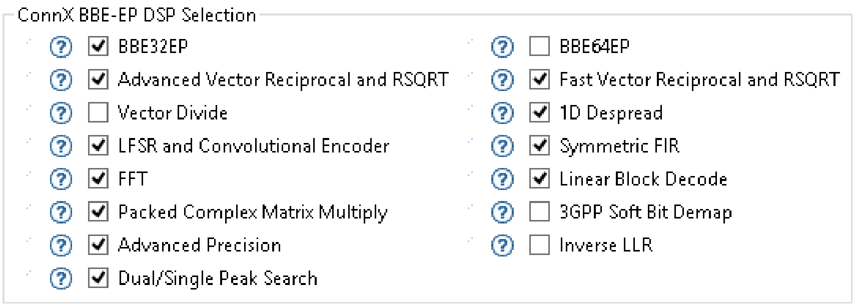
ConnX BBE-EP Family of DPUs are 32/64-way MAC DSPs optimized for use in next-generation baseband processors.

The ConnX BBE-EP Family natively supports both real and complex arithmetic operations. For digital signal processing developers, this greatly simplifies development of algorithms dominated by complex arithmetic. In addition to having the SIMD/VLIW DSP core, the ConnX BBE-EP Family contains a 32-bit scalar processor, ideal for efficient execution of control code. This combined SIMD/VLIW/Scalar design makes the ConnX BBE-EP Family ideal for building real systems where high computational throughput is combined with complex decision making.

TheConnX BBE-EP Family is built around a core vector pipeline consisting of 32/64-way 16bx16b MACs along with a set of versatile pipelined execution units. These units support flexible precision real and complex multiply-add; bit manipulation; data shift and normalization; data select, shuffle and interleave. The ConnX BBE-EP Family multipliers and their associated adder and multiplexer trees enable execution of complex multiply operations and signal processing filter structures in parallel. The results of these operations can be extended up to a precision of 40-bits per element, truncated/rounded/saturated or shifted/packed to meet the needs of different algorithms and implementations. The ConnX BBE-EP Family instruction set is optimized for several DSP kernel operations and matrix multiplies with added acceleration for a wide range of key wireless functions. In addition, the instruction set supports signed*unsigned multiplies for emulation of 32-bit wide multiplication operations.

The ConnX BBE-EP Family supports programming in C/C++ with a vectorizing compiler. Automatic vectorization of scalar C and full support for vector data types allows software development of algorithms without the need for programming at assembly level. Native C operator overloading is supported for natural programming with standard C operators on real and complex vector data-types. The ConnX BBE-EP Family has a Boolean predication architecture that supports a large number of predicated operations. This enables the ConnX BBE-EP Family compiler to achieve a high vectorization throughput even with complicated functions that have conditional operations embedded in their inner loops.

The ConnX BBE32EP and its larger cousin, the ConnX BBE64EP share a common architecture, providing a high degree of code portability between the two cores. Both cores share a common set of check box options that allow capabilities to be added/subtracted from the core. This permits the system designer to optimize the core for a particular application space reducing both area and power consumption.


ConnX BBE-EP DSP Selection

✓ BBE32EP	☐ BBE64EP
✓ Advanced Vector Reciprocal and RSQRT	✓ Fast Vector Reciprocal and RSQRT
☐ Vector Divide	✓ 1D Despread
✓ LFSR and Convolutional Encoder	✓ Symmetric FIR
✓ FFT	✓ Linear Block Decode
✓ Packed Complex Matrix Multiply	☐ 3GPP Soft Bit Demap
✓ Advanced Precision	☐ Inverse LLR
✓ Dual/Single Peak Search	

4.1.10.1 ConnX BBE32EP

32-way multiplier-accumulator (MAC), dual 16-way arithmetic logic unit (ALU) single instruction, multiple data (SIMD) Baseband Engine

The ConnX BBE32EP is based on an ultra-high performance DSP architecture designed for use in next-generation baseband processors for LTE Advanced, other 4G cellular radios and multi-standard broadcast receivers. The high computational requirements of such applications require new and innovative architectures with a high degree of parallelism and efficient I/Os. The ConnX BBE32EP meets these needs by combining a 16-way Single Instruction, Multiple Data (SIMD), 32 multiplier-accumulators (MAC) and up to 5-issue Very Long Instruction Word (VLIW) processing pipeline with a rich and extensible set of interfaces.

4.1.10.2 ConnX BBE64EP

64-way multiplier-accumulator (MAC), dual 32-way arithmetic logic unit (ALU) single instruction, multiple data (SIMD) Baseband Engine

The ConnX BBE64EP is based on an ultra-high performance DSP architecture designed for use in next-generation baseband processors for LTE Advanced, other 4G cellular radios and multi-standard broadcast receivers. The high computational requirements of such applications require new and innovative architectures with a high degree of parallelism and efficient I/Os. The ConnX BBE64EP meets these needs by combining a 32-way Single Instruction, Multiple Data (SIMD), 64 multiplier-accumulators (MAC) and up to 5-issue Very Long Instruction Word (VLIW) processing pipeline with a rich and extensible set of interfaces.

4.1.10.3 FFT

This option provides FFT and DFT support offering significant performance improvements on FFT operations of any size.

The ConnX BBE{32|64}EP FFT package provides special FFT instructions optimized to perform a range of DFT computations. These FFT instructions implement a generic Discrete Fourier Transform in a series of steps. Each step is a pass over the whole input.

In the general case, the ConnX BBE{32|64}EP architecture allows a vector/state load, a vector store, a radix4 add, a vector-multiply and a shuffle to execute all in parallel. This produces to a full vector worth of radix4 FFT results as output per cycle.

4.1.10.4 Symmetric FIR

This option inserts a pre-adder function in front of the multiplier tree, effectively doubling the number of taps that can be handled by the core MACs.

Optimized ISA support to accelerate symmetric FIR operations is available as a separate configurable option. The symmetric FIR operations double the effective MACs per cycle. This package does not support (complex data, complex coefficients) symmetric FIR filters.

Note:

- *FFT* & *Symmetric FIR* options share significant amounts of hardware. When configuring in one of these options in your ConnX BBE{32|64}EP core, it may be appropriate to add the other as the incremental cost is small.
- As a reference, several code examples are packaged with standard ConnX BBE{32|64}EP configurations in Xtensa Xplorer.

4.1.10.5 Packed Complex Matrix Multiply

Adds support for vectors where the matrix elements are ordered by matrix rather than grouped by element.

ConnX BBE{32|64}EP has support for efficient computation of small complex packed matrix multiplies - 2x2, 4x4, and variants.

Complex packed matrix multiplies in ConnX BBE{32|64}EP are accomplished by decomposing the problem into smaller tasks. A crucial step is the use of special shuffles in a multi-step execution to reorganize the order of data depending on matrix size. These *special shuffles* (patterns for matrix element reordering) are available only as a configuration option, and enhance regular shuffle operation `BBE_SHFLNX16I` and select operation `BBE_SELNX16I`.

Note: Streaming order matrix multiplies can be done without shuffling with regular machine multiplies for real or complex.

4.1.10.6 LFSR and Convolutional Encoding

Support for channel coding operations that involve LSFR code generation, such as scrambling and channel spreading, and also for convolutional encoding.

The LFSR option adds special operations to accelerate LFSR sequence generation up to 32-bits per cycle. This is done by emulating a 32x32-bit matrix by 32x1-bit multiplication. All the multiplication and reduction add operations in this option are in Galois Field (GF2).

4.1.10.7 Linear Block Decoder

Enables decoding block linear error-correction codes such as Hamming codes. In general, it can correlate a set of binary code vectors against a vector of real quantities.

The support to accelerate linear block decoding in the ConnX BBE{32|64}EP comes in the form of two optimized operations. Both operations operate on 16-bit real data and perform a 16-way multiplication based on a vector of 1-bit codes. The operations support up to eight such sets of 1-bit code vectors. The ISA HTML for these two operations has more information on how a linear block decode operation is set up.

4.1.10.8 1D Despread

Support to correlate a complex-binary vector against a complex 16i+16q or 8i+8q vector. This is particularly useful in despreading operations in 3G standards.

ConnX BBE{32|64}EP supports 1D despreading of 16-bit real or 8/16-bit complex input data. The 1D despread operations perform N-way element-wise signed, vector multiplication of real/complex inputs with a vector of real/complex codes at full precision. If `n` is the spreading factor (SF), `n` consecutive products are consecutively added (4, 8 or 16). A vector of `N/n` wide outputs, correspondingly real/complex, is produced every cycle. Say for SF=4, N-way real data and real codes

```
FOR i in {0...N/n}
  re_res[i] = SUM(j=0...n, re_data[i*4+j] * re_code[i*4+j])
```

4.1.10.9 Soft-bit Demapping

This option supports up to 256 QAM soft-bit demapping.

The soft-bit demapping operations are used to convert soft-symbol estimates, outputs of an equalizer, into soft bit estimates, or log-likelihood ratios (LLRs), later to be processed by a soft channel decoder for error correction and detection. The soft-bit demapper typically sits at the interface between complex and soft-bit domains.

Supported constellations and mappings are summarized in *Table 2: Set of Symbol Constellations Supported* on page 106. Symbol mappings for 3GPP and WiFi use different Gray Encoding formats, both supported by the soft-bit demapper operations.

Table 2: Set of Symbol Constellations Supported

Standard	Supported Constellations	Gray Encoding	Output to Soft Decoding
3GPP	QPSK	3GPP	Turbo
	16-QAM		
	64-QAM		
WiFi (IEEE 802.11)	QPSK	IEEE	Convolutional or LDPC
	16-QAM		
	64-QAM		
	256-QAM		

Note: Available with RE-2013.4 and later.

4.1.10.10 Comparison of Divide Related Options

ISA support for divide related functions in the ConnX BBE{32|64}EP family is offered as three configurable options:

- Fast vector reciprocal & reciprocal square root
- Advanced vector reciprocal & reciprocal square root
- Vector divide

Each option provides multiple operations (by adding more hardware) to the ConnX BBE{32|64}EP ISA to accelerate a specific divide related function. The table below briefly outlines the intended use for each operation along with any functional overlap with other operation(s).

For some insight into the usage of above operations, refer to the packaged code example **vector_divide** (computes 16b by 16b signed vector division) in several different ways.

Table 3: Comparison of Divide Related Options

Class	*Operations*	*Function*	*Overlaps functionally with*	*Configurable Option*
Advanced Vector Reciprocal	BBE_RECIPLUNX40_0	High precision reciprocal approximation: ~23b mantissa accuracy	32b/16b divide and BBE_FPRECIP	Advanced Vector Reciprocal & Reciprocal Square Root
	BBE_RECIPLUNX40_1	High precision reciprocal approximation: ~23b mantissa accuracy	32b/16b divide and BBE_FPRECIP	
Advanced Vector Reciprocal Square Root	BBE_RSQRTLUNX40_0	High precision reciprocal square root approximation: ~24b mantissa accuracy	BBE_FPRSQRT	
	BBE_RSQRTLUNX40_1	High precision reciprocal square root approximation: ~24b mantissa accuracy	BBE_FPRSQRT	
Fast Vector Reciprocal	BBE_RECIPUNX16_0	Unsigned integer reciprocal approximation: results with approximately 15b precision	16b/16b unsigned divide	Fast Vector Reciprocal & Reciprocal Square Root
	BBE_RECIPUNX16_1	Unsigned integer reciprocal approximation: results with approximately 15b precision	16b/16b unsigned divide	
	BBE_RECIPNX16_0	Signed integer reciprocal approximation: results with approximately 15b precision	16b/16b signed divide	
	BBE_RECIPNX16_1	Signed integer reciprocal approximation: results with approximately 15b precision	16b/16b signed divide	
	BBE_FPRECIPNX16_0	Signed floating point reciprocal approximation: results with approximately 10b precision in the typical case, 7.5b in the worst case.	16b/6b signed divide, 32b/16b divide and advanced precision recip	
	BBE_FPRECIPNX16_1	Signed floating point reciprocal approximation:	16b/6b signed divide, 32b/16b divide and	

		results with approximately 10b precision in the typical case, 7.5b in the worst case.	advanced precision recip	
	BBE_DIVADJNX16	Signed divide adjust to allow C-exact 16b/16b integer divide - works with BBE_RECIPNX16*	16b/16b signed divide	
	BBE_DIVUADJNX16	Unsigned divide adjust to allow C-exact 16b/16b integer divide	16b/16b unsigned divide	
Fast Vector Reciprocal Square Root	BBE_FPRSQRTNX16_0	Signed floating point reciprocal square root approximation: results with approximately 9.7b precision in the typical case, 7.5b in the worst case.	High precision reciprocal square root	
	BBE_FPRSQRTNX16_1	Signed floating point eciprocol square root approximation: results with approximately 9.7b precision in the typical case, 7.5b in the worst case.	High precision reciprocal square root	
16-bit Vector Divide - First Step	BBE_DIVNX16S_5STEP0_0	Signed 16b/16b vector divide	BBE_RECIPNX16	Vector Divide
	BBE_DIVNX16S_5STEP0_1	Signed 16b/16b vector divide	BBE_RECIPNX16	
	BBE_DIVNX16U_4STEP0_0	Unsigned 16b/16b vector divide	BBE_RECIPUNX16	
	BBE_DIVNX16U_4STEP0_1	Unsigned 16b/16b vector divide	BBE_RECIPUNX16	
	BBE_DIVNX16Q_4STEP0_0	Unsigned fractional 16b/16b vector divide	BBE_FPRECIPNX16	
	BBE_DIVNX16Q_4STEP0_1	Unsigned fractional 16b/16b vector divide	BBE_FPRECIPNX16	
Vector Divide - Other Steps	BBE_DIVNX16S_3STEPN_0	Last step of 32b/16b and 16b/16b signed vector divide	BBE_RECIPNX16	
	BBE_DIVNX16S_3STEPN_1	Last step of 32b/16b and 16b/16b signed vector divide	BBE_RECIPNX16	
	BBE_DIVNX16S_4STEP_0	Middle step of 32b/16b and 16b/16b signed vector divide	BBE_RECIPUNX16, high precision	

			reciprocal approximation	
	BBE_DIVNX16S_4STEP_1	Middle step of 32b/16b and 16b/16b signed vector divide	BBE_RECIPUNX16, high precision reciprocal approximation	
	BBE_DIVNX16U_4STEP_0	Middle step of 32b/16b and 16b/16b unsigned vector divide	BBE_RECIPUNX16, high precision reciprocal approximation	
	BBE_DIVNX16U_4STEP_1	Middle step of 32b/16b and 16b/16b unsigned vector divide	BBE_RECIPUNX16, high precision reciprocal approximation	
	BBE_DIVNX16U_4STEPN_0	Last step of 32b/16b and 16b/16b unsigned vector divide	BBE_RECIPUNX16, high precision reciprocal approximation	
	BBE_DIVNX16U_4STEPN_1	Last step of 32b/16b and 16b/16b unsigned vector divide	BBE_RECIPUNX16, high precision reciprocal approximation	
32-bit Vector Divide - First Step	BBE_DIVNX32S_5STEP0_0	Signed 32b/16b vector divide	High precision reciprocal approximation	
	BBE_DIVNX32S_5STEP0_1	Signed 32b/16b vector divide	High precision reciprocal approximation	
	BBE_DIVNX32U_4STEP0_0	Unsigned 32b/16b vector divide	BBE_RECIPUNX16	
	BBE_DIVNX32U_4STEP0_1	Unsigned 32b/16b vector divide	BBE_RECIPUNX16	

Note: It is recommended to use protos that combine operation sequences appropriate for the type. The packaged code examples may be used as a reference to identify such protos.

4.1.10.10.1 Vector Divide

For 16-bit/16-bit and 32-bit/16-bit vector division with 16-bit outputs.

The ConnX BBE{32|64}EP can be configured with a vector divide option. Low-precision vector division operations in the ConnX BBE{32|64}EP are multiple cycle operations and are executed stepwise – four step operations for N/2 results of 16-bit precision and each step taking one cycle. These steps are pipelined by interleaving even and odd elements of the N-element vector. For convenience of programming, two sets of these four steps, one set each for even and odd elements are bundled into an intrinsic that produces N results of vector division in 8 cycles.

This option also supports high-precision division often useful for dividing arbitrary 16-bit fixed-point formats by appropriately shifting high-precision dividends. These high-precision vector division operations differ in that the dividend vector is 32-bits/element although stored in a 40-bit/element guarded `wvec` register.

4.1.10.10.2 Fast Vector Reciprocal & Reciprocal Square Root

Reduced cycle counts for 16-bit operations as compared to the vector divide option.

For more insight into the usage of above operations, refer to the packaged code example **vector_recip_fast** (computes reciprocal of input vector using the BBE_FPRECIP operations) and **vector_rsqrt_fast** (computes reciprocal sqrt of input vector using the BBE_FPRSQRT operations)

4.1.10.10.3 Advanced Vector Reciprocal & Reciprocal Square Root

Increased precision compared to the Fast Vector Reciprocal & Reciprocal Square Root option. Inputs and outputs are generally 16-bit fixed although intermediate results can be stored in higher precision with extra effort.

4.1.10.11 Advanced Precision Multiply/Add

This option supports 23-bit (16b mantissa, 7b exponent), and 32-bit fixed point multiplication and addition. Inputs and outputs are 16-bit/32-bit fixed precision although intermediate results can be stored at a higher precision.

The ConnX BBE{32|64}EP advanced precision multiply/add is a configurable option that enables two new classes of data representation and related computations:

- 23-bit floating point - 16-bit mantissa and 7-bit exponent
- 32-bit high precision fixed point

The expected MAC performance of the 23-bit floating point is 1.5 times the cycle cost of 16-bit fixed point, and, the expected MAC performance of 32-bit fixed point is 3 times the cycle cost of 16-bit fixed point.

Note: Available with RE-2013.4 and later.

4.1.10.12 Inverse Log-likelihood Ratio (LLR)

This options is used to reciprocate the soft-bit demapping operation. It converts a set of LLR values to the mean and variance of the corresponding complex N-QAM symbol. This is useful in SIC and turbo equalization type of operations.

ConnX BBE{32|64}EP includes optional operations to facilitate efficient estimation of soft-complex QAM symbols from log-likelihood ratios (LLR). This process uses a lookup table for the computation of non-linear hyperbolic tangent (tanh). Also, different operations are needed for each size of supported constellations (4/16/64/256-QAM).

Note: Available with RE-2013.4 and later.

4.1.10.13 Dual/Single Peak Search

Dual/Single Peak Search shortdesc

ConnX BBE{32|64}EP Single and dual peak (value and its index) search of 16/32-bit, signed/unsigned, real/complex inputs. In baseband, this option accelerates real and complex correlation functions often found in, for example, frame alignment, frame acquisition, frequency correction.

The peak search configuration option is based on operations optimized in the use of 32-bit SIMD comparators in conjunction with some helper operations to extract up to two peaks and their indices. The 16-bit peak search operations internally sign extend inputs to 32 bits for use with the 32-bit comparators. In both 32-bit single peak search and 16/32-bit dual peak search, the operations process N input elements per issue. But, with 16-bit single peak search, the 2N elements are processed per issue.

Restriction: Available with RE-2014.5 and later.

4.1.11 IVP-EP Imaging and Video DSP

IVP-EP imaging and video DSP high-performance DSP optimized for image, video and vision processing

Cadence IVP-EP imaging and video DSP is a high-performance digital signal processor (DSP) optimized for image, video and vision processing. The instruction set architecture provides easy programmability in C and delivers a high sustained pixel processing performance required for advanced image, video, and vision processing and analysis applications that are rapidly growing in complexity as new image methods and entirely new scene and object recognition applications are introduced.

The IVP-EP is optimized for processing tasks characterized by both high pixel rates, often exceeding 200M pixels per second through the major processing functions, as well as high computation rates, often exceeding 1000 operations on each pixel processed.

The IVP-EP features a SIMD/VLIW architecture. Each VLIW instruction contains up to five operations, including up to three ALU operations (one of which can be a multiply or multiply-accumulate) and up to two 512-bit load/store operations (one of which can be a store operation). 32-way 16-bit, 64-way 8-bit, and 16-way 32-bit SIMD operations are supported. The rich set of operations include arithmetic, shift and logical operations and data select and shuffle operations.

Restriction: LX6++ only

4.1.11.1 IVP-EP Histogram Option

Operations that accelerate computing histograms containing a modest number of bins

The operations in the Histogram package count the number of values contained in two 2Nx8U input vectors, producing a vector of Nx16U counts. The 2Nx8U input values are scaled by shifting right 0..3 places prior to being counted. Both the range of values counted (0..N-1, N..2N-1, 2N..3N-1, …, 7N..8N-1) and the number of places to shift the input values are specified by fields in an AR register. The range field in the AR register is incremented by the count operations. Different versions of the count operations set the range to the lowest range prior to counting, use two vboolN inputs to control which input elements are counted, and generate cumulative counts.

Histograms are computed by using the count operations to count input values in the ranges of interest and accumulating the resulting counts using a separate ADD operation. N histogram bin values are updated for 4N input values per cycle using the count operations in this package. The throughput is thus proportional to the number of bins in the histogram. For histograms containing a large number of bins it may be more effective to only update the bins containing the input values instead of using this approach.

The Histogram package adds a new instruction format that supports two load operations, a count operation and an add operation. This enables full throughput even for histograms with N or fewer bins.

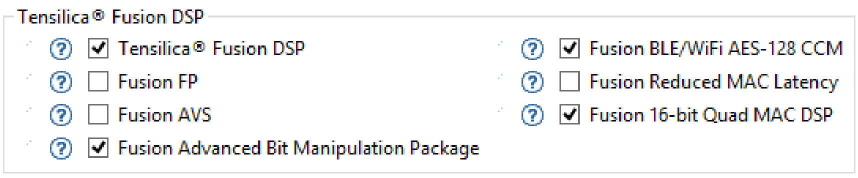
The count operations support a simple scaling function to bin values. More complex binning functions are handled separately and the count operations are used to count the computed bin values.

Detailed descriptions of the count operations are found in the IVPEP ISA HTML documentation.

Restriction: LX6.0.1++ only

4.1.12 Tensilica® Fusion DSP

A highly optimized, highly configurable, processor geared for efficient execution of dataplane algorithms needed for the Internet of Things (IoT), and other applications, such as codec chips, sensor hubs, and narrowband wireless communications


Tensilica® Fusion DSP

- [✓] Tensilica® Fusion DSP
- [ ] Fusion FP
- [ ] Fusion AVS
- [✓] Fusion Advanced Bit Manipulation Package
- [✓] Fusion BLE/WiFi AES-128 CCM
- [ ] Fusion Reduced MAC Latency
- [✓] Fusion 16-bit Quad MAC DSP

The Cadence® Tensilica® Fusion DSP Engine is a highly optimized, highly configurable, processor geared for efficient execution of dataplane algorithms needed for the Internet of Things (IoT), and other applications, such as codec chips, sensor hubs, and narrowband wireless communications. It is derived from a smaller, two-slot VLIW version of the Cadence HiFi 3 Audio Engine It supports dual issuing a load or a store together with two way SIMD ALU or MAC operations, supporting dual 16x16, 32x16, 24x24-bit MACs and single 32-bit

MACs. At a minimum, it is source code software compatible with the Cadence HiFi 2, HiFi Mini and HiFi 2 EP Audio Engines except for bit-stream and variable-length decode and encode. At a minimum, it is also compatible with the Cadence HiFi 3 Audio Engine except for bit-stream, variable-length decode and encode and HiFi 3 quad MAC instructions. The AVS option, if chosen, adds full HiFi software compatibility by adding bit-stream, variable-length decode and encode operations, and HiFi 3 quad MAC emulation capability[5].

Fusion DSP contains a wide range of configuration options to meet your needs. Each one of the six options can be selected independently. The AVS option was discussed above. The 16-bit Quad MAC option adds computation extension to support 16-bit vector Quad MAC (four MAC) for complex and dot product operations. Also included with this Quad MAC option are specialized instructions for FFT computation acceleration. The FP option adds support for single issuing single-precision, IEEE 754, floating point operations, including fused multiply accumulates, together with two-way SIMD loads or stores. The Reduced MAC Latency option halves the latency of all long latency operations, sacrificing the maximum MHz achievable to enable lower area and power at low MHz. The Advanced Bit Manipulation option add supports for bit-level operations for baseband-PHY and MAC processing. This option also supports CRC and scrambling, FEC convolutional encoding, and adds instructions for bit-level shuffling operations. The BLE/Wi-Fi AES 128-CCM option supports instructions to accelerate AES 128 CCM-mode encryption/decryption.

The Fusion DSP is a coprocessor configuration option for the Xtensa® LX processor. All Fusion operations can be used as intrinsics in standard C/C++ applications. In addition, when compiling with automatic vectorization or with the –mcoproc option, the compiler will automatically infer these operations when compiling standard C code.

Restriction: LX6++ only

4.1.12.1 Fusion BLE/WiFi AES-128 CCM

Support for AES-128 CCM encryption functions

Support for efficient implementation of AES-128 CCM encryption functions, as required, for example, in Bluetooth Low Energy and Wi-Fi protocols.

4.1.12.2 Fusion AVS

Support for codec software compatibility

Support for software compatibility with HiFi-2, HiFi 3 and HiFi mini audio, voice and speech codecs. Enables HiFi bit stream intrinsics as well as emulation intrinsics for the HiFi 3 quad multiplication instructions.

[5] Note that even without the AVS option, Fusion DSP is able to emulate all HiFi bit-stream instructions in software, albeit very slowly. This makes even the base configuration fully compatible with HiFi 2 and HiFi Mini.

4.1.12.3 Fusion FP

Support for IEEE 754 single precision floating point

Support for IEEE 754 single precision floating point. Floating point compute operations, including fused multiply-accumulates, can be issued in parallel with loads or stores. The compute operations work on scalar, 32-bit data. The load and store operations can load or store two-way SIMD, 32x2-bit data.

4.1.12.4 Fusion Advanced Bit Manipulation Package

Bit manipulation operations including CRC, LFSR, Convolutional Encoding and bit-level shuffling

This option enables speedup for bit-level operations commonly used in Baseband PHY and MAC standards like Bluetooth, Wi-Fi, and 3GPP, for:

- CRC and Scrambling (Linear Feedback Shift Register) functions
- Bit-level Convolutional Encode operations
- Bit-level shuffling and manipulation, commonly used in Baseband PHY and MAC standards.

4.1.12.5 Fusion Reduced MAC Latency

Reduces the latency required for certain instructions to complete including multiply, bit-stream instructions and the optional floating point instructions. The option results in smaller area and lower power cores at the expense of significantly lower maximum MHz

Without this option, multiply and bit-stream instructions are fully pipelined and can be issued every cycle but take an additional cycle to complete. With this option, they complete without additional cycles of latency. The optional floating point operations normally complete in four cycles, but with this option they complete in two. Using this option allows for smaller and lower energy hardware but only when synthesizing at lower MHz. This option limits the maximum MHz to approximately 2/3rd of the normal maximum but is generally only beneficial when synthesizing to at most 1/3rd of maximum frequency.

4.1.12.6 Fusion 16-bit Quad MAC DSP

Enhanced 16-bit DSP performance

This option allows for more efficient 16-bit DSP performance. In particular, this option enables complex quad 16-bit multiply instructions, real quad 16-bit multiply dot-product instructions and specialized instructions to speed up 16-bit FFTs with dynamic scaling support.

4.2 Instruction / ISA Options

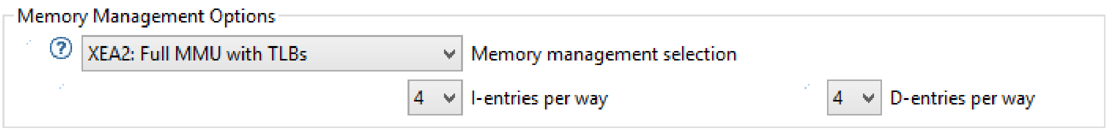
Instruction and ISA options

Sub-sections describe the available instruction and ISA configuration options.

4.2.1 Memory Management Selection

Region protection options provide coarse protection at a low gate count The Full MMU provides resource management capabilities sufficient for running Linux

The UI offers the available types of memory management



Memory Management Options

XEA2: Full MMU with TLBs — Memory management selection

4 — I-entries per way

4 — D-entries per way

LX and Xtensa processor configurations support several types of memory management:

- XEA2 Region protection
- XEA2 Region protection with translation
- XEA2 Full MMU with TLBs
- XEA2 MPU (Memory Protection Unit)

Refer to the *Xtensa LX Microprocessor Data Book* for more information on the XEA2 memory management options.

The *XEA2: MPU (extended region protection)* option statically initializes a *background* protection map which matches the configuration selections for processor interface and call windows. At runtime, software can dynamically refine the *foreground* protection map to suit specific needs. See also the *Xtensa System Software Reference Manual*.

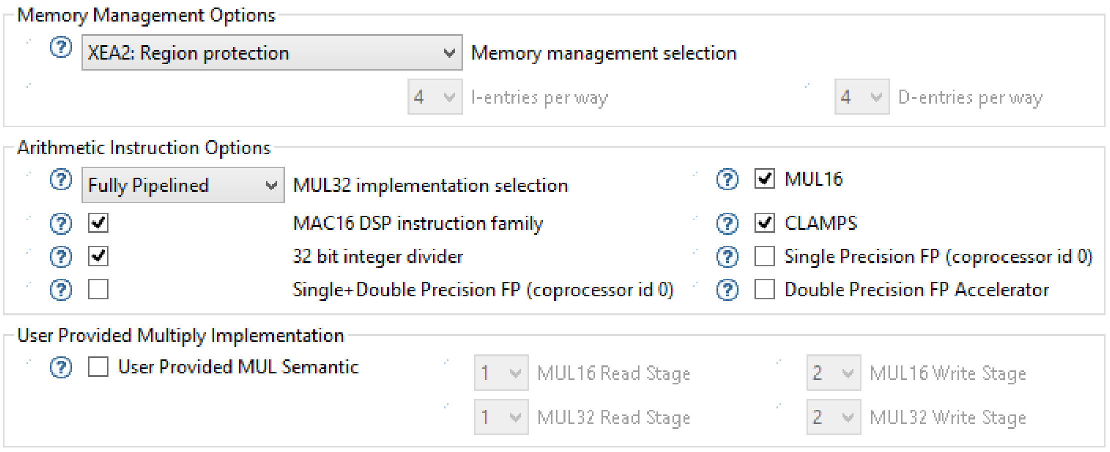
The *XEA2: Full MMU with TLBs* option allows you to select either four or eight entries per way for each of the TLBs (I-TLB and D-TLB). Selecting a higher number of entries allows more page entries to be cached in the TLB and hence, possibly fewer future TLB misses. This benefit comes with an additional hardware cost. You can select a different number of entries for I-TLB and D-TLB to match the characteristics of the software. One indication that you should increase the number of entries is to observe if there is a high number of TLB misses.

Note: If you want to select or deselect the Full MMU, select an appropriate template and then make changes. Adding or removing the Full MMU makes significant changes to memory and vector configuration which can be onerous to reconcile.

Restriction: The MPU selection is available with LX7++ only.

4.2.2 Arithmetic Instruction Options

The Xtensa Processor Generator offers a series of optional arithmetic configuration options.



Memory Management Options

XEA2: Region protection — Memory management selection

4 — I-entries per way

4 — D-entries per way

Arithmetic Instruction Options

Fully Pipelined	MUL32 implementation selection	✓	MUL16
✓	MAC16 DSP instruction family	✓	CLAMPS
✓	32 bit integer divider	☐	Single Precision FP (coprocessor id 0)
☐	Single+Double Precision FP (coprocessor id 0)	☐	Double Precision FP Accelerator

User Provided Multiply Implementation

☐ User Provided MUL Semantic

1 — MUL16 Read Stage

2 — MUL16 Write Stage

1 — MUL32 Read Stage

2 — MUL32 Write Stage

These are all selected from the Instructions page of the Xplorer Configuration Editor. Refer to the Architectural Options chapter in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for more details.

4.2.2.1 MUL32 Option

This option selects a standard 32-bit multiplier, which is used by the compiler whenever multiplying signed or unsigned variables of type int, short or char. Without this option and without any other multiplication option, the compiler will emulate all integral multiplication using shifts and adds. Emulation speed depends on the actual bit patterns being multiplied as well as on other configuration options, but can reach 100 cycles for a single multiply. If the 16 bit multiplier is included, but not the 32-bit multiplier, then the compiler will emulate 32-bit multiplies using multiple 16-bit operations.

Iterative Implementation

This option creates a MULL instruction that implements a 32-bit times 32-bit multiplication into a 32-bit product using iterative, non-pipelined hardware. This instruction takes anywhere from one to six cycles, depending on the bit patterns being multiplied. Instructions subsequent to the multiply will not begin execution until the multiply operation is complete.

Fully Pipelined Implementation

This option creates a MULL instruction that implements a 32-bit times 32-bit multiplication into a 32-bit product using fully-pipelined hardware. This instruction takes two cycles but the processor will only stall if the result of the multiplication is needed in the next cycle. The compiler will attempt to schedule other instructions, including other multiplies, in between the multiply and any use of its result.

Pipelined Plus UH/SH

In addition to the fully-pipelined MULL instruction, this option contains two fully pipelined instructions, MULSH for signed values and MULUH for unsigned, to compute the high 32-bits of a 32-bit times 32-bit into 64-bit product integral multiplication. These instructions will be automatically inferred by the compiler to aid in the multiplication of two signed or unsigned variables where at most one of the input variables is 64-bits. For other uses, you may access these instructions directly using intrinsics as follows.

```
#include <xtensa/tie/xt_mul.h>
int a, b, c;
unsigned ua, ub, uc;
...
c = XT_MULSH(a, b);
uc = XT_MULUH(ua, ub);
```

4.2.2.2 MUL16 Option

16-bit multiplier (signed/unsigned)

MUL16 supports signed and unsigned 16-bit multiplication. On configurations with only this option, the compiler will infer the use of this option from C whenever it can prove that a 16-bit multiplication is equivalent to a 32-bit multiplication. That is, whenever it can prove that both operands are 16-bits or less or the result is 16-bits or less. When that cannot be proven, the compiler will emulate a 32-bit multiplication using this option. The speed of the emulation depends on the actual bit patterns of the values being multiplied, but is approximately 10 cycles.

The MUL16 implementation is essentially free if either MAC16 or MUL32 is selected (the logic is shared).

Related Links

MUL32 Option on page 116

MAC16 DSP Instruction Family on page 117
16-bit Multiply/Accumulate with 40 bit accumulator (instruction family)

4.2.2.3 MAC16 DSP Instruction Family

16-bit Multiply/Accumulate with 40 bit accumulator (instruction family)

The MAC16 instruction family is a series of instructions allowing a 16-bit multiply accumulate into a 40-bit accumulator in parallel with two 16-bit updating loads. It allows a full iteration of a 16-bit dot product every cycle. Note that the instructions in this family that perform loads in parallel with the multiply accumulate are specialized and are not inferred by the C compiler. The only way to use these instructions is with compiler intrinsics or with hand-coded assembly. Note that using intrinsics, the specialized m registers are accessed by passing in their index, 0 to 3, directly into the intrinsic. The compiler is able to infer use of the multiply accumulate instruction that does not execute in parallel with a load. However, this instruction is typically no faster than what is enabled by the MUL16 option.

With no other multiplication options, the compiler will emulate 32-bit multiplications using the MAC16 instructions.

4.2.2.4 CLAMPS Option

Signed CLAMPS instruction (for saturating arithmetic)

The CLAMPS instruction tests whether a signed integral variable fits into a fixed number of bits ranging from 7 to 22. If so, the input value is returned. If not, the largest value with the same sign that does fit into the fixed number of bits is returned. This option is particularly useful for implementing saturating arithmetic. The compiler will automatically infer this instruction from the pattern.

`x = min(max((x,-2`^{n}`),2`^{n}`-1))=`

Min and max are first inferred from standard C conditional operations.

Consider the following example:

```
if (a < -1024) {
    result = -1024;
} else if (a > 1023) {
    result = 1023;
} else {
    result = a;
}
```

When compiled with optimization, the compiler will generate a CLAMPS instruction with an immediate value of 10.

4.2.2.5 32-Bit Integer Divider

32-bit integer divider that completes in a variable number of cycles depending on the operands

This option adds four instructions that are used to perform 32-bit integer division. The instructions compute the quotient or the remainder, for signed or unsigned numbers respectively. These instructions take from two to 13 cycles depending on the bit patterns of the dividend and the divisor. The divide instructions are implemented using iterative or non-pipelined hardware, which means that instructions subsequent to the divide will not begin execution until the divide operation is complete. The compiler will infer the use of these instructions for all 8-, 16- and 32-bit integer division and modulo computations.

4.2.2.6 Single Precision FP Option

IEEE single precision floating point instruction family. Uses coprocessor ID 0

This coprocessor supports fully pipelined, single precision floating point operations. Divide and sqrt are implemented using a series of operations. Without this option, a single-precision floating point is supported using emulation at speeds that are typically 50-100 cycles per base floating point operation. Configurations without integer multipliers are significantly slower still.

4.2.2.7 Single+Double Precision FP option

IEEE single + double precision floating point instruction family. Uses coprocessor ID 0

This coprocessor supports fully pipelined, single and double precision floating point operations. Divide and sqrt are implemented using a series of operations. Without this option, floating point is supported using emulation at speeds that are typically 50 to 200 cycles per base floating point operation. Configurations without integer multipliers are significantly slower still.

4.2.2.8 Double Precision FP Accelerator Option

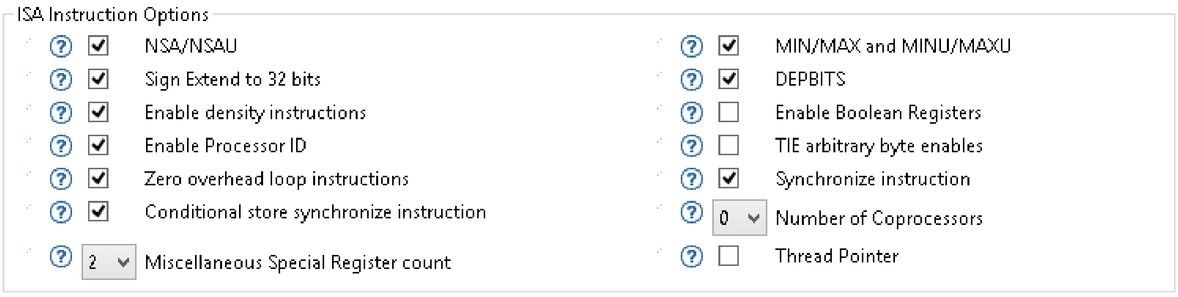
This coprocessor adds a modest 4-7K gates to speed up the emulation of double precision floating point operations. Most base operations (adds, multiplies, compares) are sped up by about a factor of two while divide is sped up by seven and square root by a factor of 11. Adds and subtracts take approximately 20 cycles and multiplies take approximately 27 cycles on configurations that contain the MULSH and MULUH instructions. When compiling an application with optimization and the -mcoproc option, this coprocessor will also speed up integer division by a factor of two to an average of about 20 cycles.

4.2.2.9 User-Provided Multiply Semantic

It may be desirable to create a processor with both custom multiply instructions written in TIE as well as one or more of the multiply instructions available as configuration options. In such a scenario, hardware can be saved by having the TIE instructions share the multiplier hardware with the core instructions. This can be achieved by allowing the user to write a single hardware description in TIE to implement both core and custom multiplication operations. By selecting the *User Provided Mul Semantic* option, Cadence will not provide an optimized implementation of the core multiply instructions and will assume that one is provided by the TIE developer. If the TIE developer does not provide such an implementation, the TIE compiler will fall back to using reference implementation of the core multiplies that may be slower and larger than necessary. When selected, the user can also set the pipeline stages used by the core multiplies, allowing for example the implementation of single cycle or three cycle core multiply instructions. Note that this option is compatible with the MUL16 and the MUL32 configuration options but not with the MAC16 instruction family.

Restriction: The separate read and write stages for MUL32 are LX5/X10++ only. For previous releases, the MUL16 read and write stages are used for both MUL16 and MUL32.

4.2.3 Miscellaneous ISA Instruction Options



ISA Instruction Options

✓ NSA/NSAU	✓ MIN/MAX and MINU/MAXU
✓ Sign Extend to 32 bits	✓ DEPBITS
✓ Enable density instructions	☐ Enable Boolean Registers
✓ Enable Processor ID	☐ TIE arbitrary byte enables
✓ Zero overhead loop instructions	✓ Synchronize instruction
✓ Conditional store synchronize instruction	0 Number of Coprocessors
2 Miscellaneous Special Register count	☐ Thread Pointer

These can all be selected from the Instructions page of the Xplorer Configuration Editor. Refer to the Architectural Options chapter in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for more details.

4.2.3.1 NSA/NSAU Option

Normalize shift amount

The NSA (normalize shift amount) instruction returns the number of contiguous sign bits in the most significant bits of a 32-bit signed value. The NSAU instruction returns the number of contiguous zero bits in the most significant bits of a 32-bit unsigned value. These instructions are not inferred by the compiler. However, library routines provided by Xtensa, in particular integer division and modulo, make heavy use of these instructions when present. As these instructions require relatively little hardware, the option is recommended for all but small configurations. The instructions can be accessed via intrinsics as follows:

```
#include <xtensa/tie/xt_misc.h>
int a, b;
unsigned ub;
...
a = XT_NSA(b);
a = XT_NSAU(ub);
```

4.2.3.2 MinMax Option

This performs a signed or unsigned integral minimum or maximum operation. These instructions require very little hardware and will be automatically inferred by the compiler. This option should be selected unless area is at an extreme premium.

4.2.3.3 SEXT (Sign Extend To 32-bits) Option

Sign extend to 32-bits (SEXT)

This replicates one of bits 7 to 22 of an integral value into all high order bits of the 32 bit integer. The compiler must perform sign extension whenever converting values of type int into variables of type short or signed char. Since the semantics of C and C++ frequently promote short or char types into ints, the compiler frequently has to sign extend when you use short or char variables. Without this instruction, the compiler must sign extended using a left shift followed by a right shift. This instruction requires little hardware, thus it should be selected unless area is at an extreme premium or your code does not utilize short or char data types.

4.2.3.4 DEPBITS Option

Deposit Bits

The DEPBITS instruction has a source register (as), a destination register (at), a four bit immediate (imm4), and a five bit immediate (imm5). Its operation is to take the imm4 low bits of ars, shift them left by imm5, and replace the corresponding bits of art with these. This instruction is inferred automatically by the compiler from standard C code.

Restriction: LX6/X11++ only

4.2.3.5 Density Instructions

Enable use of density (16-bit) instructions

This option enables the use of 16-bit instructions. The compiler will automatically use 16-bit variants of the core 24-bit instructions whenever possible to minimize code size. On average, enabling this option will reduce code size by 10% to 20%. This option should always be selected except in the very rare circumstance when you absolutely need the encoding space occupied by these instructions in order to encode a large number of 24-bit TIE instructions.

4.2.3.6 Boolean Registers option

Enable use of Boolean Register File (for TIE)

This option adds a set of 16 single-bit registers and instructions that operate on these registers. The instructions perform boolean operations on the registers and can branch based on the value of one or more of these registers. Note that no core instructions set the boolean registers. They are only set by custom TIE instructions or by Cadence coprocessors such as floating point, ConnX Vectra LX and HiFi 2. Therefore, this option is useless unless you have one of these coprocessors or custom TIE instructions. However, this option is absolutely required if you do have one of these coprocessors or custom TIE instructions. To utilize the booleans registers in your C or C++ program, you must include the xtensa/tie/xt_booleans.h file. This file defines new datatypes xtbool, xtbool2, xtbool4, xtbool8 and xtbool16, corresponding to single bit boolean variables or SIMD style sets of booleans. You may branch on xtbool conditions using standard C control flow constructs. The other operations on booleans are accessible via intrinsics.

4.2.3.7 Processor ID Option

Processor ID enables external logic to set a unique identifier for each processor in a system. This option is useful when a single piece of code executing on multiple processors wants to know which processor is invoking the instruction. The value of the id can be accessible in C as follows:

```
#include <xtensa/tie/xt_core.h>

int my_id = XT_RSR_PRID();
```

This register is also used by the XMP Shared Memory system, which reserves the lower eight bits of the register to number the cores in a subsystem in order.

4.2.3.8 TIE Arbitrary Byte Enables Option

Selecting the TIE arbitrary byte enables option allows you to use the StoreByteDisable interface in your TIE code

This option is necessary for TIE files and Cadence coprocessors that contain instructions that disable part of a memory store. A TIE developer might use this feature for developing predicated, SIMD, memory references. Example usage: the ConnX Vectra LX and ConnX D2 coprocessors use this feature for implementing unaligned data accesses. HiFi 2 uses this feature to do conditional, bit-stream writes.

4.2.3.9 Zero-Overhead Loops Option

Enable zero-overhead loop instructions (eliminates loop pipeline overhead)

These instructions enable looping with no per-iteration cycle overhead. The use of the loop instructions set up a loop trip count, beginning PC and ending PC. Whenever the hardware executes the last instruction in the loop and the loop trip count has not been exceeded, control is automatically transferred back to the first instruction without needing any explicit branch instructions. Because every taken branch on Cadence processors requires 3 to 5 cycles to execute, the zero overhead loop instructions are extremely useful for code that spends time in loops. When compiling with optimization, the compiler will typically use these instructions for every loop that does not contain a function call.

4.2.3.10 Synchronize Instruction

This option adds load-acquire and store-release instructions (L32AI and S32RI) to ease multiprocessor synchronization.

The option requires little area and is therefore recommended for multiprocessor designs communicating through shared memory.

Related Links

Memory Ordering on page 164

Conditional Store Sync option on page 123

4.2.3.11 Conditional Store Sync option

This option adds the S32C1I instruction to ease multiprocessor synchronization by providing an atomic compare and store operation.

Note that if you use the conditional store synchronization instruction for multiprocessor synchronization, you must implement PIF support for the RCW transaction. Further note, that the AXI and AHB bridges supplied by Cadence come with built-in support for this transaction that is implemented by locking the bus. Refer to the *Xtensa LX Microprocessor Data Book* for details.

These options require little area and are therefore recommended for multiprocessor designs communicating through shared memory.

Related Links

Memory Ordering on page 164

Synchronize Instruction on page 122

4.2.3.12 Number of Coprocessors option

Maximum number of coprocessors that can be defined in TIE to support efficient (lazy) register save/restore

Can be set from 0 to 8. Designating a coprocessor in TIE creates a bit in a special coprocessor enable register. When that bit is set, any access to the coprocessor's registers causes an exception. The associated exception-handling routine can then save the state of the coprocessor's registers before they are used, for example, in a new context.

4.2.3.13 Misc Special registers option

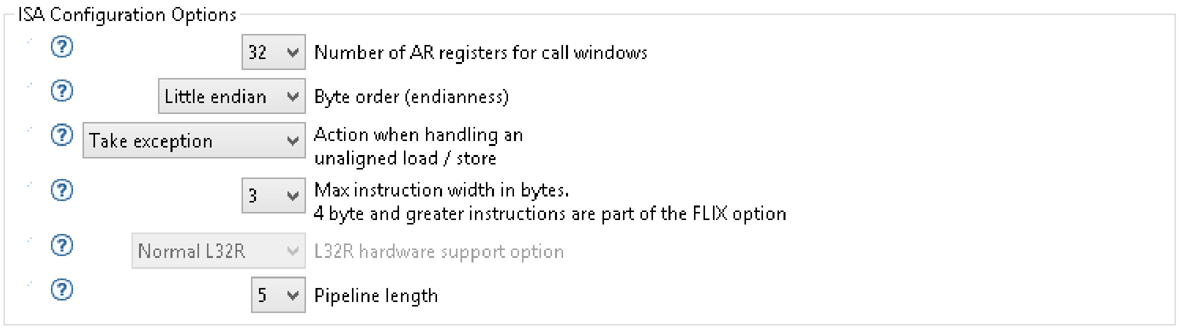
The miscellaneous special registers option provides zero to four scratch registers within the processor readable and writable by RSR, WSR, and XSR. These registers are privileged. They may be useful for some application-specific exception and interrupt processing tasks in the kernel. The MISC registers are undefined after Reset.

4.2.3.14 Thread Pointer

Thread pointer to facilitate tools/RTOS support of thread local storage

Select this configuration option if you want to run Linux on your Xtensa core. If the option is selected, your Xtensa core will include an additional 32-bit register called THREADPTR (user register 231). This is a dedicated register that will be used by the target Linux OS to support thread-local storage in NPTL (the Native POSIX Thread Library).

4.2.4 ISA Configuration Options


ISA Configuration Options

32 Number of AR registers for call windows

Little endian Byte order (endianness)

Take exception Action when handling an unaligned load / store

3 Max instruction width in bytes.
4 byte and greater instructions are part of the FLIX option

Normal L32R L32R hardware support option

5 Pipeline length

These can all be selected from the Instructions page of the Xplorer Configuration Editor. Refer to the Architectural Options chapter in the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for more details.

4.2.4.1 AR Registers Count

Number of physical AR registers. Setting to 16 registers means windowed calls are not supported

The Cadence windowed ABI allows you to have more physical AR registers than the 16 that are directly accessible by instructions in the ISA. On a call, the system rotates the AR register file, automatically giving the called function access to new scratch registers. This windowing mechanism allows for faster and smaller code. Whenever the number of physical registers is exceeded, an exception is thrown and the exception handler automatically saves and restores the excess registers to and from the memory stack. See the Xtensa Instruction Set Architecture (ISA) Reference Manual for more details.

Cadence allows 16, 32, or 64 physical registers. The selection of 16 registers necessitates the use of the CALL0 ABI. The use of 32 versus 64 physical registers does not affect application software. The choice is a trade-off between application performance and hardware area. The extra registers add approximately 5,000 gates to the processor but minimize the number of exceptions. If the number of gates is significant to your application, it is recommended that you profile your application and search for the functions beginning with _WindowUnderflow and _WindowOverflow. The more time spent in these handlers, the greater the value of having 64 physical registers.

Related Links

Application Binary Interfaces on page 19

4.2.4.2 Byte Ordering Option

Little or Big Endian

Cadence offers the choice of little or big endian as a configuration option. Every built processor configuration supports only one endianness. Cadence hardware and software supports both endianness equally. Mixing endianness in the same multiprocessor system is difficult.

4.2.4.3 Unaligned Load / Store Action Selection

How unaligned loads / stores are handled.

Traditional RISC processors expect that variables are aligned to their natural boundaries. For example, a 32-bit int variable is expected to be aligned to 32-bits. The C and C++ compilers will always align variables appropriately. However, through the use of casts, parameters or pointers might point to unaligned data. The compiler will assume that all data is properly aligned unless it’s obvious to the compiler that it is not. For example, ((int *) 0x1), is obviously unaligned.

Cadence offers three configuration options to deal with circumstances where unaligned accesses occur. With Align address, the hardware will zero the bottom bits of the address before performing the memory access. This is rarely the desired behavior and is mainly provided for compatibility with earlier Cadence processors that only offered this option. This option is also unsupported on configurations with a data memory interface of greater than 128-bits. With Take exception, the hardware will throw an exception whenever an unaligned access is attempted. Typically, unaligned accesses should be treated as programming errors and the exception is an aid to debugging. However, for those running Linux on Xtensa processors, the exception handler in Linux will emulate an unaligned hardware access using multiple-aligned accesses. Using the exception handler is slow, but is useful when running open source drivers that are not performance critical and assume support for unaligned accesses. Note that writing handlers that emulate unaligned accesses is not easy or supported for configurations with FLIX. The third option, Handled by hardware, has the hardware automatically support unaligned accesses. Note that the hardware for handling these accesses takes several cycles so that performance-critical code should still only issue aligned accesses. The advantage of having the hardware handle the unaligned accesses is that it is faster than the emulation routine available in the exception handler and is also able to work together with FLIX. The advantage of the exception is that it makes it easier to identify and fix unaligned accesses, leading to more efficient and reliable code. Diamond processors provided by Cadence use the Take exception option.

4.2.4.4 Max Instruction Width Option

Max instruction width. Xtensa core instructions are 2 or 3 bytes wide.

Cadence supports modeless intermixing of multiple instruction sizes. All configurations support 24-bit instructions. The use of 16-bit instructions is almost always recommended to save code size. The 16-bit instructions are equivalent variants of the most commonly used 24-bit instructions so that the compiler will always use them when possible. Additionally, Cadence supports customer defined FLIX (VLIW) instructions of any multiple of eight length from 32 to 128 bits. You can partition the wide instructions into custom slots, each of which is capable of executing one of a set of operations. The compiler automatically packs operations into the instructions. To utilize larger instructions, set the maximum instruction width appropriately.

4.2.4.5 L32R Hardware Support Option

Select what hardware support options to include for L32R.

The normal L32R instruction does a PC-relative load using a 16-bit offset to load literals. Literals are used to hold addresses of global variables and functions in addition to user specified literal constants. L32R is broadly supported by all Xtensa software and tools and is usually the best selection. However, on systems with instruction caches but no data caches, an L32R instruction might only be able to reach literals placed in system memory. Loading a literal from system memory on a core without a data cache will be very slow. Choosing L32R + Const16 adds the ability to load literals using 2 operations with 16-bit immediates, eliminating the need for any loads. When Const16 is enabled, the compiler (assembler) will generate that by default. Const16 requires one extra instruction to generate a literal, typically requires more memory and is not supported by all Xtensa software; Linux, for example, is not supported. However, const16 is much faster than an L32R that needs to load a literal from system memory without a data cache.

Restriction: LX6/X11++

Hardware support for Extended L32R is a legacy option which is supported through LX3/X8 processors only. Note that whether software makes use of normal L32R, extended L32R for loading literals is a separate software build option.

Related Links

Use Extended L32R Instruction (for Legacy Hardware) on page 20

4.2.4.6 Pipeline Options

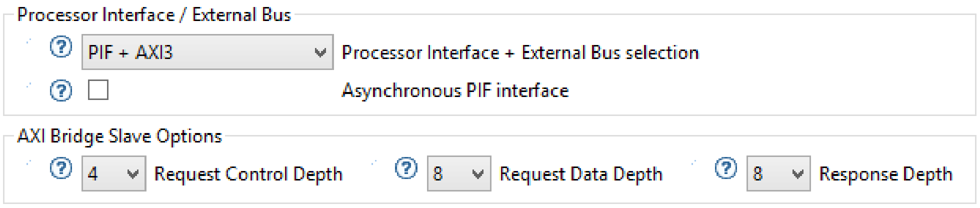
The default pipeline is 5 stage; LX processors can optionally be configured to have a 7 stage pipeline.

The base Cadence processor is a 5-stage pipeline micro-architecture with a single stage dedicated to data memory and another stage dedicated to instruction fetch. For large local memories (caches or on chip local memories) on configurations being clocked aggressively, the speed of the memory can limit the speed of the processor core. For such configurations, Cadence offers the option of adding two extra stages to the pipeline, one to the instruction fetch and another to the data memory. The use of these extra stages allow for larger and slower local memories without impacting the processor clock rate. These extra stages increase the branch penalty by one cycle and the load-use delay by one cycle. Therefore, they potentially slow down an application's speed, as measured in number of clocks required to complete the application. Some, applications, mostly DSP, will not slow down appreciably. However, control type code can slow down 15% or more.

4.3 Interface Options

Options which affect interfaces.

4.3.1 Bus and Bridge Selections


Processor Interface / External Bus

PIF + AXI3 — Processor Interface + External Bus selection

☐ Asynchronous PIF interface

AXI Bridge Slave Options

4 — Request Control Depth; 8 — Request Data Depth; 8 — Response Depth

These can all be selected from the Interfaces page of the Xplorer Configuration Editor.

Restriction: AXI4 and the AXI bridge slave options are LX5/X10++ only

4.3.1.1 PIF / Bus Selection

PIF interface to external memories with optional bus bridge

The PIF is the processor's main interface to memory, and is required for an Xtensa processor that is configured with caches and/or system memories.

In addition to the PIF, an external bus bridge can be selected - either AHB-Lite, or AXI.

Note: If you want to add or remove the processor interface, select an appropriate template and then make changes. Adding or removing the PIF makes significant changes to memory and vector configuration which can be tedious to reconcile.

Restriction: LX5/X10++ support both AXI3 and AXI4 bridges; earlier Xtensa versions only support the AXI3 bridge.

4.3.1.2 Asynchronous AMBA Bridge Interface

Asynchronous AMBA bridge interface for AXI or AHB-Lite bridge

If selected, the processor and bus clocks are asynchronous. The PIF asynchronous FIFO synchronizes PIF requests and responses. A standard asynchronous FIFO, which is instantiated between the core and bus bridge in the Xttop module, is used to synchronize transfers between the processor and bus clock domains.

4.3.1.3 Bridge Slave Options

4.3.1.3.1 AXI Slave Request Control Depth option

AXI slave Request Control Buffer Depth can be set to one of 1,2,4,8,16 entries. The default is 4 entries. Each entry consists of the AXI request control signals used by the bridge viz address, length, size, burst type and ID bits. This selection applies to both Read Address and Write Address channels.

Restriction: The AXI bridge slave options are LX5/X10++ only

4.3.1.3.2 AXI Slave Request Data Depth option

AXI slave Request Data Buffer Depth can be set to one of 1,2,4,8,16 entries. The default is 8 entries. Each entry consists of AXI write request data signals used by the bridge viz data, strobes, and ID bits. This selection only applies to Write Data channels.

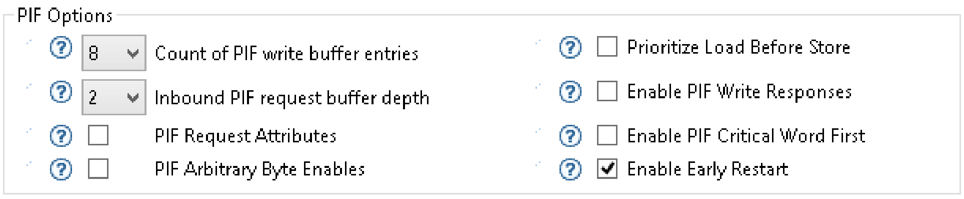
Restriction: The AXI bridge slave options are LX5/X10++ only

4.3.1.3.3 AXI Slave Response Depth option

AXI slave Response Buffer Depth can be set to one of 1,2,4,8, or 16 entries. The default is 8 entries. This parameter changes the number of entries in the read response buffer. Each entry in the read response buffer stores read data bits along with error control and ID bits. Note that the selection of this parameter only applies to Read Response channel. The depth of AXI slave write response buffer is not configurable and is set to 16 entries.

Restriction: The AXI bridge slave options are LX5/X10++ only

4.3.2 PIF Options



PIF Options

8 Count of PIF write buffer entries	☐ Prioritize Load Before Store
2 Inbound PIF request buffer depth	☐ Enable PIF Write Responses
☐ PIF Request Attributes	☐ Enable PIF Critical Word First
☐ PIF Arbitrary Byte Enables	☑ Enable Early Restart

These can all be selected from the Interfaces page of the Xplorer Configuration Editor.

4.3.2.1 Count of PIF Write Buffer Entries

Write buffer is used to mask the latency of large bursts of writes, data-cache dirty-line castouts, and register spills.

4.3.2.2 Prioritize Loads Before Stores

Allows the processor to reorder load requests before store requests on the PIF to reduce the load-miss penalty and improve performance. The decision to bypass store requests is made by comparing the eight address bits above the cache index - if the processor configuration has a data cache - or log2(LoadStore width) - if the processor has no data cache - for store operations currently residing in the write buffer.

4.3.2.3 Inbound PIF Request Buffer Depth

Inbound PIF allows an external PIF master to read/write to Xtensa's internal memories. When the inbound-PIF request option is enabled, an inbound-PIF request buffer is added to the processor. The depth of the inbound-PIF request buffer limits the maximum block-read or -write request that the Xtensa processor can handle. Therefore, the inbound-PIF request buffer size should be configured to be greater than, or equal to the largest possible inbound-PIF block-request size. Note that this control is enabled when the inbound-PIF option is selected for one or more local memories.

4.3.2.4 PIF Write Responses option

Enable PIF Write Responses. If enabled, a "Write Error" interrupt may be configured

A PIF transaction is composed of a request and a response. For a PIF write, the Xtensa processor can be configured to not count the write response. In that case, any write transaction that has been requested by the Xtensa processor, a PIF master, is assumed to be completed in the future. Any write response from a slave is accepted but ignored. Write responses may be important for memory ordering or synchronization purposes. However, on long latency systems, enabling write responses might significantly impact application performance.

When the write-response configuration option is selected, 16 unique IDs are allocated to write requests, and no more than 16 write requests can be outstanding at any time. In addition, store-release, memory-wait, and exception-wait instructions will wait for all pending write responses to return before those instructions commit.

Note: If the inbound-PIF configuration option is selected, the write responses option causes the PIF to send write responses to external devices that issue inbound-PIF write requests.

4.3.2.5 PIF Request Attributes option

The PIF can be configured to have an additional Request Attribute output for a PIF master (POReqAttribute[11:0]) and input for a PIF slave (PIReqAttribute[11:0]). This option is useful for implementing system level features, such as a second level cache.

Note: This option has been extended with LX5/X10++. It is compatible with the prior PIF3.2 option

4.3.2.6 PIF Critical Word First option

Enable loading of critical word first on a block request

The Xtensa processor can be configured to issue block-read transactions as Critical Word First Transactions. Any block-read can be issued such that a specified PIF width of the block will arrive first, with the remaining PIF widths arriving in sequential order and wrapping around to the beginning of the block e.g. a block-read of 8 PIF widths could arrive as 5, 6, 7, 0, 1, 2, 3, 4.

This requires that the Early Restart option be selected. Selecting both Critical Word First and Early Restart will improve the processor miss penalty by approximately 1 + (Cache_line_size/PIF_size) cycles. If only Early Restart is selected, you will get the same benefit if a miss is to the first element of a cache line but as little as one cycle of benefit if the miss is to the last element of the cache line.

Restriction: LX5/X10++ only

Related Links

Early Restart option on page 130
Enable Early Restart as soon as the critical word has been filled to a cache-line

4.3.2.7 PIF Arbitrary Byte Enable option

The Xtensa processor can be configured to use Arbitrary Byte Enables on write requests. This may allow more efficient execution when Arbitrary Byte Enable TIE is used that generates stores that have arbitrary byte enable patterns.

Note: LX5 and X10 processors will only generate Arbitrary Byte Enables on write single transactions.

Restriction: LX5/X10++ only

4.3.2.8 Early Restart option

Enable Early Restart as soon as the critical word has been filled to a cache-line

This option enables early restart on both instruction cache line misses and data cache line misses. For data, the instruction will wait in the M-stage of the pipeline and the data for the cache line miss data will be forwarded to it there. For instruction fetch, the fetch will stall in the I-stage and the cache line miss data will be forward to it there.Refill of the cache lines occurs in the background while the pipeline proceeds. This improves cache miss performance, at the cost of area.

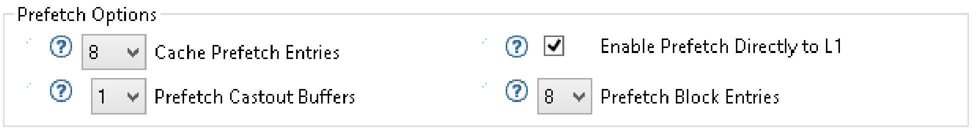
Selecting both Critical Word First and Early Restart will improve the processor miss penalty by approximately 1 + (Cache_line_size/PIF_size) cycles. If only Early Restart is selected, you will get the same benefit if a miss is to the first element of a cache line but as little as one cycle of benefit if the miss is to the last element of the cache line.

Restriction: LX5/X10++ only

Related Links

PIF Critical Word First option on page 129
Enable loading of critical word first on a block request

4.3.3 Prefetch Options


Prefetch Options
8 Cache Prefetch Entries
1 Prefetch Castout Buffers
✓ Enable Prefetch Directly to L1
8 Prefetch Block Entries

These can all be selected from the Interfaces page of the Xplorer Configuration Editor.

Selecting a non-zero count of cache prefetch entries option selects whether prefetch is included; the other options modify its behaviour.

Xtensa processors can prefetch to instruction and data caches if the relevant cache configuration options are compatible. If not, then prefetch will only be supported to the data cache. The requirements for prefetching to both instruction and data caches are as follows:

- Xtensa LX5 and X10 processors and earlier: The instruction and data cache line widths must be the same, plus the instuction cache and data cache access widths must be the same
- Xtensa LX6 and X11 processors and later: The instruction and data cache line widths must still be the same, but the access width restriction is relaxed as follows.
    - Either:
        - The instruction and data cache access widths must be the same
    - Or the following must all be satisfied:
        - Early Restart is selected
        - Data cache access width and PIF width are the same
        - Instruction cache access width is double the data cache access width

4.3.3.1 Cache Prefetch Entries

Controls how many cache lines can be held in the prefetch buffer

Select > 0 entries to include the cache prefetch unit.

4.3.3.2 Prefetch Directly to L1 option

Without this option, the prefetch engine will keep the prefetched data in the prefetch buffers until it is requested by a cache miss. If this is selected, it will speculatively move the prefetched data into the cache in order to try and avoid the cache miss entirely. This option can meaningful improve performance on systems with small latencies to memory, but adds a meaningful hardware overhead.

Restriction: LX5/X10++ only

Related Links

Prefetch Castout Buffers option on page 131
Number of Prefetch Castout Buffers that will be available when prefetch will be replacing a dirty cache line

Prefetch Block Count option on page 132
Number of blocks configured for Block Prefetch

4.3.3.3 Prefetch Castout Buffers option

Number of Prefetch Castout Buffers that will be available when prefetch will be replacing a dirty cache line

When Prefetch to L1 is configured, as well as write back caches, the prefetch logic may need to castout dirty lines from the cache. It does this in the background using the prefetch castout buffer

Restriction: LX5/X10++ only

Related Links

Prefetch Directly to L1 option on page 131

4.3.3.4 Prefetch Block Count option

Number of blocks configured for Block Prefetch

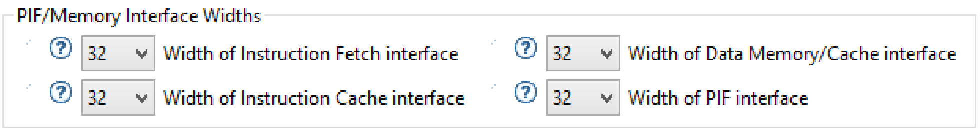
Enable block prefetches by setting the number of entries to eight. With block prefetch, software can ask the hardware to prefetch a large, multi-line, range of addresses into the cache. The hardware will transfer these blocks in the background enabling the hiding of the cache miss latency. Variants of block prefetch can also be used to prepare a region of memory to be stored without suffering any bus transactions to move the old values into the cache. Further variants allow for writing back a region of memory from the cache out to main memory.

Restriction: LX6/X11++ only

Related Links

Prefetch Directly to L1 option on page 131

4.3.4 Interface Width Options



PIF/Memory Interface Widths

32 Width of Instruction Fetch interface

32 Width of Data Memory/Cache interface

32 Width of Instruction Cache interface

32 Width of PIF interface

These can all be selected from the Interfaces page of the Xplorer Configuration Editor. The following simplified diagram shows the memory interface widths:

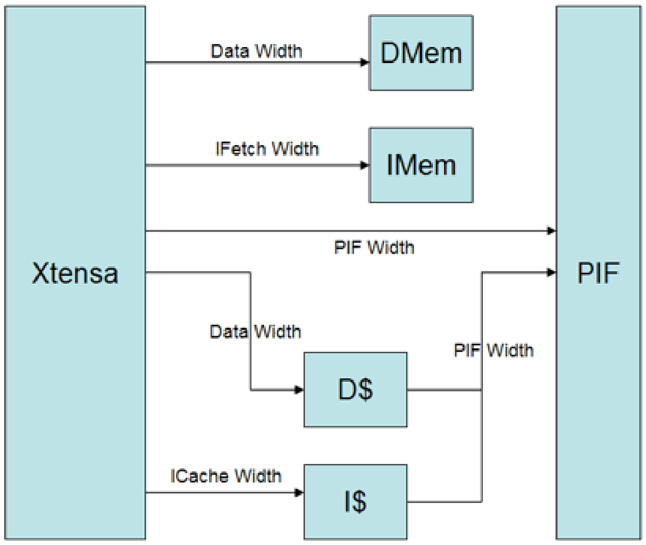


Figure 22: Memory Interface Widths

Cadence allows you to separately control the instruction fetch width, the data cache/memory width, the instruction cache/memory width and the width of the PIF inter- face. In general, wider widths give higher performance at a higher cost in area. The instruction fetch width controls how many bits are fetched in a cycle from the Icache or local memory into holding buffers. This parameter can be set to 32, 64 or 128. Configurations with 64-, 96- or 128-bit FLIX instructions are required to set this parameter to 64- or 128-bits respectively. For configurations without wide instructions, the use of a 64 or 128-bit fetch may still have two potential advantages. First, a taken branch on Cadence processors suffers a minimum two cycle penalty (three on configurations with a 7-stage pipe) assuming that the entire target instruction can be fetched with a single load from the local instruction memory. The fetch unit always fetches at least as much data as the size of the largest supported instruction. However, the fetch unit always fetches aligned data. If the target instruction crosses a 32-bit boundary assuming a 32-bit fetch width or a 64- or 128-bit boundary assum-ing a 64- or 128-bit fetch width, then there is one additional cycle of penalty. Fewer instructions cross 64- or 128-bit boundaries than 32-bit boundaries. Therefore a processor with a wider fetch will suffer fewer branch bubbles. Second, for straight line code, the use of a 64- or 128-bit fetch implies that the fetch unit needs to fetch fewer times. Fetching a single 64- or 128-bit chunk of instructions consumes less power than fetching multiple, 32-bit chunks. Of course, fetching 64- or 128-bits consumes more power than fetching 32-bits, and if the application branches sufficiently frequently, the use of a 64- or 128-bit fetch will not cut the number of fetches in half. Therefore, which configuration option consumes less power depends on how frequently branches are taken.

The data cache or memory width controls how many bits are transferred from external memory into the cache per cycle as well as how many bits can be loaded or stored from the cache or local data memory every cycle. The width must be at least as large as the largest load or store instruction and must be at least as large as the PIF. The PIF width is the width of the memory interface from external memory to the local memories or caches. It is also the data transfer width for non-local, uncached memory references. Larger PIF widths enable faster handling of cache misses. It is typically better to make your PIF width match your system bus width rather than externally bridge the processor to a smaller system bus width.

The Xtensa processor supports an optional hardware and software prefetch mechanism for systems with large memory latency. When the processor detects a stream of cache misses (either data or instruction), it can speculatively prefetch ahead up to four cache lines and place them in a buffer close to the processor. In addition, the user can explicitly control prefetching using the DPFR instruction. Prefetch is enabled by setting the number of Cache Prefetch Entries.

The interactions between the widths and other parameters are fairly complex, and are enforced by the Xplorer Configuration Editor. The basic rules are as follows:

- ICache width >= IFetch width
- ICache width <= max(IFetch, PIF)
- IFetch width >= Max instruction width
- Data width >= PIF width

4.3.4.1 Width of Instruction Fetch Interface

The instruction fetch width must be at least as large as the maximum instruction size.

Related Links

Interface Width Options on page 132

4.3.4.2 Width of Data Memory/Cache Interface

Width of interface to Data RAM, ROM, XLMI, Data Cache

The maximum width of data for a load or store instruction. Core instructions are 32 bits or less; TIE instructions can access up to this width in a single operation. Most Cadence DSP coprocessors make heavy use of wide loads and stores.

Restriction: Widths > 128 bits are LX4++ only

4.3.4.3 Width of Instruction Cache Interface

Normally the instruction cache width should be the same as the instruction fetch width, either 32-, 64-, or 128-bits. The instruction fetch width is the amount of data fetched from the cache on each instruction fetch read access, and there is rarely any benefit to having the instruction cache width wider than this, since wider memories usually consume more power. One exception is when instruction cache refill time is critical, since a wider instruction cache reduces this refill time, assuming a wider PIF interface is also used.

Must be at least the width of Instruction fetch Width and cannot exceed max(PIF, InstFetch).

4.3.4.4 Width of PIF Interface

The PIF can be configured to be 32, 64, or 128 bits wide. The PIF read and write data buses are the same width.

4.3.5 Port / Queue Options

4.3.5.1 GPIO32 Option

32-bit General purpose Input output TIE port interface. Contains a 32-bit output port (EXPSTATE) and 32-bit input port (IMPWIRE)

The GPIO32 option provides a pair of preconfigured output and input ports that allow SOC designers to integrate the processor core more tightly with peripherals. For example, these GPIO ports can be used to receive and send out control signals from/to other devices and RTL blocks.

4.3.5.2 QIF32 Option

32-bit TIE Queue interface (FIFO). Contains a 32-bit data output FIFO interface and 32-bit data input FIFO interface with handshaking signals

The QIF32 option provides preconfigured 32-bit input/output queue interfaces. These queue interfaces can be used to connect to standard FIFOs and thus, provide a very useful mechanism for the processor to communicate with other RTL blocks, devices, and even other processors, without ever using the system bus. These interfaces are accessed directly from the data path of the processor by using push-pop type of instructions, thus leading to very high communication throughput with the rest of the SOC.

4.3.6 Memory Error Type Selection

The configured memory error type applies to the corresponding selected caches and local memories

The processor's local instruction RAMs, local data RAMs, and memory caches can optionally be configured with memory parity or error-correcting code (ECC). These options add extra memory bits that are read from and written to the RAMs and caches. The processor uses the extra bits to detect errors that may occur in the cache data (and sometimes, to correct these errors as well). The parity option allows the processor to detect single-bit errors. The ECC option allows the processor to detect and correct single-bit errors and to detect double-bit errors.

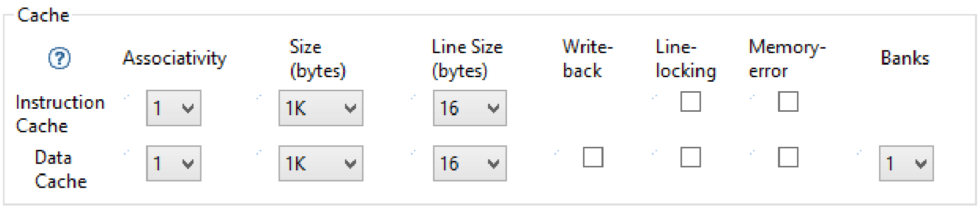
Data memory error width defaults to 1 byte - parity and ECC is computed on each individual byte. Setting it to 4 causes the parity and ECC calculations to be performed on 32-bit words.

Restriction: Data memory error width selection is available with LX6/X11++ only

A single memory error type can be configured for each of the instruction and data sides. Separately, each cache and memory has a selector as to whether memory errors are configured for it.

4.3.7 Caches and Local Memories

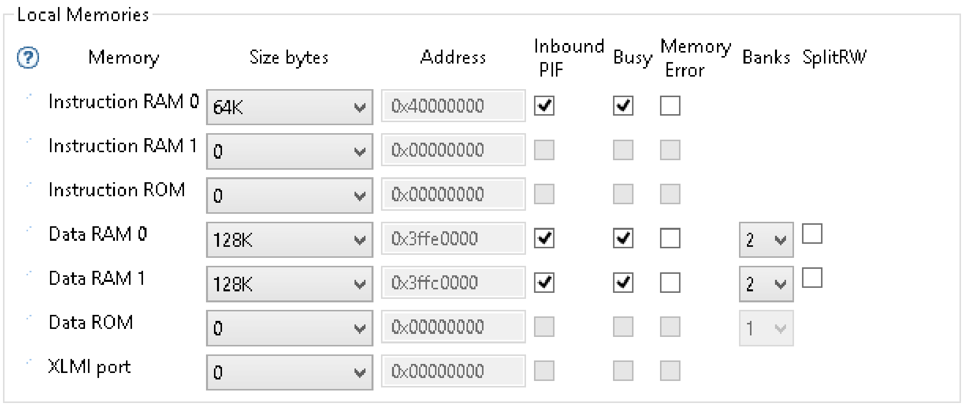
Subsections describe the individual cache and local memory options in more detail.



Cache

	Associativity	Size (bytes)	Line Size (bytes)	Write-back	Line-locking	Memory-error	Banks
Instruction Cache	1	1K	16		☐	☐	
Data Cache	1	1K	16	☐	☐	☐	1

Figure 23: Cache Options



Local Memories

Memory	Size bytes	Address	Inbound PIF	Busy	Memory Error	Banks	SplitRW
Instruction RAM 0	64K	0x40000000	✓	✓	☐		
Instruction RAM 1	0	0x00000000	☐	☐	☐		
Instruction ROM	0	0x00000000	☐	☐	☐		
Data RAM 0	128K	0x3ffe0000	✓	✓	☐	2	☐
Data RAM 1	128K	0x3ffc0000	✓	✓	☐	2	☐
Data ROM	0	0x00000000	☐	☐	☐	1	
XLMI port	0	0x00000000	☐	☐	☐		

Figure 24: Local Memory Options

Cadence allows up to six local memory interfaces on each of the instructions and data sides. Each interface might be a local RAM, local ROM or cache. Each way of a set-associative cache counts as one interface. The caches can be anywhere from 1 Kilobyte to 128 Kilobytes, from direct-mapped to 4-way set associative, with line sizes from 16 bytes to 256 bytes.

Caches allow reasonably robust performance with minimal effort. Local memories potentially allow higher performance and efficiency, but not always. Local memories support external DMA engines through the processor’s inbound PIF port. DMA allows you to work on one block of data while loading another block in the background. DMA potentially completely avoids the penalties of a cache miss. Of course, this only works if the working set sizes of the current block plus the block being loaded in parallel together are small enough to fit inside the local memory.

Caches work well when the total memory being used is significantly larger than the local memory size but the working set at any given time is sufficiently small. Local memories are much harder to use in such scenarios. Data must be explicitly and manually moved into and out of the local memory. Partitioning code is not always easy. You may try to use both local memories and caches, putting your frequently used data or code in local memories while leaving caches to handle the rest. This can be very effective if some code or data is small and used frequently, and other code or data is very large and is being streamed into the processor. Frequently, however, making such a clean partition is difficult; hardware does a better job of dynamically allocating memory to caches than you can statically. Local memories require less power to access than equivalently sized caches. Direct-mapped caches require significantly less power than set associative caches. Direct-mapped caches can perform well, but performance can be less robust. Small changes to an application can have a dramatic performance impact if two pieces of code or data suddenly fall into the same cache location. With direct-mapped caches, be certain to utilize some of the performance tuning and measuring methodologies described in Chapter 2 to make sure that you are not thrashing the cache. In particular, the Cache Explorer allows you to automatically simulate performance and power usage for various cache systems on your actual application, and the Link Order tool allows you to rearrange your code to minimize instruction cache misses.

Two local memories of size n/2 require less power than one local memory of size n. Two local memories can also increase the performance of DMA because the DMA engine writing into one memory will not compete for bandwidth with the processor trying to access the other memory. However, with two local memories, you must partition the data or code between the two local memories. Cadence also supports line locking of all but one way in a set associative cache. Line locking provides some of the benefits of local memories in a cache. In order to effectively utilize line locking, you must explicitly identify data or code that is small and frequently used. As with local memories, it is often hard to statically partition as well as the hard- ware caching mechanism is able to automatically partition.

Caches and local data memories can be divided into one to four banks. The data memory is divided into banks so that successive data memory width sized accesses go to different banks. At most one load or store can go to any one bank in a cycle. On configurations that support multiple loads or stores per cycle or on systems with DMA, using more banks will minimize the number of stalls due to bank conflicts.

Restriction: The limit in LX3 and Xtensa 8 and earlier processors was four local memory interfaces.

See the appropriate Data Book for more detailed information.

4.3.7.1 Instruction Cache Details

The parent topic (*Caches and Local Memories* on page 136) compares caches and local memories and considers the configuration choices.

- **Associativity**: - 1 through 4 way associativity is supported.
- **Size**: Total cache size of all configured ways. Minimum size of a way is 512 bytes; minimum cache size is 1KB.

- **Line Size**: Size of a cache line in bytes. The *critical word first* option can be used to reduce the penalty when a line is being filled from external memory.
- **Line Locking**: keeps a line in the cache until it is unlocked. Once locked, the line behaves similar to local memory. This is useful when working with a small piece of code without incurring an expense of having a local memory. Refer to the Local Memory Usage and Options chapter in the appropriate *Xtensa LX Microprocessor Data Book* for more information.
- **Memory Errors**: The instruction cache interface can optionally be configured with memory error protection using either parity or an error-correcting code (ECC).

  If memory-error protection is configured, the interface ports to each cache RAMs' data-array and tag-array (for both read and write data) are supplemented with memory-check bits that are read back from the cache with the cached data and are written to the cache memory in the same way as the regular data. The parity-protection scheme for the instruction cache uses one additional memory bit per 32 bits of data-array width and one additional memory bit for each tag array. For the ECC protection scheme, the instruction cache uses seven extra memory bits per 32 bits of data array width and seven extra bits for each tag array.

The dynamic cache way disable capability gives the ability to disable and re-enable the use of cache ways in both Instruction- Cache and Data-Cache independently to facilitate power savings. New and modified instructions enable the user to clean cache ways before disabling them and to initialize cache ways while enabling them. When a Cache Way is disabled, it removes that cache memory block from service. Therefore it reduces total cache capacity by 1/(number of ways in service).

Restriction: LX6/X11++ only.

Related Links

PIF Critical Word First option on page 129
Enable loading of critical word first on a block request

Early Restart option on page 130
Enable Early Restart as soon as the critical word has been filled to a cache-line

Memory Error Type Selection on page 135
The configured memory error type applies to the corresponding selected caches and local memories

4.3.7.2 Data Cache Details

The parent topic (*Caches and Local Memories* on page 136) compares caches and local memories and considers the configuration choices.

- **Associativity**: - 1 through 4 way associativity is supported.
- **Size**: Total cache size of all configured ways. Minimum size of a way is 512 bytes; minimum cache size is 1KB.

- **Line Size**: Size of a cache line in bytes. The critical word first option can be used to reduce the penalty when a line is being filled from external memory.
- **Write-back**: The data cache is a write-through cache by default. If this option is selected, the data cache can be programmatically toggled between write-back and write-through.
- **Line Locking**: The data cache allows line locking, which keeps a line in the cache until it is unlocked. Once locked, the line behaves similar to local memory. This is useful when working with a small piece of code/data without incurring an expense of having a local memory. Refer to the Local Memory Usage and Options chapter in the appropriate *Xtensa LX Microprocessor Data Book* for more information.
- **Memory Errors**: The data cache interface can optionally be configured with memory error protection using either parity or an error-correcting code (ECC).

  If memory-error protection is configured, the interface ports to each cache RAMs' data-array and tag-array (for both read and write data) are supplemented with memory-check bits that are read back from the cache with the cached data and are written to the cache memory in the same way as the regular data. The parity-protection scheme for the data cache uses one extra memory bit per byte of data array width and one extra memory bit for each tag array. The ECC protection scheme for the data cache uses five extra memory bits per byte of data array width, and seven memory bits for the tag array.
- **Banks**: If multiple banks are configured, then the data cache is divided into banks so that successive data memory width sized accesses go to different banks. At most one load or store can go to any one bank in a cycle.

  Restriction: Multiple banks are LX5++ only

The dynamic cache way disable capability gives the ability to disable and re-enable the use of cache ways in both Instruction- Cache and Data-Cache independently to facilitate power savings. New and modified instructions enable the user to clean cache ways before disabling them and to initialize cache ways while enabling them. When a Cache Way is disabled, it removes that cache memory block from service. Therefore it reduces total cache capacity by 1/(number of ways in service).

Restriction: LX6/X11++ only.

Related Links

PIF Critical Word First option on page 129
Enable loading of critical word first on a block request

Early Restart option on page 130
Enable Early Restart as soon as the critical word has been filled to a cache-line

Memory Error Type Selection on page 135
The configured memory error type applies to the corresponding selected caches and local memories

4.3.7.3 Local Memories

The parent topic (*Caches and Local Memories* on page 136) compares caches and local memories and considers the configuration choices.

Local Memory Options

To configure a local memory, choose a non-zero size. Note that by default the option "automatically select memory addresses" is enabled, and Xplorer will move memories to maintain proximity and alignment.

Note that the "normal" L32R instruction which is used to load literals has a range of 256K bytes preceding the current PC, so the default positioning of instruction / data puts data memories before instruction memories so the data memory can be used for literal storage. If there is no data memory within range, the editor warns because the compiler may have problems generating literals if compiling code into that memory.

Attributes that can be selected for local memory interfaces (inbound PIF, busy and memory error) must be consistent for each memory type. E.g. if you configure 2 data RAMs then either both must have inbound PIF configured, or neither.

Selecting **Inbound PIF** allows an external PIF master to read/write to Xtensa's internal memories. When the inbound-PIF request option is enabled, an inbound-PIF request buffer is added to the processor.

Selecting **Busy** will create an external interface to the processor that the processor will check before writing to the memory.

Banks: Data RAM and Data ROM can optionally be configured in 2 or 4 banks. If multiple banks are configured, then the data RAM is divided into banks so that successive data memory width sized accesses go to different banks. At most one load or store can go to any one bank in a cycle.

Split Read-Write port: For multiple load-store configurations, this brings out the interfaces for all load-store units so you can handle the multiplexing and banking ouside of the core. The alternative is to select CBOX which handles the multiplexing inside the core.

Restriction: Multiple Bank and Split Read-Write port options are LX5++ only

See the appropriate Data Book for more detailed information.

Related Links

Memory Error Type Selection on page 135
The configured memory error type applies to the corresponding selected caches and local memories

Automatically Select Memory Addresses on page 141
Xplorer can automatically position local memories for alignment and proximity needs

4.3.7.4 Automatically Select Memory Addresses

Xplorer can automatically position local memories for alignment and proximity needs

By default the option **Automatically select memory addresses** is enabled. In this mode, the Configuration Editor chooses appropriate local memory addresses to keep them naturally aligned and close together such that literals can be loaded from data memory with L32R.

If you want to ensure exact compatibility with addresses of some other processor configuration it is appropriate to uncheck the auto-placement option so you can enter specific addresses. Consider also the memory regions in which system and local memories are placed such that your cache attributes and power considerations are met.

Note also that exception vectors need to be placed inside valid memories. If vectors are marked as being placed in an instruction memory, Xplorer's default behaviour is to automatically match the vector location with that of the containing memory.

Local Memory root address: If the full MMU is selected, then the configuration editor requires that local memories be "auto-placed" to ensure that there are no address space collisions. This option lets you choose whether that local memory auto-placement occurs in kernel space or user space.

4.3.7.5 Load/Store Units

Cadence supports the use of one or two load/store units. Dual load/store units potentially allow your application to issue two load/stores every cycle. The processor will only take advantage of the second load/store unit if you have custom FLIX TIE instructions with loads or stores in multiple slots, or if you utilize the dual load/store variant of a ConnX processor, which come pre-built with two load/store units.

Data caches on dual load/store configurations must be at least 2-way set associative and must have at least two banks. The memory is banked so that successive data memory access width size references go to different banks. The processor can not issue multiple memory references to the same bank in the same cycle. The compiler will try to compile code to avoid bank conflicts.

Local data memory can be optionally banked. In addition, the user can select the connection box option. With this option, if two memory references go to different local data memories, the processor will not stall. If two memory references go to the same bank of the same local data memory, the connection box will stall the processor for one cycle. With no banking and no connection box, dual load/store configurations require the use of dual-ported memory.

Restriction: Xtensa X and TX processors support just one load / store unit.

Related Links

Connection Box on page 142

The Connection Box is useful for DSP's with X-Y memory configurations

4.3.7.6 Connection Box

The Connection Box is useful for DSP's with X-Y memory configurations

This option implements an internal CBOX for configurations with multiple load-store units where multiplexing is handled inside the core. An alternative for local memories on LX5++ processors is to instead select split read-write port on the memories which brings out the interfaces for all the load-store units so multiplexing can be handled outside the core.

Related Links

Load/Store Units on page 141

5. TIE Programming

Topics:

TIE gives you great power to customize your processor for an application or an application domain while still maintaining a high level programming model

This chapter provides an overview of the software aspects of TIE; the software constructs in the TIE language and how to utilize TIE from your C application.

5.1 Software Constructs in TIE

This section covers some of the software related issues when writing TIE files. The following assumes a basic familiarity with writing TIE. If you are using TIE but not writing TIE, you can safely skip this section.

5.1.1 Custom Register Files

TIE allows you to create custom register files. Variables of the standard C datatypes are never by default placed in TIE register files. Instead, you must define a custom datatype to be associated with each custom register file. The C programmer will be able to declare variables of the custom data type, and the compiler will automatically load and store and register allocate the variables as needed.

In the simple case, you may want a single datatype to be associated with a register file. However, just like both an `int` and a `short` live in the standard `AR` register file, there may be scenarios where multiple datatypes will be associated with a single custom register file. In the simple case of a single datatype, the essential programming support for a register file is generated automatically from the register file description. A new datatype is created with the same name as the register file. Custom instructions to load, store, and move variables of the datatype are automatically generated. The compiler automatically knows to use these instructions when loading or storing variables of the custom datatype. Any instruction that expects an argument from a custom register file can be accessed in C via an intrinsic that expects a corresponding argument of the associated datatype.

If you want multiple datatypes, you must add information to the TIE description to tell the software tools how to utilize the multiple datatypes. First, you must declare the set of desired types using the `ctype` construct.

```
ctype name size alignment regfile-name
```

`ctype` is a keyword. *name* is the name of the type both in the TIE description and in the associated C files. `size` is the size of the type and `alignment` is the required alignment, both in bits. The compiler will always align variables of the ctype to the required alignment. On configurations with a memory access width of 128-bits or less, the standard libraries and run-time provided by Cadence align the stack to128 bits and malloc returns a 128-bit aligned value. On configurations with 256- or 512-bit memory accesses, malloc returns a 256- or 512-bit aligned valued. The stack is by default still only aligned to 128-bits. The compiler will automatically adjust the stack dynamically whenever compiling functions that require greater alignment.

For every register operand of a C intrinsic, the compiler expects a variable of one particular type. If a register file has only one `ctype`, that `ctype` is the expected type. If there are multiple ctypes defined for a register file, you must associate a specific ctype with every operand of that register file. Globally, for every register file there is a default `ctype`. You can select the default by adding the default keyword to the ctype definition in the TIE compiler will select one.

If a given instruction is meant to take an argument of a type other than a default type, you must specify the interface of the instruction using a TIE proto statement. A TIE proto is a mechanism for the TIE designer to express how TIE register files and instructions should be used by the application programmer and by the software tools such as the compiler, debugger and operating systems. See the *Tensilica Instruction Extension (TIE) Language Reference Manual* for details.

Consider the following simple example:

```
proto foo {inout mytype m} {} {
    foo m
}
```

A proto named `foo` is defined that takes a single argument of type `mytpe` and that proto invokes the single instruction `foo`. Every proto creates a C intrinsic recognized by the compiler. If the `foo` intrinsic is called with a variable of type *mytype*, the compiler will generate the foo instruction. In this simple example, the proto was associated with a single instruction of the same name, but the proto construct is more general. A single proto (and therefore a single intrinsic) might be associated with a series of TIE instructions.

For instructions that operate on the core AR register file, TIE predefines `ctypes` associated with the standard base ctypes: `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32` and `int64` corresponding to the unsigned and signed versions of `char`, `short`, `int` and `long long` respectively. The default type for the AR register file is `uint32`, meaning that any instruction with a proto that operates on the AR register file will be associated with an intrinsic that expects a variable of type `unsigned`. If the intrinsic is invoked with a variable of a different type, standard C rules will be used to convert the argument to the type `unsigned`. Note that converting a small datatype into an `int` or an `unsigned int` is free because the 32-bit AR register file represents all small datatypes as their 32-bit equivalent. On the other hand, converting an `int` or an `unsigned` to a shorter datatype requires the compiler to truncate the value into the number of bits available in the shorter datatype. For example, if a TIE proto is written to take an input argument of type `uint16` and the corresponding intrinsic is invoked on a variable of type `unsigned int`, the compiler must truncate the input argument. This is typically wasteful because the TIE instruction is usually written to truncate its input argument. Therefore, it is usually better to use `int32` or `uint32` for all input arguments to protos. On the other hand, if the TIE instruction is producing a result of a shorter datatype, the proto should be written to return the corresponding shorter datatype. That way if the result is assigned to a variable of corresponding type, the compiler does not need to truncate the result before completing the assignment.

Given a variable of a particular `ctype`, the compiler needs to know how to load and store the variable. If there is only one `ctype`, the TIE compiler automatically generates load, store, and move instructions for the `ctype`. If there is more than one `ctype`, you must explicitly create the instructions and tell the compiler which instructions to use using a set of stylized protos with the name `<ctype>_loadi`, `<ctype>_storei` and `<ctype>_move`, as follows:

```
proto ctype_loadi {out ctype c, in ctype *p, in immediate o} {} {
    ...
}
proto ctype_storei {in ctype c, in ctype *p, in immediate o} {} {
    ...
}
proto ctype_move (in ctype cin, out ctype cout) {} {
    ...
}
```

All the protos expect their arguments as shown. The load and store protos expect the data to be the first argument. The address of the corresponding load or store must be the value given by the sum of the pointer to the `ctype` and an immediate. Indexed versions are also optionally described using the `<ctype>_loadx` and the `<ctype>_storex` stylized protos as follows:

```
proto ctype_loadx {out ctype c, in ctype *p, in uint32 x} {} {
    ...
}
proto ctype_storex {in ctype c, in ctype *p, in uint32 x} {} {
    ...
}
```

The body of the protos must contain one or more instructions that implement the load, store or move as defined. The second set of brackets contain an optional set of temporary variables. These are often needed when the load or store proto is implemented using multiple load or store instructions in sequence.

The compiler may use these protos arbitrarily to spill or copy variables of the associated `ctypes`, therefore it is essential that the instruction sequence implemented inside the proto preserve all the bits needed to represent a variable of the `ctype` and have no other side effects. Note that it is not always necessary to preserve all the bits in the register file; a 16-bit type living in a 32-bit register can utilize protos that only save and restore 16-bits of the register file. However, the default `ctype` is also used by operating systems and debuggers to save and restore registers containing unknown variables so the default `ctype` must be associated with protos that save and restore all the bits in the register file.

Note that even if a register file is associated with only a single `ctype`, you might still explicitly write the protos for that type manually. If it is not possible to load or store a `ctype` with a single instruction or if for example the register is wider than 512 bits, you must explicitly provide the protos.

With multiple datatypes, you may want to convert a variable of one datatype to another. The compiler needs to know what instruction sequence to use for such conversions. Again, stylized protos are used: `<ctype1>_mtor_<ctype2>`, `<ctype1>_rtom_<ctype2>` and

`<ctype1>_rtor_<ctype2>`. At any point in time, a variable might be in a register file or in memory. To convert, different instruction sequences are required depending on whether the variables are in memory or in register files. You can supply all three protos (it is not possible to directly move a variable from one memory location to another), and the compiler should use the most efficient sequence for the situation. If a proto is missing, `<ctype1>_rtor_<ctype2>` for example, the compiler will use a two proto sequence to convert into memory and then load the destination type from memory into a register file.

Unlike C++, the compiler will not implicitly convert using transitive data rules, other than those defined in standard C. If the C program assigns to a variable of custom datatype `type3` an expression of type `type1`, and the TIE programmer provides protos to convert from `type1` to `type2` and from `type2` to `type3`, the C compiler will not do the conversion and will instead generate an error.

5.1.2 TIE State

TIE also allows the creation of state. While a register file includes multiple identical entries and the compiler will allocate variables to different entries, a state contains only one entry. A C/C++ programmer uses a piece of state implicitly by using an instruction that utilizes the state. There are no types or variables associated with that state. State is defined in TIE through the use of the `state` keyword.

```
state name width {reset value} {add_read_write} {export} {shared_or}
```

The `add_read_write` keyword instructs the TIE compiler to create instructions, `RUR.`*<name>* and `WUR.`*<name>* copy the state to and from the standard AR register file. These instructions can be used directly by the programmer, but are also automatically used by the debugger and by operating systems to access and context switch the value of the state. If a state is larger than 32 bits, name is suffixed by the strings *_0*, *_1*, etc. as needed.

Sticky Bits

Given multiple invocations of instructions that write the same state, the compiler will ensure that separate invocations are kept in order. For many uses of state, that is correct and necessary. However, there are circumstances where it is not necessary and requiring a strict ordering can negatively impact performance. For example, imagine defining a coprocessor with a single bit overflow state that is set to 1 whenever any instruction encounters an overflow. The programmer does not want to check for overflow after every instruction. Instead, the programmer might check for overflow after a loop or after a phase of the algorithm, and in the rare cases of overflow might, for example, scale the data and invoke the phase or loop again. Therefore, the programmer doesn't care which operation caused the overflow, she just wants to know if any operation caused an overflow.

To tell the compiler that accesses to a state can be reordered, add the following property to your TIE file.

```
property ignore_state_output state name {op1, op2, ...}
```

The property lists the state and all the operations accessing the state that can be reordered. Note that not all operations accessing the state can be reordered. In particular, there is typically an operation that checks the value of the state and another that clears the state. It is important that the compiler not move a computation past the operation that actually checks for overflow.

Note that if none of the other outputs of an instruction marked with this property are used, the instruction may be deleted by the compiler. Consider the following example.

```
void func(int n, mytype *data)
{
    int i;
    mtype sum;
    for (i=0; i<n; i++)
        sum = myadd(sum, data[i]);
}
```

Assume that `myadd` performs an addition and potentially sets an overflow state.The compiler may delete the entire body of the function since the result of the computation, `sum`, is a local variable that is never used.

The use of overflow is a particular issue on processors utilizing VLIW. If a single VLIW instruction can issue multiple operations in one instruction, it is useful if multiple of those operations can write the bit. If they keyword, `shared_or` is added to a state declaration, the compiler is allowed to issue multiple writes to the state in the same VLIW instruction, and the hardware will set the state to the bitwise OR of the result of the two operations. The `shared_or` attribute is only useful together with the `ignore_state_output` property. However, there are circumstances where the property is useful even though it is not legitimate for two operations to write the state in the same cycle.

5.1.3 Inference

There are multiple TIE constructs designed to enable TIE inference in the compiler; that is, to enable the C compiler to automatically utilize a TIE instruction given a standard C program or given a C program using intrinsics for more simplified TIE.

Although you can hand write all these constructs, caution is advised. Using these constructs effectively, in a way that the compiler will be able to infer automatically, is difficult and error prone.

5.1.3.1 Imap

`imap` is a general construct to tell the compiler that a sequence of input instructions, either core instructions or base TIE instructions, is equivalent to a single output instruction. The compiler uses a general graph matching algorithm to search for the input pattern, and if it finds the pattern, the compiler will replace the input series of instructions with the single output instruction.

In order to successfully match a pattern, the pattern must be identical to the sequence of instructions generated by the compiler at the phase in the compilation process where matching occurs. The correct sequence is often not obvious.

Sometimes, it may be desirable for the output operation's behavior to not match exactly the behavior of the sequence of input operations. For example, a multiply-add instruction might be more accurate than the sequence of a multiply followed by an add if the multiply and the add instruction each round their results while the multiply-add might do a single round at the end of both. In such scenarios, the user might still want the compiler to automatically infer the use of the multiply-add, while still giving the C programmer enough control to disable the inference in cases where the exact rounding behavior is needed. For such cases, the TIE developer can specify that an imap has inexact behavior by adding the property `imap_nonExact` to the `imap` in the TIE file. XCC will try to use the `imap` by default only at optimization level `-O3`, and regardless of optimization level, this inference can be explicitly turned on with the compiler option `-menable-non-exact-imaps` and can be turned off with `-mno-enable-non-exact-imaps`.

A detailed description and syntax of the imap construct can be found in the *Tensilica Instruction Extension (TIE) Language Reference Manual*.

5.1.3.2 Updating Loads and Stores

The `loadi` and `storei` stylized protos described in *Custom Register Files* on page 144 allow the C programmer to utilize custom load and store instructions implicitly, without intrinsics. Given an array variable `bar` of custom type `foo`, you can assign to the variable using the simple assignment syntax `bar[...] = ...`, no intrinsics are needed. A common optimization is the use of updating load and store instructions that increment the address register as well as perform a load or store. The compiler will automatically utilize updating loads or stores if the TIE writer describes them using customized protos: `ctype_loadiu`, `ctype_storeiu`, `ctype_loadxu`, `ctype_storexu`.

```
proto ctype_loadiu {out ctype c, inout ctype *p, in immediate o} {} {
    ...
}
proto ctype_loadxu {out ctype c, inout ctype *p, in uint32 x} {} {
    ...
}
proto ctype_storeiu {in ctype c, inout ctype *p, in immediate o} {} {
    ...
}
proto ctype_storexu {in ctype c, inout ctype *p, in uint32 x} {} {
    ...
}
```

These protos expect the first argument to be the data being loaded or stored from the address given by the sum of the pointer and immediate or index register. The body of the protos must contain a single load or store instruction that loads from the given address. The instructions must be pre-incrementing loads and stores; that is, the instructions must first add the two arguments to compute the address and then on output must update the pointer with the address just accessed.

Similar, post-incrementing loads and stores are also supported. For these, the address being loaded or stored is simply the base argument and after the load or store the address is updated by the immediate or index variable.

```
proto ctype_loadip {out ctype c, inout ctype *p, in immediate o} {} {
    ...
}
proto ctype_loadxp {out ctype c, inout ctype *p, in uint32 x} {} {
    ...
}
proto ctype_storeip {in ctype c, inout ctype *p, in immediate o} {} {
    ...
}
proto ctype_storexp {in ctype c, inout ctype *p, in uint32 x} {} {
    ...
}
```

5.1.3.3 Specialization

Cadence allows the creation of 32-, 64-, 96- or 128-bit FLIX, or VLIW instructions. Each instruction will be packed by the compiler with core or TIE operations. A 32-bit instruction does not have much space to encode even a pair of operations. Therefore, the constituent operations must be small. One technique to limit the size of an instruction is to limit the legal values of its immediates. For example, you might create a variant of a 24-bit instruction that is equivalent to the original instruction except that the immediate range is more limited. You would probably like the compiler to automatically choose between the variants, using the limited range variant when possible and profitable and the full range variant otherwise. For the compiler to automatically choose, the compiler needs to be told that the two instructions are equivalent. That can be done using a property statement, as follows:

```
property specialized_op general_operation specialized_operation
```

The `specialized_operation` must be exactly equivalent to the `general_operation` other than the allowed range of immediates. In the source C program, either intrinsic can be used and the compiler will automatically choose between the two variants if possible when optimizing for performance.

The `specialized_op` property can also be used when one of the operations takes an `AR` register argument rather than an immediate. In such cases, the compiler will automatically switch from the immediate version to the register version if the immediate value is out of range.

5.1.4 Coprocessors

Custom register files can significantly increase the amount of state that needs to be saved and restored by a multi-threaded operating system. Often, a custom register file might only be used by one or by a few threads in a system. Saving and restoring the register file on every context switch can be wasteful. Therefore register files can be grouped together into coprocessors. Register files in coprocessors need not be saved on context switch. Instead, the OS can set them up to throw an exception on use. That way, the save and restore is delayed until use, and in the case that a particular thread does not use the coprocessor, can be completely eliminated.

Register files are grouped into coprocessors using the coprocessor statement as follows.

```
coprocessor name number {list of register files}
```

The number is a number between 0 and 7.

TIE queues, ports and lookups can also be grouped together into coprocessors to allow a protected operating system, such as Linux, to control access to these resources. The operating system can chose to force the hardware to throw an exception if an unauthorized task tries to access these interfaces.

5.2 Type-based TIE

The C compiler will take advantage of TIE instructions, automatically using inference techniques, manually using a technique called Type-based TIE or using a combination of the two. FLIX, or VLIW, is a special case. It is not possible to manually program in terms of FLIX instructions at the C level; you must program in terms of base operations and allow the compiler to schedule and bundle automatically. The remainder of this section describes Type-based TIE.

5.2.1 Intrinsics

In the simplest case, a TIE developer can create arbitrary instructions that operate on pre-existing core states, most typically the AR register file. For each such created instruction, XCC automatically supports an intrinsic with the same name. To avoid name space pollution, you must include a special header file before using the intrinsic, as follows:

```
#include <xtensa/tie/tiefilename.h>
```

If a configuration has a single TIE file, `tiefilename` is the name of that TIE file. If a configuration has multiple TIE files, the TIE developer has a choice of using one include file for all the TIE files or using a separate one for each. If you are uncertain, the easiest thing is to grep for the instruction name in all the files in the directory `xtensa/tie`, located under `tdk/include` when using the dynamic TDK flow or under `workspace/XtDevTools/install/builds/release_version/config_name/xtensa-elf/arch/include`, when the TIE file is part of the configuration.

The include file is necessary to avoid name space pollution. Otherwise, if you create a TIE instruction with the name max, for example, it would not be possible for any C code to define a function with the same name.

Intrinsics are not macros and are not implemented using asms, inline assembly. Intrinsics are implemented internally in the XCC compiler. Since the compiler knows what state is read or written by every instruction, the compiler knows how to correctly and efficiently optimize in the presence of TIE intrinsics.

In TIE, an argument can be one of `in`, `out` or `inout`. It is also possible to implement predication in TIE, where an `out` or `inout` argument is conditionally killed based on some user-defined criteria. To the C compiler, a conditionally killed argument is treated as an `inout` parameter, one whose value is either the new value or the old value, in the case that the operand is killed.

If a TIE instruction has at most one `out` and no `inout` (or predicated) operands, that one output argument is the return value of the intrinsic. All the `in` operands of the instruction are the arguments to the intrinsic, in order. If there are multiple `out` arguments or any `inout` or predicated arguments, the intrinsic will not return a value. Instead, the intrinsic will take each argument in the order given by the TIE file. `Out` and `inout` arguments will be treated as C++ reference parameters (or macro arguments), even if the input program is C. In other words, calling the intrinsic will change the value of the `out` arguments, without requiring you to apply the `&` operator to the argument Consider a TIE instruction, `incr`, that takes a single `inout` argument from the AR register file and increments it. The intrinsic `incr`, can be used as follows.

```
int a=2;
incr(a); // after this instruction, a will have the value 3
```

The included header file contains a comment for each `out` or `inout` argument Consider, for example, the `MULA18_0` (18-bit multiply add) instruction present in the ConnX Vectra LX coprocessor. A grep for the instruction in the header file is shown as follows.

```
extern void _TIE_xt_vectralx_MULA18_0(vec4x40 t /*inout*/, vec8x20 s, vec8x20 r);
#define MULA18_0 _TIE_xt_vectralx_MULA18_0
```

The `#define` in the second line shows that the intrinsic name, `MULA18_0`, is really an alias for the internal compiler name, `_TIE_xt_vectralx_MULA18_0`. The first line shows that the intrinsic returns no value, has an `inout` first argument of type `vec4x40`, used for the accumulator, and two in arguments of type `vec8x20` for the two multiplicands.

TIE instruction operands can be register operands or immediates. Register operands will appear in the intrinsic declaration with their associated C datatype. Immediate operands will appear in the declaration to be of type `immediate`. Intrinsics expecting immediates must be passed immediates with a value matching a legal immediate value of the instruction (unless the `specialized_op` property is used). Otherwise, the compiler will give an error.

TIE states, as opposed to register files, are not operands. An intrinsic call will implicitly use any state that is read or modified by the corresponding TIE instructions.

TIE allows the use of protos, corresponding to a sequence of instructions. For each proto, XCC will recognize an intrinsic with the same name as the proto. From the C programmer’s point of view, a proto is just like an instruction, except that XCC will generate a series of instructions from the intrinsic rather than just one.

5.2.2 TIE Datatypes

For instruction operands from the core AR register file, the corresponding C intrinsic expects an argument of a standard base C datatype: `signed` or `unsigned` versions of `char`, `short`, `int` and `long long`. These arguments can be consumed or produced by any C construct, either standard or another TIE intrinsic. For example, the expression `a+foo(b)` takes the result of the intrinsic `foo` and then adds it to `a` using the core addition instructions.

TIE also allows the creation of custom register files. Standard C variables will never live by default in the custom register files (although using various inference techniques, the XCC compiler might temporarily move them to custom register files). Cadence could have provided a system where each intrinsic explicitly specified which register of a custom register file to use. However, programming in that manner is not much easier than programming in assembly. Instead, TIE provides a mechanism for creating custom datatypes that live in the custom register files. The C programmer can declare variables of the custom type, and the compiler automatically loads and stores the variables as needed. The compiler automatically allocates the variables to the register file, and if there are not enough registers, then the compiler will automatically spill the registers as needed.

The use of TIE datatypes is best illustrated by example. Consider the ConnX Vectra LX coprocessor provided by Cadence. It supports a SIMD datatype holding eight, 16-bit elements, called vec8x16. It also supports a vec8x20 datatype with four extra guard bits of precision for intermediate calculations. It supports an instruction ADD20, to do an 8-way SIMD add and another instruction, RADD20, that adds together all the elements in a single vec8x20 variable. Below is an example C program that adds up all of the elements in an array. The function takes as input an array of vec8x16 elements and returns a short representing the sum of all the elements. This example uses a local variable sum, to hold all the temporary, vector results. Note that sum is initialized using the short value 0. The TIE developer has specified a way to automatically convert short values (via replication) into vec8x20 variables. While ADD20 and RADD20 are specified via intrinsics, the compiler knows how to automatically load a[i] into a vector register and how to schedule all the operations together in VLIW bundles. The inner loop of this example executes in one cycle per iteration.

```
#include <xtensa/tie/xt_vectralx.h>
short addmeup(vec8x16 *a)
{
    int i;
    vec8x20 sum;
    sum = (short) 0;
    for (i=0; i<100; i++) {
        sum = ADD20(sum, a[i]);
    }
    return RADD20(sum);
}
```

Figure 25: Vector Reduction Add

TIE register files are by default all caller saved. That means that every leaf function can assume that it may freely use any TIE register. If a variable is defined before a function call and used after a function call, the compiler will automatically spill the variable to memory before the call and load it from memory after the call, just in case the called function needs to use the register.The TIE developer can choose to make some of the registers in a TIE register file to be callee saved by using the callee_saved property in their TIE file. The compiler will try to use callee saved registers for variables defined before calls and used after calls, and the compiler will try to use caller saved registers for other variables.

Variables of one TIE type can only be converted to variables of another TIE type if the TIE developer has defined a direct conversion sequence from the first type to the second. Transitive conversions are not allowed. So, for example, if the TIE developer has defined a conversion from an int to a TIE type foo, the C programmer cannot convert from a short variable to a foo, even though there is a conversion from a short to an int. This rule is designed to avoid ambiguity that might lead to confusion. Given two types, there is only one conversion path. Use of unsupported conversions will lead to compile time errors.

5.2.3 Operator Overloading

When a TIE instruction has unusual semantics, the C programmer must specify its use through the use of an intrinsic. However, often a TIE instruction has semantics that are similar to or analogous to standard C/C++ operations working on custom datatypes. In such cases, rather than use intrinsics, it is nice to be able to use the standard C/C++ operators. TIE provides a mechanism to specify that a proto is equivalent to a standard operator when applied to TIE ctypes.

```
operator "symbol" proto_name
```

For example, to specify that the ADD20 intrinsic is equivalent to the operator "+" , use the following.

```
operator "+" ADD20
```

With the operator overloading predefined for ConnX Vectra LX, the example in *Figure 25: Vector Reduction Add* on page 154 can be replaced with the code in the example below. Note that there is no standard C operator for reduction adds and therefore RADD20 must still be invoked using an intrinsic. Note that unlike operator overloading in C++, the types given to the operator must exactly much the type of the proto. Implicit type conversions are not allowed as they might lead to ambiguities.

```
#include <xtensa/tie/xt_vectralx.h>
short addmeup(vec8x16 *a)
{
    int i;
    vec8x20 sum;
    sum = (short) 0;
    for (i=0; i<100; i++) {
        sum = sum + a[i];
    }
    return RADD20(sum);
}
```

Figure 26: Vector Reduction Add Using Operator Overloading

5.2.4 Structured ctypes

Normally ctypes are associated with a single entry of a register. However, it is sometimes useful for a single ctype to be stored in multiple registers. As an example, consider the multiply instructions in the ConnX Vectra LX coprocessor. ConnX Vectra LX register files are 160 bits and a register can hold either a `vec8x20` or a `vec4x40` ctype corresponding to eight, 20-bit elements or four, 40-bit elements. Typically, the 20-bit elements are used as operands to multiply instructions while the 40-bit elements are used to hold the results. Note that two multiply instructions are required to multiply two input vectors (one for the even elements and one for the odd) and note that the results of these two multiply instructions must live in two different registers. Nonetheless, from the programmer's point of view, it is often more convenient to deal with a single multiply intrinsic, specified in TIE with a single proto, where that multiply intrinsic returns a `vec8x80 ctype` that lives in two registers and requires two instructions to compute. To specify a structured ctype, augment the ctype declaration with a list of pairs corresponding to base types and element names as described with the following syntax.

```
ctype name size alignment regfile-name {base_ctype0 name0, base_ctype1 name1, ...}
```

To use a structured `ctype`, a proto needs a way to refer to the constituent elements. This is accomplished using the `->` syntax. For example, a proto with an argument `a` of type `ctype`, can refer to element `name0` of a using the syntax `a->name0`.

For the case of ConnX Vectra LX, `vec8x40` type and the `MUL18` proto that produces a vec8x40 type are defined as follows.

```
ctype vec8x40 320 128 vec { vec4x40 hi, vec4x40 lo }
proto MUL18 {out vec8x40 t, in vec8x20 s, in vec8x20 r} {} {
    MUL18.0    t->lo, s, r;
    MUL18.1    t->hi, s, r;
}
```

Structured ctypes are particularly useful together with operator overloading. There is no standard C/C++ operator for multiplying the even elements of a vector. The only way to use the `*` operator to multiply two, `vec8x20` types, is to have a `vec8x40` type for the result.

5.3 Display Format

All TIE datatypes and register file entries are displayed in the debugger as hex or decimal numbers. If, for example, a 33-bit TIE register file contains three, 11-bit SIMD elements, seeing such a variable in hex makes debugging difficult. Xplorer allows you to define custom display formats for TIE datatypes and registers so that, for example, the 33-bit register can be displayed as a parenthesized, comma separated list of the three elements. The custom format can be defined in Xplorer itself or default formats can be specified in the TIE file.

To create a format, use two properties.

```
property display_format_name name "format"
property display_format_map {state-name | reg-name | ctype-name} "format-type"
```

The first property associates a format name with a particular format. The syntax for the format is described in the Xplorer On-line Help pages. The second property statement associates a particular format with a register file, state or TIE ctype. The format-type is one of `mem_ctype`, `reg_ctype`, `state`, `user_register`.TIE ctypes are separated into memory and register format types because it’s possible that the binary representation in memory is different than the binary representation in a register file. Xplorer knows whether a variable is in memory or in a register file and will use the appropriate format.

As an example, consider a 32-bit ctype, `xr_16x2`, consisting of two, 16-bit SIMD elements. The following properties will cause variables of type `xr_16x2` to be displayed as parenthesized, comma-separated hex representations of the two constituent types. In this case, both the register format and the memory format are assumed to be identical.

```
ctype xr_16x2 32 32 XR
property display_format_name xr_16x2 "(%h[31:16], [15:0])"
property display_format_map {xt_16x2} xr_16x2 "mem_ctype"
property display_format_map {xt_16x2} xr_16x2 "reg_ctype"
```

5.4 Communicating between TIE files and C/C++ files

It may be useful to be able to share more arbitrary information between a TIE file and a C/C++ file. For example, one version of a TIE file might support 8-way SIMD operations while another version might support 16-way. It would be ideal if a single C/C++ application could be written that works with either version. That C/C++ file needs some standard way to know the vector length of the targetted TIE file. The header_include property in TIE allows the TIE developer to place user defined C/C++ code inside a header file that can be used by the application. The header file can either be the standard TIE header file, in which case the C/C++ code is placed at the end of the header file, or it can be any other named file.

```
property header_include include_file.h "#define VECTOR_LENGTH 16\n"
```

6. Memory Layout

Topics:

Cadence architectures provide great flexibility in designing your memory subsystem. You can create caches of many different types and sizes. You can also create local instruction or data memories that can be accessed by the processor in a single cycle. Given a memory subsystem design, the programmer needs a way to place different pieces of code and data at different addresses.

The system software developer may have complicated needs with regards to memory layout. Handlers must be placed at specific addresses; the memory space of devices must be reserved against general usage. These uses are beyond the scope of this guide and are instead covered in the *Xtensa Linker Support Packages (LSPs) Reference Manual*.

Sometimes you want to dynamically load an application into memory. If you are using an operating system, Linux, for example, the operating system may have sophisticated mechanisms for dynamically loading programs and sharing libraries across programs. For users running without an operating system, Cadence has tools to help, but they are also beyond the scope of this guide. These tools are covered in detail in the *Xtensa System Software Reference Manual*.

This guide focuses on the needs of application developer; how they place code or data in local IRAM or DRAM memory.

6.1 Using Local Memories Only

By default, your application will not use local memories. If your configuration contains no caches, every load, store, and instruction fetch will go to system memory and take many cycles. In the simplest case, you may want to use only local memories. There are several ways to achieve this goal.

First, if you are developing using the default, sim, or the min-rt LSP, linker support package, you can instead use the sim-local or min-rt-local LSP. For command-line users, simply link your application with the flag `-mlsp=sim-local` or `-mlsp=min-rt-local`. For Xplorer users, simply select the appropriate LSP from the Linker tab of the Xtensa Project Build Properties dialog box.

If you are using a different LSP, the easiest way on the command line is to create an entire new set of lsps that default to use local memories with the tool `xt-regenlsps` as follows:

```
xt-regenlsps -dstbase dir -mlocalmems
```

Once created, the new LSPs can be used on the command line using the flag `-mlsp=dir/lspname` or in Xplorer by selecting a custom LSP.

Alternatively, from Xplorer, right click on the configuration and you can select an option to create, or clone, a new configuration that defaults to using all local memories.

6.2 Using Local Memories for Some Code and Data

Most users do not want to put everything in local memories, only important code and data. For code that you are writing, a function or variable can be put into local memories using attributes. For example, to put a function `foo` into the single, local IRAM, declare it as follows:

```
extern void foo(void) __attribute__((section(".iram.text")));
```

Make sure that the declaration appears in the same file as the function definition and precedes the definition. For configurations with two local IRAMs, the predefined sections `.iram0.text` and `.iram1.text` are associated with each one respectively. The section `.iram.text` is an alias for `.iram0.text`.

Similarly, global variables can be placed into the local DRAM as follows:

```
int globvar __attribute__((section(".dram.data"))) = 0;
```

For configurations with two local DRAMs, the predefined sections `.dram0.data` and `.dram1.data` are associated with each one respectively. The section `.dram.data` is an alias for `.dram0.data`.

The compiler may create read-only data to support the compilation of a particular function. For example, the compiler may create a read-only table to implement a C switch statement. If a function is sufficiently important to place in IRAM, the generated jump table might be important enough to go in DRAM. The following attribute on a function declaration tells the compiler to both allocate the function in IRAM and any generated read only data in DRAM.

```
extern void foo(void)__attribute__((section(".iram.text"), rodata_section(".dram.rodata")));
```

Note that variables are not affected by the `rodata_section` attribute, only compiler generated data. Variables must be explicitly attributed.

Local variables and dynamically allocated variables cannot be allocated to memories using attributes; they always reside in the `stack` and `heap` respectively. The stack and heap can be moved to a local DRAM using the Memory Map editor in Xplorer or on the command line using the `xt-regenlsps` tool. See the *Xtensa Linker Support Packages (LSPs) Reference Manual* for details.

Sometimes you want to place code that you did not write into the IRAM or DRAM. A common example is all or part of the C library. This can be done using *xt-objcopy*.

For example, to place any code pulled from the C library into instruction RAM, place any associated literals (if using PC-relative literals) into data RAM, and leave other sections in their default location. You can do this by renaming sections within the C library as follows:

```
mkdir tmp
xt-objcopy --rename-section .text=.iram0.text \
    --rename-section .literal=.dram0.literal \
    <xtensa_root>/xtensa-elf/lib/libc.a tmp/libc.a
```

where *`<xtensa_root>`* is the location of your Xtensa core package. Telling the linker to use an alternate version of a library or object file specified by the LSP `specs` file, such as the C library, is best done using the `-L <searchdir>` flag to XCC. This tells the linker to look in the specified directory ahead of its standard search path. Place your alternate library in a separate directory free of other unwanted files, and add that directory to the linker search path, for example, `-L tmp` for the above code.

The link order tool described in *Memory System* on page 27 allows an automated way to map the most important functions, based on profile information, into the IRAM.

7. Devices and Synchronization

Topics:

The C/C++ programming model mostly assumes that everything your program does is expressed directly in C/C++ or indirectly via C libraries. In reality, many programs need to communicate with external devices or with other processors. When communicating with the outside world, the programmer needs control to ensure that memory references communicating with the outside world actually happen, that memory references happen in the intended order and that multiple processors are able to atomically read or write multiple blocks of memory. Such control can be difficult to achieve when compilers reorder memory references or delete unused variables, when processors cache data without any attempt to keep multiple caches coherent and when multiple processors can attempt to write to the same locations in arbitrary orders. While the C/C++ language gives a little help in the form of *volatile* attributes, that help is coarse and incomplete. Instead, the programmer must go outsides the bounds of the language when extra control is needed.

7.1 Shared Memory Programming

Cadence provides a library, XMP, for programming multiprocessors in a shared-memory environment. It includes a methodology for sharing data among processors and a library for synchronization. Many of the issues discussed in this chapter are handled automatically by the library. Documentation for the library can be found in the *Xtensa System Software Reference Manual*.

7.2 Memory Ordering

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

This section assumes that the memory locations being used for communications are uncached. Caches add an additional level of complexity that is discussed in *Cache Coherence* on page 168.

Programmers often try to guarantee sequential semantics by using the `volatile` attribute. This is different from using `volatile` to guarantee that a memory reference to a memory mapped device with side effects is not deleted. Unfortunately, the use of `volatile` for this purpose is inefficient and potentially dangerous. The C and C++ standards guarantee that two volatile references are not reordered with respect to each other, but they do not guarantee that a volatile reference is not reordered with respect to a non-volatile one. Thus, to be safe, both flags and data must be marked as `volatile`, but doing so can be inefficient. Because `volatile` can be used for memory consistency, the compiler is forced to be overly conservative when a strict ordering is not required but volatile is used for devices with side effects. The Xtensa hardware may reorder memory references unless separated with a MEMW or other synchronization instruction. XCC separates all volatile references with a MEMW instruction by default, even though this is usually unnecessary for devices with side effects. The flag `-mno-serialize-volatile` instructs XCC to omit the MEMW instructions.

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

```
#pragma flush_memory
```

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

Consider the example below where one processor is signaling to another that data has been produced while the other is spinning waiting for the flag to be set.

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

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

Figure 27: Communicating with Flags

All the data is guaranteed to be written by the first processor before the flag, `flag_data_ready`, is written. None of the data consumed by the second processor is consumed until the second processor sees that the flag has been written.

For examples with a single flag variable used on configurations with the Synchronize Instruction option, Cadence offers two instructions, S32RI and L32AI, that allow you to more efficiently combine the effects of storing or loading the flag variable together with the MEMW ordering guarantees in a single instruction. For example, the flag setting example above can be replaced with the code in this example:

```
#include <xtensa/tie/xt_sync.h>
...
for (i=0; i<n; i++) {
    data[i] = ...
}
XT_S32RI(1,&flag_data_ready,0);
```

```
#include <xtensa/tie/xt_sync.h>
...
while (!XT_L32AI(&flag_data_ready,0)) {
    ;
}
for (i=0; i<n; i++) {
    ... = data[i];
}
```

Figure 28: S32RI and L32AI

The semantics of the *S32RI* instruction ensure that any prior load or store instructions are externally visible before the value being stored by the *S32RI* becomes visible. Similarly when testing a flag variable, the *L32AI* load instruction can load the flag variable and guarantee that the flag is loaded before any subsequent memory references.

7.2.1 TIE Ports

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

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

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

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

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

TIE Ports and Deadlock

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

Consider a simple two processor system as shown in *Figure 29: TIE Ports and Deadlock* on page 167. Each processor is sending data to the other via a small queue The first processor writes into a queue and then tries to get a response back from the second queue. The second processor reads data from the first queue and then sends back a response to the second queue. If the compiler for the first processor rearranges the references so that the first processor tries to read its input queue before it writes its output queue, the read will cause the processor to block while waiting for data before it has a chance to write its output data. The system will deadlock because the second processor will never write its queue until it receives its data from the first processor.

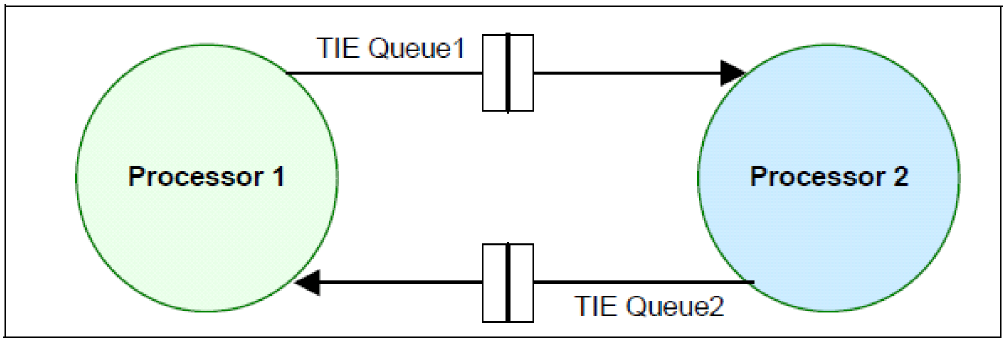


Figure 29: TIE Ports and Deadlock

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

7.3 Cache Coherence

Cadence processors optionally support caches as an easy to use mechanism to improve the memory performance of an application. With caches, memory being accessed is automatically copied into close, fast cache memory, and is only copied back into system memory whenever different memory values require the same locations in the cache.

Devices that are not performance critical or that are accessed through narrow interfaces are often mapped into uncached memory regions. However, every uncached write generates a bus transaction and every uncached load requires at least six cycles and also generates a bus transaction. Therefore, performance critical devices that access blocks of data are typically either cached or DMA'd into and out of local memories.

Caches on multiprocessor systems without hardware cache coherence complicate the memory consistency issues discussed in *Memory Ordering* on page 164. First, caches, at least write-back caches, delay stores an arbitrary amount of time. Second, loads may read a memory value from a local cache even if some other processor or device has already modified the global value. Therefore, on multiprocessor systems, caches may be inconsistent with each other. Consider again the examples in *Figure 27: Communicating with Flags* on page 165 and in *Figure 28: S32RI and L32AI* on page 165. Let us now assume that the variable *flag* and the array *data* are in cached memory. When the first processor writes the flag, the flag and/or some of the data might stay in the cache for an arbitrary amount of time. Therefore, the second processor might wait forever for the flag to be set or worse, it might read a mixture of old and new data if some but not all of the data written by the first processor has been sent back to main memory. Similarly, when the second processor is spinning, waiting for the flag to be set, if the flag is not set on the first iteration, it will never be set because the cache of the second processor will not get updated. Perhaps worse, if some but not all of the old data remains in the second processor’s cache, the second processor will get a mixture of old and new data. To be safe, the first processor must force the data and the flag to be written back to main memory and the second processor must invalidate the flag and the data before reading them.

The XMP library provides macros, *XMP_INVALIDATE_ELEMENT* and *XMP_WRITEBACK_ELEMENT*, to writeback and invalidate variables. Alternatively, the Cadence HAL (Hardware Abstraction Layer) software provides functions. In fact, the XMP macros call the HAL functions. The function *xthal_dcache_line_writeback(void *address)* ensures that data is sent back from the cache to system memory. The function *xthal_dcache_line_invalidate(void *address)* ensures that the data at a particular address is deleted from the cache. When writing or reading a large amount of data, the functions *xthal_dcache_region_writeback(void *address, unsigned size)* and *xthal_dcache_region_invalidate(void *address, unsigned size)* can be used to writeback or invalidate multiple addresses with a single function call. Further details about the HAL functions are in the *Xtensa System Software Reference Manual*. The example in *Figure 27: Communicating with Flags* on page 165 is rewritten to handled cached flags and data below.

```
#include <xtensa/config/core.h>
...
for (i=0; i<n; i++) {
    data[i] = ...
}
xthal_dcache_region_writeback(data,size);
flag_data_ready = 1;
xthal_dcache_line_writeback(&flag_data_ready);
```

```
#include <xtensa/config/core.h>
...
xthal_dcache_line_invalidate(&flag_data_ready);
while (!flag_data_ready) {
    xthal_dcache_line_invalidate(&flag_data_ready);
}
xthal_dcache_region_invalidate(data, size);
for (i=0; i<n; i++) {
    ... = data[i];
}
```

Figure 30: Coherent Use of Flags

An additional problem occurs because data is written to and from cache in units of lines that are anywhere from 16 to 256 bytes. The compiler will place unrelated variables in the same cache line. Imagine a situation where one processor is writing one flag while another processor is writing a different flag, but both flags are next to each other in memory and share a cache line. If each processor sets its respective flag at the same time, the cache of each processor will have its flag set but not the other. Then depending on which cache line gets written back to memory last, only one of the flags will remain set. To avoid such scenarios, you must ensure that a variable being communicated not be shared with any unrelated variables. The easiest way to do that is to both align the variable to the size of the cache line and make sure that the size of the variable is a multiple of the cache line size. Variables can be aligned to 64 bytes, for example, using attributes as follows:

```
int flag __attribute__ ((aligned (XCHAL_DCACHE_LINESIZE)));
```

7.4 Volatile Devices

In many cases, you may have a device in uncached memory that is accessed through a load or store to a single, memory mapped location. You may not care if loads or stores to the device are reordered with respect to other memory references. However, you do care that the device is accessed exactly as many times as specified in the original C/C++ code. You do not want the compiler to move a load from a memory mapped hardware queue outside of a loop, for example, because it appears to the compiler that successive loads must all be loading the same value. For such cases, the use of volatile is appropriate. Note however that the compiler will insert `MEMW` instructions around all volatile references because the compiler does not know that you do not care about memory ordering. The flag `-mno-serialize-volatile` instructs XCC to omit the `MEMW` instructions.

Index