HiFi 5 DSP User's Guide

For Xtensa LX Processor Configurations

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

For further assistance, contact Cadence Online Support at https://support.cadence.com/.

Product Release: RI-2022.9

Last Updated: 06/2022

Modification: 749531

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

Contents

List of Tables

List of Figures

Changes from the Previous Version

The following changes were made to this document for the Cadence Tensilica RI-2022.9 release:

- Updated *Introduction* on page 13 to reflect changes done in HiFi 5 hardware for RI-2022.9 release.
- Updated *HiFi 5 Architecture Overview* with some additional details
- Updated *VLIW Slots and Formats* on page 45 with new FLIX formats added to improve code size.
- Updated *Data Types* on page 51 with support added for new data types.
- Updated *Store Operations* on page 94 by adding new store variants.
- Updated *Load Operations* on page 93 by adding new load variants.
- Updated *Configuring a HiFi 5 DSP* with details about new sigmoid/tanh config option.
- Added a new section *New ISA Added in RI-2022.9*.
- Added a new section *ISA Enhancements to Support Activation Functions* on page 130 to describe the following topics:
    - *Sigmoid and Tanh Functions*
    - *Activation Function Specific Operations*

The following changes were made to this document for the Cadence Tensilica RI-2021.8 release.

- Updated *Optional Configuration Templates for HiFi 5 DSP*
- Updated *Extending HiFi 5 DSP with User TIE*

The following changes were made to this document for the Cadence Tensilica RI-2021.6 release.

- *Introduction* on page 13 updated to reflect changes done in HiFi 5 hardware for RI-2021.6 release.
- *Programming the DSP* on page 49 updated at few places to replace reference to “xcc compiler” by “Xtensa C/C++ compiler”
- *Store Operations* on page 94 updated with new store operations introduced in HiFi 5 hardware for RI-2021.6 release
- *Multiply and Accumulate Operations Overview* on page 96, *Neural Networks Multiplication Operations* on page 106, and *Neural Network (NN) Examples* on page 153 updated with new NN MAC operations introduced in HiFi 5 hardware for RI-2021.6 release
- *Optional Floating Point Unit Operations* on page 125 updated with description of 3-cycle FMA based changes done in the HiFi 5 hardware for RI-2021.6 release

The following changes were made to this document for the Cadence Tensilica RI-2020.5 release.

- The trademark page was updated

1. Introduction

Topics:

- *Purpose of this User's Guide*
- *Installation Overview*
- *HiFi 5 DSP Architecture Overview*
- *HiFi 5 DSP Prefetching*
- *HiFi 5 DSP Instruction Set Overview*
- *Upgrading the Previous HiFi DSPs*

The Cadence HiFi 5 DSP is a high-performance embedded digital signal processor (DSP) optimized for front-end, far-field and near-field audio and voice processing. It is also designed for enabling efficient implementations of neural network (NN) based speech recognition algorithms.

The HiFi 5 DSP is a five-slot VLIW machine which can execute up to eight 32x32-bit MACs, sixteen 32x16-bit MACs, sixteen 16x16-bit MACs per cycle. It can issue two 128-bit loads per cycle, or one load and one store of 128-bit per cycle for parallel loads and stores of the operand and results. The HiFi 5 DSP offers additional floating-point precision support for enhanced audio and voice processing through an optional Single Precision vector floating-point unit (SP FPU), which can perform eight single-precision IEEE-754 floating-point MACs per cycle. Because the front-end audio and voice processing is done in frequency domain, both the floating-point and fixed-point MAC operations in the HiFi 5 DSP are enhanced to operate on complex data types.

For supporting neural network-based speech recognition algorithms, the HiFi 5 DSP provides a Neural Network Extension option that enables the hardware to perform up to thirty-two 8x16, 4x16, and 8x8-bit MACs per cycle. The multiplication operations support both signed and unsigned operands, as well as operands with 4-bit precision. Some speech neural network implementations also use half-precision floating-point variables. For such networks, the HiFi 5 DSP includes a Half Precision floating point unit (HP FPU) option that provides up to sixteen half-precision IEEE-754 floating-point MACs per cycle. The instruction set is designed to address both dot products and convolution operations to cover several types of neural network implementations used in the speech recognition.

Starting with the RI-2022.9 Xtensa release, HiFi 5 DSP instruction set has been upgraded with a set of new FLIX formats for better code size, and ISA enhancements to improve performance for various NN kernels such as 4X8-

bit kernels, rounding methods used in TFLM operators and various NN applications. HiFi 5 DSP vector floating point unit (FPU) is also upgraded to improve the SP FPU performance for Eigen library with better SIMD float and complex float data type support. The programming model for HiFi 5 DSP hardware is improved to have better Out-of-Box (OOB) cycle performance. Reference C code developed using standard C integer and float data types and code using basic operators like STL 2019 BASOPs will be benefited. With this ISA enhancements, the HiFi 5 core is not binary compatible with the HiFi 5 DSP library/executable built using the previous HW version. It will be backward compatible at the C/C++ source level for the programs optimized using HiFi 4, HiFi 3z, HiFi 3, HiFi Mini and HiFi 2/EP intrinsics and also with HiFi 5 DSP code developed for earlier releases. Source code compatibility for the legacy codes developed on previous HiFi 5 DSP hardware shall be maintained. You can differentiate the code written for newer HiFi 5 DSP hardware by using the compile time switch `XCHAL_HW_VERSION>= XTENSA_HWVERSION_RI_2022_9` in the source code.

With the RI-2021.6 Xtensa tool chain release, HiFi 5 NN extension ISA was upgraded with a set of new instructions for the Asymmetric int8 quantization support and the depth-wise separable convolution. HiFi 5 vector floating point unit (FPU) is also upgraded to improve the SP FPU and HP FPU performance with 3 cycle latency Fused Multiply Add (FMA) hardware. Even with this ISA upgrade, the HiFi 5 core is still binary compatible with the HiFi 5 library/executable built using previous release of HW version. It will be backward compatible at the C/C++ source level for the programs optimized using HiFi 4, HiFi 3z, HiFi 3, HiFi Mini and HiFi 2/EP intrinsics and also with HiFi 5 code developed for earlier releases. Source code compatibility for the legacy codes developed on previous HiFi 5 hardware shall be maintained. You can differentiate the code written for newer HiFi 5 hardware by using the `XCHAL_HW_VERSION >= XTENSA_HWVERSION_RI_2021_6` conditional check in code. *Code Portability from Previous HiFi DSPs* on page 77 provides the details related to porting such software.

The HiFi 5 DSP is a configuration option that can be included with the Xtensa LX7 (and later versions) processor. All the HiFi 5 DSP operations can be used as intrinsics in the standard C/C++ applications. In addition, when compiling with automatic vectorization or with the `-mcoproc` option, the compiler will automatically use the HiFi 5 DSP operations when compiling the standard C code.

1.1 Purpose of this User's Guide

This guide provides an overview of the HiFi 5 DSP architecture and its instruction set. It will help programmers using HiFi 5 DSP by identifying some of the techniques that are commonly used to optimize algorithms. It provides guidelines to achieve high performance by using HiFi 5 DSP's instructions, intrinsics, protos, and primitives. This guide also serves as a C and C++ usage reference for the appropriate way to use HiFi 5 DSP features in C/C++ software development. This guide will also assist Xtensa HiFi 5 DSP users who wish to add additional instructions to the HiFi 5 DSP architecture.

To use this guide most effectively, a basic level of familiarity with the Xtensa software development flow is highly recommended. For more details, see the *Xtensa Software Development Toolkit User's Guide*.

Conventions

In this document, the symbol <*xtensa_root*> refers to the installation directory of a user's Xtensa configuration.

For example, <*xtensa_root*> refers to the directory *\usr\xtensa\XtDevTools\install\builds \RI-2018.0-win32\<s1>* if *<s1>* is the name of your Xtensa configuration.

In the examples in this guide, replace <*xtensa_root*> with the installation directory of your Xtensa distribution

1.2 Installation Overview

To install a HiFi 5 DSP configuration, follow the same procedures described in the *Xtensa Development Tools Installation Guide*. The HiFi 5 DSP header files are in the following directories and files:

<xtensa_root>/xtensa-elf/arch/include/xtensa/config/defs.h

<xtensa_root>/xtensa-elf/arch/include/xtensa/tie/xt_hifi5.h

For easier migration of existing HiFi codes, you can use any of xt_hifi2.h, xt_hifi3.h, or xt_hifi4.h.

1.3 HiFi 5 DSP Architecture Overview

The HiFi 5 DSP is a single-instruction/multiple-data (SIMD) processor and can process up to four 32-bit data items or eight 16-bit data elements in parallel. It is a VLIW architecture, supporting the execution of up to five operations in parallel.

The HiFi 5 DSP contains thirty-two 64-bit AE_DR registers and sixteen AR registers. Each AE_DR register typically contains either two 32-bit operands, four 16-bit operands or eight 8-

bit operands. The operands can be interpreted as either integer, fixed-point or floating-point values. AR registers typically contain normal short, integer values used in baseline Xtensa operations or they contain pointers, counters, offsets, and shift values used in the source code. The AR and AE_DR register sets provide operands and collect results from the core HiFi 5 VLIW based compute engine.

This paragraph gives a brief description of the performance and functioning of several operations in various slots.

Slot 0 operations of the HiFi 5 DSP operation-bundles include scalar operations, vector and scalar load, store, bitstream and Huffman operations. In addition, a few ALU, select and shift operations are also available in slot 0.

Slot 1 operations have similar but fewer set of operations as in slot 0. Also, slot 1 does not allow any store operations.

Slot 2 and 3 contain Vector MAC, ALU, and optional floating point operations. Slot 2 also contains the special MAC operations used in the Neural Network Extension package.

Slot 4 has a limited set of operations that are useful in specific scenarios like FFT processing.

Using the above resources, the HiFi 5 DSP can perform up to four 32-bit and eight 16-bit ALU operations in parallel. Additionally the HiFi 5 DSP can perform up to eight 32-bit MAC operations per cycle or sixteen 16x16-bit or 32x16-bit multiplications per cycle.

The neural network extensions enable thirty-two 8x8, 8x16, 4x16 MAC per cycle.

The floating point optional units also support up to eight floating point MACs per cycle and 16 half-precision floating point MACs per cycle.

The HiFi 5 DSP operations are encoded in several different VLIW operation formats. Most of these formats are encoded in 128 bits and support up to five slots. A couple of formats with operations bundled in two or three slots are encoded in 64-bit wide operation words.

The HiFi 5 DSP supports either caches or local memories with the full flexibility provided by Xtensa configurations. The programmer can select either or both cache and local memory for storing operations and data.

The audio software packages supplied by Cadence do not use the DMA. Hence, most customers either use caches or make local memories sufficiently large to cover desired applications. The NN implementations that use a large acoustic model can decide to use local memories. The programmer can decide to use the iDMA for transferring data and weights to and from the internal memories to system DRAM. The HiFi 5 DSP can only be configured to use little-endian byte ordering for loading or storing different values residing in the memory. The following diagram illustrates the register, load/store units, and execution units in different slots added to an Xtensa LX processor based HiFi 5 DSP architecture.

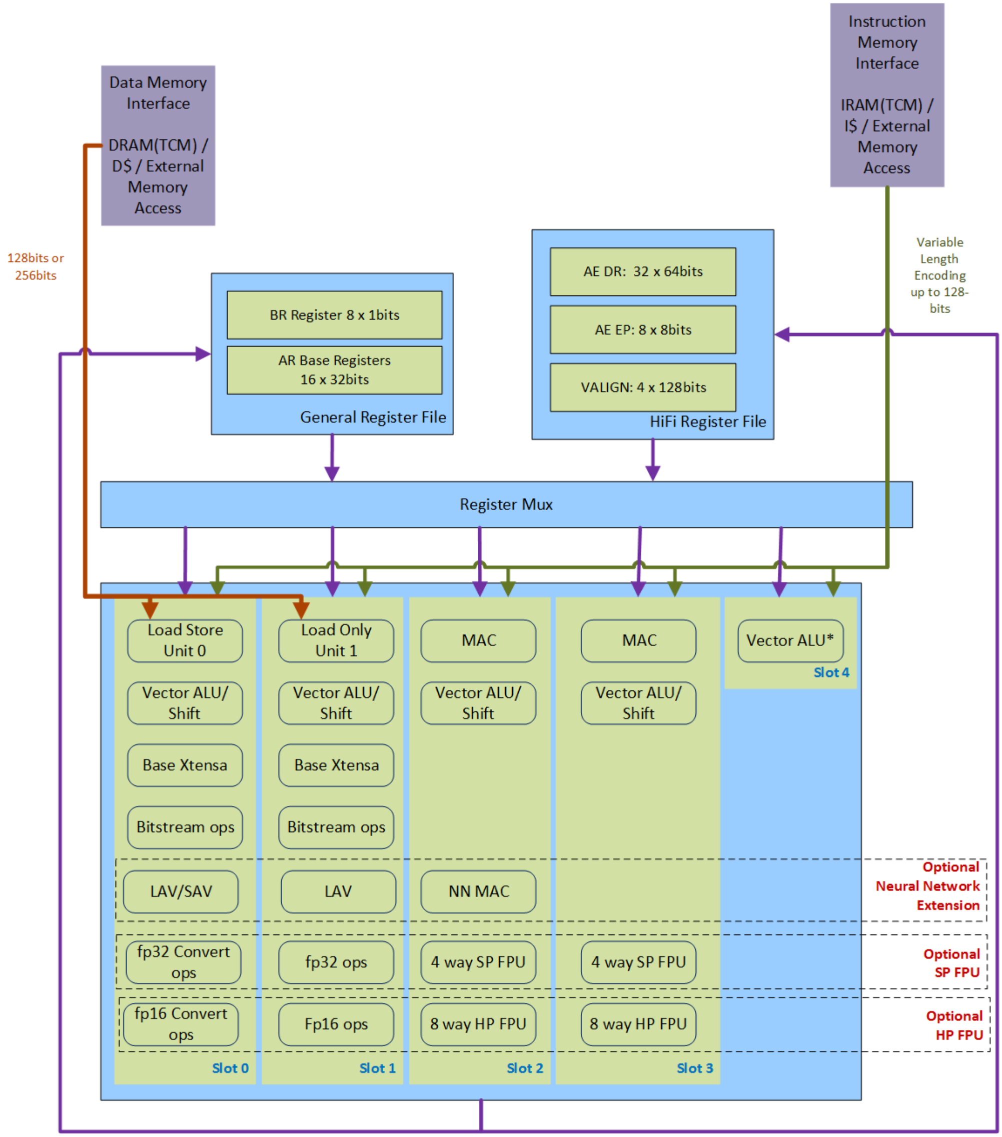


Figure 1: HiFi 5 DSP Architecture

The HiFi 5 operations can be issued in one of the several VLIW formats. Most of the formats are encoded in 128 bits and support up to five slots. The formats that are encoded in 64-bits can support three or two slots in each execution. There is a five slot format that allows few carefully selected operations to be executed in the fifth slot, to improve the efficiency of FFT and few other min/max finding algorithms. The operation from each slot can be executed independently or in parallel according to the static bundling expressed in the machine code. For example, two load operations can execute concurrently with two multiply/accumulate operations because loads are in slots 0 and 1, while multiply/accumulate operations are in slots 2 and 3.

Following are the main hardware resources in the HiFi 5 DSP subsystem:

- Two load units
- Store unit
- Multiply/accumulate units
- Arithmetic/logic units
- Shift units
- A 32-entry 64-bit register, AE_DR
- A 4 entry 128-bit VALIGN register
- A 4-entry register AE_EP to hold eight extra guard bits for the AE_DR registers.

The load/store unit is capable of loading or storing up to two 64 bit elements, four 32-bit SIMD elements, eight 16-bit SIMD elements or sixteen 8-bit SIMD elements. The load/store unit supports unaligned accesses, whereby a stream is first primed and then 64 or 128 unaligned bits can be loaded or stored in every cycle. The ability of dual issuing the load operations, or issuing a load and store operation in parallel enhances the per-cycle load/store throughput to 256-bits per cycle for all the standard precisions. 24-bit load/store is supported for legacy purposes. If 24-bit data is contained inside 32-bit envelopes, up to eight 24-bit elements can be loaded in single cycle. If the 24-bit data is packed together into 24-bits of memory, Eight packed elements can be loaded or stored in three operations.

If optional Single Precision Floating Point Unit (SP FPU) or Half Precision Floating Point Unit (HP FPU) is selected, it brings the resources for the floating point operations. Standard number format operations (used to convert the numbers between integer and float formats) are available in slot 0, while the resource for floating point maths (such as multiply, add, and sub) are available on slot 2 and slot 3. These operations can be dual issued. HP FPU also includes operations/resource for reduction maths, matrix multiplication and convolution.

If Neural Network Extension is selected, It brings low precision 8x16, 8x8, and 4x16 MAC and variable aligning load and store operations. These special MAC operations go into the third slot of a three slot format and use separate power efficient NN multiplier multiply/accumulate unit to achieve 32 MACs per cycle throughput. The variable aligning load/store operations are helpful to load/store partial elements into 16x4 and 8x8 vectors.

1.4 HiFi 5 DSP Prefetching

The HiFi 5 DSP includes a prefetch option geared for systems with long memory latency. When the HiFi 5 DSP detects a positive stride-1 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 on the data side optionally into the L1 data cache. There is no support for prefetching directly into the L1 instruction cache. In addition, you can manually issue prefetch instructions.

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 increase 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_long(unsigned long long mode);
```

The value returned is not meant for direct use or interpretation; however, it is 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 Unit Option chapter in the *Xtensa LX7 Microprocessor Data Book* for details.
- A bit-wise OR of two cache prefetch mode constants, 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 Unit Option chapter in the *Xtensa LX 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.

Software Prefetching

Prefetching can also be individually controlled via software. On configurations that support block prefetch, the following macros are supported:

```
#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 mark a set of lines as being present in the cache without actually fetching the data from memory. The `_modify` macros, which are equivalent to writing random values into memory, 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 issues 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.

Other block prefetch macros are also available, but are less useful for application level code. See the *Xtensa LX7 Microprocessor Data Book* or the *Xtensa System Software Reference Manual* for more details.

Single cache lines can also be prefetched using the following GCC extension. This extension is supported on all configurations that support prefetching and not just ones that support block prefetching.

```
__builtin_prefetch(addr);
```

1.5 HiFi 5 DSP Instruction Set Overview

The HiFi 5 DSP is built on the baseline Xtensa RISC architecture, which implements a rich set of generic scalar instructions optimized for efficient embedded processing. The power of the HiFi 5 DSP comes from a comprehensive DSP and audio instruction set.

A wide variety of load/store operations support multiple addressing modes, with support for 8/16/32/64-bit scalar and 8/16/24/32/64-bit vectors. Vector data management is supported with select, convert, and shifting operations. A set of bitstream and variable length instructions allows for efficient access of serial bitstreams, including Huffman encode and decode.

The HiFi 5 DSP enhances the multiplication operation for all precisions. The HiFi 5 DSP is specifically enhanced for performing MAC and ALU operations on complex and complex-conjugate operands which would be useful in the implementation of frequency-domain adaptive-signal processing algorithms. Multiply operations from base core include 32x32-bit, 32x16-bit, and 16x16-bit. Multiply operations come in fixed-point and integer variants. They come in high precision and low precision variants. The high precision multipliers come with a wider accumulator than the operands. Both low precision and high precision multipliers can perform two, four, or eight independent SIMD multiplications.

The HiFi 5 DSP supports dual/quad multiplies, where the results of two/four multiplies are added or subtracted together before being added into the accumulator.

The optional single precision floating point unit supports two 2-way or 4-way SIMD units of IEEE-754 single- precision floating point operations. Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for more details about the core single precision floating point support, on which the 2-way or 4-way SIMD units are based. The dot product operations from Neural Network Extension supports 16x8-bit, 16x4-bit and 8x8-bit operations.

The Neural Network Extension also supports octal multiplications, where the results of the eight multiplications are added together before they are added to the accumulator.

The optional half precision floating point unit supports two 8-way SIMD units of IEEE-754 half precision floating point operations. The 8-way operations are specially designed for matrix vector multiplications and convolution multiplication. Along with the 8-way SIMD multiply operations, standard half precision ALU, MUL, convert etc. instructions are also supported.

1.6 Upgrading the Previous HiFi DSPs

This section lists the ways of upgrading the previous HiFi DSPs to the HiFi 5 DSP.

Upgrading the HiFi 2, HiFi EP, HiFi Mini, HiFi 3, HiFi 3z and HiFi 4 DSPs

The HiFi 5 DSP is backward compatible at the C/C++ source level with the HiFi 4 DSP. The HiFi 5 DSP is also backward compatible at the C/C++ source level with the HiFi 2, HiFi EP, HiFi Mini, HiFi 3, and HiFi 3z DSPs with some exceptions.

Any algorithm written in C/C++, including all HiFi packages from Cadence, can simply be recompiled on the HiFi 5 DSP to get the same performance improvements. Due to difference in the data-memory-bus-width, some applications may show slight degradation, which needs to be tuned on the HiFi 5 DSP. A few applications will need some minor modifications optimized using intrinsics.

Refer to the *Porting Code to the HiFi 5 DSP* on page 77 for more details.

2. HiFi 5 DSP Features Overview

Topics:

This chapter gives an overview of the HiFi 5 DSP features, instruction naming conventions, fixed point values and fixed point arithmetic, VLIW slots and formats, operation descriptions, and optional floating point units and optional Neural Network Extension.

2.1 HiFi 5 DSP Features

The HiFi 5 DSP contains a 32-entry, 64-bit register, AE_DR.

Each register can hold:

- One or two, 24- or 32-bit operands,
- One or four 16-bit operands,
- One or eight 8-bit operands or
- One 56- or 64-bit operand

as shown in the *Figure 2: AE_DR Register* on page 26. The 24-bit and 56-bit operands are sign extended to fill their 32- or 64-bit containers. The separate halves or quarters of the register are always separate data items. For example, if you shift a 32-bit element to the left, the L element does not spill over into the high element. When a vector is stored in a 64-bit register, the separate data items in the vector are called elements. When a 32-bit, 16-bit or 8-bit scalar value is stored, it is the same as holding the same value in all the elements of the vector.

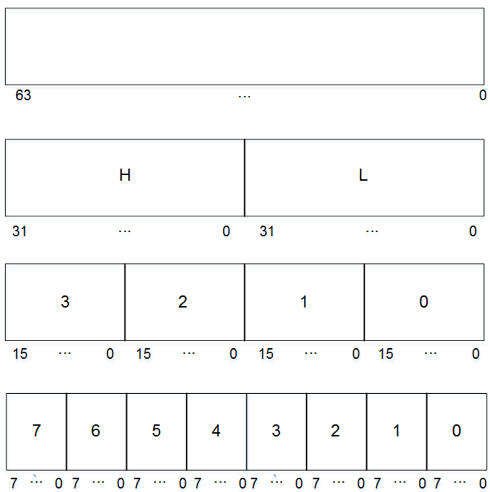


Figure 2: AE_DR Register

When a register is stored to memory, the higher half of the register is always stored in and loaded from the lower memory address. Operations that access individual 24- or 32-bit elements of AE_DR registers refer to the elements with selectors L and H in the mnemonics. Operations that access individual 16-bit elements refer to the elements with sectors 3, 2, 1, and 0 in the mnemonics. Operations that access individual 8-bit elements refer to the elements with sectors 7, 6, 5, 4, 3, 2, 1 and 0 in the mnemonics.

For legacy HiFi 2/EP and HiFi Mini instructions, a 32-bit data item might occupy the middle of an entire AE_DR register and a 16-bit data item might occupy the middle of a 32-bit half register. When using such legacy instructions, a register holds half as many elements; hence the instruction exploits less parallelism. Such instructions should only be used in legacy code.

HiFi 5 DSP supports a 4-entry, 8-bit extra precision register, AE_EP. This register is used in conjunction with some multiply, add, shift, and saturate instructions to provide a total of 72 bits of precision. This register is accessed one cycle later than the AE_DR register. Accumulations can still happen every cycle, but there is an extra cycle of delay when trying to shift or saturate the extra accumulator back down to 64 bits.

The HiFi 5 DSP supports a 4-entry, 128-bit alignment register, AE_VALIGN. This register can be used either in a 64-bit mode (ae_valign) or a 128-bit mode (ae_validnx2). Using this register in the 128-bit mode allows the hardware to load or store a SIMD stream that is not 128-bit aligned at a rate of 128 bits per cycle. Using this register in a 64-bit mode allows the hardware to load or store a SIMD stream that is not 64-bit aligned at a rate of 64 bits per cycle.

The aligning loads are present on both slot 0 and slot 1 and the aligning stores are present only on slot 0. It is hence possible to dual issue the loads or issue loads and store in parallel. In this scenario, both the aligning operations should read from/write into different streams. The aligning load store instructions can also be issued in parallel with normal loads and stores. The aligning mechanism also allows 24-bit data to be packed densely into 24-bit containers. These mechanisms are described in more detail in *Load and Store Operations Overview* on page 83.

The TIE state registers in the DSP are shown in *Table 1: DSP Subsystem State Registers* on page 27.

Table 1: DSP Subsystem State Registers

State Register	Bit Size	Description
AE_OVERFLOW	1	Indicates whether any arithmetic operation has saturated since the time when AE_OVERFLOW was last reset to zero.
AE_SAR	14	Contains the shift amount for various DSP shift operations. For 24 and 32-bit shifts, contains a vector of two shift amounts. This also holds the number of minimum headroom bits computed by dynamic range detection instruction AE_CALCRNG32, useful for algorithms such as FFT.

The following state registers pertain to the bitstream and variable-length encode/decode support subsystem of the DSP. Programmers generally will not need to worry about the details of how each of these state registers is used by the instructions, but for completeness the state registers (understandable for those familiar with the variable-length encode/decode instructions) are documented in *Table 2: Bitstream and Variable-length Encode/Decode Support Subsystem State Registers* on page 28.

Table 2: Bitstream and Variable-length Encode/Decode Support Subsystem State Registers

State Register	Bit Size	Description
AE_BITHEAD	32	Contains the bits at the head of the bitstream. The high half has the current 16 bits and the low half has the next 16 bits. Only the high half is used for output bitstreams.
AE_BITPTR	4	Offset within the 16 most-significant bits of the bitstream head. For an input bitstream, this value signifies the number of most significant bits of AE_BITHEAD that have been consumed already by the application. For an output bitstream, this value signifies the number of most significant bits of AE_BITHEAD that have already been initialized.
AE_BITSUSED	4	Contains the number of bits consumed or produced in the last table lookup by a variable-length encode/decode instruction. This value is coded in binary, with the exception that all zeroes are interpreted as the value 16.
AE_TABLESIZE	4	Contains one less than the base-2 logarithm of the current decoding table size for variable-length decode. 0 corresponds to a 2-entry table; 15 corresponds to a 65536-entry table.
AE_FIRST_TS	4	Contains the correct value of AE_TABLESIZE for the first level in the lookup-table hierarchy. This state is an optimization so that no AE_VLDSHT instruction is needed between consecutive decoding operations using the same codebook.
AE_NEXTOFFSET	27	This state is used for three different things. • In variable-length decode: Before an AE_VLDL16T or AE_VLDL32T instruction, AE_NEXTOFSET is the index of the table entry corresponding to the current bitstream prefix to look up. • After an AE_VLDL16T or AE_VLDL32T instruction, AE_NEXTOFFSET is the offset of the base of the next decoding lookup table. • In variable-length encode: After an AE_VLEL16T or AE_VLEL32T instruction, the low bits of AE_NEXTOFFSET hold the codeword bits produced by the most recent lookup.
AE_SEARCHDONE	1	This state tells the AE_VLDL16C instruction to prepare AE_NEXTOFFSET (using AE_FIRST_TS) for a fresh decoding search starting with the first table in the decoding hierarchy. This state is an optimization so that no AE_VLDSHT instruction is needed between consecutive decoding operations using the same codebook.

The state registers shown in *Table 3: Circular Buffer Support State Registers* on page 29 pertain to the circular buffer support and are shared between the DSP subsystem and the bitstream and variable-length encode/decode support subsystem of the DSP.

Table 3: Circular Buffer Support State Registers

State Register	Bit Size	Description
AE_CBEGIN0	32	Contains the start address of the circular buffer.
AE_CEND0	32	Contains the end address of the circular buffer.
AE_CBEGIN1	32	Contains the start address of the circular buffer.
AE_CEND1	32	Contains the end address of the circular buffer.
AE_CBEGIN2	32	Contains the start address of the circular buffer.
AE_CEND2	32	Contains the end address of the circular buffer.
AE_CWRAP	1	Indicates whether any circular buffer operation has wrapped around since the time when AE_CWRAP was last reset to zero. Set only for bitstream instructions.

The state registers shown in *Table 4: Zero Bias State Registers* on page 29 are used in multiplication operations in which operands are in quantized 8 bit format. When optional Neural Network Extension is selected, Some of the 8-bit multiplications accept the operands in quantized 8 bit format. These instructions subtract the zero biases stored in these registers from the operands. This feature is explained in *Neural Network (NN) Examples* on page 153.

Table 4: Zero Bias State Registers

State Register	Bit Size	Description
AE_ZBIASC8	8	This register holds the zero bias for operand 1, usually coefficients/ weights in neural networks.
AE_ZBIASV8	8	This register holds the zero bias for operand 2, usually vectors in neural networks.

The state registers shown in *Table 5: Floating Point Support State Registers* on page 29 pertain to the optional floating point support.

Table 5: Floating Point Support State Registers

State Register	Bit Size	Description
RoundMode	2	Control the rounding mode of floating point operations. A value of 0 rounds to nearest, a value of 1 rounds toward 0, a value of 2 rounds towards infinite and a value of 3 rounds toward negative infinite.
InvalidFlag	1	Invalid exception flag.
DivZeroFlag	1	Divide-by-zero flag.
OverflowFlag	1	Overflow exception flag.

State Register	Bit Size	Description
UnderflowFlag	1	Underflow exception flag.
InexactFlag	1	Inexact exception flag.

The TIE state registers are grouped as follows into six user registers for the purposes of efficient save and restore operations. The set of states can be read using AE_RUR_NAME and AE_WUR_NAME, where NAME is the name given below. The associated number is not relevant for the application programmer.

If optional Neural Network Extension is selected, the zero bias state registers are grouped into a user register named AE_ZBIASV8C. These registers are read from and written to AE_DR (ae_int32x2) register using special instructions AE_MOVZBVCDR and AE_MOVDRZBVC.

```
user_register AE_OVF_SAR 240 {
       AE_SAR[13:6],
    AE_OVERFLOW[0],
    AE_SAR[5:0] }
user_register AE_BITHEAD 241 AE_BITHEAD[31:0]
user_register AE_TS_FTS_BU_BP 242 {
             AE_TABLESIZE[3:0],
        AE_FIRST_TS[3:0],
        AE_BITSUSED[3:0],
        AE_BITPTR[3:0] }
user_register AE_CW_SD_NO 243 {
             AE_CWRAP[0],
        AE_SEARCHDONE[0],
        AE_NEXTOFFSET[26:0] }
user_register AE_CBEGIN0 246 AE_CBEGIN0[31:0]
user_register AE_CEND0 247 AE_CEND0[31:0]
user_register AE_CBEGIN1 248 AE_CBEGIN1[31:0]
user_register AE_CEND1 249 AE_CEND1[31:0]

user_register AE_CBEGIN2 236 AE_CBEGIN2 [31:0]
user_register AE_CEND2   237 AE_CEND2   [31:0]
user_register AE_CBEGIN3 238 AE_CBEGIN3 [31:0]
user_register AE_CEND3   239 AE_CEND3   [31:0]
// With Neural Network Extension, the following user register is defined.
user_register AE_ZBIASV8C {AE_ZBIASV8, AE_ZBIASC8}

// With the floating point option, the following user register is used to control and
// detect rounding and exception behavior. See Chapter 4 of the Xtensa® Instruction Set
// Architecture (ISA) Reference Manual for more details.
user_register FCR_FSR
{RoundMode,InvalidFlag,DivZeroFlag,OverflowFlag,UnderflowFlag,InexactFlag}
```

In addition to specialized instructions sequences used to save and restore entire user registers efficiently from memory, instructions are provided to read and write individual state registers. Both types are listed in *Table 6: State Register Access Instructions* on page 31.

Table 6: State Register Access Instructions

Instruction	Intrinsic	Description
RUR.AE_OVERFLOW	RUR_AE_OVERFLOW, RAE_OVERFLOW	Read state register AE_OVERFLOW
RUR.AE_SAR	RUR_AE_SAR, RAE_SAR	Read the lower half of state register AE_SAR
AE_MOVASAR	AE_MOVASAR	Read all of state register AE_SAR
RUR.AE_TABLESIZE	RUR_AE_TABLESIZE, RAE_TABLESIZE	Read state register AE_TABLESIZE
RUR.AE_FIRST_TS	RUR_AE_FIRST_TS, RAE_FIRST_TS	Read state register AE_FIRST_TS
RUR.AE_BITHEAD	RUR_AE_BITHEAD, RAE_BITHEAD	Read state register AE_BITHEAD
RUR.AE_BITSUSED	RUR_AE_BITSUSED, RAE_BITSUSED	Read state register AE_BITSUSED
RUR.AE_BITPTR	RUR_AE_BITPTR, RAE_BITPTR	Read state register AE_BITPTR
RUR.AE_SEARCHDONE	RUR_AE_SEARCHDONE, RAE_SEARCHDONE	Read state register AE_SEARCHDONE
RUR.AE_NEXTOFFSET	RUR_AE_NEXTOFFSET, RAE_NEXTOFFSET	Read state register AE_NEXTOFFSET
RUR.AE_CBEGIN0	RUR_AE_CBEGIN0, RAE_CBEGIN0, AE_GETBEGIN0	Read state register AE_CBEGIN0. AE_GETCBEGIN0 returns a void * value.
RUR.AE_CEND0	RUR_AE_CEND0, RAE_CEND0, AE_GETCEND0	Read state register AE_CEND0. AE_GETCEND0 returns a void * value.
RUR.AE_CBEGIN1	RUR_AE_CBEGIN1, RAE_CBEGIN1, AE_GETBEGIN1	Read state register AE_CBEGIN1. AE_GETCBEGIN1 returns a void * value.
RUR.AE_CEND1	RUR_AE_CEND1, RAE_CEND1, AE_GETCEND1	Read state register AE_CEND1. AE_GETCEND1 returns a void * value.

Instruction	Intrinsic	Description
RUR.AE_CBEGIN2	RUR_AE_CBEGIN2, RAE_CBEGIN2, AE_GETBEGIN2	Read state register AE_CBEGIN2. AE_GETCBEGIN2 returns a void * value.
RUR.AE_CEND2	RUR_AE_CEND2, RAE_CEND2, AE_GETCEND2	Read state register AE_CEND2. AE_GETCEND2 returns a void * value.
RUR.AE_CWRAP	RUR_AE_CWRAP, RAE_CWRAP	Read state register AE_CWRAP
RUR.FCR	RUR_FCR	Read register FCR containing state RoundMode
RUR.FSR	RUR_FSR	Read register FSR corresponding to state registers InvalidFlag, DivZeroFlag, OverflowFlag,UnderflowFlag and InexactFlag
AE_MOVVFCRFSR	AE_MOVVFCRFSR	Copy user register FCR_FSR into a vector register which can be stored to memory.
WUR.AE_OVERFLOW	WUR_AE_OVERFLOW, WAE_OVERFLOW	Write state register AE_OVERFLOW
WUR.AE_SAR	WUR_AE_SAR, WAE_SAR	Write duplicate values to each half of state register AE_SAR
AE_MOVSARA7X2	AE_MOVSARA7X2	Write each half of state register AE_SAR from two different AR registers
AE_MOVSARD7	AE_MOVSARD7	Write each half of state register AE_SAR from each half of an AE_DR register
WUR.AE_TABLESIZE	WUR_AE_TABLESIZE, WAE_TABLESIZE	Write state register AE_TABLESIZE
WUR.AE_FIRST_TS	WUR_AE_FIRST_TS, WAE_FIRST_TS	Write state register AE_FIRST_TS
WUR.AE_BITHEAD	WUR_AE_BITHEAD, WAE_BITHEAD	Write state register AE_BITHEAD

Instruction	Intrinsic	Description
WUR.AE_BITSUSED	WUR_AE_BITSUSED, WAE_BITSUSED	Write state register AE_BITSUSED
WUR.AE_BITPTR	WUR_AE_BITPTR, WAE_BITPTR	Write state register AE_BITPTR
WUR.AE_SEARCHDONE	WUR_AE_SEARCHDONE, WAE_SEARCHDONE	Write state register AE_SEARCHDONE
WUR.AE_NEXTOFFSET	WUR_AE_NEXTOFFSET, WAE_NEXTOFFSET	Write state register AE_NEXTOFFSET
WUR.AE_CBEGIN0	WUR_AE_CBEGIN0, WAE_CBEGIN0, AE_SETCBEGIN0	Write state register AE_CBEGIN0. AE_SETCBEGIN0 take a void *value.
WUR.AE_CEND0	WUR_AE_CEND0, WAE_CEND0, AE_SETCEND0	Write state register AE_CEND0 AE_SETCEND0 take a void * value.
WUR.AE_CBEGIN1	WUR_AE_CBEGIN1, WAE_CBEGIN1, AE_SETCBEGIN1	Write state register AE_CBEGIN1. AE_SETCBEGIN1 take a void * value.
WUR.AE_CEND1	WUR_AE_CEND1, WAE_CEND1, AE_SETCEND1	Write state register AE_CEND1 AE_SETCEND1 take a void * value.
WUR.AE_CBEGIN2	WUR_AE_CBEGIN2, WAE_CBEGIN2, AE_SETCBEGIN2	Write state register AE_CBEGIN2. AE_SETCBEGIN2 take a void *value.
WUR.AE_CEND2	WUR_AE_CEND2, WAE_CEND2, AE_SETCEND2	Write state register AE_CEND2 AE_SETCEND2 take a void * value.
WUR.AE_CWRAP	WUR_AE_CWRAP, WAE_CWRAP	Write state register AE_CWRAP
WUR.FCR	WUR_FCR	Write register FCR containing state RoundMode
WUR.FSR	WUR_FSR	Write register FSR corresponding to state registers InvalidFlag, DivZeroFlag, OverflowFlag,UnderflowFlag and InexactFlag
AE_MOVFCRFSRV	AE_MOVFCRFSRV	Set user register FCR_FSR from a vector register which can be loaded from memory.

Instruction	Intrinsic	Description
AE_MOVDRZBVC	AE_MOVDRZBVC	Reads state registers AE_ZBIASV8 and AE_ZBIASC8 into a AE_DR register's high and low 32 bits respectively. AE_SZBIASVC is a proto implemented using this operation to store the value to memory. This instruction is available only if Neural Network Extension is selected.
AE_MOVZBVCDR	AE_MOVZBVCDR	Sets state registers AE_ZBIASV8 and AE_ZBIASC8 with the values from a AE_DR register's High and low 32 bits respectively. AE_LZBIASVC is a proto implemented using this operation to load the values from memory. This instruction is available only if Neural Network Extension is selected.

In the operation descriptions in *Standard DSP Operations by Type* on page 81 each mnemonic is listed with assembly syntax showing placeholders for its operands. The register files of the operands are implied by the placeholders, as in *Table 7: Operand Register Types* on page 34.

Table 7: Operand Register Types

Placeholder	Register File	Legal Values	Example
A, ah, al, a0, a1, ax	AR	a0 – a15	a3
q, q0, q1, d, d0, d1, dh, dl, fr, fs, ft	AE_DR	aed0 – aed15	aed2
acc_ep	AE_EP	aep0 – aep3	aep1
b	BR	b0 – b15	b3
bhl	BR2	b0 – b14 (even)	b0
b3210	BR4	b0-b12 (multiple of 4)	b0
u	AE_VALIGN	u0-u3	u0

Table 8: Operand Immediate Types

Placeholder	Value Range	Stride
i16	-16..14	2
i16pos	0..14	2
i32	-32..28	4
i32pos	0..28	4
i64	-64..56	8
I64half	-32..24	8
i64pos	0..56	8
I64neg	-32..-8	8
I	Operation-dependent	1

Given multiple formats, the slotting constraints on each operation can be quite complicated. Refer to the generated *ISA HTML* available via the Xtensa Xplorer.

Each operation description is also annotated with the C syntax showing the intrinsic name and prototype for the operation. A discussion of using C data types and intrinsics to program the HiFi 5 DSP is included in *Programming the DSP* on page 49.

All the HiFi 2 , HiFi 3 and HiFi 4 C types and intrinsics are available in HiFi 5 DSP to ensure C/C++ source code portability. Notes on HiFi 2 , HiFi 3 and HiFi 4 code portability and matching intrinsics are included in the operation description for the relevant operations as well as in *Programming the DSP* on page 49.

Enhancements

The HiFI 5 DSP offers significant enhancements over the previous DSPs in the HiFi family.

Multiplies

The HiFi 3 DSP supports:

- Two 32x32-bit multiplies per cycle
- Four 32x16-bit multiplies per cycle
- Four 16x16-bit multiplies per cycle
- Four 24x24-bit multiplies per cycle

The HiFi 3z DSP supports:

- Two 32x32-bit, four 24x24-bit, or 32x16-bit multiplies per cycle

- Also has limited support for eight 16x16-bit multiplies per cycle (primarily used in dot product or complex multiply)

The HiFi 4 DSP supports:

- Four 32x32-bit multiplies per cycle.
- Has limited support for eight 32x16-bit multiplies per cycle.
- Limited support for eight 16x16-bit multiplies per cycle (primarily used in dot product or complex multiply). Same as the HiFi 3 DSP; however, the algorithms using 16x16-bit multiplies benefit from the enhanced the load/store unit.

The HiFi 5 DSP brings the advantages of both the HiFi 3z and HiFi 4 DSP with a rich set of multiplications.

The HiFi 5 DSP supports:

- Eight 32x32-bit multiplies per cycle
- Sixteen 32x16-bit and 16x16-bit multiplies per cycle

The HiFi 5 DSP also enhances the support for complex numbers. See *Multiply and Accumulate Operations Overview* on page 96 for details on the HiFi 5 multipliers.

The HiFi 3/3z DSPs support four 24x24-bit multiplies but only two 32x32-bit, hence the algorithms that use 24-bit precisions gives good performance as compared to the algorithms with 32-bit. There is no performance advantage of using the 24x24-bit instead of the 32x32-bit in the HiFi 4 and HiFi 5 DSPs, as most of the 24x24-bit multiplies actually perform 32x32-bit multiplication. The 24x24-bit multiplies are provided as Compatibility intrinsics to allow old applications to correctly compile and run. While the HiFi 4 and HiFi 5 DSPs compile and run the algorithms with 24-bit precision, the output may differ on wraparound conditions and saturation. If the algorithm is written assuming the kind of wraparound or saturation found in HiFi 3/3z or HiFi 2/EP, the implementation may need to be fixed to ensure correct output on the HiFi 4 and HiFi 5 DSP. The HiFi 5 DSP implementation is bit exact with HiFi 4 DSP for all the compatible intrinsics and may show different behaviour against the HiFi 3/3z DSPs.

The HiFi 3/3z DSPs supports high precision multiplies that accumulate in 64 bits. The HiFi 4 and HiFi 5 DSPs provide limited support for 72-bit accumulators. While the main AE_DR register in HiFi 4 and HiFi 5 is 64 bits just like HiFi 3/3z, you can use an auxiliary 8-bit register, AE_EP, together with AE_DR, to provide a total of 72 bits of precision. Support for 72 bits is available in multiply, multiply accumulate, addition, subtraction, shift, saturation, sign extension, and move operations. The 72-bit support is described in *Multiply and Accumulate Operations Overview* on page 96, *Add, Subtract and Compare Operations* on page 110, *Shift Operations* on page 112, *Saturate Operations* on page 117 and *Move Operations* on page 119.

Circular Buffer Support

The HiFi 3 DSP supports a single circular buffer, HiFi 4 supports two, and HiFi 5 supports three. The additional circular buffer in HiFi 5 DSP is mainly for 128-bit and 64-bit load/store operations. Each circular buffer can be accessed through its own set of intrinsics, except the first one, all others are prefixed with 1 or 2. On the HiFi 4 and HiFi 5 DSP, most of the load/store operations are supported with first circular buffer, but limited load/store operations are provided with remaining circular buffers. Refer to *Load and Store Operations Overview* on page 83 for the Load/Store operations supported by each circular buffers. The bitstream/variable length instructions use only the first circular buffer. The HiFi 4 and HiFi 5 DSP support a special instruction AE_ADDICIRC for emulation of an arbitrary number of additional circular buffers. The circular buffer registers are listed in *Table 3: Circular Buffer Support State Registers* on page 29.

Dynamic Normalization

Dynamic normalization is a very useful feature for the FFT implementation. The HiFi 4 DSP has native support for 32-bit dynamic normalization while the HiFi 3z DSP provides support for 16-bit dynamic normalization. Both the HiFi 3z and HiFi 4 DSPs support only a single channel dynamic normalization. HiFi 5 DSP includes both 32-bit and 16-bit dynamic normalisation, and also extends the native support to two channels. These instructions help to implement two-channel (Stereo) FFT with dynamic normalisation. Refer to *Complex FFT Example* on page 146 for an example of the stereo complex FFT implementation using the special instructions used for dynamic normalizations.

Single Precision Floating Point Unit

The HiFi 4 DSP has optional two copies of a 2-way SIMD single precision floating point unit (SP FPU).

The HiFi 3 and HiFi 3z DSPs optionally support one copy.

The HiFi 5 also has two copies of the SP FPU, which is backward compatible with HiFi 4. It provides 4-way SIMD variant for performance critical instructions, such as Multiply and Subtract.

Half Precision Floating Point Unit

Another optional floating point package included in the HiFi 5 DSP is the 4-way SIMD half precision floating point unit (HP FPU). Some performance critical multiply instructions on HP FPU implement 8-way SIMD, which can be issued in two parallel slots to achieve 16 MACs per cycle throughput. HP FPU also provides reduction maths operations and convolution multiplies. HP FPU can be used to implement Matrix Multiplication and Convolution used in Neural Networks in half precision data type.

Both SP FPU and HP FPU provide instrucitons that are IEEE-754 complaint. Refer to *Optional Floating Point Unit Operations* on page 125 for details on both SP FPU and HP FPU.

Neural Network Extension

The HiFi 5 DSP includes an optional package called the Neural Network Extension. This package extends the HiFi 5 DSP's capability with specialized instructions targeted for matrix vector multiplication and 2D convolution, and also for variable aligning load/store operations. The matrix vector multiplications and convolution arithmetic used in Neural Network are computationally intensive due to its large dimensions. When storing the weights, low precision 8-bit or below is used, so these instructions work on data as low as 4 bit. The supported precisions are 8x16-bit, 8x8-bit and 4x16-bit. The Neural Network Extension includes the instructions for both signed and unsigned 8 and 4-bit numbers and signed 16-bit. This extension also includes instructions to handle the multiplication and convolution in Quantized 8-bit format used in ANN. These instructions are usually issued in 3 slot VLIW format, in parallel with Load operations and provide throughput of 32 multiplications per cycle. Refer to the *Neural Networks Multiplication Operations* on page 106 for more details. The state registers used by Neural Network Extension instructions for Quantized 8-bit format are listed in section *Table 4: Zero Bias State Registers* on page 29. Chapter *Neural Network (NN) Examples* on page 153 provides examples of the Neural Network Extension usage. The variable aligning load/store operations are documentd as part of *Load and Store Operations Overview* on page 83.

Programmers are required to modify the code to benefit from these enhancements.

2.2 Instruction Naming Conventions

All HiFi 5 DSP integral and fixed point DSP operation mnemonics shown in *Table 9: Operation Mnemonics* on page 38 begin with the string `AE_` to avoid colliding with any other space of names. The optional floating point instructions use the standard Xtensa floating point intrinsic names that add an XT_ prefix to the operation name and replace the .S with `_S`.

Following the `AE_` prefix, each mnemonic has a string of one or more characters signifying the type of operation such as load, shift, add, *etc*. For example, `AE_L` is the prefix denoting DSP loads. Operation types are shown in *Table 9: Operation Mnemonics* on page 38.

The remaining portion of each operation mnemonic typically includes reminders of various aspects of the operation's details. Multiplies, loads and stores have more regular naming conventions that are described in their respective sections.

The Operation types and Operation Mnemonics are listed in *Table 10: Operation Type* on page 40 *Table 9: Operation Mnemonics* on page 38

Table 9: Operation Mnemonics

Mnemonic	Meaning
A	Depending on the context, either denotes the accumulator for MAC operations and some ALU operations,

Mnemonic	Meaning
	-or- Indicates the operations use the AR register (particularly for shift or convert operations, and some ALU operations). This convention is used when similar operations exist that use AE_DR register (A) of Immediate value (I)
E / EP	Denotes AE_EP Register
ASYM	Denotes asymmetric rounding (*e.g.*, AE_ROUND32X2F64S**ASYM**)
B / BR	Denotes AE_BR register
C	Denotes Complex operations (*e.g.*, AE_MULF**C**32RA) or Carry (*e.g.*, ADD**C**32) depending on the context.
CNV	Denotes convolution operation (*e.g.*, AE_MUL2X4Q4X16**CNV**_L).
D	Refers to dual operations or indicates that the operation uses AE_DR register depending on the context.
EXP	Used to indicate that the instruction arithmetic is on the exponent part of a number.
EXT	Denotes extraction or bit-extraction.
F	Denotes fractional arithmetic (*e.g.*, AE_MULZAA**F**D32S.HH.LL) or the value False in a conditional move (*e.g.*, AE_MOVF64).
H and L	Combinations of H and L are used to refer to halves of registers (*e.g.*, AE_MULZAAFD32S.**HH**.**LL**) If HP FPU option is enabled, H is also the suffix to indicate the SIMD nature of the HP FPU instructions , that don't start with AE_. (Usually comes as _H [Scalar] / _HX4 [2 Way SIMD] / _HX4X2 [4 Way SIMD]).
0,1,2,3	Combinations of 0,1,2 and 3 are used to refer to quarters of registers (*e.g.,* AE_MULF32X16.L**0**)
I	Denotes use of an immediate operand (*e.g.,* AE_SRA**I**P32)
J / CJ	Used to indicate that one of the operands in the arithmetic is imaginary value *i*. The exact arithmetic is explained in the instructions.
M	Used in legacy instruction to indicate that the number is stored mid of DR registers.
O	Denotes Octal MAC operations, if Neural Network Extension is selected.
P	Denotes Pair vector operands for SIMD operations Some legacy HiFi2 instructions use this letter to denote P register of HiFi 2's p register, which is mapped to AE_DR Register on HiFi 5 DSP
Q	Denotes Quad MAC/MSU operations Few legacy HiFi 2 instructions use this letter to denote P register of HiFi 2's q register, which is mapped to AE_DR Register on HiFi 5 DSP
R	Symmetric Round, same as SYM, used in MUL operations

Mnemonic	Meaning
RA	Asymmetric Round. Same as ASYM, used in MUL operations
RAS	Asymmetric Rounding and Saturation, used in MUL operations
RNG	Indicates the operation is specially designed for range (shift) detection and normalize the numbers useful in implementing dynamic FFT implementation.
S	Denotes saturating arithmetic (*e.g.*, AE_MULF32**S**.LL) or the use of the AE_SAR state register as a shift amount (*e.g.*, AE_SRA**S**P32), depending on the context. If SP FPU is enabled, S is also the suffix to denote that the SIMD nature of the SP FPU instructions , that don't start with AE_. (usually comes as _S [scalar] / _SX2 [2 Way SIMD]/ _SX2X2 [4 Way SIMD])
SYM	Denotes symmetric rounding (*e.g.*, AE_ROUND32X2F64S**SYM**)
T	Denotes the value True in a conditional move (*e.g.*, AE_MOVT64) A few multiplication instructions use T to indicate truncate
U	Denotes unsigned arithmetic (*e.g.*, AE_MULS32U.LL)
V	Denotes Variable shift amount to each lane, used in shift instructions
W	Denotes the output precision of the instruction is higher than the input operands.
X	Denotes use of an index register in an address computation (*e.g.*, AE_L64.**X**P)
X2	Denotes a 2-way SIMD operation in contexts (*e.g.*, AE_L32**X2**.I) where scalar operations are also available
X4	Denotes a 4-way SIMD operation (*e.g.*, AE_L16X4.XC)
X2X2	Denotes a 2-way SIMD operation, in which each operand is spread over 2 vector registers (first X2 indicates the elements in the vector and latter X2 refers to the number of vectors)
X4X2	Denotes a 4-way SIMD operation, in which each operand is spread over 2 vector registers
X8	Denotes an 8-way SIMD operation.
Z	Denotes zeroing out the accumulator before performing accumulation.

The following table briefly explains various operation types used in HiFi 5 DSP. For the complete details of the instruction, refer to the ISA HTML.

Table 10: Operation Type

Operation Type	
ACCW	This type of operation adds two numbers and accumulates the result into high precision output. (For example, AE_ACCW8 add two 8-bit numbers and accumulate into 16 bit)
ADD[W][RNG]	This type of operation adds/subtracts two numbers.

Operation Type	
SUB[W][RNG]	W refers to high precision output. RNG refers to special kind of fused operations used in FFT
RADD	Reduction add operations, used to add the elements within vectors
ADDSUB SUBADD ADDANDSUB	Instructions that perform both add and sub. ADDSUB performs the arithmetic within a vector register, while ADDANDSUB across the registers. ADDANDSUB is useful with complex arithmetic.
EXPADD EXPSUB	Add or subtract exponents of a user define emulated float number. Refer to *User Defined Emulated Float Example* on page 149
ADDCEXP	Adds mantissa of two emulated float number, with carry updating the exponent. Refer to *User Defined Emulated Float Example* on page 149
ADD[I]CIRC	Special type of operations to manipulate the pointer used for circular buffer access. I indicates that the operand 2 is immediate. Note, these instructions work on AR registers.
ADDINV	This type of instructions are used to find 1-x of a fixed point number.
ABS	These SIMD instructions find absolute value of each element in a vector
NEG	These SIMD instructions negate each element in a vector. There are negate instructions also to negate certain specific elements in a vector
MIN[ABS] MAX[ABS]	These SIMD instructions find minimum and maximum between elements across two vectors ABS refers absolute minimum or maximum
BMIN, BMAX	These SIMD instructions find minimum and maximum between elements across two vectors. They are same as MAX and MIN, except that these instructions also set Boolean flags
RMIN, RMAX	Finds minimum and maximum within the elements of a vector.
MINMAX	Instruction to limit/clip signed elements in a register within the range specified by the minimum and maximum signed values in the input registers.
L	Load Operations, used for both scalar and vector load operations. In case of vector load, these operations expect the pointers to be aligned.
LA	Aligning Load operations, used to load vectors from a stream. These instructions perform aligning the data as the pointers need not be aligned. These instructions require priming of the pointer before using.
S	Store operations, used for both scalar and vector store operations. In case of vector store, these operations expects the pointers to be aligned.
SA	Aligning Store operations, used to store vectors to a stream. These instructions perform aligning the data as the pointers need not be aligned. These instructions require priming of the pointer before using.

Operation Type	
LAV	Variable Aligning Load operations, used to load partially from a stream. These instructions require priming of the pointer before using.
SAV	Variable Aligning Store operations, used to store partial elements from a vector into a stream. These instructions require priming of the pointer before using.
LALIGN	Instructions used for priming a stream to use AE_LA operations.
ZALIGN	Instructions used for priming a stream to use AE_SA operations.
SALIGN	Instructions used to flush out the stream from VALIGN registers.
RNG	Special kind of operation useful in dynamic FFT implementation
CALCRNG	Instructions that compute the shift value for the dynamic range determination for Fast Fourier Transform (FFT).
CONJ	Find complex conjugate of a number
NSA	Instructions to find the (left) shift amount to normalize a number. (referred as headroom).
MUL	Multiply operation
MOV	Instruction to move between registers, within DR, across DR, AR, BR and VALIGN registers.
CLAMPS	Instruction that clamps AR register value to 16-bits signed
LE | LT | EQ	These SIMD instructions compare each element of two vectors. LT : Less Than LE : Less than or Equal EQ - Equal
AND | OR | NAND | XOR	These SIMD instructions bitwise operations on each element of two vectors
SLA | SRA	Arithmetic Left / Right Shift operations
SRL	Logical Right Shift Operations
SORT	Sort elements of a vector
[D]SEL	Selects elements across two vectors to form a vector based on the index pattern given. DSEL is a 2-way SIMD variant of SEL
SHFL	Shuffles elements within a vector, based on the index pattern given
ROUND	Instruction to demote a fixed point number to lower precision with rounding logic
TRUNC	Instruction to demote a fixed point number to lower precision with truncate logic
SAT	Instruction to demote an integer number to lower precision with saturate logic.

Operation Type	
CVT	Instruction to promote a fixed point number with user defined shift and sign extend. These instructions are useful in adjusting Q formats.
SEXT	Instruction to promote an integer number to high precision using sign extension.
ZEXT	Instruction to promote an integer number to high precision using zero extension.
VLD	Instruction for Variable Length Decoding.
VLE	Instructions for Variable Length Encoding.
LB | LBK | LBS	Instructions for loading bits into AR register from a bitstream.
DB	Instruction for discarding bits from a bitstream.
SB|SBF	Instructions for storing bits into a bitstream from Bitstream. SBF is used to flush the bitstream head into bitstream.
BITSWAP	Instruction for swapping or reversing the order of bits in an AR register.
SHORTSWAP	Instruction for swapping or reversing the order of elements in a DR register.
SHA	Swap Half-word Access instruction to swap bytes in the two halfwords in an AR, typically for endianness change during a memcpy()-like operation
PKSR	Pack-Shift-Round Instructions used for acceleration of the biquad filters.
ARDECNORM	Instruction used for Arithmetic Decoding.
DIV	Denotes Division on helper function for implementing division

2.3 Fixed Point Values and Fixed Point Arithmetic

The HiFi 5 DSP contains instructions for implementing fixed point arithmetic. This section describes the representation and interpretation of fixed point values as well as some operations on fixed-point values.

Representation of Fixed Point Values

A fixed point data type *m.n* contains a sign bit, some number of bits *m-1*, to the left of the decimal and some number of bits *n*, to the right of the decimal. When expressed as a binary value and stored into a register file, the least significant *n* bits are the fractional part, and the most significant *m* bits are the integer part expressed as a signed 2’s complement number. If the binary value is interpreted as a 2’s complement signed integer, converting from the binary value to a fixed point number requires dividing the integer by 2n.

Thus, for example, the 24-bit 1.23 number 0.5 is represented as 0x400000.

Table 11: Example1: 24-bit 1.23 Number 0.5

Signed Integer (1 bit)	Fractional (23 bits)
0	100 0000 0000 0000 0000 0000
0x0	0x40 0000

And the 64-bit 17.47 number -1.5 is represented as (-2 + 0.5 = 0xff 4000 0000 0000).

Table 12: Example2: 64-bit 17.47 Number -1.5

Signed Integer (17 bit)	Fractional (47 bits)
1 1111 1111 1111 1110	100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x1fffe	0x4000 0000 0000

HiFi 5 DSP fractional instructions use fractional operations on 1.15, 1.23, 9.23, 1.31, 17.47 and 1.63, described in more details as follows. The extra precision 72-bit accumulators are only supported with integer operations.

- 1.15: 16-bit fixed point data type with 1 sign bit and 15 bits to the right of the decimal. The largest positive value 0x7fff is interpreted as $1.0 – 2^{-15}$. The smallest negative value 0x8000 is interpreted as -1.0. The value 0 is interpreted as 0.0.
- 9.23: 32-bit fixed point data type with a 9-bit signed integer and 23 bits to the right of the decimal. The largest positive value 0x7fffffff is interpreted as $256.0 – 2^{-23}$. The smallest negative value 0x80000000 is interpreted as -256.0. The value 0 is interpreted as 0.0.
- 1.23: 24-bit fixed point data type with 1 sign bit and 23 bits to the right of the decimal. The largest positive value 0x7fffff is interpreted as $1.0 – 2^{-23}$. The smallest negative value 0x800000 is interpreted as -1.0. The value 0 is interpreted as 0.0. Since register halves hold 32 bits, not 24 bits, typical 24-bit fractional variables are 9.23. However, 24-bit fixed-point multiply instructions ignore the upper 8 bits, thereby treating them as 1.23.
- 1.31: 32-bit fixed point data type with 1 sign bit and 31 bits to the right of the decimal. The largest positive value 0x7fffffff is interpreted as $1.0 – 2^{-31}$. The smallest negative value 0x80000000 is interpreted as -1.0. The value 0 is interpreted as 0.0.
- 17.47: 64-bit fixed point data type with a 17-bit signed integer and 47 bits to the right of the decimal. The largest positive value 0x7fff ffff ffff ffff is interpreted as $65536.0 – 2^{-47}$. The smallest negative value 0x8000 0000 0000 0000 is interpreted as -65536.0. The value 0 is interpreted as 0.0.
- 1.63: 64-bit fixed point data type with 1 sign bit and 63 bits to the right of the decimal. The largest positive value 0x7fff ffff ffff ffff is interpreted as $1.0 – 2^{-63}$. The smallest negative value 0x8000 0000 0000 0000 is interpreted as -1.0. The value 0 is interpreted as 0.0.
- 9.55: 64-bit fixed point data type with a 9-bit signed integer and 55 bits to the right of the decimal. The largest positive value 0x7fff ffff ffff ffff is interpreted as $256.0 – 2^{-55}$. The

smallest negative value 0x8000 0000 0000 0000 is interpreted as -256.0. The value 0 is interpreted as 0.0.

Arithmetic with Fixed Point Values

When multiplying fixed point numbers *m.n0* * *m.n1* with a standard signed integer multiplier, the natural result of the multiple will be an m.n data type where n = n0+n1 and m = m0+m1. So, for example, multiplying a 1.31 typed variable by a 1.31 typed variable generates a 2.62 typed variable. The HiFi 5 DSP supports both the 1.63 and the 17.47 data type. Fixed point multiply instructions that generate 1.63-based data shift the 2.62 result to the left by 1-bit and then saturate the result. Multiply instructions that generate 17.47-bit data, round away the bottom 15 bits.

HiFi 5 DSP contains both saturating and non-saturating instructions. Overflowing the supplied guard bits with a non-saturating instruction is a program error that will cause the result to wrap around. For saturating operations, the processor also sets the overflow state, which can later be checked programmatically. In the instruction descriptions that follow, it is explicitly stated if an operation saturates.

Other Fixed Point Representations

Programmers are free to use fixed-point representations other than the ones listed above. Most HiFi 5 DSP operations are independent of fixed-point representation; *e.g.*, a fixed point add is equivalent to an integer one. Even for multiplies, the multiply instructions are compatible with any representations that expect the result to be shifted by the same amount. So, if the input data is actually a 2.30 data type rather than a 1.31 data type, the 32-bit fixed point multiply instructions will correctly produce a 3.61 typed variable. The programmer is responsible for knowing what type of data is in what variables; if manual conversions are needed, you can always use shift instructions.

2.4 VLIW Slots and Formats

HiFi 5 DSP can issue up to five operations in a single 128-bit instruction bundle or two operations in a 64-bit bundle using Xtensa LX FLIX (VLIW) technology. HiFi 5 DSP supports several different formats with different slotting constraints. Every instruction belongs to one format, but different formats may pack a different number of operations in a single instruction.

In general:

- the first slot supports scalar and vector loads and stores, and scalar Xtensa operations
- the second slot supports scalar and vector loads, and scalar Xtensa operations
- the last three slots mostly support vector ALU and vector multiply operations.

Note: One of the 64-bit formats (slot '`ae5_slot1`' in format '`ae_format_5`') uses a dummy second slot to allow multiply operations in the third nominal slot.

Xtensa operations are all available in 24-bit Inst format or an even smaller 16-bit format. A subset of the core Xtensa operations is also available in the first two VLIW slots.

Understanding the slotting is important when optimizing code for HiFi 5 DSP. Often a loop is limited by operations that can only go in one slot or another. For example, it is never possible to issue more than one (possible SIMD) store per cycle. If a loop is limited by the operations in one slot, there is no point in trying to optimize the operations in another slot.

See the generated HTML description available with Xtensa Xplorer for details on available slotting.

The following sub-section lists the common operations available in various FLIX formats:

First Format: ae_format

This is the main four slot, 128-bit format that includes ae_slot0, ae_slot1, ae_slot2, and ae_slot3.

Second Format: ae_format_2

This is a three slot, 128-bit format that includes ae2_slot0, ae2_slot1, and ae2_slot2. The purpose of this format is to support specialized multiply operations and branch operations that require arguments or immediates that are too large to fit into a single slot of the four slot format.

Third Format: ae_format_3

This is a two slot, 64-bit format that includes ae3_slot0 and ae3_slot1. The purpose of this format is to combine two memory or core operations in a smaller, 64-bit instruction.

Fourth Format: ae_format_4

This is a specialized five slot, 128-bit format used for operations used in FFT computation. It includes ae4_slot0, ae4_slot1, ae4_slot2, ae4_slot3, and ae4_slot4.

Fifth Format: ae_format_5

This is effectively a two slot, 64-bit format that includes ae5_slot0, ae5_slot1 (dummy), and ae5_slot2. The purpose of this format is to allow more compact operations including multipliers.

Sixth Format: ae_format_6

This is a four slot, 128-bit format partly specialized for operations used in FFT computation. It contains ae6_slot0, ae6_slot1, ae6_slot2, and ae6_slot3.

Seventh Format: ae_format_7

This is a four slot, 128-bit format used for high throughput vector multiply operations and does not include any vector stores. This format contains ae7_slot0, ae7_slot1, ae7_slot2, and ae7_slot3.

Eighth Format: ae_format_8

This is a specialized three slot, 128-bit format used for multiply operations from the Neural Network (NN) extension. This format is present only when the NN extension is configured, and contains ae8_slot0, ae8_slot1, and ae8_slot2.

Ninth Format: ae_format_9

This is a specialized four slot, 128-bit format used for high throughput single/half-precision floating-point multiply operations and vector stores. This format is present only when SP FPU and/or HP FPU options are configured, and contains ae9_slot0, ae9_slot1, ae9_slot2, and ae9_slot3.

Tenth Format: ae_format_10

This is a specialized four slot, 128-bit format used for single/half-precision floating-point dual MAC operations, and does not include any vector stores. This format is present only when SP FPU and/or HP FPU options are configured, and contains ae10_slot0, ae10_slot1, ae10_slot2, and ae10_slot3.

Eleventh Format: ae_format_1

Along with the RI-2022.9 Xtensa tool chain release, the following FLIX format is added to improve the code size of control code modules.

This new 2-slot, 48-bit FLIX format is added to reduce code-size primarily of scalar (Xtensa) control code. This format contains ae1_slot0 and ae1_slot1.

In addition, more scalar and vector operations are slotted in the narrow (48 and 64 bit) FLIX formats for general code-size reduction. Wide-loops operations are also slotted in HiFi 5 FLIX formats.

3. Programming the DSP

Topics:

- *Data Types*
- *Xtensa Xplorer Display Format Support*
- *Programming Styles*
- *Auto-Vectorization of Standard C/C++*
- *ITU-T/ETSI Intrinsics*
- *Operator Overloading*
- *Intrinsic-Based Programming*
- *Code Portability from Previous HiFi DSPs*
- *Important Compiler Switches*

Cadence recommends you read and become familiar with the *Xtensa C and C++ Application Programmer's Guide* before attempting to obtain optimal results by programming HiFi 5 DSP.

Note that this chapter does not attempt to duplicate the material in this guide.

HiFi 5 DSP offers eight 32-bit MACs per cycle or sixteen 16-bit MACs per cycle. With the optional Neural Network Extension, HiFi 5 also offers 8x16-bit and 8-bit support with 32 MACs per cycle. It offers equivalent support for both integer and fractional arithmetic. The C and C++ languages support integer arithmetic on 32x32-bit or 16x16-bit data. Therefore, while standard applications can effectively utilize HiFi 5 DSP's resources, applications that require fractional arithmetic or 24-bit or 32x16-bit multiplication must be modified to express those semantics. These modifications can be as simple as declaring variables of the appropriate custom data types and relying on built-in operator overloading, or using explicit intrinsics to express the exact operations desired. For 16-bit applications, the ITU-T/ETSI intrinsics [2009] are fully supported.

Applications that require 8x16 or 8-bit multiplications and arithmetic should be modified using explicit intrinsics for exact operations desired.

Similarly, the application that requires 8x16 or 8-bit multiplications and arithmetic should also be modified using explicit intrinsics for exact operations desired.

All the HiFi 5 DSP instructions can be accessed from C/C ++ level intrinsics.

The Xtensa C/C++ compiler (xt-clang) will efficiently register-allocate HiFi 5 DSP variables and schedule HiFi 5 DSP instructions, relieving the programmer from the hardest aspects of writing in assembly. It is not required to resort to assembly at any time.

HiFi 5 DSP is also a 2/4/8-way SIMD (Single Instruction/ Multiple Data) architecture. Applications that do not take advantage of the SIMD will run up to eight times slower than applications that do. For the 16- or 32-bit integer applications and for applications written using the ITU-T/ ETSI intrinsics, the compiler can automatically vectorize code to take advantage of the SIMD architecture. Even so, it is typical for programmers to do some work to fully exploit the available performance. Sometimes only minor tweaks are required. Other times the algorithm's computations and data structures must be completely reordered to create parallel loops that can operate on contiguous variables in memory.

For 24-bit and 32x16-bit applications (and 8x16-bit and 8-bit applications if Neural Network Extension is enabled), the compiler does not automatically vectorize. The application writer must write the code using explicit vector data types or intrinsics.

This chapter describes multiple approaches to programming HiFi 5 DSP and illustrates them with some simple examples. *Variable-Length Encode and Decode* on page 135, *Audio DSP Examples* on page 143 and *Neural Network (NN) Examples* on page 153 goes into more detail with more complicated examples.

3.1 Data Types

Several C data types are provided by the HiFi 5 DSP to facilitate programming the DSP in C and C++ using instruction intrinsics and operator overloading.

The intrinsic prototype for each HiFi operation is described in the ISA Html and the data types used in the operations are listed in *Table 13: HiFi 5 DSP C Types* on page 52 .

HiFi 5 DSP supports 16-, 24-, 32-, and 64-bit types. If the Neural Network Extension is selected, HiFi 5 DSP also supports 8-bit types. All types come in both integer and fractional versions. For intrinsic programmers using 8-, 16-, 32-, and 64-bit types, both integer and fractional versions can usually be used interchangeably. A variable of an integer type can be assigned to a fractional variable, and vice-versa, without changing the bit pattern in registers or memory. It is up to the programmer to use the appropriate intrinsic to achieve the desired computation.

For programmers using operator overloading, however, the fractional and integer types map to different instructions. In particular, fractional types use fractional multiplies and saturating arithmetic while integer types use integer multiplies and non-saturating arithmetic. 24-bit fractional and integer types have an additional difference. 24-bit integer types are stored in memory in the low 24 bits of a 32-bit word, equivalent to the storage representation for 32-bit integers. 24-bit fractional types are stored in memory in the high 24 bits of a 32-bit word, equivalent to a 1.31-bit representation, with the low-precision bits all set to 0.

All types (other than the 56- and 64-bit types) come in both scalar and vector versions. In general, computation happens on vector variables. Scalar variables are stored in registers by replicating the scalar variable into every element of the vector. Assigning a vector variable to a variable of the equivalent scalar type will replicate the element in the lowest bit-position into all the elements of the vector. Assigning a scalar to a vector will not change the bit pattern in the register.

All 64-bit data types must be aligned to 64 bits. Variables declared as custom data types will automatically be properly aligned. Variables cast from standard C data types must be aligned by the user.

If dual vector load/store operations (128-bit load/store) are used, both 128-bit and 64-bit data types must be aligned to 128 bits. As the vector size is 64 bits, the user may use 64-bit C data type and cast the pointer to 128-bit data type to access the memory. In this scenario, the alignment is the user's responsibility.

Assigning a low precision variable to a high precision variable sign extends the variable for signed types and zero extends for unsigned types. Assigning a high precision variable to a low precision variable discards the upper bits for integer types and discards the lower bits for fractional types.

With the optional floating point option, HiFi 5 DSP supports a two 2-way SIMD, single precision floating point type `xtfloatx2`. This type can be converted to and from ae_int32x2 using the standard integer to floating point conversions.

With this option, the data type xtfloatx4 is provided, which is mainly for setting up the pointer for dual vector load/store operations. Along with the RI-2022.9 Xtensa tool chain release, support for xtfloatx4 is improved to handle various arithmetic and multiply operations. xtcomplexfloatx2 data type is added to support complex float data types. This new data type enables various arithmetic and multiply operations on complex data and provides OOB performance improvements for code involving complex float data.

Conversions can also be implicitly applied to intrinsic invocations. For example, just like assigning a scalar variable to a vector variable does not change the bit pattern in the register, a scalar variable can be assigned to an intrinsic expecting an input vector argument without first changing the bit pattern of the scalar.

All legacy HiFi 2 types are supported so that the HiFi 2 code can work out-of-the-box. They should only be used on HiFi 2 code, but can be freely intermixed when porting HiFi 2 code to HiFi 5 DSP. Note that for compatibility with HiFi 2, assigning variables of vector types to variables of type ae_p24s or ae_p24f does not replicate the elements and instead leaves the bit patterns unchanged.

Table 13: HiFi 5 DSP C Types on page 52 contains a complete list of the DSP data types with a brief description.

Table 13: HiFi 5 DSP C Types

Type	Description
ae_int4x16	64-bit type containing sixteen 4-bit elements. The memory format for this type is sixteen elements packed into adjacent bytes such that each byte has two elements. In memory, this type is 8-byte aligned. This data type is provided to support low bit width (LBW) neural networks.
ae_int8	8-bit type consisting of a single integer element stored in memory. When this type is converted to an ae_int8x8 type in an AE_DR register, the data is replicated into the eight 8-bit register elements.
ae_int8x8	64-bit type containing eight 8-bit integer elements. This type normally represents the 64-bit contents of an AE_DR register when the register entry holds four data elements. The memory format for this type is eight elements stored in adjacent bytes. In memory, this type is 8-byte aligned.
ae_int8x16	128-bit type containing sixteen 8-bit integer elements. This is a composite type containing two ae_int8x8 types. Its main use is to support dual vector (128 bit) load store operations.
ae_int16	16-bit type consisting of a single integer element stored in memory. When this type is converted to an ae_int16x4 type in an AE_DR register, the data is replicated into the four 16-bit register elements.
ae_int16x4	64-bit type containing four 16-bit integer elements. This type normally represents the 64-bit contents of an AE_DR register when the register entry holds four data elements. The memory

Type	Description
	format for this type is four elements stored in adjacent 16-bit words. In memory, this type is 8-byte aligned.
ae_int16x8	128-bit type containing eight 16-bit integer elements. This is a composite type containing two ae_int16x4 types. Its main use is to support dual vector (128 bit) load store operations.
ae_int24	24-bit type containing a single integer element stored in the least significant 24 bits of a 32-bit word. In memory, this type is 4-byte aligned. This type is loaded and stored in a way that is equivalent to loading and storing the ae_int32 type.
ae_int24x2	48-bit type containing two 24-bit integer elements. The memory format for this type is two elements, each stored in the least significant 24 bits of adjacent 32-bit words. In memory, this type is 8-byte aligned. This type is loaded and stored in a way that is equivalent to loading and storing the ae_int32x2 type.
ae_int32	32-bit type consisting of a single integer element stored in memory. When this type is converted to an ae_int32x2 type in an AE_DR register, the data is replicated into the two 32-bit register elements.
ae_int32x2	64-bit type containing two 32-bit integer elements. The memory format for this type is two elements stored in adjacent 32-bit words. In memory, this type is 8-byte aligned.
ae_int32x4	128-bit type containing four 32-bit integer elements. This is a composite type containing two ae_int32x2 types. Its main use is to support operator overloading for 32x16-bit multiplication.
ae_int64	64-bit type representing the contents of an AE_DR register when the register entry holds a single integer element.
ae_int64x2	128-bit type containing two 64-bit integer elements. This is a composite type containing two ae_int64 types. Its main use is to support dual vector (128 bit) load store operations.
ae_int64x4	256-bit type containing four 64-bit integer elements. This is a composite type containing four ae_int64 types. Its main use is to support dot product vectorization using quad multiply instructions
ae_int16u	16-bit type consisting of a single unsigned integer element stored in memory. When this type is converted to an ae_int16x4 type in an AE_DR register, the data is replicated into the four 16-bit register elements.
ae_int32u	32-bit type consisting of a single unsignedinteger element stored in memory. When this type is converted to an ae_int32x2 type in an AE_DR register, the data is replicated into the two 32-bit register elements.
ae_f16	16-bit type consisting of a single fractional element stored in memory. When this type is converted to an ae_f16x4 type in an AE_DR register, the data is replicated into the four 16-bit register elements.

Type	Description
ae_f16x4	64-bit type containing four 16-bit fractional elements. The memory format for this type is four elements stored in adjacent 16-bit words. In memory, this type is 8-byte aligned.
ae_f24	24-bit type containing a single 24-bit fractional elements. The memory format for this is an element stored in the most significant 24 bits of a 32-bit word making it equivalent to a 1.31-bit representation. In registers, this occupies the lower 24 bits of each 32-bit half of a register, allowing for extra guard bits of precision.
ae_f24x2	48-bit type containing two 24-bit fractional elements. The memory format for this type is two elements, each stored in the most significant 24 bits of adjacent 32-bit words making it equivalent to a 1.31-bit representation. In registers, this occupies the lower 24 bits of each 32-bit half of a register, allowing for extra guard bits of precision.
ae_f32	32-bit type consisting of a single fractional element stored in memory. When this type is converted to an ae_f32x2 type in an AE_DR register, the data is replicated into the two 32-bit register elements.
ae_f32x2	64-bit type containing two 32-bit fractional elements. The memory format for this type is two elements stored in adjacent 32-bit words. In memory, this type is 8-byte aligned.
ae_f32x4	128-bit type containing four 32-bit fractional elements. This is a composite type containing two ae_f32x2 types. Its main use is to support operator overloading for 32x16-bit multiplication.
ae_f64	64-bit type representing the contents of an AE_DR register when the register entry holds a single fractional element.
ae_ep	8-bit type used in conjunction with ae_int64/ae_f64 for extended precision accumulation.
xtfloat	For configurations with the optional SIMD IEEE-754 single precision floating point unit, a type containing 32-bit IEEE floating point scalar value.
xtfloatx2	For configurations with the optional SIMD IEEE-754 single precision floating point unit, a type containing two 32-bit IEEE floating point values.
xtfloatx4	For configurations with the optional SIMD IEEE-754 single precision floating point unit, 128-bit type containing four 32-bit floating point values. This is a composite type containing two xtfloatx2 types. Its main use is to support dual vector (128 bit) load store operations.
xtcomplexfloat	It is scalar, complex, single-precision floating-point data type. For configurations with the optional SIMD IEEE-754 single precision floating point unit, this data type contains a pair of 32-bit elements representing a complex single precision floating point data. Its main use is to support operations on complex data.
xtcomplexfloatx2	For configurations with the optional SIMD IEEE-754 single precision floating point unit, 128-bit type containing two 32-bit complex floating point values. This is a composite type containing two xtcomplexfloat types. Its main use is to support operations on complex data.

Type	Description
xthalf	For configurations with the optional SIMD IEEE-754 half precision floating point unit, a type containing 16-bit IEEE floating point scalar value.
xthalfx4	For configurations with the optional SIMD IEEE-754 half precision floating point unit, a type containing four 16-bit IEEE half precision floating point values.
xthalfx8	For configurations with the optional SIMD IEEE-754 half precision floating point unit, 128-bit type containing eight 16-bit floating point values. This is a composite type containing two xthalfx2 types. Its main use is to support dual vector (128 bit) load store operations.
ae_valign	VALIGN register for 64-bit aligning load
ae_valignx2	VALIGN register for 128-bit aligning load
HiFi-2 Compatibility Types	
ae_p16x2s	This type ensures HiFi 2 target code compatibility. 32-bit type containing two 16-bit elements. This type lives only in memory, and represents two elements in a 1.15 format. It can be automatically converted into an ae_p24x2s object, in which case the low 8 bits of each resulting element are zero and the upper 8 bits are sign-extended.
ae_p24x2s	This type ensures HiFi 2 target code compatibility. 48-bit type containing two 24-bit elements. The memory format for this type is two elements, each stored in the least significant 24 bits of adjacent 32-bit words. In memory, this type is 8-byte aligned. In HiFi 5 DSP , this type is loaded and stored in a way that is equivalent to loading and storing the ae_p32x2s type.
ae_p24x2f	This type ensures HiFi 2 target code compatibility. This type occupies 64 bits in memory, but should be thought of as a 48-bit type containing two 24-bit fractional elements. This type exists only in memory, and represents two elements in a 1.31 format; the low 8 bits of each of the elements are ignored. It can be automatically converted into an ae_p24x2s object, in which case the low 8 bits of each element are discarded—the 1.31-bit value in memory is converted to a 9.23-bit value in the register.
ae_p16s	This type ensures HiFi 2 target code compatibility. 16-bit type consisting of a single element stored in memory. This type can be automatically converted into an ae_p24x2s. In such a conversion, the ae_p16s object's bits are padded with zeroes and duplicated to form the two 24-bit elements of the resulting ae_p24x2s object. In HiFi 5 DSP, each 24-bit element is sign extended to 32 bits.
ae_p24s	This type ensures HiFi 2 target code compatibility. It is a 24-bit type consisting of a single element stored in the low 24 bits of a 32-bit memory word. This type exists only in memory and can be automatically converted into an ae_p24x2s object. In such a conversion, the ae_p24s object's bits are duplicated to form the two 24-bit elements of the resulting ae_p24x2s object. In HiFi 5 DSP, this type is loaded and stored in a way that is equivalent to loading and storing the ae_p32s type.

Type	Description
ae_p24f	This type ensures HiFi 2 target code compatibility. It is a 24-bit type consisting of a single element stored in the high 24 bits of a 32-bit memory word. This type exists only in memory and can be automatically converted into an ae_p24x2s object. In such a conversion, the ae_p24f object's bits are duplicated to form the two 24-bit elements of the resulting ae_p24x2s object. In HiFi 5 DSP, the 1.31-bit value in memory is converted to a 9.23-bit value in the register.
ae_q56s	This type ensures HiFi 2 target code compatibility. It is a 56-bit type representing the contents of an AE_DR register. The memory format for this type has the bits of the ae_q56s object stored in the low 56 bits of a 64-bit double word. In HiFi 5 DSP , this type is loaded and stored in a way that is equivalent to loading and storing the ae_int64 type.
ae_q32s	This type ensures HiFi 2 target code compatibility. It is a 32-bit type representing a value in memory that will be padded with 16 zeroes at the low end and sign extended by 8 bits at the high end to form a 56-bit value when converted to an ae_q56s object (*i.e.*, when loaded into an AE_DR register). In HiFi 5 DSP , the 1.31-bit value in memory is converted to a 17.47-bit value in the register.

Example C to Load, Store, and Convert Fractions and Other Memory Types

The examples below demonstrate how to efficiently load, store, and convert various data types in C using HiFi 5 DSP. The examples do not enumerate all possible conversions between core C and HiFi 5 DSP types. Generally, conversion between register (local) variables and data in memory (arrays, struct fields, etc.) should be done through pointer typecasting, while conversion between register variables should be done through direct use of the appropriate HiFi 5 DSP conversion intrinsics.

- Take a 32-bit value and replicate as two 32-bit elements in AE_DR.

```
int mem32 = …;
ae_int32x2 p = mem32;
```

- Load two 32-bit values in AE_DR. `&mem32[i]` must be 64-bit aligned.

```
int *mem32 = …;
ae_int32x2 p = *((ae_int32x2 *) &mem32[i]);
```

- Move two 32-bit values in AR to the two 32-bit elements in AE_DR.

```
int ah = …;
int al = …;
ae_int32x2 p = AE_MOVDA32X2(ah, al);
```

Convert and sign-extend a 32-bit (1.31) fraction in AR to a 9.55-bit value in AE_DR.

```
int a = …;
ae_int64 q = AE_CVTQ56A32S(a);
```

- Convert and sign-extend the low (L) 1.31-bit fraction in AE_DR to a 9.55 value in AE_DR.

```
ae_int32x2 p = …;
ae_f64 q = AE_CVTQ56P32S_L(p);
```

- Saturate and truncate two 9.55-bit values in AE_DR to the two 1.31-bit fraction elements of AE_DR.

```
ae_int64 qh = …;
ae_int64 ql = …;
ae_int32x2 p = AE_TRUNCI32X2F64S(qh, ql, 8);
```

- Saturate two 1.31-bit values in AE_DR into two 1.23-bit fraction elements in AE_DR. This allows the resultant values to be safely used in future 24-bit multiply instructions.

```
ae_f32x2 = …;
ae_f24x2 p = AE_SAT24S(d);
```

Changing Types

Sometimes it is necessary to treat a variable as one type for one computation and another for a follow-on computation. For example, one might want to do a fractional multiply on a 24-bit variable that is stored in memory in the low 24 bits rather than the high 24 bits of a word. For such uses, HiFi 5 DSP supports conversion intrinsics that do not change the bit-representation of a variable.

They are all of the form `AE_MOV<dest_type>_FROM<SRC_TYPE>`. The following example shows how to coerce an `ae_f64` variable into `ae_int24x2`.

```
ae_f64 = …;
ae_int24x2 p = AE_MOVINT24X2_FROMF64(d);
```

3.2 Xtensa Xplorer Display Format Support

Xtensa Xplorer provides support for a variety of display formats, which makes using and debugging data types easier. The formats allow vector register data contents to be displayed in an easier to read format. Variables are displayed by default in a format matching their vector data types. Registers are by default always displayed as ae_int64, but you can change the format to any other format.

The display formats for the different types are shown in *Table 14: HiFi 5 DSP Display Types* on page 58.

Table 14: HiFi 5 DSP Display Types

Type	Display Format
ae_int32x2	Hex and decimal for each element of the vector.
ae_f32x2	Hex and decimal for each element of the vector assuming a 1.31 representation.
ae_int24x2	Hex and decimal for each element of the vector. The upper 8 bits of the variable, whether in register or in memory is not displayed.
ae_f24x2	Hex and decimal for each element of the vector. If the variable is in memory, it is displayed as a 1.31 variable. If it is in a register, it is displayed as a 9.23.
ae_int16x4	Hex and decimal for each element of the vector.
ae_f16x4	Hex and decimal for each element of the vector assuming a 1.15 representation.
ae_int8x8	Hex and decimal for each element of the vector.
ae_f8x8	Hex and decimal for each element of the vector assuming a 1.7 representation.
ae_int32	Hex and decimal.
ae_f32	Hex and decimal assuming a 1.31 representation.
ae_int24	Hex and decimal. The upper 8 bits of the variable, whether in register or in memory is not displayed
ae_f24	Hex and decimal. If the variable is in memory, it is displayed as a 1.31 variable. If it is in a register, it is displayed as a 9.23.
ae_int16	Hex and decimal.
ae_f16	Hex and decimal assuming a 1.15 representation.
ae_int8	Hex and decimal.
ae_f8	Hex and decimal assuming a 1.7 representation.
ae_int64	Hex and decimal.
ae_f64	Hex and decimal assuming a 17.47 representation. A 1.63 representation can be seen by explicitly selecting it in Xtensa Xplorer.
ae_int32x4	Hex and decimal for each element of the vector.
ae_f32x4	Hex and decimal for each element of the vector assuming a 1.31 representation.
ae_p24x2f	Hex for each element of the vector. All 24 bits of an element are displayed, even if 0.

Type	Display Format
ae_p24s	Hex. All 24 bits are displayed, even if 0.
ae_p24f	Hex. All 24 bits are displayed, even if 0.
ae_p16x2s	Hex for each element of the vector. All 16 bits of an element are displayed, even if 0.
ae_p16s	Hex. All 16 bits are displayed, even if 0.
ae_q32s	Hex. All 32 bits are displayed, even if 0.
ae_q56s	Hex, with the 8-guard bits separated from the other 48 bits. All 48 bits are displayed, even if 0.

3.3 Programming Styles

Typically, a code can be run efficiently on any fixed-point DSP. If the reference code is floating point, it must be converted into fixed point, unless the optional floating point unit is utilized. Doing such conversions is beyond the scope of this guide. It is often desirable to convert to fixed point, one function at a time. Reference codes are frequently written in terms of basic fixed point intrinsic libraries.

As a first step, it is often desirable to implement the existing intrinsic library in terms of HiFi 5 DSP intrinsics. When implementing such an intrinsic library, the programmer has a choice of using the standard C/C++ data types as external interfaces or using the native HiFi 5 DSP data types. If the body of a library is ported to use HiFi 5 DSP intrinsics, but the interface remains standard C/C++, the implementation must convert to and from the HiFi 5 DSP data types. The compiler can sometimes, but not always, eliminate these conversions. If instead, the interfaces of the libraries are changed to use HiFi 5 DSP data types, performance will be better, but all the code that calls into the library must be changed to handle the HiFi 5 DSP data types, which is not always possible.

Cadence provides an optimized implementation of the ITU-T/ETSI intrinsics used often in voice codecs. The interface uses standard C/C++ data types, but the implementation has been carefully crafted to allow the compiler to eliminate the conversions.

The most common scenario is that important functions in the application are optimized directly for HiFi 5 DSP, and the original library is left for the less important functions.

There are several basic programming styles that can be used, depending on application needs, in increasing order of manual effort. These are:

- Auto-vectorizing standard scalar C/C++ code
- Auto-vectorization code written on top of the ITU-T/ETSI intrinsics
- C/C++ code with HiFi 5 DSP data types and operator overloading
- Use of intrinsic functions for computation instruction along with HiFi 5 DSP data types and implicit loads and stores
- Use of intrinsic functions for both computation and loads and stores

These different styles can be freely intermixed. For maximum performance, it is typically necessary to use at least some amount of explicit intrinsics for computation. However, it is often not necessary to use intrinsics for loads or stores.

For each of these strategies, one can write either scalar or vector code. One general strategy is to port a single function at a time. If the desired semantics match standard C/C++ code or the ITU-T/ETSI intrinsics, start with that and automatic vectorization. For 24-bit or 32x16-bit applications, start with scalar code, using operator overloading where the desired semantics match the available overloads and intrinsics where a specialized semantic is needed. Either way, the code is then profiled. Those parts of the code that are computationally important can then be manually vectorized. At any point, if the performance goals for the code have been met, the optimization can cease. By starting with what can be done easily and refining only the most computationally-intensive portions of code manually, the engineering effort can be directed to where it has the most effect, which is discussed in the following sections.

3.4 Auto-Vectorization of Standard C/C++

Auto-vectorization of scalar C code can produce effective results on simple loop nests, but has its limits. It can be improved through the use of compiler pragmas and options, and effective data marshalling to make data accesses (loads and stores) regular and aligned.

The Xtensa C/C++ compiler provides several options and methods of analysis to assist in vectorization. These options are discussed in detail in the *Xtensa C and C++ Application Programmer's Guide*, in particular in the SIMD Vectorization section. We recommend studying this guide in detail. However, following are some guidelines in summary form:

- Vectorization is triggered with the compiler options `-O3 -fvectorize` (or `-O3 -LNO:simd`), or by selecting the Enable Automatic Vectorization option in Xplorer. The `-LNO:simd_v` and `-save-temps` (or `-S`) options give feedback on vectorization issues and keeps intermediate results, respectively. Xplorer's Vectorization Assistant is a graphical tool to help the programmer understand what did and did not vectorize.
- Data should be aligned to 8-byte boundaries. The Xtensa C/C++ compiler will naturally align arrays to start on 8-byte boundaries. But the compiler cannot assume that pointer arguments are aligned. The compiler needs to be told that data is aligned by one of the following methods:
  - Using global or local arrays rather than pointers
  - Using `#pragma aligned(<pointer>, n <alignment_byte_boundary>)`
- Pointer aliasing causes problems with vectorization. The `__restrict__` attribute for pointer declarations (*e.g.*, `short * __restrict__ cp;`) tells the compiler that the pointer does not alias.
- There are global compiler aliasing options, but these can sometimes be dangerous.

- Subtle C/C++ semantics in loops may make them impossible to vectorize. The Vectorization Assistant can help in identifying small changes that allow effective vectorization.
- Irregular or non-unity strides in data array accessing can be a problem for vectorization. Changing data array accesses to regular unity strides can improve results, even if some “unnecessary computation” is necessary.
- Outer loops can be simplified wherever possible to allow inner loops to be more easily vectorized. Sometimes trading outer and inner loops can improve results.
- Loops containing function calls and conditionals may prevent vectorization. It may be better to duplicate code and perform a little "unnecessary computation" to produce better results.
- Array references, rather than pointer dereferencing, can make code (especially mathematical algorithms) both easier to understand and easier to vectorize.
- At –O3, the compiler will perform optimizations that, while mathematically correct, might change the exact bit results of floating point computations. For example, the compiler might replace a += b*c with a fused multiply-accumulate operation that avoids a round between the multiply and the accumulate. If bit-exact answers are needed, compile with fno-unsafe-math-optimizations.

Consider a simple example that performs a 16-bit energy calculation:

```
int Energy (short a[], int n)
     {
         int i;
         int s = 0;

             for (i = 0; i < n; i++)
             {
                 s = s + a[i]*a[i];
             }
         return s;
     }
```

The program can be compiled either with or without automatic vectorization. Note that even without automatic vectorization it is still important to select the Use DSP co-processor option, or equivalently the `-mcoproc` compiler option. These optimizations allow the compiler to automatically use HiFi 5 DSP instructions for scalar code.

Without vectorization, the compiler generates an inner loop that performs one 16-bit multiply every cycle.

```
loopgtz a3,L
         {
                ae_l16.ip aed0,a2,2
                nop
                ae_mula16x4 aed1,aed2,aed0,aed0
         }
        L:
```

Note that the `ae_mula16x4` instruction performs four multiplies, but because `ae_l16.ip` performs a single 16-bit load that replicates the data, each of the four multiplies is multiplying the same operand.

Note that operations within brackets {, }, in assembly code are part of the same instruction and execute in parallel.

With vectorization, the compiler generates a loop that executes four multiply-adds every cycle.

```
loopgtz a3,L
 {
        ae_la16x4.ip   aed0,u0,a2
        nop
        ae_mula16x4    aed1,aed2,aed0,aed0
  }
  {
        ae_la16x4.ip   aed3,u0,a2
        nop
        ae_mula16x4    aed1,aed2,aed3,aed3
  }
 L:
```

Note that since the input array is a parameter, and since no special compiler flags or pragmas have been used, the compiler must assume that it might not be aligned. Therefore, the compiler uses the aligning load instructions.

If our example used `int` instead of `short`, the compiler would generate a loop that executes two 32-bit multiply-adds per cycle.

3.5 ITU-T/ETSI Intrinsics

To use the ITU-T/ETSI Intrinsics, include one or both of the following header files:

```
#include <hifi2/basic_op_xtensa.h
#include <hifi2/oper_32b_xtensa.h
```

The standard intrinsics can be used either with or without automatic vectorization, just like standard C/C++ code.

Consider our energy calculation example, modified to use the intrinsics.

```
#include <hifi2/basic_op_xtensa.h>

int Energy( short a[], int n)
{
    int i, s = 0;
    for (i = 0; i < n; i++)
    {
            s = L_mac (s, a[i], a[i]);
```

```
    }
    return s;
}
```

Without vectorization, the compiler generates an inner loop that does a multiplication every cycle.

```
loop a3, L
{
      ae_l16.ip aed0,a2,2
      nop
      ae_mulaf16ss.00 aed1,aed0,aed0
  }
 L:
```

With vectorization, the compiler generates an inner loop that does two multiplications every cycle.

```
      loopgtz a3,L
 {
      ae_la16x4.ip aed0,u0,a2
      nop
      ae_mulaafd16ss.33_22 aed1,aed2,aed2
 }
 {
      ae_la16x4.ip   aed2,u0,a2
      nop
      ae_mulaafd16ss.11_00  aed1,aed2,aed2
 }
 {
      nop
      nop
      ae_mulaafd16ss.33_22 aed1,aed0,aed0
 }
 {
      nop
      nop
      ae_mulaafd16ss.11_00 aed1,aed0,aed0
  }
 L:
```

Looking at the loop, you may ask why the compiler cannot generate a loop that executes four multiplies per cycle.

While vectorizing, the compiler maintains bit-exactness with the reference intrinsics. These intrinsics require a saturation step after every multiplication and addition. HiFi 5 DSP only supports a dual-mac instruction that performs sequential saturations. It is not possible to issue two dual-mac instructions in parallel because the second multiply-add needs to wait for the first one to complete. Otherwise, it might not correctly saturate. Only if an ITU-T/ETSI application is doing independent multiply-accumulates in parallel can the compiler generate code that sustains four multiplies per cycle.

3.6 Operator Overloading

Common HiFi operations can be accessed in C or C++ by applying standard C operators to the HiFi data types. For example, the C code below infers operation `AE_ADD32`:

```
ae_int32x2 p0, p1;
ae_int32x2 p = p0 + p1;
```

Table 15: Operators Supported for HiFi 5 DSP on page 64 describes operators supported. Unless noted otherwise, the operators return variables with the same type as the input operand types. If at least one of the input operands has a SIMD type, the return type will also be SIMD.

Table 15: Operators Supported for HiFi 5 DSP

Operator	Operand Types	Operation	Description
+	ae_f32, ae_f32x2,ae_f32x4	AE_ADD32S	Signed saturating 32-bit addition
-	ae_f32, ae_f32x2,ae_f32x4	AE_SUB32S	Signed saturating 32-bit subtraction
-	ae_f32, ae_f32x2,ae_f32x4	AE_NEG32S	Signed saturating 32-bit negation
*	ae_f32x2	AE_MULFP32X2RAS	Signed SIMD fixed-point 1.31x1.31-bit into 1.31-bit multiplication with an `ae_f32x2` return type
*	ae_f32	AE_MULFP32X2RAS	Signed fixed-point 1.31x1.31-bit into 1.31-bit multiplication with an `ae_f32` return type
*	ae_f32x4 * ae_f16x4	AE_MULFP32X16X2RAS.L AE_MULFP32X16X2RAS.H	Signed SIMD fixed-point 1.31x1.15-bit into 1.31-bit multiplication
*	ae_f32 * ae_f16	AE_MULFP32X16X2RAS.H	Signed fixed-point 1.31x1.15-bit into 1.31-bit multiplication
&	ae_f32, ae_f32x2, ae_f32x4, ae_int32, ae_int32x2, ae_int32x4 ae_f24, ae_f24x2, ae_int24, ae_int24x2, ae_f64, ae_int64, ae_f16, ae_f16x4, ae_int16, ae_int16x4, ae_int8, ae_int8x8	AE_AND64	Binary AND

<table>
<tr><th>Operator</th><th>Operand Types</th><th>Operation</th><th>Description</th></tr>
<tr><td>|</td><td>ae_f32, ae_f32x2, ae_f32x4, ae_int32, ae_int32x2, ae_int32x4
ae_f24, ae_f24x2, ae_int24, ae_int24x2,
ae_f64, ae_int64, ae_f16, ae_f16x4,
ae_int16,
ae_int16x4,
ae_int8, ae_int8x8</td><td>AE_OR64</td><td>Binary OR</td></tr>
<tr><td>^</td><td>ae_f32, ae_f32x2, ae_f32x4, ae_int32, ae_int32x2, ae_int32x4
ae_f24, ae_f24x2, ae_int24, ae_int24x2,
ae_f64, ae_int64, ae_f16, ae_f16x4,
ae_int16,
ae_int16x4,
ae_int8, ae_int8x8</td><td>AE_XOR64</td><td>Binary Exclusive OR</td></tr>
<tr><td>~</td><td>ae_f32, ae_f32x2, ae_f32x4, ae_int32, ae_int32x2, ae_int32x4,
ae_f24, ae_f24x2, ae_int24, ae_int24x2,
ae_f64, ae_int64, ae_f16, ae_f16x4,
ae_int16,
ae_int16x4,
ae_int8, ae_int8x8</td><td>AE_NAND64</td><td>Binary NOT</td></tr>
<tr><td>>></td><td>ae_f32, ae_f32x2, ae_f32x4,
ae_int32, ae_int32x2, ae_int32x4,
ae_in24, ae_int24x2</td><td>AE_SRAI32</td><td>Signed arithmetic 32-bit right shift by an immediate shift amount</td></tr>
<tr><td>>></td><td>ae_f32, ae_f32x2, ae_f32x4,
ae_int32, ae_int32x2, ae_int32x4</td><td>AE_SRAA32</td><td>Signed arithmetic 32-bit right shift by a variable shift amount</td></tr>
</table>

Operator	Operand Types	Operation	Description
	ae_int24, ae_int24x2		
<<	ae_f32, ae_f32x2, ae_f32x4	AE_SLAI32S	Signed saturating 32-bit left shift by an immediate shift amount
<<	ae_f32, ae_f32x2, ae_f32x4	AE_SLAA32S	Signed saturating 32-bit left shift by a variable shift amount
<	ae_f32x2, ae_int32x2 ae_f24x2, ae_int24x2,	AE_LT32	Signed less-than comparison with an `xtbool2` return type
<=	ae_f32x2, ae_int32x2 ae_f24x2, ae_int24x2,	AE_LE32	Signed less-than-or-equal comparison with an `xtbool2` return type
==	ae_f32x2, ae_int32x2 ae_f24x2, ae_int24x2,	AE_EQ32	Equal comparison with an `xtbool2` return type
>=	ae_f32x2, ae_int32x2 ae_f24x2, ae_int24x2,	AE_LE32	Signed greater-than-or-equal comparison with an `xtbool2` return type
>	ae_f32x2, ae_int32x2 ae_f24x2, ae_int24x2,	AE_LT32	Signed greater-than comparison with an `xtbool2` return type
+	ae_int32, ae_int32x2, ae_int32x4, ae_int24, ae_int24x2	AE_ADD32	Signed 32-bit addition
-	ae_int32, ae_int32x2, ae_int32x4, ae_int24, ae_int24x2	AE_SUB32	Signed 32-bit subtraction
-	ae_int32, ae_int32x2, ae_int32x4, ae_int24, ae_int24x2	AE_NEG32	Signed 32-bit negation
*	ae_int32x2	AE_MULP32X2	Signed SIMD 32x32 into 32-bit multiplication with an `ae_int32x2` return type
*	ae_int32	AE_MULP32X2	Signed 32x32 into 32-bit multiplication with an `ae_int32` return type
*	ae_int32x4 *ae_int16x4	AE_MULP32X16X2.L AE_MULP32X16X2.H	Signed SIMD 32x16-bit into 32-bit multiplication

Operator	Operand Types	Operation	Description
*	ae_int32 * ae_int16	AE_MULP32X16X2.H	Signed 32x16-bit into 32-bit multiplication
<<	ae_int32, ae_int32x2, ae_int32x4, ae_int24, ae_int24x2	AE_SLAI32	Signed 32-bit left shift by an immediate shift amount.
<<	ae_int32, ae_int32x2, ae_int32x4, ae_int24, ae_int24x2	AE_SLAA32	Signed 32-bit left shift by a variable shift amount
+	ae_f24, ae_f24x2	AE_ADD24S	Signed saturating 24-bit addition.
-	ae_f24, ae_f24x2	AE_SUB24S	Signed saturating 24-bit subtraction.
-	ae_f24, ae_f24x2	AE_NEG24S	Signed saturating 24-bit negation.
*	ae_f24	AE_MULFP24X2RA	Signed SIMD fixed-point 1.23x1.23-bit into 9.23-bit multiplication with an `ae_f32` return type
*	ae_f24x2	AE_MULFP24X2RA	Signed SIMD fixed-point 1.23x1.23-bit into 9.23-bit multiplication with an `ae_f32x2` return type
>>	ae_f24, ae_f24x2	AE_SRAI24	Signed arithmetic 24-bit right shift by an immediate shift amount
>>	ae_f24, ae_f24x2	AE_SRAS24	Signed arithmetic 24-bit right shift by a variable shift amount
<<	ae_f24, ae_f24x2	AE_SLAI24S	Signed saturating 24-bit left shift by an immediate shift amount
<<	ae_f24, ae_f24x2	AE_SLAS24S	Signed saturating 24-bit left shift by a variable shift amount
*	ae_int24x2	AE_MULP32X2	Signed SIMD 32x32 into 32-bit multiplication with an `ae_int32x2` return type
*	ae_int24	AE_MULP32X2	Signed 32x32 into 32-bit multiplication with an `ae_int32` return type
+	ae_f64	AE_ADD64S	Signed saturating 64-bit addition
-	ae_f64	AE_SUB64S	Signed saturating 64-bit subtraction.
-	ae_f64	AE_NEG64S	Signed saturating 64-bit negation
>>	ae_f64, ae_int64	AE_SRAI64	Signed arithmetic 64-bit right shift by an immediate shift amount

Operator	Operand Types	Operation	Description
>>	ae_f64, ae_int64	AE_SRAA64	Signed arithmetic 64-bit right shift by a variable shift amount
<<	ae_f64	AE_SLAI64S	Signed saturating 64-bit left shift by an immediate shift amount.
<<	ae_f64	AE_SLAA64S	Signed saturating 64-bit left shift by a variable shift amount
<	ae_f64, ae_int64	AE_LT64	Signed less-than comparison with an `xtbool` return type
<=	ae_f64, ae_int64	AE_LE64	Signed less-than-or-equal comparison with an `xtbool` return type
==	ae_f64, ae_int64	AE_EQ64	Equal comparison with an `xtbool` return type
>=	ae_f64, ae_int64	AE_LE64	Signed greater-than-or-equal comparison with an `xtbool` return type
>	ae_f64, ae_int64	AE_LT64	Signed greater-than comparison with an `xtbool` return type
+	ae_int64	AE_ADD64	Signed 64-bit addition
-	ae_int64	AE_SUB64	Signed 64-bit subtraction
-	ae_int64	AE_NEG64	Signed 64-bit negation
<<	ae_int64	AE_SLAI64	Signed 64-bit left shift by an immediate shift amount
<<	ae_int64	AE_SLAA64	Signed 64-bit left shift by a variable shift amount
+	ae_f16, ae_f16x4	AE_ADD16S	Signed saturating 16-bit addition
-	ae_f16, ae_f16x4	AE_SUB16S	Signed saturating 16-bit subtraction
-	ae_f16, ae_f16x4	AE_NEG16S	Signed saturating 16-bit negation
*	ae_f16x4	AE_MULF16X4SS	Signed SIMD fixed-point 1.15x1.15-bit into 1.31-bit multiplication with an `ae_f32x4` return type
>>	ae_f16, ae_f16x4	AE_SRAI16	Signed arithmetic 16-bit right shift by an immediate shift amount
>>	ae_f16, ae_f16x4	AE_SRAA16S	Signed saturating arithmetic 16-bit right shift by a variable shift amount
<<	ae_f16, ae_f16x4	AE_SLAI16S	Signed saturating 16-bit left shift by an immediate shift amount

Operator	Operand Types	Operation	Description
<<	ae_f16, ae_f16x4	AE_SLAA16S	Signed saturating 16-bit left shift by a variable shift amount
<	ae_f16x4, ae_int16x4	AE_LT16	Signed less-than comparison with an `xtbool4` return type
<=	ae_f16x4, ae_int16x4	AE_LE16	Signed less-than-or-equal comparison with an `xtbool4` return type
==	ae_f16x4, ae_int16x4	AE_EQ16	Equal comparison with an `xtbool4` return type
>=	ae_f16x4, ae_int16x4	AE_LE16	Signed greater-than-or-equal comparison with an `xtbool4` return type
>	ae_f16x4, ae_int16x4	AE_LT16	Signed greater-than comparison with an `xtbool4` return type
+	ae_int16, ae_int16x4	AE_ADD16	Signed 16-bit addition
-	ae_int16, ae_int16x4	AE_SUB16	Signed 16-bit subtraction
-	ae_int16, ae_int16x4	AE_MOVI, AE_SUB16	Signed 16-bit negation
*	ae_int16x4	AE_MUL16X4	Signed SIMD 16x16 into 32-bit multiplication with an `ae_int32x4` return type
>>	ae_int16, ae_int16x4	AE_SRAI16	Signed 16-bit right shift by an immediate shift amount
>>	ae_int16, ae_int16x4	AE_SRAA16S	Signed 16-bit right shift by a variable shift amount
+	ae_int8, ae_int8x8	AE_ADD8	Signed 8-bit addition
-	ae_int8, ae_int8x8	AE_SUB8	Signed 8-bit subtraction
-	ae_int8, ae_int8x8	AE_MOVI, AE_SUB8	Signed 8-bit negation
>>	ae_int8, ae_int8x8	AE_SRAI8	Signed 8-bit right shift by an immediate shift amount
>>	ae_int8, ae_int8x8	AE_SRAA8S	Signed 8-bit right shift by a variable shift amount
<	ae_int8x8	AE_LT8	Signed less-than comparison with an int8 return type
<=	ae_int8x8	AE_LE8	Signed less-than-or-equal comparison with an int8 return type
==	ae_int8x8	AE_EQ8	Equal comparison with an int8 return type
>=	ae_int8x8	AE_LE8	Signed greater-than-or-equal comparison with an int8 return type

Operator	Operand Types	Operation	Description
>	ae_int8x8	AE_LT8	Signed greater-than comparison with an int8 return type

Table 16: Operators Supported for the Optional Floating Point Unit on page 70 describes the supported operators for the optional floating point unit. Scalar types should all be programmed using the standard C/C++ float support.

Table 16: Operators Supported for the Optional Floating Point Unit

Operator	Operand Types	Operation	Description
+	xtfloatx2	ADD.S	SIMD floating point addition.
-	xtfloatx2	SUB.S	SIMD floating point subtraction.
-	xtfloatx2	NEG.S	SIMD floating point negation.
*	xtfloatx2	MUL.S	SIMD floating point multiplication.
/	xtfloatx2	DIV.S	SIMD floating point division
<	xtfloatx2	OLT.S	SIMD floating point less than comparison
<=	xtfloatx2	OLE.S	SIMD floating point less than or equal comparison
==	xtfloatx2	OEQ.S	SIMD floating point equal comparison

Table 17: Operators Supported for the Legacy HiFi 2 DSP's Data Types on page 70 describes the supported operators for the legacy HiFi 2 data types. Note that the overloading choices for the HiFi 2 types are quite different than for HiFi 5 DSP.

Table 17: Operators Supported for the Legacy HiFi 2 DSP's Data Types

Operator	Operand Types	Operation	Description
+	ae_p24s, ae_p24f, ae_p24x2s, ae_p24x2f	AE_ADDSP24S	Signed saturating 24-bit addition
-	ae_p24s, ae_p24f, ae_p24x2s, ae_p24x2f	AE_SUBSP24S	Signed saturating 24-bit subtraction
-	ae_p24s, ae_p24f, ae_p24x2s, ae_p24x2f	AE_NEGSP24S	Signed saturating 24-bit negation
*	ae_p24s, ae_p24f	AE_MULFP24S.LL	Signed single fixed-point 1.23x1.23-bit into 9.47-bit multiplication with an `ae_q56s` return type

Operator	Operand Types	Operation	Description
*	ae_p24x2s, ae_p24x2f	AE_MULZAAFP24S.HH.LL	Signed dual fixed-point 1.23x1.23-bit into 9.47-bit multiplication with an `ae_q56s` return type
&	ae_p24s, ae_p24f, ae_p24x2s, ae_p24x2f	AE_ANDP48	Binary AND
|	ae_p24s, ae_p24f, ae_p24x2s, ae_p24x2f	AE_ORP48	Binary OR
^	ae_p24s, ae_p24f, ae_p24x2s, ae_p24x2f	AE_XORP48	Binary Exclusive OR
~	ae_p24s, ae_p24f, ae_p24x2s, ae_p24x2f	AE_NANDP48	Binary NOT
>>	ae_p24s, ae_p24f, ae_p24x2s, ae_p24x2f	AE_SRAIP24	Signed arithmetic 24-bit right shift by an immediate shift amount
>>	ae_p24s, ae_p24f, ae_p24x2s, ae_p24x2f	AE_SRASP24	Signed arithmetic 24-bit right shift by a variable shift amount
<<	ae_p24s, ae_p24f, ae_p24x2s, ae_p24x2f	AE_SLLISP24S	Signed saturating 24-bit left shift by an immediate shift amount
<<	ae_p24s, ae_p24f, ae_p24x2s, ae_p24x2f	AE_SLLSSP24S	Signed saturating 24-bit left shift by a variable shift amount
<	ae_p24x2s, ae_p24x2f	AE_LTP24S	Signed less-than comparison with an `xtbool2` return type
<=	ae_p24x2s, ae_p24x2f	AE_LEP24S	Signed less-than-or-equal comparison with an `xtbool2` return type
==	ae_p24x2s, ae_p24x2f	AE_EQP24	Equal comparison with an `xtbool2` return type
>=	ae_p24x2s, ae_p24x2f	AE_LEP24S	Signed greater-than-or-equal comparison with an `xtbool2` return type

Operator	Operand Types	Operation	Description
>	ae_p24x2s, ae_p24x2f	AE_LTP24S	Signed greater-than comparison with an `xtbool2` return type
+	ae_q56s	AE_ADDQ56	56-bit addition
-	ae_q56s	AE_SUBQ56	56-bit subtraction
-	ae_q56s	AE_NEGQ56	56-bit negation
&	ae_q56s	AE_ANDQ56	Binary AND
|	ae_q56s	AE_ORQ56	Binary OR
^	ae_q56s	AE_XORQ56	Binary Exclusive OR
~	ae_q56s	AE_NANDQ56	Binary NOT
>>	ae_q56s	AE_SRAIQ56	Signed arithmetic 56-bit right shift by an immediate shift amount
>>	ae_q56s	AE_SRAAQ56	Signed arithmetic 56-bit right shift by a variable shift amount
<<	ae_q56s	AE_SLLIQ56	56-bit left shift by an immediate shift amount
<<	ae_q56s	AE_SLLAQ56	56-bit left shift by a variable shift amount
<	ae_q56s	AE_LTQ56S	Signed less-than comparison with an `xtbool` return type
<=	ae_q56s	AE_LEQ56S	Signed less-than-or-equal comparison with an `xtbool` return type
==	ae_q56s	AE_EQQ56	Equal comparison with an `xtbool` return type
>=	ae_q56s	AE_LEQ56S	Signed greater-than-or-equal comparison with an `xtbool` return type

Operator	Operand Types	Operation	Description
>	ae_q56s	AE_LTQ56S	Signed greater-than comparison with an `xtbool` return type.

Note that all the non-legacy multiply overloads produce results of the same low precision as the operands. This is because there are no high-precision SIMD multiplies. The high-precision dual multiplies in HiFi 5 DSP add (or subtract) together the two multiply results into a single result, and it is less natural to define the semantics of multiplying two ae_f24x2 variables, for example, to be a single ae_f64 that is the dot-product of the two variables. This is in contrast to the legacy HiFi 2/EP data types, such as ae_p24x2f, where multiplying two such variables does indeed do a dot product. Those semantics were chosen because HiFi 2/EP has no true SIMD multiplies.

If you want to use the new, high-precision multiplies in HiFi 3, HiFi 4, or HiFi 5 DSPs you must use intrinsics.

Operator Overloading: Energy Calculation Example

Consider our energy calculation example where the input data is stored in memory as a 1.31 fixed point value. The following code is a standard C reference code.

```
    s = 0;
    for (i=0; i<n; i++)
{
      s += ((long long) a[i]*a[i]) >> 31;
    }
    return s;
```

The code can be converted into HiFi 5 DSP code as follows.

```
ae_f32 *ap = (ae_f32 *)a;
ae_f32 s = 0;
for (i=0; i<n; i++) {
  s += ap[i]*ap[i];
}
return s;
```

The main loop uses operator overloading to perform a 32-bit, fixed point multiply. The `ae_f32` typed array is implicitly loaded, just like any standard C/C++ type. The accumulator is of type `ae_f32`. The assignment of the result to an `int` does not change the bit pattern. Hence this routine returns a 1.31 value stored as an `int`.

The compiler generates the following inner loop.

```
loop a3, L
{
```

```
            ae_l32.ip    aed0,a2,4
            nop
            ae_mulafp32x2ras aed1,aed0,aed0
     }
     L:
```

HiFi 5 DSP can issue a multiply and a load every cycle. Note that the compiler automatically generates the multiply-add instruction, `ae_mulafp32x2ras`.

HiFi 5 DSP can perform two multiplies and two loads per cycle. Even ignoring the lack of SIMD, this code should be able to run twice as fast. The problem is that we are accumulating into a single accumulator. We cannot issue more than one multiply accumulate into that accumulator every cycle. However, we can change the code as follows to use two partial sums:

```
ae_f32 s = 0;
ae_f32 s2 = 0;
for (i=0; i<n; i+=2) {
      s += ap[i]*ap[i];
      s2 += ap[i+1]*ap[i+1];
}
return s+s2;
```

Now the compiler generates a perfect inner loop that does two loads and two multiply-accumulates per cycle:

```
loop a3, L

{      # format ae_format88
       ae_l32.i aed0,a2,-4
       ae_l32.ip aed2,a2,8
       ae_mulafp32x2ras aed1,aed0,aed0
       ae_mulafp32x2ras_s2 aed3,aed2,aed2
 }
L:
```

However, note that if the multiply instructions saturate, using partial sums is not bit-exact to the original code.

The inner loop is perfect, except that no SIMD is used. By changing `ae_f32` into `ae_f32x2`, and cutting the trip count in half, we convert the example into a 2-way SIMD example. The main loop is computing four partial sums in parallel. After the loop, we must add together the four partial sums into a single sum using a SIMD add followed by the `AE_ADD32_HL_LH` intrinsic.

```
ae_f32x2 *ap = (ae_f32x2 *) a;
ae_f32x2 s, s2 = 0;

for (i = 0; i < n>>1; i+=2)
{
        s = s + ap[i]*ap[i];
        s2 = s2 + ap[i+1]*ap[i+1];
```

```
    }
    s = s+s2;
    return AE_ADD32_HL_LH(s,s);
```

The compiler generates the following inner loop.

```
    loop a3, L
    {
          ae_l32x2.i aed0,a2,-8 ae_l32x2.ip
              aed2,a2,  ae_mulafp32x2ras aed1,  aed0,aed0
              ae_mulafp32x2ras_s2    aed3,aed2,aed2
     }
     L:
```

The generated code can now perform four multiplies every cycle.

Note that the optimized code assumes that n is a multiple of four. If that is not guaranteed, the last iteration[s] of the loop must be executed separately using scalar code.

This example code uses fixed point arithmetic. If you want to use integral arithmetic instead, simply use the integral rather than the fixed-point types.

Operator Overloading: 32X16-bit Dot Product Example

Consider now a scenario where we wish to do 32x16-bit multiplication, rather than 32x32-bit multiplication. An energy calculation only has a single input operand, while the 32x16-bit requires two. So, we convert our energy example into a dot product. Because four 16-bit elements can fit into a register, we vectorize by four rather than by two. The number of elements in the 32-bit operand must be the same as the number of elements in the 16-bit operand. Therefore, the HiFi 5 DSP defines an `ae_f32x4` (and an `ae_int32x4`) data type. These are structure data types that occupy two registers. Most operations defined on these types result in two instructions, thus they are no faster than the 2-way SIMD types. However, their use is necessary when doing 32x16-bit multiplication using operator overloading. In the following example, note that the result is reduced into a single int using the `AE_F32X4_RADD` intrinsic. This is a convenience intrinsic that translates into a 3-instruction sequence.

```
        ae_f32x4 *ap = (ae_f32x4 *) a;
        ae_f16x4 *bp = (ae_f16x4 *) b;
        ae_f32x4 s,s2 = 0;

        for (i = 0; i < n>>2; i+=2)
        {
                s += bp[i]*ap[i];
                s2 += bp[i+1]*ap[i+1];
        }
        return AE_F32X4_RADD(s+s2);
```

3.7 Intrinsic-Based Programming

The next programming style is to use explicit intrinsics. Even if operator overloading is not sufficient, it may not be necessary to use intrinsics everywhere, as the compiler may, for example, infer the right vector loads and stores. Sometimes adding just a few strategic intrinsics may be sufficient to achieve maximum efficiency. The compiler can still be counted on for efficient scheduling and optimization.

Every HiFi 5 DSP instruction can be directly accessed by an intrinsic of the same name (except that "." in instruction names is converted into "_" in intrinsic names). See Chapter 2 for the list of prototypes of the supported intrinsics, along with the instruction descriptions.

Consider a simple example that does a 32-bit, fixed-point energy calculation, but wants to keep all the intermediate results in high precision. Operator overloading always uses the low-precision multipliers. Therefore, we must use intrinsics for the multiply.

```
ae_f32x2 *ap = (ae_f32x2 *) a;
ae_f64 s = AE_ZERO64();

for (i = 0; i < n>>1; i++)
{
         AE_MULAAFD32S_HH_LL(s, ap[i], ap[i]);
}
ae_f32 result = AE_ROUND32F64SASYM(s);
```

In addition to the dual-multiply intrinsic, intrinsics are used to initialize the accumulator to zero, and to round the final result back down to 32 bits. Note that we have chosen to use a 1.31x1.31 into 1.63-bit multiply instruction. If we need guard bits, we can also chose to perform a 1.31x1.31 into 17.47-bit multiply instruction or use the 72 bits available with the EP instructions.

Note the following points:

- There is no need to use explicit vector loads.
- The intrinsics are not assembly operations. They need not be manually scheduled into FLIX bundles. Variables need not be manually allocated into particular registers, the compiler takes care of that (and the code still remains quite "C-like"). The compiler generates a perfect inner loop with a dual, updating load and a dual multiply instruction.
- The compiler will automatically select load/store instructions, but programmers may in some cases be able to optimize results using their own selection, by using the correct intrinsic instead of leaving it to the compiler

Consider now a similar example where the operand is stored in the circular buffer. The assumption is that the operand array might cross the end of the buffer. After loading the last element in the buffer, the code needs to continue to the first element. There is no way to implicitly utilize the circular buffer load instructions; you need to use the explicit load intrinsics as shown in the following code.

```
ae_f32x2 tmp;
ae_f32x2 *ap = (ae_f32x2 *) a;
ae_f64 s = AE_ZERO64();

for (i = 0; i < n>>1; i++)
{
        AE_L32X2_XC(tmp, ap, 8);
        AE_MULAAFD32_HH_LL(s, tmp, tmp);
}
ae_f32 result = AE_ROUND32F64SASYM (s);
```

The operand pointer is loaded using the updating, circular load intrinsic, `AE_L32X2_XC`. This example assumes that the boundaries of the circular buffer have been set elsewhere.

See *Audio DSP Examples* on page 143 for more details.

3.8 Code Portability from Previous HiFi DSPs

This section details the ways to port code to the HiFi 5 DSP.

Porting Code to the HiFi 5 DSP

The HiFi 5 DSP implements all HiFi 2, HiFi 3 and HiFi 4 C types, intrinsics and operator overloads to ensure that existing C and C++ target source code can compile and run on a HiFi 5 DSP processor with some exceptions listed in this section. Similarly, all the C types, intrinsics and operator overloads of HiFi 3z are also implmented on HiFi 5 DSP, except AE_CALCRNG3 (details below). HiFi 2, HiFi 3, or HiFi 4 assembly target code must be manually modified to build and run on a HiFi 5 DSP. The HiFi 5 DSP ISA is not binary compatible with previous HiFi DSPs. i.e., a binary compiled on a previous HiFi DSPs will not execute correctly on a HiFi 5 DSP.

The HiFi Mini DSP supports 2-way SIMD 8-bit load instructions AE_LP8X2F.I and AE_LP8X2F.IU that have no equivalent on HiFi 5 DSP. The HiFi 5 DSP instead supports 8-bit data type natively and has instruction to load eight 8-bit elements. HiFi Mini also supports some core operations that are intended to be inferred by the compiler and not used as intrinsics. These operations are also not supported on HiFi 5 DSP.

Guidelines for porting the HiFi 2, HiFi 3, HiFi 3z, or HiFi 4 DSP target code to the HiFi 5 DSP:

There is no precision difference between HiFi 4 and HiFi 5. For all other HiFi DSPs, the HiFi 5 behaviour matches the HiFi 4 DSP, hence all the guidelines for porting to HiFi 4 are also applicable for porting to HiFi 5.

- *Mapping*:

HiFi 5 implements all the instructions from HiFi 2, HiFi 3 and HiFi 4 DSPs. HiFi 5 also implements all instructions except AE_CALCRNG3 from HiFi 3z DSP. To ensure efficient HiFi 5 DSP hardware implementation, some are implemented as compatibility intrinsics. Refer to *Standard DSP Operations by Type* on page 81 for notes on the operation and intrinsic mapping.

If the application uses any of the following deprecated operations, they must be modified.

- **AE_MUL[|A|S][R]FQ32SP24S[.H|.L]**: HiFi 4 and HiFi 5 do not support the 32x24-bit MAC (AE_MUL[|A|S][R]FQ32SP24S[.H|.L]) of HiFi 2 EP, HiFi 3, and HiFi 3z. If these instructions are used in the code, it may not compile on HiFi 5. The programmer must modify the code to make use of 32X32 instructions as appropriate.
- **AE_MUL[A]FC[R|I]32RAS**: On HiFi 3z DSP, AE_MUL[A]FC32RAS is implemented as proto using AE_MUL[A]FCR32RAS and AE_MUL[A]FCI32RAS. HiFi 5 DSP implements AE_MUL[A]FC32RAS as a operation, hence the AE_MUL[A]FCR32RAS and and AE_MUL[A]FCI32RAS are not implemented.
- **AE_MUL[Z]SA64P24S.HL.LH and AE_MUL[Z]SAF48P24S.HL.LH**: These intrinsics from HiFi 3 DSP use 24-bit arithmetic deprecated on HiFi 4 and HiFi 5, hence are not supported.

The HiFi 3z DSP supports AE_CALCRNG3, which computes the shift value for the dynamic range determined for 16 bitstream. Similarly the HiFi 4 DSP also implements AE_CALCRNG1, AE_CALCRNG2, and AE_CALCRNG3, which compute the shift value for the dynamic range determined for 32 bitstream. HiFi 5 provides better support for both 32 bit and 16 bit through new instructions AE_CALCRNG32 and AE_CALCRNG16, respectively. HiFi 5 DSP implements all the three CALCRNG instructions of HiFi 4 DSP as compatibility intrinsics using AE_CALCRNG32 operation. It is not implementing the AE_CALCRNG3 of HiFi 3z. If the FFT implementation on HiFi 3z uses AE_CALCRNG3, that section of code need to be modifed to use AE_CALCRNG16. Refer to *FFT (Fast Fourier Transform) Operations* on page 121 and *Complex FFT Example* on page 146 for details about AE_CALCRNG32 and AE_CALCRNG16.

- *Precision*:

  Some HiFi 2 and HiFi 3-specific intrinsics (DSP operations, loads and stores) provide wider precision than the intrinsics available in the HiFi 2/3/3z ISA. For example, the `AE_ADDP24` intrinsic is implemented through operation `AE_ADD32`—if a computation overflowed the 24 bits in HiFi 2, in HiFi 5 DSP the computation will maintain the extra precision in the 8 MSBs of each 32-bit AE_DR element. If a HiFi 2 application code assumes wraparound due to the limited register width, it may need to be fixed to ensure correct execution on HiFi 5 DSP. HiFi 5 DSP does not support high precision 24-bit multiplication. Such HiFi 2/3/3z intrinsics are implemented using 32-bit multiplications. On HiFi 5 DSP, fractional 1.31x1.31->1.63 saturate, whereas on HiFi 2 and HiFi 3 1.23x1.23->17.47 (HiFi 3) or 9.47 (HiFi 2) does not. Therefore, very long multiply-accumulate sequences that would have wrapped around the extra 8 or 16 guard bits on HiFi 2/3 will instead saturate on the HiFi 5 DSP.
- *Performance*:

To ensure efficient HiFi 5 DSP hardware implementation, some HiFi 2 intrinsics that map to a single operation in the HiFi 2 ISA are implemented through a sequence of two or more operations in HiFi 5 DSP. For example, HiFi 2 intrinsic `AE_MULZASFQ32SP16S_HH` is implemented through a sequence of four operations in HiFi 5 DSP . If a HiFi 2 application relies on such intrinsics, it may need to be manually re-optimized to ensure efficient execution on HiFi 5 DSP. In many cases however, the additional VLIW slots, extra registers, and MACs provided by HiFi 5 DSP will be sufficient to compensate.

Like HiFi 3z and HiFi 4 DSP processors, HiFi 5 also supports dual load and simultaneous load and store. HiFi 5 also supports instruction to load/store 128 bits, i.e., two AE_DR registers. To accommodate the 128-bit load/store, the data bus width is increased to 128 bit / 256 bits, while all other DSPs supports 64 bits data bus and single DR register load/ store. This difference causes different memory access pattern and memory stalls. The implementation that assumes the 64-bit width memory banks may introduce more stalls on HiFi 5. The programmer may need to re-tune such implementations to avoid performance degradation.

3.9 Important Compiler Switches

The following compiler switches are important:

- `-mcoproc` – as discussed in the *Xtensa C Application Programmer's Guide* and *Xtensa XT-CLANG User's Guide.* In particular for HiFi 5 DSP, the use of this flag allows the compiler to emulate standard C/C++ operations using the HiFi 5 DSP instructions. In comparison to HiFi 2/EP, this flag may have a much larger impact on HiFi 5 DSP.
- Optimization level – When optimizing code, the code should be compiled with either the `-O2` or `-O3` level of optimization. On average, `-O3` will give higher performance, but not always. It is recommended that critical functions be compiled both ways to compare performance.
- Compiling for code size – Less performance critical functions should be compiled with `-Os` (in addition to either `-O2` or `-O3`). This will meaningfully shrink the code size required. In addition to saving on memory, smaller code might improve performance on real systems with more limited instruction cache sizes.

4. Standard DSP Operations by Type

Topics:

- *Load and Store Operations Overview*
- *Load Operations*
- *Store Operations*
- *Core Load and Store Operations*
- *Multiply and Accumulate Operations Overview*
- *32x32-bit Multiplication Operations*
- *32x16-bit Multiplication Operations*
- *16x16-bit Multiplication Operations*
- *32x16-bit Legacy Multiplication Operations*
- *16x16-bit Legacy Multiplication Operations*
- *Neural Networks Multiplication Operations*
- *Add, Subtract and Compare Operations*
- *Shift Operations*
- *Normalization Operations*
- *Divide Step Operations*
- *Truncate Operations*
- *Round Operations*
- *Saturate Operations*
- *Convert Operations*
- *Move Operations*
- *Pack-Shift-Round Operations*
- *FFT (Fast Fourier Transform) Operations*

This section serves as a quick overview of the groups of HiFi operations.

Note: Refer to the HiFi 5 DSP ISA HTML for detailed description of operations.

4.1 Load and Store Operations Overview

HiFi 5 DSP supports loading and storing scalars, vectors or dual vectors of 8, 16, 24, 32, and 64 bit elements.

- Each scalar load/store accesses 8, 16, 24, 32, or 64 bits
- Each vector accesses 64 bits or 48 bits for packed 24-bit data
- Each dual vector accesses 128 bits into two data registers

Vector loads and stores, content at the high address in memory is always stored in the least significant bits in the register. Reverse vector loads and stores reverse the elements in a register so that the content at low address in memory is stored in the least significant bits in the register. This way, whether accessing data in a stride one or stride negative one fashion, the earliest data to be accessed is always in the same position in the register.

Data caches and internal data memory 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 cannot 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. Due to Dual vector load, the data bus width should be at least 128 bit. If the data is consequently placed and accessed through the load instructions issues simultaneously, this could cause memory banking conflict for single vector loads and scalar loads.

Special support is provided for retaining full throughput when vectors of data are not aligned to 64 bits (or 128-bits in case of dual load/store.) HiFi 5 DSP also supports three circular buffers that can be used with either aligned or unaligned data.

Aligning Loads and Stores

HiFi 5 DSP has support for loading or storing vector streams of data 64 bits at a time even if the data is not aligned to 64 bits. Note that while the vector variables need not be aligned to 64 bits, they must still be aligned according to the requirements of each scalar element, *i.e.,* 32 bits for vectors of ints.

Such loads and stores are called aligning loads and stores. Support is available for 8-, 16-, 24-, and 32-bit data. The aligning vector load and store instructions use the HiFi 5 DSP alignment register file to provide a throughput of one aligning load or store operation per instruction.

A special priming instruction, AE_LA64.PP, is used to begin the process of loading an array of unaligned data. This instruction loads the alignment register with data from the start of the stream. The subsequent aligning load instruction loads from the next location in memory, merging it with the data already in the alignment register. The exact details of how the aligning instructions work are not relevant to the programmer. Simply invoke the AE_LA64_PP priming intrinsic with the first address (aligned or not) to be loaded and continue loading with the appropriate aligning loads to achieve a subsequent throughput of one aligning load per instruction.

The design of the priming load and aligning load instructions is such that they can be used in situations where the alignment of the address is unknown. The load sequence works whether the starting address is aligned or not.

Consider a simple example that adds up the 32-bit elements in an array.

```
void add(int * a, int n)
{
  ae_int32x2 *ap=(ae_int32x2 *) &a[0];
  ae_int32x2 tmp;
  extern ae_int32x2 V;
  ae_valign align;
  int i;

  align = AE_LA64_PP(ap);  // prime the stream
  for(i = 0; i < n; i = i + 2)
  {
    AE_LA32X2_IP(tmp,align,ap); // load the next element
    V = V + tmp;
  }
}
```

Similarly, when accessing the data with a stride of negative one, prime the stream by passing in the address of the first scalar element to be loaded `(a[n-1])`, as follows.

```
void add(int * a, int n)
{
  ae_int32x2 *ap=(ae_int32x2 *) &a[n-1];
  ae_int32x2 tmp;
  extern ae_int32x2 V;
  ae_valign align;
  int i;

  align = AE_LA64_PP(ap);  // prime the stream
  for(i = 0; i < n; i = i + 2)
  {
    AE_LA32X2_RIP(tmp,align,ap); // load the next element
    V = V + tmp;
  }
}
```

Note that in the negative stride case, the start of the stream is handled differently in the aligned versus the non-aligned case. With aligned loads, one passes in the address of `a[n-2]` because that is the address of the first 64-bit word being loaded. With aligning loads, one passes in the address of the first 32-bit scalar being loaded, `a[n-1]`, because the priming load loads from memory, the aligned 64-bit envelope containing its argument and `a[n-2]` might not be in the same 64-bit envelope as `a[n-1]`.

HiFi 5 DSP supports storing 24-bit data in a packed format that requires only 24 bits per data element. Using this support can potentially save 25% of the memory required for a 24-bit variable. Support for this packed data is implemented using the alignment mechanism. In the examples above, simply use `AE_LA24X2` intrinsic instead of `AE_LA32X2` as shown below. Note that we have used `char *` for the pointer type. While not strictly necessary, it is helpful to indicate that the packed stream is an unaligned byte stream.

```
void add(int * a, int n)
{
  char *ap=(char *) &a[0];
  ae_int24x2 tmp;
  extern ae_int32x2 V;
  ae_valign align;
  int i;

  align = AE_LA64_PP(ap);  // prime the stream
  for(i = 0; i < n; i = i + 2)
  {
    AE_LA24X2_IP(tmp,align,ap); // load the next element
    V = V + (ae_int32x2) tmp;
  }
}
```

For packed data, even scalar streams are unaligned, so support is also available for `AE_LA24` intrinsics. Because the memory format for packed data is different, packed data can only be used in cases where all loads and stores of a stream are done using the packing loads and stores. While the packing loads and stores can be used on any 24-bit variable, since a priming load and a finalizing store is required for every stream, it is often only efficient to use them on stride one or stride negative one streams. Similarly, since there are only four alignment registers, it is only efficient to use them on loops that have at most four streams.

Aligning stores operate in a slightly different manner. Before starting a stream, the alignment variable needs to be zeroed using the `AE_ZALIGN64()` intrinsic. On an unaligned store, each aligning store instruction merges some of the data with data already in the alignment register and writes the result to memory. The remaining data is written into the alignment register for use in the next aligning store. If the data happens to be aligned, each aligning store simply writes its data to memory. After completing the stream, you must finalize the stream using a finalization instruction. If the data happens to be unaligned, that finalization instruction writes out the remaining data from the alignment register. The finalization instruction also zeroes the alignment register so that a follow-on stream can skip the use of the `AE_ZALIGN64()` intrinsic.

Following is a simple example that zeroes an `n` element array of `ints` named `a`.

```
ae_int32x2 V_con = (ae_int32x2)(0);
ae_int32x2 *addr = (ae_int32x2 *) a;
ae_valign align = AE_ZALIGN64();  // zero alignment reg
for(i = 0; i <= n; i = i + 2)
{
  AE_SA32X2_IP(V_con, align, addr); // store
```

```
}
AE_SA64POS_FP(align, addr); // finalize the stream
```

Negative strided streams work analogously to the case of loads, with the use of `RIP` intrinsics. Note that there are separate flush instructions for the positive stride and negative stride streams.

Aligning load/store is also supported for dual-vector load/store operations (128-bit). In this case, the priming is done using AE_LA128.PP. negative stride is not supported by 128-bit aligning load/store operations. The usage of 120-bit aligning load/store operations is same as the 64-bit aligning load/store operations explained below, except that they use different set of priming and flushing instructions.

Circular Buffers

HiFi 5 DSP has support for three circular buffers, which can be accessed in either the forward or the backward direction.

The circular buffer boundaries are specified through four 32-bit states.

Table 18: Circular Buffer States

State	Description
AE_CBEGIN0	The start address of the circular buffer.
AE_CEND0	The end address of circular buffer, *i.e.*, the start address plus the byte size of the buffer.
AE_CBEGIN1	The start address of the second circular buffer.
AE_CEND1	The end address of the second circular buffer, *i.e.*, the start address plus the byte size of the buffer.
AE_CBEGIN2	The start address of the third circular buffer.
AE_CEND2	The end address of third circular buffer, i.e., the start address plus the byte size of the third buffer.

Use the following intrinsic functions to read from the circular buffer states in C:

```
void * AE_GETCBEGIN0 (void);
void * AE_GETCEND0 (void);
void * AE_GETCBEGIN1 (void);
void * AE_GETCEND1 (void);
void * AE_GETCBEGIN2 (void);
void * AE_GETCEND2 (void);
```

Use the following intrinsic functions to write to the circular buffer states in C:

```
void AE_SETCBEGIN0 (const void * addr);
void AE_SETCEND0 (const void * addr);
void AE_SETCBEGIN1 (const void * addr);
void AE_SETCEND1 (const void * addr);
void AE_SETCBEGIN2 (const void * addr);
void AE_SETCEND2 (const void * addr);
```

All circular buffer operations follow a “post-increment” convention; that is, in every case the effective address is the base address while the updated base address is formed by adding the register offset to the base address with circular wrap-around.

The address increment is specified in terms of number of bytes. The increment can be either positive (wrap-around at the end of the buffer), or negative (wrap-around at the beginning of the buffer).

Both aligned and unaligned accesses are supported. However for unaligned accesses, `AE_CBEGIN0`, `AE_CBEGIN1`, `AE_CEND0`, and `AE_CEND1`, `AE_CEND2`, and `AE_CEND2` must all be aligned to 64 bits for single vector load/store or 128 bit for dual vector laod/store. For aligned accesses, `AE_CBEGIN0`, `AE_CBEGIN1`, `AE_CBEGIN2`,`AE_CEND0`,`AE_CEND1`, and `AE_CEND2` must all be aligned to the size of the data being loaded or stored. Unaligned accesses use the alignment mechanism described in *Aligning Loads and Stores* on page 83. Priming loads use the PC or PC1 suffix with separate instructions for positive and negative stride. For unaligned references, only stride one and stride negative one are supported. Packed 24-bit loads are supported. 128-bit load instructions don't support negative stride.

`AE_CBEGIN0` need not be smaller than `AE_CEND0`.If an instruction accesses data past the `AE_CEND0` boundary, data will continue to be accessed at `AE_CBEGIN0` regardless of whether it is before or after `AE_CEND0`.

Circular buffer support is available for DSP loads and stores to the `AE_DR` register file as well as bitstream loads and stores to the AR register file.

Following is an example C code snippet demonstrating how to initialize and use the first circular buffer. The buffer is used to store 24-bit vector data in the 24 MSBs of each 32-bit word with a negative stride starting from the last element of the buffer.

```
/* Allocate the buffers. */
void *buf = malloc(buf_size);

/* Initialize the circular buffer boundaries. */
AE_SETCBEGIN0(buf);
AE_SETCEND0(buf + buf_size);

/* Point to the first element to be loaded/stored. */
ae_f24x2 *buf_ptr = (ae_f24x2 *)(buf + buf_size - sizeof(ae_f24x2));
…
for (…) {
  ae_f24x2 p;
  …
  AE_S32X2F24_XC(p, buf_ptr, -sizeof(ae_f24x2));
```

```
    …
}
```

Circular buffers can also be used to support Ping-Pong buffers. `AE_CEND` can be set to the end of the current buffer, while `AE_CBEGIN` is set to the beginning of the other buffer.

Load and Store Naming Scheme

The mnemonic of most load and store operations contains a size indicating the size of operands it will load or store. The sizes are listed in *Table 19: Load and Store Operation Sizes* on page 88.

Table 19: Load and Store Operation Sizes

Size	Definition	Description
16	16-bit scalar	This operation accesses an aligned 16-bit quantity.
24	24-bit scalar	This operation accesses a 24-bit quantity that is packed into memory so as to occupy only 24 bits in memory.
32	32-bit scalar	This operation accesses an aligned 32 bit quantity. This size is also used for legacy 24-bit integers, which are stored in a 32-bit memory location right-justified and with 8 bits of sign extension.
32F24	Left-justified 24-bit fraction	This operation accesses a 24-bit fraction, which is stored left-justified in a 32-bit memory location. It shifts the value right by 8 bits and sign extends on the left by 8 bits. The address must be 32-bit aligned.
64	64-bit scalar	This operation accesses an aligned 64-bit quantity.
24X2	Vector of two 24-bit elements	This operation accesses two of the size “24” above, occupying 48 bits in memory.
32X2	Vector of two 32-bit elements	This operation accesses two of the size “32” above. Some instructions need the pair to be 64-bit aligned while others do not.
32X2F24	Vector of left-justified 24-bit fraction	This operation accesses two of the size “32F24” above. Some instructions need the pair to be 64-bit aligned while others do not.
16X4	Vector of four 16 bit elements	This operation accesses four of the size “16” above. Some instructions need the quartet to be 64-bit aligned while others do not.
8X4F	Vector of four left-justified 8 bit fraction into 16 bits	This operation accesses four aligned 8-bit values and converts them into four 1.15-bit values by adding eight zeroes to the bottom of each element. The address must be 32-bit aligned.

Size	Definition	Description
8X4S	Vector of four right-justified 8 bit integer sign extended into 16 bits.	This operation accesses four aligned 8-bit values and converts them into four 16-bit integer values by sign extending them. The address must be 32-bit aligned.
8X4U	Vector of four right-justified 8 bit integer zero extended into 16 bits.	This operation accesses four aligned 8-bit values and converts them into four 16-bit integer values by zero extending them. The address must be 32-bit aligned.
8	8-bit scalar	This operation accesses an aligned 8-bit quantity
8X8	Vector of eight 8-bit elements	This operation accesses eight of the size "8" above. Some instructions need the octet to be 64-bit aligned while others do not.
64X2	Vector of two 64-bit elements	This operation accesses two of the size "64" above. Some instructions need the pair to be 64-bit aligned while others do not
32X2X2	Two vectors of two 32-bit elements.	This operation accesses two of the size "32X2" above.
16X4X2	Two vectors of four 16-bit elements	This operation accesses two of the size "16X4" above.
8X8X2	Two vectors of eight 8-bit elements	This operation accesses two of the size "8X8" above.

The mnemonic of most load and store operations contains a suffix indicating how the effective address is computed and whether the base address register is updated. The suffixes are listed in the following table.

Operations with suffix `IP`, `XP`, `IC`, `IC1`, `XC`, `XC1`, or `XC2` follow a “post-increment” convention where the effective address is the base AR register, and the base address register is updated by adding an immediate, constant or register offset. Operations with suffix `IU` or `XU` follow a “pre-increment” convention where the effective address is the result of adding the immediate or register offset to the base address register’s contents and the base address register is updated with the effective address. Operations with suffix `I` or `X` do not increment, but create an effective address which is the sum of the base address register and an immediate or offset register. See *Table 20: Load and Store Operation Suffixes* on page 89 for load and store operation suffixes.

Table 20: Load and Store Operation Suffixes

Suffix & Definition	Effective Address	Base Reg Update	Description
I Immediate	Reg + immed	[none]	The effective address is a base AR register plus an immediate value. The base AR register is not updated.

Suffix & Definition	Effective Address	Base Reg Update	Description
X Indexed	Reg + Reg	[none]	The effective address is a base AR register plus an index AR register value. The base AR register is not updated.
IP Post Update Immediate	Reg	Reg + Immed	The effective address is a base AR register. The base AR register is updated with the base AR register plus an immediate or constant value.
XP Post Update Indexed	Reg	Reg + Reg	The effective address is a base AR register. The base AR register is updated with the base AR register plus an offset AR register value.
IC Post Update Implied Immediate with Circular buffer	Reg	Reg + Const folded back into circular buffer	The effective address is a base AR register. The base AR register is updated with the base AR register plus a positive constant value equal to one element. If the address is less than AE_CEND0 and the updated value is greater than or equal to AE_CEND0, then AE_CEND0-AE_CBEGIN0 is subtracted from it.
IC1 Post Update Implied Immediate with Circular buffer	Reg	Reg + Const folded back into circular buffer	The effective address is a base AR register. The base AR register is updated with the base AR register plus a positive constant value equal to one element. If the address is less than AE_CEND1 and the updated value is greater than or equal to AE_CEND1, then AE_CEND1-AE_CBEGIN1 is subtracted from it.
XC Post Update Indexed with Circular Buffer	Reg	Reg + Reg folded back into circular buffer	The effective address is base AR register. The base AR register is updated with the base AR register plus an offset AR register value. For positive updates, if the address is less than AE_CEND0 and the updated value is greater than or equal to AE_CEND0, then AE_CEND0-AE_CBEGIN0 is subtracted from it. For negative updates, if the address is greater than or equal to AE_CBEGIN0 and the updated value is less than AE_CBEGIN0, then AE_CEND0-AE_CBEGIN0 is added to it.
XC1 Post Update Indexed with Circular Buffer	Reg	Reg + Reg folded back into circular buffer	The effective address is base AR register. The base AR register is updated with the base AR register plus an offset AR register value. For positive updates, if the address is less than

Suffix & Definition	Effective Address	Base Reg Update	Description
			AE_CEND1 and the updated value is greater than or equal to AE_CEND1, then AE_CEND1-AE_CBEGIN1 is subtracted from it. For negative updates, if the address is greater than or equal to AE_CBEGIN1 and the updated value is less than AE_CBEGIN1, then AE_CEND1-AE_CBEGIN1 is added to it.
RI Reverse Immediate	Reg + Immed	[none]	The effective address is a base AR register plus an immediate value. The base AR register is not updated. The vector elements in the result register are also swapped.
RIP Reverse Post Update	Reg	Reg + Immed	The effective address is a base AR register. The base AR register is updated with the base AR register minus an immediate. For some operations, the immediate is implicitly equal to the size of the element being loaded or stored. The vector elements in the result register are also swapped.
RIC Reverse Post Update Implied Immediate with Circular buffer	Reg	Reg + Const folded back into circular buffer	The effective address is a base AR register. The base AR register is updated with the base AR register minus a positive constant value equal to one element. If the address is greater than or equal to AE_CBEGIN0 and the updated value is less than AE_CBEGIN0, then AE_CEND0-AE_CBEGIN0 is added to it. The vector elements in the result register are also swapped.
RIC1 Reverse Post Update Implied Immediate with Circular buffer	Reg	Reg + Const folded back into circular buffer	The effective address is a base AR register. The base AR register is updated with the base AR register minus a positive constant value equal to one element. If the address is greater than or equal to AE_CBEGIN1 and the updated value is less than AE_CBEGIN1, then AE_CEND1-AE_CBEGIN1 is added to it. The vector elements in the result register are also swapped.
PP Prime	See Instruction	See Instruction	This addressing mode is used for priming instructions which set up the beginning of an unaligned load sequence.

Suffix & Definition	Effective Address	Base Reg Update	Description
PC Circular Prime	See Instruction	See Instruction	This addressing mode is used for priming instructions which set up the beginning of an unaligned load sequence in a circular buffer.
PC1 Circular Prime	See Instruction	See Instruction	This addressing mode is used for priming instructions which set up the beginning of an unaligned load sequence in the second circular buffer.
FP Flush	See Instruction	See Instruction	This addressing mode is used for flushing the last part of an unaligned store sequence
IU Immediate with Update	Reg + Immed	Reg + Immed	The effective address is a base AR register plus an immediate value. The base AR register is updated with the effective address. These instructions are used for legacy HiFi 2/EP operations only.
XU Indexed with Update	Reg + Reg	Reg + Reg	The effective address is a base AR register plus an offset AR register value. The base AR register is updated with the effective address. These instructions are used for legacy HiFi 2/EP operations only.
IC2 Post Update Implied Immediate with Circular buffer	Reg	Reg + Const folded back into circular buffer	The effective address is a base AR register. The base AR register is updated with the base AR register plus a positive constant value equal to one element. If the address is less than AE_CEND2 and the updated value is greater than or equal to AE_CEND2, then AE_CEND2-AE_CBEGIN2 is subtracted from it.
XC2 Post Update Indexed with Circular Buffer	Reg	Reg + Reg folded back into circular buffer	The effective address is base AR register. The base AR register is updated with the base AR register plus an offset AR register value. For positive updates, if the address is less than AE_CEND2 and the updated value is greater than or equal to AE_CEND2, then AE_CEND2-AE_CBEGIN2 is subtracted from it. For negative updates, if the address is greater than or equal to AE_CBEGIN2 and the updated value is less than AE_CBEGIN2, then AE_CEND2-AE_CBEGIN2 is added to it.
PC2 Circular Prime	See Instruction	See Instruction	This addressing mode is used for priming instructions which set up the beginning of an

Suffix & Definition	Effective Address	Base Reg Update	Description
			unaligned load sequence in the third circular buffer.

4.2 Load Operations

Table 21: Load Overview on page 93 gives an overview of the various types of load operations. The first column indicates a set of load operations which includes all those with the size <sz> and the address mode <adr> replaced by any of the values in the second and third columns. The fourth column summarizes the purpose of that group of operations.

Table 21: Load Overview

Instruction	Size <sz>	Suffix <adr>	Purpose
AE_L<sz>.<adr>	64, 32, 32F24, 16, 8	I, X, IP, XP, XC, XC1, XC2	Aligned loads of scalars
AE_L<sz>.<adr>	32X2, 32X2F24, 16X4	I, X, IP, RI, RIP, XP, XC, XC1,, XC2, RIC, RIC1	Aligned loads of vectors
AE_LA<sz>.<adr>	64, 128	PP	Prime for Unaligned loads using IP
AE_L<sz>.<adr>	8X4F, 8X4U, 8X4S	I, IP X, XP	Aligned loads of vectors
AE_LA<sz>POS.<adr>	32X2, 16X4, 24, 24X2, 8x8, 32X2F24, 64X2, 32X2X2, 16X4X2, 8X8X2	PC	Prime for Unaligned loads using IC with positive stride
AE_LA<sz>NEG.<adr>	32X2, 16X4, 24, 24X2, 8X8, 32X2F24	PC	Prime for Unaligned loads using IC with negative stride
AE_LA<sz>POS.<adr>	32X2, 16X4, 24, 24X2, 8x8, 32X2F24, 64X2, 32X2X2, 16X4X2, 8X8X2	PC1	Prime for Unaligned loads using IC1 with positive stride
AE_LA<sz>NEG.<adr>	32X2, 16X4, 24, 24X2,8X8, 32X2F24	PC1	Prime for Unaligned loads using IC1 with negative stride
AE_LA<sz>.<adr>	32X2, 32X2F24, 16X4, 24, 24X2, 8X8	IP, IC, IC1, IC2	Unaligned loads for accessing vectors of aligned scalars with positive update

Instruction	Size <sz>	Suffix <adr>	Purpose
AE_LA<sz>.<adr>	32X2, 32X2F24, 16X4, 24, 24X2, 8X8	RIP, RIC, RIC1	Unaligned loads for accessing vectors of aligned scalars with negative update
AE_LALIGN<sz>.I	64,128		Load of alignment register
AE_L<sz>M.<adr>	16X2, 32, 16	I, X, XC, XC1, IU, XU XC2	Legacy Loads
AE_L<sz>.<adr>	64X2, 32X2X2, 16X4X2, 8X8X2, 8X8	I, X, IP, XP, XC, XC1, XC2	Aligned loads of vectors
AE_LA<sz>POS.<adr>	32X2, 16X4, 24, 24X2, 8x8, 32X2F24, 64X2, 32X2X2, 16X4X2, 8X8X2	PC2	Prime for Unaligned loads using IC2 with positive stride
AE_LA<sz>_<adr>	16X4X2, 32X2X2, 64X2, 8X8X2	IP, IC, IC1, IC2	Unaligned loads for accessing vectors of aligned scalars with positive update
AE_LA<sz>_<adr>	8X4S, 8X4U	IP, XP	Aligned loads of vectors
AE_L<sz>.<adr>	16S, 16U	I.N	Narrow Loads
AE_L_<sz><adr>	32,16,16U		Scalar Loads
AE_LAV<sz>_<adr>	8X8X2, 16X4X2, 32X2X2	XP	Variable elements Load. These operations are available only when Neural Network Extension is selected.
AE_LAVUNSQZ<sz>_<adr>	16X4, 8X8	XP	Variable elements Load and Unsqueeze These operations are available only when Neural Network Extension is selected.

4.3 Store Operations

Table 22: Store Operations Overview on page 95 gives an overview of the various types of store instructions. The first column indicates a set of store instructions which includes all those with the size <sz> and the address mode <adr> replaced by any of the values in the

second and third columns. The fourth column summarizes the purpose of that group of instructions.

Table 22: Store Operations Overview

Instruction	Size <sz>	Suffix <adr>	Purpose
AE_S<sz>.<adr>	64	I, X, IP, XP, XC, XC1, XC2	Aligned stores of scalars
AE_S<sz>.<adr>	32X2, 16X4 8X8, 32X2F24	I, X, IP, XP, XC, XC1, XC2, RIP, RIC, RIC1 RIP, RIC, RIC1 variants are not available for 8X8	Aligned stores of vectors
AE_S<sz>.<adr>	8X4U, 8X4UX2	I, X, IP, XP	Aligned stores of vectors. It is a vector store operation that picks four least significant 8-bit values from a 16x4 vector operand and stores the resultant 8x4 vector to memory.
AE_S<sz>.L.<adr>	32, 32F24, 16,8	I, X, IP, XP, XC, XC1, XC2	Aligned stores of scalars from the low part of a register. 16-bit variant uses .0 suffix instead of .L (AE_S16.0.<adr>). 8-bit variant does not use any suffix, instead the index is passed as immediate value.
AE_S<sz>.<adr>	32RA64S, 24RA64S	I, IP, X, XP, XC, XC1	Aligned stores of scalars from the middle part of a register with rounding and saturation
AE_S<sz>.<adr>	32X2RA64S, 24X2RA64S16X4,16X4RA32S	IP	Aligned stores of two scalars from the middle part of a register with rounding and saturation
AE_SA<sz>.<adr>	32X2, 32X2F24, 16X4, 8x8, 24X2,	IP, IC, IC1,IC2, RIP, RIC, RIC1	Unaligned stores for accessing vectors of aligned scalars

Instruction	Size <sz>	Suffix <adr>	Purpose
AE_SA<sz>POS.FP	64 ,128		Flush after unaligned store with positive stride
AE_SA64NEG.FP			Flush after unaligned store with negative stride
AE_SALIGN<sz>.I	64,128		Store of alignment register
AE_ZALIGN64			Zero alignment register
AE_S<sz>M.<adr>	16X2, 32, 16	I, X, XC, XC1, XC2, IU, XU (no XC1 for S32M)	Legacy stores, 16-bit variant has a .L Suffix (AE_S16M.L.<adr>)
AE_SA<sz>.L.<adr>	24	IP,IC,IC1,RIP,RIC,RIC1,IC2	Legacy 24-bit store
AE_S<sz>.<adr>	64X2,32X2X2,16X4X2,8x8x2	I, X, IP, XP, XC, XC1,XC2	Aligned stores of dual vectors
AE_SA<sz>_<adr>	8X8X2,16X4X2, 32X2X2,64X2	IP,IC,IC1,IC2	Aligning stores of dual vectors
AE_S<sz>.H.<adr>	32	I, X, IP, XP, XC, XC1	Aligned stores of scalars from the higher part of a register
AE_SAV<sz>_<adr>	8X8X2, 16X4X2, 32X2X2	XP	Variable elements store These operations are available only when Neural Network Extension is selected.

4.4 Core Load and Store Operations

HiFi 5 DSP provides three instructions AE_L16SI.N, AE_L16UI.N, and AE_S16I.N that are limited immediate versions of the core L16SI, L16UI, and S16I instructions respectively. These instructions are inferred automatically by the C/C++ compiler.

4.5 Multiply and Accumulate Operations Overview

The HiFi 5 DSP ISA supports a rich collection of single, dual, and quad multiply/accumulate operations with different input and output precision, scaling, rounding and saturation modes.

HiFi 5 supports up to:

- Eight 32x32-bit multiply-accumulate operations per cycle
- Sixteen 32x16-bit multiply-accumulate operations per cycle
- Sixteen 16x16-bit multiply-accumulate operations per cycle

The following types of multiplication are supported:

- Integer multiplication with and without saturation
- Fractional multiplication with and without rounding and/or saturation accumulations
- Point-wise SIMD multiplication
- Multiplications that perform partial or full dot product
- Sliding multiplications useful in FIR or correlation type operations
- Multiplications useful in multiply-accumulate of complex and complex conjugate numbers

To ensure source level software compatibility, all the legacy HiFi 2/EP, HiFi 3, HiFi 3z, and HiFi 4 multiply operations (including 24-bit multiply) are supported; some are provided as intrinsics.

If the Neural Network (NN) Extension option is selected, it comes with a set of low-precision integer 8-bit and 16-bit multiplications useful in neural network implemenations. HiFi 5 with the NN Extension can support up to:

- thirty-two 8x16 multiply and accumulate per cycle
- thirty-two 4x16 multiply and accumulate per cycle
- thirty-two 8x8 multiply and accumulate per cycle

The following different types of low precision integer multiplications are included when NN Extension option is enabled.

- Multiplications useful in Matrix-Vector and dot product of 4-bit, 8-bit and 16-bit data
- Multiplications useful in convolution of 4-bit, 8-bit, and 16-bit data
- Signed and unsigned multiplications of 4-bit, 8-bit, and signed 16-bit numbers
- Asymmetrically quantized signed and unsigned 8-bit data type support for multiplications.

The accumulators used in these multplications are mostly 32-bit and 64-bit accumulators

The naming conventions used for the multiply instructions included in base HiFi 5 config and the NN Extension of HiFi 5 is explained in *Instruction Naming Conventions* on page 38.

An overview for the multiplication types is provided in the subsequent subsections.

Following are the instruction description conventions used:

- Two-Way Single MUL acc_32: This indicates the instruction results in two accumulator outputs, each containing a 32-bit integer value resulting from single multiply or multiply and accumulate operation.
- Two-Way Dual MUL acc_1.31: This indicates the instruction results in two accumulator outputs, each containing a 1.31 fractional value resulting from dual multiply or dual-multiply and accumulate operation.

- Two-Way Single CMUL acc_32 (or simply Two-Way CMUL acc_32): This indicates the instruction results in two complex accumulator outputs, each containing two 32-bit integer values (real and imaginary part) resulting from a complex multiply or complex-multiply and accumulate operation. Therefore a single CMUL operation indicates there are 2 dual element multiply and sub/add operations to create real and imaginary outputs.

4.6 32x32-bit Multiplication Operations

The input operands for 32x32-bit multiplication are elements of `AE_DR` registers. Each `AE_DR` register holds two 32-bit elements for each `AE_DR` register operand to a multiplication; one of the two elements must be selected as the input to the multiplication through an `H` or an `L` suffix. The result of each multiply/accumulate operation goes into an `AE_DR` register. Some instructions that perform four or eight MACs accept two AE_DR registers for each operand. This section also lists operations that support 32x32-bit into 72-bit multiplication.

Table 23: Fractional 32X32

One-Way Single MUL acc_1.63	One-way Single MUL acc_17.47	Two-Way Single MUL acc_1.31
AE_MULF32S .[LL|LH|HH] , AE_MULAF32S .[LL|LH|HH]	AE_MULF32R.[LL|LH|HH] , AE_MULAF32R.[LL|LH|HH] , AE_MULSF32R.[LL|LH|HH] , AE_MULF32RA.[LL|LH|HH] , AE_MULAF32RA.[LL|LH|HH] , AE_MULSF32RA.[LL|LH|HH]	AE_MULFP32X2RS, AE_MULFP32X2RAS, AE_MULFP32X2TS , AE_MULAFP32X2RS, AE_MULAFP32X2RAS, AE_MULAFP32X2TS , AE_MULSFP32X2RS, AE_MULSFP32X2RAS, AE_MULSFP32X2TS

One-Way Dual MUL acc_1.63	One-Way Dual MUL acc_17.47	Two-way Dual MUL acc_1.31 (Mul result add-sub into different accumulator)
AE_MULZAAFD32S. [HH.LL | HL.LH], AE_MULAAFD32S. [HH.LL | HL.LH] , AE_MULZASFD32S. [HH.LL | HL.LH], AE_MULASFD32S. [HH.LL | HL.LH] , AE_MULZSAFD32S. HH.LL, AE_MULSAFD32S. HH.LL , AE_MULZSSFD32S. [HH.LL | HL.LH], AE_MULSSFD32S. [HH.LL | HL.LH]	AE_MULZAAFD32RA. [HH.LL| HL.LH], AE_MULZASFD32RA. [HH.LL|HL.LH], AE_MULZSAFD32RA. HH.LL, AE_MULZSSFD32RA. [HH.LL| HL.LH], AE_MULAAFD32RA. [HH.LL|HL.LH], AE_MULASFD32RA. [HH.LL| HL.LH], AE_MULSAFD32RA. HH.LL, AE_MULSSFD32RA. [HH.LL|HL.LH]	AE_MULADDF32RAS, AE_MULSUBF32RAS, AE_MULADDF32RS, AE_MULSUBF32RS

Two-Way Single MUL acc_1.63	Two-Way Single MUL acc_17.47	Four-way Single MUL acc_1.31
AE_MULFP32X2S.[HH.LL|HL.LH], AE_MULAFP32X2S.[HH.LL|HL.LH], AE_MULSFP32X2S. [HH.LL|HL.LH]	AE_MULF32X2RA. [HH.LL|HL.LH], AE_MULAF32X2RA. [HH.LL|HL.LH], AE_MULSF32X2RA. [HH.LL|	AE_MULF2P32X4RAS, AE_MULAF2P32X4RAS, AE_MULSF2P32X4RAS,

Two-Way Single MUL acc_1.63	Two-Way Single MUL acc_17.47	Four-way Single MUL acc_1.31
	HL.LH] , AE_MULF32X2R. [HH.LL|HL.LH], AE_MULAF32X2R. [HH.LL|HL.LH], AE_MULSF32X2R. [HH.LL|HL.LH]	AE_MULF2P32X4RS, AE_MULAF2P32X4RS, AE_MULSF2P32X4RS

Two-Way Dual MUL acc_1.63	Two-Way Dual MUL acc_17.47
AE_MULZAAF2D32S. [HH.LL|HL_LH] , AE_MULZASF2D32S. [HH.LL|HL_LH] , AE_MULZSAF2D32S. [HH.LL|HL_LH] , AE_MULZSSF2D32S. [HH.LL|HL_LH] , AE_MULAAF2D32S. [HH.LL|HL_LH] , AE_MULASF2D32S. [HH.LL|HL_LH] , AE_MULSAF2D32S. [HH.LL|HL_LH] , AE_MULSSF2D32S. [HH.LL|HL_LH]	AE_MULZAAF2D32RA. [HH.LL|HL_LH] , AE_MULZASF2D32RA. [HH.LL|HL_LH] , AE_MULZSAF2D32RA. [HH.LL|HL_LH] , AE_MULZSSF2D32RA. [HH.LL|HL_LH] , AE_MULAAF2D32RA. [HH.LL|HL_LH] , AE_MULASF2D32RA. [HH.LL|HL_LH] , AE_MULSAF2D32RA. [HH.LL|HL_LH] , AE_MULSSF2D32RA. [HH.LL|HL_LH]

Special FIR Instructions		
Two-Way Dual FIRMUL acc_1.63 ()	**Two-Way Dual FIRMUL acc_17.47 (Special FIR instructions)**	**Two-Way Dual MUL acc_1.63 (Point-wise Op {a*b + c*d})**
AE_MULFD32X2S.FIR. [H|L] , AE_MULAFD32X2S.FIR. [H|L]	AE_MULFD32X2RA.FIR. [H|L], AE_MULAFD32X2RA.FIR. [H|L]	AE_MULF2D32X2WS

Complex MUL and Complex Conjugate MUL Instructions		
One-Way CMUL acc_1.63	**One-Way CMUL acc_17.47**	**One-Way CMUL acc_1.31**
AE_MULFC32W, AE_MULAFC32W, AE_MULFCJ32W, AE_MULAFCJ32W	AE_MULFC32RA, AE_MULAFC32RA,	AE_MULFC32RAS, AE_MULAFC32RAS, AE_MULFCJ32RAS, AE_MULAFCJ32RAS

Table 24: Integer 32X32

One-Way Single MUL acc_64	One-Way Single MUL acc_64 (saturated)
AE_MUL32. [LL|LH|HH] , AE_MULA32. [LL|LH|HH] , AE_MULS32. [LL|LH|HH]	AE_MUL32S. [LL|LH|HL|HH] , AE_MULA32S. [LL|LH|HL|HH] , AE_MULS32S. [LL|LH|HL|HH]

One-Way Dual MUL acc_64	Two-Way Single MUL acc_32 (Results from Top or bottom half of 32bits)
AE_MULZAAD32. [HH.LL|HL_LH], AE_MULZASD32. [HH.LL|HL_LH], AE_MULZSAD32. HH.LL, AE_MULZSSD32. [HH.LL|HL_LH], AE_MULAAD32. [HH.LL|HL_LH], AE_MULASD32. [HH.LL|HL_LH], AE_MULSAD32. HH.LL, AE_MULSSD32. [HH.LL|HL_LH]	AE_MULP32X2, AE_MULAP32X2, AE_MULSP32X2 , AE_MULP32X2T, AE_MULAP32X2T, AE_MULSP32X2T

One-Way Dual MUL acc_64 (saturated)	Two-Way Single MUL acc_64 (saturated)
AE_MULZAAD32S. [HH.LL|HL_LH], AE_MULZASD32S. [HH.LL|HL_LH], AE_MULZSAD32S. [HH.LL,|HL.LH], AE_MULZSSD32S. [HH.LL|HL_LH], AE_MULAAD32S. [HH.LL|HL_LH], AE_MULASD32S. [HH.LL|HL_LH], AE_MULSAD32S. [HH.LL|HL_LH], AE_MULSSD32S. [HH.LL|HL_LH]	AE_MUL32X2S. [HH.LL|HL.LH], AE_MULA32X2S. [HH.LL|HL.LH], AE_MULS32X2S. [HH.LL|HL.LH]

Two-Way Single MUL acc_64
AE_MULZAA32X2.HH.LL, AE_MULZSS32X2. [HH.LL], AE_MULAA32X2.HH.LL, AE_MULSS32X2.HH.LL

ELEMENT-WISE		
Two-Way Single MUL acc_32 (saturated)	**Four-Way Single MUL acc_32 (saturated)**	**Four-Way Single MUL acc_32 (Results from Top or bottom half of 32bits)**
AE_MULP32X2S	AE_MUL2P32X4S	AE_MUL2P32X4, AE_MULA2P32X4, AE_MULS2P32X4, AE_MUL2P32X4T, AE_MULA2P32X4T, AE_MULS2P32X4T

Complex MUL and Complex Conjugate MUL	
One-way CMUL acc_64	**One-way CMUL acc_32 (with truncation)**
AE_MULC32W, AE_MULAC32W, AE_MULCJ32W, AE_MULACJ32W	AE_MULC32, AE_MULAC32, AE_MULCJ32, AE_MULACJ32

EP MUL	
One-Way Single MUL acc_72 (72-bit wide results from 32x32 signed)	**One-Way Single MUL acc_72 (72-bit results 32x32 unsigned x signed)**
AE_MUL32EP. HH, AE_MULA32EP. HH, AE_MULS32EP. HH	AE_MUL32USEP.LL, AE_MUL32USEP.LH , AE_MULA32USEP.LH

One-Way Dual MUL acc_72 (72-bit wide results from 32x32 signed)	One-Way Dual MUL acc_72 (72-bit results 32x32 unsigned x signed)
AE_MULZAAD32EP.HH.LL , AE_MULZSSD32EP.HH.LL , AE_MULAAD32EP.HH.LL , AE_MULSSD32EP.HH.LL	AE_MULZAAD32USEP.HL.LH , AE_MULAAD32USEP.HL.LH

Special
Non-MUL OP (Complex zout = j*zin)
AE_MUL32JS

4.7 32x16-bit Multiplication Operations

The input operands for 32x16-bit multiplication operations are elements of `AE_DR` registers. The first multiplicand holds two 32-bit elements. The second multiplicand holds four 16-bit elements. For operations that allow operand selection within a register, each 32-bit operand is specified through an `H` or `L` suffix and each 16-bit operand is selected through a 3, 2, 1, or 0 suffix.

Table 25: Fractional 32X16

One-Way Single MUL acc_17.47	Two-way Single MUL acc_1.31	Two-Way Single MUL acc_17.47
AE_MULF32X16.[L|H][0|1|2|3], AE_MULAF32X16.[L|H][0|1|2|3], AE_MULSF32X16.[L|H][0|1|2|3,	AE_MULFP32X16X2S. [L|H, AE_MULFP32X16X2RS.[L|H], AE_MULFP32X16X2RAS.[L|H], AE_MULAFP32X16X2S.[L|H], AE_MULAFP32X16X2RS.[L|H], AE_MULAFP32X16X2RAS.[L|H], AE_MULSFP32X16X2S.[L|H], AE_MULSFP32X16X2RS.[L|H], AE_MULSFP32X16X2RAS.[L|H]	AE_MULFP32X16.[L|H], AE_MULAFP32X16.[L|H], AE_MULSFP32X16.[L|H]

One-Way Dual MUL acc_17.47	One-Way Quad MUL acc_17.47
AE_MULZAAFD32X16.[H0.L1|H1.L0|H2.L3|H3.L2], AE_MULZASFD32X16.[H1.L0|H3.L2], AE_MULZSAFD32X16.[H1.L0|H3.L2], AE_MULZSSFD32X16.[H1.L0|H3.L2], AE_MULAAFD32X16[H0.L1|H1.L0|H2.L3|H3.L2], AE_MULASFD32X16[H1.L0|H3.L2], AE_MULSAFD32X16[H1.L0|H3.L2], AE_MULSSFD32X16[H1.L0|H3.L2]	AE_MULZAAAAFQ32X16, AE_MULAAAAFQ32X16

ELEMENT-WISE
Four-Way Single MUL acc_1.31
AE_MULF2P32X16X4S, AE_MULF2P32X16X4RS, AE_MULF2P32X16X4RAS, AE_MULAF2P32X16X4S, AE_MULAF2P32X16X4RS, AE_MULAF2P32X16X4RAS, AE_MULSF2P32X16X4S, AE_MULSF2P32X16X4RS, AE_MULSF2P32X16X4RAS

Special FIR instructions	
Two-Way Dual FIRMUL acc_17.47	Two-Way Quad FIRMUL acc_2.62
AE_MULFD32X16X2.FIR.[LL|LH|HL|HH], AE_MULAFD32X16X2.FIR.[LL|LH|HL|HH]	AE_MUL2Q32X16.FIR.[L|H], AE_MULA2Q32X16.FIR.[L|H]

Complex MUL and Complex Conjugate MUL instructions		
One-Way CMUL acc_1.31	One-Way CMUL acc_17.47	Two-Way CMUL acc_1.31
AE_MULFC32X16RAS.[L|H], AE_MULAFC32X16RAS.[L|H]	AE_MULFC32X16W.[L|H], AE_MULAFC32X16W.[L|H], AE_MULFCJ32X16W.[L|H], AE_MULAFCJ32X16W.[L|H]	AE_MULFPC32X16X2RAS, AE_MULAFPC32X16X2RAS, AE_MULFPCJ32X16X2RAS, AE_MULAFPCJ32X16X2RAS

Table 26: Integer 32X16

One-Way Single MUL acc_64	**Two-Way Single MUL acc_32**
AE_MUL32X16.[L|H].[0|1|2|3], AE_MULA32X16.[L|H].[0|1|2|3], AE_MULS32X16.[L|H].[0|1|2|3]	AE_MULP32X16X2.[L|H], AE_MULAP32X16X2.[L|H], AE_MULSP32X16X2.[L|H]

One-Way Dual MUL acc_64
AE_MULAAD32X16.[H0.L1|H1.L0|H2.L3|H3.L2], AE_MULZAAD32X16[H0.L1|H1.L0|H2.L3|H3.L2], AE_MULZASD32X16.[H1.L0|H3.L2], AE_MULZSAD32X16[H1.L0|H3.L2], AE_MULZSSD32X16[H1.L0|H3.L2], AE_MULASD32X16[H1.L0|H3.L2], AE_MULSAD32X16[H1.L0|H3.L2], AE_MULSSD32X16[H1.L0|H3.L2]

One-Way Quad MUL acc_64	**Two-Way Quad MUL acc_64**
AE_MULAAAAQ32X16, AE_MULZAAAAQ32X16	AE_MULAAAA2Q32X16, AE_MULZAAAA2Q32X16

Complex MUL and Complex Conjugate MUL instructions		
One-way CMUL acc_32 (with truncation)	One-way CMUL acc_64	Two-way CMUL acc_32 (with truncation)
AE_MULAC32X16.[L|H], AE_MULC32X16.[L|H]	AE_MULC32X16W.[L|H], AE_MULAC32X16W.[L|H]	AE_MULPC32X16X2, AE_MULAPC32X16X2

4.8 16x16-bit Multiplication Operations

The input operands for 16x16-bit multiplication operations are elements of AE_DR registers. Each AE_DR register holds four 16-bit elements; for each AE_DR register operand to a multiplication, one of the four elements must be selected as the input to the multiplication through a 3, 2, 1, or 0 suffix. If the pair of (3, 2) or (1, 0) are used as operand to the multiplication, they are denoted by suffix H and L respectively.

Table 27: Fractional 16x16

One-Way Single MUL acc_1.31	One-way Dual MUL acc_1.31	Two-Way Dual MUL acc_1.31
AE_MULAF16SS.[00|10|11|20|21|22|30|31|32|33] AE_MULF16SS[00|10|11|20|21|22|30|31|32|33] AE_MULSF16SS[00|10|11|20|21|22|30|31|32|33]	AE_MULAAFD16SS.[11_00|13_02|33_22] AE_MULZAAFD16SS.[11_00|13_02|33_22] AE_MULSSFD16SS.[11_00|13_02|33_22] AE_MULZSSFD16SS.[11_00|13_02|33_22]	AE_MULAAFD16SS.[HH_LL|HL_LH] AE_MULZAAFD16SS.[HH_LL|HL_LH] AE_MULSSFD16SS.[HH_LL|HL_LH] AE_MULZSSFD16SS.[HH_LL|HL_LH]

Four-Way Dual MUL acc_1.15	Four-Way Dual MUL acc_1.31
AE_MULFD16X16X4RAS	AE_MULFD16X16X4WS

ELEMENT-WISE	
Four-Way Single MUL acc_1.15	**Four-Way Single MUL acc_1.31**
AE_MULFP16X4S AE_MULFP16X4RS AE_MULFP16X4RAS	AE_MULF16X4SS AE_MULAF16X4SS AE_MULSF16X4SS

Special FIR instructions
Two-Way Quad FIRMUL acc_33.31
AE_MULFQ16X2.FIR.[0|1|2|3] AE_MULAFQ16X2.FIR.[0|1|2|3]

Complex and Complex Conjugate MUL instructions	
Two-Way CMUL acc_1.15	Two-Way CMUL acc_1.31
AE_MULFC16RAS	AE_MULFC16S
AE_MULAFC16RAS	AE_MULAFC16S

Complex and Complex Conjugate MUL instructions	
Two-Way CMUL acc_1.15	Two-Way CMUL acc_1.31
AE_MULFCJ16RAS	AE_MULFCJ16S
AE_MULAFCJ16RAS	AE_MULAFCJ16S

Table 28: Integer 16x16

One-Way Single MUL acc_32	Two-Way Single MUL acc_32	Two-Way Dual MUL acc_32
AE_MUL16.00 AE_MULA16.00 AE_MUL16S.00 AE_MULA16S.00 AE_MULS16S.00	AE_MULP16S.[L|H] AE_MULAP16S.[L|H] AE_MULSP16S.[L|H]	AE_MULAA2D16SS.[HH_LL|HL_LH] AE_MULZAA2D16SS.[HH_LL|HL_LH] AE_MULSS2D16SS.[HH_LL|HL_LH] AE_MULZSS2D16SS.[HH_LL|HL_LH]

One-Way Quad MUL acc_64	Two-Way Quad MUL acc_64
AE_MULAAAAQ16, AE_MULZAAAAQ16	AE_MULAAAA2Q16, AE_MULZAAAA2Q16, AE_MULAAAA2Q16X8, AE_MULZAAAA2Q16X8

ELEMENT-WISE	
Four-Way Single MUL acc_16	**Four-Way Single MUL acc_32**
AE_MULP16X16X4S, AE_MULAP16X16X4S, AE_MULSP16X16X4S	AE_MUL16X4, AE_MULA16X4, AE_MULS16X4, AE_MUL16X4S, AE_MULA16X4S, AE_MULS16X4S

Complex and Complex Conjugate MUL instructions		
One-way CMUL acc_32 (with saturation)	One-way CMUL acc_64	Two-way CMUL acc_16 (with saturation)
AE_MULC16S.[L|H], AE_MULAC16S.[L|H], AE_MULC16JS.[L|H], AE_MULAC16JS.[L|H]	AE_MULC16W.[L|H], AE_MULAC16W.[L|H]	AE_MULC16S, AE_MULAC16S

Two-way CMUL acc_32 (with saturation)
AE_MUL2C16S, AE_MULA2C16S

Special
Non-MUL OP (Complex zout = j*zin)
AE_MUL16JS

4.9 32x16-bit Legacy Multiplication Operations

HiFi 5 DSP provides a basic set of legacy 32x16-bit MAC operations for efficient execution of HiFi 2 target code. The legacy 32- and 16-bit operand formats can only store half as many elements in a register; therefore they are less efficient than the HiFi 4-specific 32x16-bit operations. The 32-bit input operand comes from bits 47 through 16 of the AE_DR register.

The 16-bit input operand comes from bits 23 through 8 of the L 32-bit AE_DR element.

The following legacy intrinsics are provided on HiFi 5, either as operations or implemented as intrinsics.

Table 29: Legacy 32x16 operations

One-Way Single MUL acc_17.47 (fractional)	One-Way Single MUL acc_64 (Integer)
AE_MULF48Q32SP16S.L, AE_MULF48Q32SP16U.L, AE_MULAF48Q32SP16S.L, AE_MULAF48Q32SP16U.L, AE_MULSF48Q32SP16S.L, AE_MULSF48Q32SP16U.L AE_MULFQ32SP16S.[H|L] , AE_MULAFQ32SP16S.[H|L] , AE_MULSFQ32SP16S.[H|L] AE_MULFQ32SP16U.[H|L] , AE_MULAFQ32SP16U.[H|L] , AE_MULSFQ32SP16U.[H|L]	AE_MULAQ32SP16S.L, AE_MULAQ32SP16U.L, AE_MULQ32SP16S.L, AE_MULQ32SP16U.L, AE_MULSQ32SP16S.L, AE_MULSQ32SP16U.L AE_MULQ32SP16S.H , AE_MULAQ32SP16S.H , AE_MULSQ32SP16S.H AE_MULQ32SP16U.H , AE_MULAQ32SP16U.H , AE_MULSQ32SP16U.H

One-Way Dual MUL acc_17.47 (fractional)	One-Way Dual MUL acc_64 (Integer)
AE_MULZAAFQ32SP16S.[HH|LH|LL] AE_MULZAAFQ32SP16U.[HH|LH|LL] AE_MULZASFQ32SP16S.[HH|LH|LL] AE_MULZASFQ32SP16U.[HH|LH|LL] AE_MULZSAFQ32SP16S.[HH|LH|LL]	AE_MULZAAQ32SP16S.[HH|LH|LL] AE_MULZAAQ32SP16U.[HH|LH|LL] AE_MULZASQ32SP16S.[HH|LH|LL] AE_MULZASQ32SP16U.[HH|LH|LL] AE_MULZSAQ32SP16S.[HH|LH|LL]

One-Way Dual MUL acc_17.47 (fractional)	One-Way Dual MUL acc_64 (Integer)
AE_MULZSAFQ32SP16U.[HH|LH|LL] AE_MULZSSFQ32SP16S.[HH|LH|LL] AE_MULZSSFQ32SP16U.[HH|LH|LL]	AE_MULZSAQ32SP16U.[HH|LH|LL] AE_MULZSSQ32SP16S.[HH|LH|LL] AE_MULZSSQ32SP16U.[HH|LH|LL]

4.10 16x16-bit Legacy Multiplication Operations

The input operands for legacy 16x16-bit multiplication operations are elements of AE_DR registers. Each AE_DR register holds two 16-bit elements; for each AE_DR register operand to a multiplication, one of the two elements must be selected as the input to the multiplication through an H or an L suffix. The result of each multiply/accumulate operation goes into an AE_DR register.

The following legacy intrinsics are provided on HiFi 5, either as operations or implemented as intrinsics.

Table 30: 16x16-bit Legacy Multiplication Operations

One-Way Single MUL acc_17.47 (fractional)
AE_MULS32F48P16S.[LL|LH|HH|HL]
AE_MULAS32F48P16S.[LL|LH|HH|HL]
AE_MULSS32F48P16S.[LL|LH|HH|HL]

One-Way Single MUL (integer)
AE_MULFS32P16S.LL[.LH|.HH|.HL]
AE_MULAFS32P16S.LL[.LH|.HH|.HL]
AE_MULSFS32P16S.LL[.LH|.HH|.HL]

4.11 Neural Networks Multiplication Operations

The input operands for all low precision multiplication operations are elements of AE_DR registers. Higher and lower 4 elements of 8-bit or eight elements of 4-bit operands are specified through H or L suffix. HH, HL, LH, LL are the suffixes used for 4 element segment of 4-bit operands.

With RI-2021.6 Xtensa tools release, the following operations are added to enable asymmetric int8 quantization support and to improve Depth-wise separable convolution computation. Operations for Depth-wise separable convolution support both asymmetric signed and unsigned 8-bit data type.

- Instructions to enable asymmetric int8 quantization support
  - Convolution operations:
    - AE_MUL[A]ZB8Q8X8CNV_[H|L]
    - AE_MUL[A]ZB2X4Q8X8CNV_[H|L]
    - AE_MUL[A]ZB4O8X8CNV_[H|L]
  - Matrix vector multiplication operations:
    - AE_MUL[A]ZB8Q8X8
    - AE_MUL[A]ZB4O8X8
- Instructions to enable Depth-wise separable Convolution computations
  - AE_MUL[A]ZB3X3O8X8
  - AE_MUL[A]UUZB3X3O8X8

CNV suffix refers to convolution operation.

Table 31: Multiply Vector Multiplication/MAC Operations

Precision	Octal MAC - Quad 32-bit output	Quad MAC Octal 32-bit Output
8X8	AE_MUL8Q8X8	AE_MUL4O8X8
	AE_MULA8Q8X8	AE_MULA4O8X8
	AE_MULUU8Q8X8	AE_MULUU4O8X8
	AE_MULAUU8Q8X8	AE_MULAUU4O8X8
	AE_MULUS8Q8X8	AE_MULUS4O8X8
	AE_MULAUS8Q8X8	AE_MULAUS4O8X8
	AE_MULSU8Q8X8	AE_MULSU4O8X8
	AE_MULASU8Q8X8	AE_MULASU4O8X8
Asymmetric unsigned and signed 8-bit	AE_MULUUZB8Q8X8	AE_MULUUZB4O8X8
	AE_MULAUUZB8Q8X8	AE_MULAUUZB4O8X8
	AE_MULZB8Q8X8	AE_MULZB4O8X8
	AE_MULAZB8Q8X8	AE_MULAZB4O8X8
4X16	AE_MUL8Q4X16	AE_MUL4O4X16
	AE_MULA8Q4X16	AE_MULA4O4X16
	AE_MULUS8Q4X16	AE_MULUS4O4X16
	AE_MULAUS8Q4X16	AE_MULAUS4O4X16
8X16	AE_MUL8Q8X16	AE_MUL4O8X16
	AE_MULA8Q8X16	AE_MULA4O8X16

Precision	Octal MAC - Quad 32-bit output	Quad MAC Octal 32-bit Output
	AE_MULUS8Q8X16 AE_MULAUS8Q8X16	AE_MULUS4O8X16 AE_MULAUS4O8X16

Precision	Octal MAC - Quad 64-bit output	Quad MAC - Octal 64-bit output
8X16	AE_MUL8QW8X16 AE_MULA8QW8X16 AE_MULUS8QW8X16 AE_MULAUS8QW8X16	AE_MUL4QW8X16 AE_MULA4QW8X16 AE_MULUS4QW8X16 AE_MULAUS4QW8X16

Table 32: Convolution Operations

Precision	Octal MAC - Quad 32-bit output	Quad MAC Octal 32-bit Output
8X8	AE_MUL8Q8X8CNV_[H|L] AE_MULA8Q8X8CNV_[H|L] AE_MULUU8Q8X8CNV_[H|L] AE_MULAUU8Q8X8CNV_[H|L] AE_MULUS8Q8X8CNV_[H|L] AE_MULAUS8Q8X8CNV_[H|L] AE_MULSU8Q8X8CNV_[H|L] AE_MULASU8Q8X8CNV_[H|L] AE_MUL2X4Q8X8CNV_[H|L] AE_MULA2X4Q8X8CNV_[H|L] AE_MULUU2X4Q8X8CNV_[H|L] AE_MULAUU2X4Q8X8CNV_[H|L] AE_MULUS2X4Q8X8CNV_[H|L] AE_MULAUS2X4Q8X8CNV_[H|L] AE_MULSU2X4Q8X8CNV_[H|L] AE_MULASU2X4Q8X8CNV_[H|L]	AE_MUL4O8X8CNV_[H|L] AE_MULA4O8X8CNV_[H|L] AE_MULUU4O8X8CNV_[H|L] AE_MULAUU4O8X8CNV_[H|L] AE_MULUS4O8X8CNV_[H|L] AE_MULAUS4O8X8CNV_[H|L] AE_MULSU4O8X8CNV_[H|L] AE_MULASU4O8X8CNV_[H|L]
Asymmetric unsigned and signed 8-bit	AE_MULUUZB8Q8X8CNV_[H|L] AE_MULAUUZB8Q8X8CNV_[H|L] AE_MULUUZB2X4Q8X8CNV_[H|L] AE_MULAUUZB2X4Q8X8CNV_[H|L] AE_MULZB8Q8X8CNV_[H|L]	AE_MULUUZB4O8X8CNV_[H|L] AE_MULAUUZB4O8X8CNV_[H|L] AE_MULZB4O8X8CNV_[H|L] AE_MULAZB4O8X8CNV_[H|L]

Precision	Octal MAC - Quad 32-bit output	Quad MAC Octal 32-bit Output
	AE_MULAZB8Q8X8CNV_[H|L] AE_MULZB2X4Q8X8CNV_[H|L] AE_MULAZB2X4Q8X8CNV_[H|L]	
Depth-wise separable CNN (for Asymmetric signed and unsigned 8-bit). Note: These are operations which provide SIMD Triple MAC Octal 32-bit Output		AE_MULUUZB3X3O8X8 AE_MULAUUZB3X3O8X8 AE_MULZB3X3O8X8 AE_MULAZB3X3O8X8
4X16	AE_MUL8Q4X16CNV_[H|L] AE_MULA8Q4X16CNV_[H|L] AE_MULUS8Q4X16CNV_[H|L] AE_MULAUS8Q4X16CNV_[H|L] AE_MUL2X4Q4X16CNV_[H|L] AE_MULA2X4Q4X16CNV_[H|L] AE_MULUS2X4Q4X16CNV_[H|L] AE_MULAUS2X4Q4X16CNV_[H|L]	AE_MUL4O4X16CNV_[HH|HL|LH|LL] AE_MULA4O4X16CNV_[HH|HL|LH|LL] AE_MULUS4O4X16CNV_[HH|HL|LH|LL] AE_MULAUS4O4X16CNV_[HH|HL|LH|LL]
8x16	AE_MUL8Q8X16CNV AE_MULA8Q8X16CNV AE_MULUS8Q8X16CNV AE_MULAUS8Q8X16CNV AE_MUL2X4Q8X16CNV AE_MULA2X4Q8X16CNV AE_MULUS2X4Q8X16CNV AE_MULAUS2X4Q8X16CNV	AE_MUL4O8X16CNV_[H|L] AE_MULA4O8X16CNV_[H|L] AE_MULUS4O8X16CNV_[H|L] AE_MULAUS4O8X16CNV_[H|L]

Precision	Quad/Quad MAC - Octal 32-bit output
8X16 (for edge processing)	AE_MULQQ8X16CNV AE_MULAQQ8X16CNV AE_MULUSQQ8X16CNV AE_MULAUSQQ8X16CNV
4X16	AE_MULQQ4X16CNV_[H|L] AE_MULAQQ4X16CNV_[H|L] AE_MULUSQQ4X16CNV_[H|L]

Precision	Quad/Quad MAC - Octal 32-bit output
	AE_MULAUSQQ4X16CNV_[H|L]

4.12 Add, Subtract and Compare Operations

The HiFi 5 DSP supports a rich collection of Add, Subtract and Compare operations.

Table 33: ALU Operations

Precision	ABS	Negate	Add	Sub
8	AE_ABS8 AE_ABS8S	AE_NEG8S	AE_ADD8 AE_ADD8S	AE_SUB8 AE_SUB8S
16	AE_ABS16 AE_ABS16S	AE_NEG16S AE_CONJ16S	AE_ADD16S AE_ADD16	AE_SUB16S AE_SUB16
24	AE_ABS24S	AE_NEG24S	AE_ADD24S	AE_SUB24S
32	AE_ABS32 AE_ABS32S	AE_NEG32 AE_NEG32S AE_NEG32_L	AE_ADD32 AE_ADD32S	AE_SUB32 AE_SUB32S
56	AE_ABSSQ56S	AE_NEGSQ56S	AE_ADDSQ56S	AE_SUBSQ56S
64	AE_ABS64 AE_ABS64S	AE_NEG64 AE_NEG64S	AE_ADD64 AE_ADD64S	AE_SUB64 AE_SUB64S
72			AE_ADD72 AE_ADD72X64	AE_SUB72 AE_SUB72X64

Table 34: ALU Operation with wider output and Reduction Math Operations

Precision	WADD	WACC	WSUB
8-bit	AE_ADDW8 AE_ADDW8U	AE_ACCW8 AE_ACCW8U	AE_SUBW8 AE_SUBW8U
16-bit	AE_ADDW16	AE_ACCW16	AE_SUBW16
32-bit	AE_ADDW32	AE_ACCW32	AE_SUBW32

Table 35: Reduction Math Operations

Precision	Reduction ADD	Reduction MIN/MAX	ADD,SUB
8-bit	AE_RADD8X8.H AE_RADD8X8.L AE_RADDA8X8.H AE_RADDA8X8.L	AE_RMAX8X8 AE_RMIN8X8	
16-bit	AE_RADD16X4 AE_RADDA16X4	AE_RMAX16X4 AE_RMIN16X4	
32-bit	AE_ADD32_HL_LH AE_ADD32S_HL_LH		AE_ADDSUB32 AE_ADDSUB32S AE_SUBADD32 AE_SUBADD32S AE_ADDSUB32_HL_LH AE_ADDSUB32S_HL_LH AE_ADDANDSUB32S AE_ADDANDSUB32J AE_ADDANDSUB32JS

Table 36: Bitwise Logical Operations

Operations
AE_AND, AE_NAND, AE_OR, AE_XOR

Table 37: Compare operations

Precision	LE, LT, EQ	MIN / MAX	MAXABS/ MINABS	MINMAX	BMIN/ BMAX
8-bit	AE_LE8 AE_LT8 AE_EQ8	AE_MAX8 AE_MIN8			AE_BMIN8X8.H AE_BMIN8X8.L AE_BMAX8X8.H AE_BMAX8X8.L
16-bit	AE_LE16 AE_LT16 AE_EQ16	AE_MAX16 AE_MIN16	AE_MAXABS16S	AE_MINMAX16	AE_BMIN16X4 AE_BMAX16X4
32-bit	AE_LE32 AE_LT32 AE_EQ32	AE_MAX32 AE_MIN32	AE_MAXABS32S AE_MINABS32S	AE_MINMAX32	AE_BMIN32X2 AE_BMAX32X2

Precision	LE, LT, EQ	MIN / MAX	MAXABS/ MINABS	MINMAX	BMIN/ BMAX
64-bit	AE_LE64 AE_LT64 AE_EQ64	AE_MAX64 AE_MIN64	AE_MAXABS64S AE_MINABS64S		

Table 38: Special Arithmetic Operations

Add/Sub with Carry	Additve Inverse	Add or Subtract mantissa of Emulated Float	Add or Subtract exponent of Emulated Float
AE_ADDC32 AE_SUBC32 AE_ADDC32U AE_SUBC32U	AE_ADDINV32S AE_ADDINV16S	AE_ADDCEXP32.H AE_ADDCEXP32.L	AE_EXPADD16.H AE_EXPADD16.L AE_EXPSUB16.H AE_EXPSUB16.L

4.13 Shift Operations

HiFi 5 DSP comes with a large variety of shift operations, supporting 8-, 16-, 24-, 32-, and 64-bit shifts, as well as legacy HiFi 2 shift operations. The shift amount can come from an immediate, an AR register, AE_DR register, or the AE_SAR shift register. Variable shifts are bidirectional, meaning that the direction of the shift changes if the shift amount is negative. Variable shifts using the AR shift register can do a shift without having to set the AE_SAR shift register, but the AE_SAR variants are available in ae_slot2 and can hence be issued in parallel with both a load and store and a multiply. For 32- and 24-bit shifts, the AE_SAR and AE_DR variants also allow for different shift amounts for the high and low halves. Shift instructions using an AR register or the AE_DR or the AE_SAR state will truncate the shift amount based on the size of the data being shifted. For example, shifting a 16-bit element by 17 will truncate the shift amount from 17 down to 1.

All shift operations start with the prefix AE_S. The following letter is either L or R signifying whether the primary shift direction is left or right. The next letter is either L or A signifying whether a shift is logical (fill in 0’s on a right shift) or arithmetic (sign-extend on a right shift). The next letter, I, is for immediate shifts, A for AR shifts, V for DR variable shifts and S for AE_SAR shifts. Following is a number signifying the size of the element being shifted, the optional SYM for right shifts that round symmetrically, and an optional R for right shifts that round asymmetrically rather than truncate, and an optional S for left shifts that saturate.

The following table lists the shift operations supported by HiFi 5 DSP.

Table 39: Arithmetic Left Shift Operations

	AR reg based	IMM value based	SAR reg based
8-bit	AE_SLAA8S AE_SLAA8	AE_SLAI8S AE_SLAI8	
16-bit	AE_SLAA16S AE_SLAA16	AE_SLAI16S AE_SLLI16S AE_SLAI16	
24-bit		AE_SLAI24 AE_SLAI24S AE_SLLI24 AE_SLLI24S AE_SLLIP24 AE_SLLISP24S	AE_SLAS24 AE_SLAS24S AE_SLLSP24 AE_SLLSSP24S
32-bit	AE_SLAA32 AE_SLAA32S	AE_SLAI32 AE_SLAI32S AE_SLLI32 AE_SLLI32S AE_SLLI_32	AE_SLAS32 AE_SLAS32S
56-bit	AE_SLAAQ56 AE_SLAASQ56S AE_SLLAQ56 AE_SLLASQ56S	AE_SLAIQ56 AE_SLAISQ56S AE_SLLIQ56 AE_SLLISQ56S	AE_SLASQ56 AE_SLASSQ56S AE_SLLSQ56 AE_SLLSSQ56S
64-bit	AE_SLAA64 AE_SLAA64S	AE_SLAI64 AE_SLAI64S AE_SLLI64S	AE_SLAS64 AE_SLAS64S
72-bit		AE_SLAI72	

Table 40: Arithmetic Right Shift Operations

	AR reg based	IMM value based	SAR reg based
8-bit	AE_SRAA8S AE_SRAA8RS	AE_SRAI8 AE_SRAI8R	

	AR reg based	IMM value based	SAR reg based
16-bit	AE_SRAA16S AE_SRAA16RS AE_SRAA16SYMS	AE_SRAI16 AE_SRAI16R AE_SRAI16SYM	
24-bit		AE_SRAI24 AE_SRAIP24	AE_SRAS24 AE_SRASP24
32-bit	AE_SRAA32 AE_SRAA32RS AE_SRAA32S AE_SRAA32SYMS	AE_SRAI32 AE_SRAI32R AE_SRAI_32 AE_SRAI32SYM	AE_SRAS32
56-bit	AE_SRAAQ56	AE_SRAIQ56	AE_SRASQ56
64-bit	AE_SRAA64 AE_SRA64_32	AE_SRAI64	AE_SRAS64
72-bit		AE_SRAI72	

Table 41: Logical Shift Operations

	AR reg based	IMM value based	SAR reg based
8-bit	AE_SRLA8	AE_SRLI8	
16-bit	AE_SRLA16	AE_SRLI16	
24-bit		AE_SRLI24 AE_SRLIP24	AE_SRLS24 AE_SRLSP24
32-bit	AE_SRLA32	AE_SRLI32 AE_SRLI_32	AE_SRLS32
56-bit	AE_SRLAQ56	AE_SRLIQ56	AE_SRLSQ56
64-bit	AE_SRLA64	AE_SRLI64	AE_SRLS64

AE_DR based Bidirectional Arithmetic Shift
AE_SRAV32RS AE_SRAV16RS

4.14 Normalization Operations

The following normalization operations are supported by HiFi 5 DSP.

Table 42: Normalization Operations

Precision	Normalization Operations
16-bit	AE_NSAZ16_0 AE_NSA16X4
32-bit	AE_NSAZ32X4 AE_NSA32X4 AE_NSAZ32.L
64-bit	AE_NSA64

4.15 Divide Step Operations

HiFi 5 DSP provides two divide step operations

Table 43: Divide Helper Functions

Divide Step Operations
AE_DIV64D32.H
AE_DIV64D32.L

4.16 Truncate Operations

This section lists the truncate operations.

	Source	
Destination	**64 bit**	**32 bit**
32 bit	AE_TRUNCA32F64S.L AE_TRUNCA32X2F64S AE_TRUNCI32F64S.L AE_TRUNCI32X2F64S	

	Source	
Destination	**64 bit**	**32 bit**
16 bit	AE_TRUNCA16X4F64S AE_TRUNCI16X4F64S	AE_TRUNCI16X4F32S AE_TRUNCA16X4F32S

The following operations are provided to ensure HiFi 2 code compatibility, some may be implemented as intrinsics.

```
ae_int24x2 AE_TRUNCP24A32X2 {uint32 ah, uint32 al};
ae_p24x2s AE_TRUNCP24Q48X2 {ae_q56s d0,  ae_q56s d1};
ae_int24x2 AE_TRUNCP16 {ae_int24x2 d0};
ae_q56s AE_TRUNCQ32 {ae_q56s d0};
int32 AE_TRUNCA32Q48 {ae_q56s d0};
int32 AE_TRUNCA16P24S.L {ae_int24x2 d0};
int32 AE_TRUNCA16P24S.H {ae_int24x2 d0};
```

4.17 Round Operations

This section lists and describes the round operations.

	Source		
Destination	**64 bit**	**32 bit**	**16 bit**
32 bit	AE_ROUND32X2F64SASYM AE_ROUND32X2F64SSYM		
16 bit		AE_ROUND16X4F32SASYM AE_ROUND16X4F32SSYM	
8 bit		AE_ROUND8X4F32SASYM_L AE_ROUND8X4F32SSYM_L	AE_ROUND8x8F16SASYM AE_ROUND8x8F16SSYM

The following operations are provided to ensure HiFi 2 code compatibility, some may be implemented as intrinsics.

```
ae_f32x2 AE_ROUND32X2F48SSYM {ae_f64 d0, ae_f64 d1};
ae_f32x2 AE_ROUND32X2F48SASYM {ae_f64 d0, ae_f64 d1}
ae_f24x2 AE_ROUND24X2F48SSYM {ae_f64 d0, ae_f64 d1}
ae_f24x2 AE_ROUND24X2F48SASYM {ae_f64 d0, ae_f64 d1}
ae_f24x2 AE_ROUNDSP16Q48X2SYM {ae_f64 d0, ae_f64 d1}
ae_f24x2 AE_ROUNDSP16Q48X2ASYM {ae_f64 d0, ae_f64 d1}
ae_f32x2 AE_ROUNDSP16F24SYM {ae_f32x2 d0}
ae_f32x2 AE_ROUNDSP16F24ASYM {ae_f32x2 d0}
ae_int64 AE_ROUNDSQ32F48SYM {ae_int64 d0}
ae_int64 AE_ROUNDSQ32F48ASYM {ae_int64 d0}
ae_f32x2 AE_ROUND32F48SSYM {ae_f64 d0};
ae_f32x2 AE_ROUND32F48SASYM {ae_f64 d0};
ae_f24x2 AE_ROUND24F48SSYM {ae_f64 d0};
ae_f24x2 AE_ROUND24F48SASYM {ae_f64 d0};
ae_f24x2 AE_ROUNDSP16Q48SYM {ae_f64 d0};
ae_f24x2 AE_ROUNDSP16Q48ASYM {ae_f64 d0};
ae_p24x2s AE_ROUNDSP24Q48ASYM {ae_q56s d0};
ae_p24x2s AE_ROUNDSP24Q48SYM {ae_q56s d0};
ae_p24x2s AE_ROUNDSP16SYM {ae_p24x2s d0};
ae_p24x2s AE_ROUNDSP16ASYM {ae_p24x2s d0};
ae_q56s AE_ROUNDSQ32SYM {ae_q56s d0};
ae_q56s AE_ROUNDSQ32ASYM {ae_q56s d0};
```

4.18 Saturate Operations

This section lists and describes the saturate operations.

Source Destination	**72 bit**	**64 bit**	**32 bit**	**16 bit**
64 bit	AE_SAT64S			
32 bit		AE_SAT32X2 AE_SATU32X2		
16 bit			AE_SAT16X4 AE_SATU16X4	
8 bit			AE_SATU8X4X32_L AE_SAT8X4X32_L	AE_SAT8X8X16

The following operations are provided to ensure HiFi 2 code compatibility, some may be implemented as intrinsics.

```
ae_f64 AE_SAT48S {ae_f64 d1} ;
ae_f64 AE_SATQ56S {ae_f64 d1};
```

```
ae_q56s AE_SATQ48S {ae_q56s d1};
ae_f24x2 AE_SAT24S {ae_f32x2 d1};
```

4.19 Convert Operations

This section lists and describes the convert operations.

	Input Data Type				
Output Data Type	**ae_int64**	**ae_int32X2**	**ae_int16x4**	**ae_int8x8**	**uint32**
ae_f64		AE_CVT64F32.H			AE_CVT64A32
ae_f32x2			AE_CVTI32X4F16 AE_CVTI32X4F16S AE_CVTA32X4F16 AE_CVTA32X4F16S AE_CVTI32X4F16U AE_CVTI32X4F16US AE_CVTA32X4F16U AE_CVTA32X4F16US AE_CVT32X2F16.[10|32] AE_SEXT32X2D16.[10|32]	AE_CVTI32X4F8.[H | L] AE_CVTI32X4F8S.[H | L] AE_CVTA32X4F8.[H | L] AE_CVTA32X4F8S.[H | L] AE_CVTI32X4F8U.[H | L] AE_CVTI32X4F8US.[H | L] AE_CVTA32X4F8U.[H | L] AE_CVTA32X4F8US.[H | L]	
ae_f16x4				AE_CVTI16X4X2F8 AE_CVTI16X4X2F8S AE_CVTA16X4X2F8 AE_CVTA16X4X2F8S AE_CVTI16X4X2F8U AE_CVTI16X4X2F8US	

	Input Data Type				
Output Data Type	**ae_int64**	**ae_int32X2**	**ae_int16x4**	**ae_int8x8**	**uint32**
				AE_CVTA16X4X2F8U AE_CVTA16X4X2F8US	
ae_int32x2		AE_SEXT32			
ae_ep	AE_SEXT72				

The following operations are provided to ensure HiFi 2 code compatibility, some may be implemented as intrinsics.

```
int32 AE_CVTA32F24S.L {ae_int24x2 d0};
int32 AE_CVTA32F24S.H {ae_int24x2 d0};
ae_int24x2 AE_CVTP24A16X2.LL {uint32 ah, uint32 al};
ae_int24x2 AE_CVTP24A16X2.LH {uint32 ah, uint32 al};
ae_int24x2 AE_CVTP24A16X2.HL {uint32 ah, uint32 al};
ae_int24x2 AE_CVTP24A16X2.HH {uint32 ah, uint32 al};
ae_q56s AE_CVTQ56A32S {uint32 a0};
ae_f64 AE_CVT48A32 {uint32 a0};
ae_f64 AE_CVTQ56P32S.L {ae_int32x2 d0};
ae_f64 AE_CVTQ56P32S.H {ae_int32x2 d0};
ae_f64 AE_CVT48F32.L {ae_f32x2 d0};
ae_f64 AE_CVT48F32.H {ae_f32x2 d0};
int32 AE_CVTA32P24.L {ae_p24x2s d0};
int32 AE_CVTA32P24.H {ae_p24x2s d0};
ae_q56s AE_CVTQ48A32S {uint32 a};
ae_q56s AE_CVTQ48P24S.L {ae_p24x2s d0};
ae_q56s AE_CVTQ48P24S.H {ae_p24x2s d0};
ae_int24x2 AE_CVTP24A16 {uint32 ai};
ae_int24x2 AE_CVTP24A16X2 {uint32 ah, uint32 al};
```

4.20 Move Operations

This section lists the move operations which move data between the address register (AR), DR, Core Boolean Register (BR) and EP registers.

Special Move instructions are also described in the following table.

Table 44: Move Operations

Register Name	**AR**	**AE_DR**	**BR**	**AE_EP**
AR		AE_MOVAD8	AE_MOVAB2	AE_MOVAE

Register Name	AR	AE_DR	BR	AE_EP
		AE_MOVAD32[.H|.L] AE_MOVAD16[.3|.2|.1|.0]	AE_MOVAB4 AE_MOVAB	
AE_DR	AE_MOVDA32 AE_MOVDA32X2 AE_MOVDA16 AE_MOVDA16X2 AE_MOVDA8	AE_MOVDX2 AE_MOV		
BR	AE_MOVBA1X2 AE_MOVBA2 AE_MOVBA4 AE_MOVBA		AE_MOVB2 AE_MOVB4	
AE_EP	AE_MOVEA			AE_MOVEEP

Table 45: Special Move Operations

Operations	Descriptions
AE_MOVI	Copy and replicate the immediate (from -16 to 47) into the two halves of output 32x2 variable.
AE_MOVT64 AE_MOVT32X2 AE_MOVT16X4 AE_MOVF64 AE_MOVF32X2 AE_MOVF16X4	Conditional Move Operations for different kind of vector data types
AE_MOVT16X8 AE_MOVT8X16.H AE_MOVT8X16.L	Conditional Move Operations for lower precision, available only with Neural Network Extension
AE_MOVNEG32S_T	Conditional Move Operations for 32X2 vector elements, based on the sign of another 32X2 vector elements
AE_MOVDEXT	Moves Fractional part of the fixed point number to another AE_DR register

Operations	Descriptions
AE_MOVADEXT.H AE_MOVADEXT.L	Moves Integer part of the fixed point number into AR registers
AE_MOVBD1X4 AE_MOVBD1X2	Moves Least significant bit of AE_DR register to AE_BR register.

4.21 Pack-Shift-Round Operations

This section lists and describes the Pack-Shift-Round operations.

The PKSR operations are used in the implementation of the acceleration of biquad filters.

HiFi 5 comes with PKSR operations for various precision as shown in the following table.

Table 46: PKSR Operations

Pack-Shift-Round operations
AE_PKSR24 AE_PKSR32 AE_PKSRF32 AE_PKSR16

4.22 FFT (Fast Fourier Transform) Operations

Special instructions to maintain dynamic range:

Fixed word length registers and ALU, MACs have limited dynamic range to deal with the data. After every pass within FFT/IFFT algorithm, the ALU and MAC operations cause the data bits width to grow. This means that the total data bits width from the FFT/IFFT grows proportionally to the number of stages. So it is important to avoid data overflows in the intermediate stages, by scaling input data appropriately, while maintaining the best possible precision at the output of every stage. There are two ways to address data-overflows/ saturations.

The first way is fixed scaling, where the data is shifted right by a constant amount after every pass to keep sufficient headroom. Shifting after every pass by a constant amount can guarantee that the data will not saturate, but might throw away bits at the bottom. But this results in loss of accuracy if the magnitude of input data is small. Alternately, dynamic scaling or normalization where at each stage of the FFT, detect the largest output value and normalize intermediate outputs. This improves the precision and also overcome the limitation of fixed scaling. However, dynamic scaling operations can cause major cycle overhead unless the ISA has specialized instructions for dynamic scaling.

HiFi 5 DSP ISA includes normalization instructions designed for efficient dynamic scaling, which are particularly useful in FFT/Inverse FFT kernels.

HiFi 5 DSP has support for dynamic normalization that normalizes as much as needed. In each pass of the code, the intrinsics AE_S32X2X2RNG_XP store the result of the pass and also calculate how many guard bits are available in the stored data. At the end of the pass, the AE_CALCRNG32 instruction determines the minimum headroom for the data generated in the current pass and sets the AE_SAR register using the minimum headroom value to ensure that there will be at least three bits of headroom. The AE_ADDRNG32 and AE_SUBRNG32 and AE_ADDANDSUBRNG32_ intrinsics in the subsequent pass add/subtract the two arguments and shift the result right by the value in AE_SAR (set in the previous pass), guaranteeing that this pass will not overflow.

The FFT input must be pre-scaled so that there is at least three bits of headroom to ensure better precision and no overflow in the first pass. (in this case AE_SAR is set to zero at the start of the first pass.) The ISA for dynamic range adjustment is designed carefully to support mono or stereo FFT computation. It supports appropriate shifts required for both Radix-2 and Radix-4 FFT variants.

The following table lists the special instructions to maintain dynamic range.

Table 47: Instruction to Maintain Dynamic Range

Precision	Store Data and Detect Range	Calculate Range	Add/Sub and Apply Shift
32 bit	AE_S32X2RNG_[I\X\IP\XP] AE_S32X2X2RNG_[I\X\IP\XP] AE_RNG32X2 AE_RNG32X4	AE_CALCRNG32 AE_CALCRNG3 AE_CALCRNG2 AE_CALCRNG1	AE_ADDANDSUBRNG32 AE_ADDANDSUBRNG32_[H/L] AE_ADDRNG32 AE_SUBRNG32
16 bit	AE_S16X4RNG_[I\X\IP\XP] AE_S16X4X2RNG_[I\X\IP\XP]	AE_CALCRNG16 AE_CALCRNG3	AE_ADDANDSUBRNG16RAS AE_ADDANDSUBRNG16RAS_S1 AE_ADDANDSUBRNG16RAS_S2

The example is provided in *Computations of Operations in N point FFT* on page 148.

HiFi 5 also includes instructions to implement FFT with fixed scaling. The ADD/SUB instructions required for this purpose are listed in *Add, Subtract and Compare Operations* on page 110.

4.23 Selection and Permutation Operations

The select and permute operations allow 16-, 24-, or 32-bit SIMD elements from two elements to be combined together. Not all combinations are supported; only the most commonly used ones.

Operation	Description
AE_SEL16I	Combines four 16-bit elements from two input registers into an output register based on an immediate value.
AE_SEL16I.N	Restricted version of AE_SEL16I
AE_SEL16X4	Combines four 16-bit elements from two input registers into an output register based on the value in SS vector shift/select/shuffle (vsa) register.
AE_SEL8X8I	Combines eight 8-bit elements from two input registers into an output register based on an immediate value.
AE_SEL8X8	Combines eight 8-bit elements from two input registers into an output register based on the value in SS vector shift/select/shuffle (vsa) register.
AE_DSEL16X4	Combines four 16-bit elements from two input registers into an output register based on the indexed value of input elements in SS vector shift/select/shuffle (vsa) register.
AE_DSEL8X8	Combines eight 8-bit elements from two input registers into an output register based on the indexed value of input elements in SS vector shift/select/shuffle (vsa) register.
AE_SHFL16X4	Shuffles four 16-bit elements of the input register based on the value in SS vector shift/select/shuffle (vsa) register
AE_SHFL8X8	Shuffles eight 8-bit elements of the input register based on the value in SS vector shift/select/shuffle (vsa) register
AE_SORT16X4	Sort 4-element (16-bit) vector with indices

4.24 Circular Buffer Helper Operations

HiFi 5 DSP implments the following circular buffer helper operations.

Table 48: Circular Buffer Helper Instructions

Circular Buffer Helper Instructions	
AE_ADDCIRC.XC1	AE_ADDCIRC.XC
AE_ADDCIRC.XC2	AE_ADDICIRC
AE_ADDCIRC.XC3	

4.25 Bit Reversal Operations

HiFi 5 DSP implements a helper operation AE_ADDBRBA32, which may be used in combination with indexed loads and stores (.X) to perform bit-reversed addressing in optimized FFT implementations.

4.26 ZERO Operations

AE_ZERO() set all bits of an AE_DR register d to zero. This intrinsic is implemented in terms of the AE_MOVI instruction. This operation is supported for various data types through multiple intrinsics.

4.27 Core ALU Operations

HiFi 5 DSP implements the following operations that are simplified version of core ALU operations encoded in 16-bits for better code density. They are all inferred automatically by the C/C++ compiler.

Table 49: Core ALU Operations

Core ALU operations
Sign Extend, Zero Extend, CLAMPS
AE_SEXT16 AE_ZEXT16 AE_ZEXT8 AE_CLAMPS16

4.28 Bitstream and Variable-Length Encode and Decode Operations

The HiFi bitstream encoding and decoding operations described in this section provide efficient support for serial access to bitstreams (bits stored in memory in serial byte order, with the most significant bit first). The encoding operations are used to create a bitstream from a list of values and their bit-widths. The decoding operations are used to read the bitstreams into elements using the list of bit-widths.

The HiFi bitstream engine supports both fixed length and variable length encoding and decoding. Variable length (Huffman) encode and decode operations are specialized operations, in which the elements with variable bit-widths are encoded or decoded. The operations are assisted by a special set of tables generated from Huffman encoding/decoding schemes used in the algorithm. These tables are generated offline and their entries capture the bit-widths, bit-pattern and values. The format of the table entries are specified in *Codebook Formats*. For details on how the variable-length encode/decode operations should be used, refer to *Variable-Length Encode and Decode* on page 135.

Internally, the operations share the state registers described in *Table 2: Bitstream and Variable-length Encode/Decode Support Subsystem State Registers* on page 28. Therefore, the program cannot switch between encoding and decoding modes without storing and restoring their values.

The following Bitstream and Variable-Length Encode and Decode Operations are included in <HiFi 5 DSP

Table 50: Bitstream and Variable-Length Encode and Decode Operations

Bitstream Look Ahead Operations	Discard Bits Operations	Bitstream Store Operations
AE_LB[| .I | .K | .KI | .S | .SI]	AE_DB[| .IP | .IC | .IC1]	AE_SB[| .IP | .IC | .IC1]
AE_LBKI_DBI[| .IP | .IC]	AE_DBI[| .IP | .IC | .IC1]	AE_SBI[| .IP | .IC | .IC1]
AE_LBI_DBI[| .IP | .IC]		AE_SBF[| .IP | .IC | .IC1]
AE_LBK_DB[| .IP | .IC]		
AE_LB_DB[| .IP | .IC]		

Varaible Length Encode Operations	Variable Length Decode Operations	Other Bitstream Maniplulation operations
AE_VLEL32T	AE_VLDL32T	AE_BITSWAP
AE_VLEL16T	AE_VLDL16T	AE_SHA32
AE_VLEL16C	AE_VLDL16C	AE_ARDECNORM16
AE_VLEL16C.IP	AE_VLDL16C.IP	AE_SHORTSWAP
AE_VLEL16C.IC	AE_VLDL16C.IC	
AE_VLEL16C.IC1	AE_VLDL16C.IC1	
	AE_VLDSHT	

4.29 Optional Floating Point Unit Operations

HiFi 5 DSP supports an optional pair of 4-way SIMD IEEE Single Precision Floating Point Unit (SP FPU) and a 8-way SIMD IEEE half Precision Floating Point Unit (HP FPU)).

The floating point units share the AE_DR register file with the rest of HiFi 5 DSP. Therefore, standard loads, stores, and selects operations are shared and comes with intrinsics to work together with floating point compute operations.

The floating point operations typically have four cycles of latency, but are fully pipelined. With RI-2021.6 xtensa tools release, HiFi 5 core is provided with a more optimized vector floating-point feature, where the latency of single-precision and half-precision FMA is reduced from 4 cycles (in HW version LX7.1.0 - LX7.1.5) to 3 cycles (HW version LX7.1.6 and later). This change improves the hardware efficiency and it also improves the compiler performance for floating point codes that use VFPU (SP FPU and HP FPU) instructions in general. For optimized legacy code, in most cases recompiling the code on the latest tools will help. If for some kernels degradation in performance is observed with out of box compilation, analyze the .s assembly files and re-tune the intrinsic-based code and pragmas. A compiler macro XCHAL_HAVE_HIFI5_3CYCLE_FMA will be defined in the tools which can be used to conditionalize the code for hardware version LX7.1.6 or later.

Divide (IEEE754 exact), reciprocal, reciprocal square root, and square root (IEEE754 exact) operations are implemented using instruction sequences. These intrinsics currently use 2-way SIMD operations. 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.

In general, 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. The operations that support 4-way come with two variants, Q_S and _SX2X2. The Q_S variant allows reuse of one operand across the SIMD lane to reduce the number of ports, so it is slotted with a set of other instructions. The _SX2X2 variants are 4-way operations.

The following tables list the floating point operations.

Table 51: Floating Point Operations

SP FPU	HP FPU
Number Conversion	
TRUNC.S, TRUNC.SX2, UFLOAT.S, UFLOAT.SX2, UTRUNC.S, UTRUNC.SX2, FICEIL.S, FIFLOOR.S, FIRINT.S, FIROUND.S, FITRUNC.S, FLOAT.S, FLOAT.SX2, FLOATEXP.S,	CVTF16S.H, CVTSF16.H, FICEIL.H, FIFLOOR.H, FIRINT.H, FIROUND.H, FITRUNC.H, FLOAT16.H, FLOAT16.HX4, TRUNC16.H, TRUNC16.HX4, UFLOAT16.H, UFLOAT16.HX4, UTRUNC16.H, UTRUNC16.HX4,
Arithmetic Operations	
ABS.S, ADD.S, ADD_HL_LH.S, ADDANDSUB.S, ADDANDSUBJC.S, CONJC.S, MSUB.S, MSUBN.S, MSUBQ.S, SUB.S, SUB.SX2X2, NEG.S, NEG.SX2X2, ABS.SX2X2, ADD.SX2X2, CONJC.SX2X2, MSUB.SX2X2,	ABS.H, ABS.HX4X2, ADD.H, ADD.HX4X2, CONJC.H, CONJC.HX4X2, MSUB.H, MSUB.HX4X2, MSUBN.H, NEG.H, NEG.HX4X2, RMAXNUM.H, RMINNUM.H, SUB.H, SUB.HX4X2,

Min Max Operations	
MAX.S, MAXNUM.S, MAXNUMABS.S, MIN.S, MINNUM.S, MINNUMABS.S, BMAXNUM.S, BMAXNUMABS.S, BMINNUM.S, BMINNUMABS.S,	MAX.H, MAXNUM.H, MIN.H, MINNUM.H,

Compare And Conditional Move Operations	
OEQ.S, OLE.S, OLT.S, UEQ.S, ULE.S, ULT.S, UN.S, MOVEQZ.S, MOVF.S, MOVGEZ.S, MOVLTZ.S, MOVNEZ.S, MOVT.S,	OEQ.H, OLE.H, OLT.H, UEQ.H, ULE.H, ULT.H, UN.H,

Multily, MAC/MSU helper functions	
MADD.S, MADD.SX2X2, MADDA.S, MADDMUX.S, MADDMUX.SX2X2, MADDMUXQ.S, MADDN.S, MADDQ.S, MUL.S, MUL.SX2X2, MULJC.S, MULJC.SX2X2, MULMUX.S, MULMUX.SX2X2, MULMUXQ.S, MULQ.S,	MADD.H, MADD.HX4X2, MADDN.H, MADDQ.H, MUL.H, MUL.HX4X2, MULACNVH.HX4X2, MULACNVL.HX4X2, MULCNVH.HX4X2, MULCNVL.HX4X2, MULJC.H, MULJC.HX4X2, MULQ.H,

Other Helper functions	
CLSFY.S, CONST.S, CONST.SX2X2, ADDEXP.S, ADDEXPM.S, DIV0.S, DIVN.S, NEXP01.S, RECIP0.S, RSQRT0.S, SQRT0.S, MKDADJ.S, MKSADJ.S, FREXP.S,	ADDEXP.H, ADDEXPM.H, CLSFY.H, CONST.H, CONST.HX4X2, DIV0.H, DIVN.H, NEXP0.H, NEXP01.H, RSQRT0.H, SQRT0.H, MKDADJ.H, MKSADJ.H, RECIP0.H,

Operations where latency is reduced to 3 cycles (effective from LX7.1.6 HW)	
DIVN.S, MADDA.S, MADDMUXQ.S, MADDMUX.S, MADDMUX.SX2X2 MADDN.S, MADDQ.S, MADD.S, MADD.SX2X2, MSUBN.S MSUBQ.S, MSUB.S, MSUB.SX2X2,MULMUXQ.S, MULMUX.S, MULMUX.SX2X2, MULQ.S, MUL.S, MUL.SX2X2	ADD.H, ADD.HX4X2, DIVN.H, MADD.H, MADD.HX4X2, MADDN.H MADDQ.H, MSUB.H, MSUB.HX4X2, MSUBN.H, MULACNVH.HX4X2, MULACNVL.HX4X2 MULCNVH.HX4X2, MULCNVL.HX4X2, MUL.H, MUL.HX4X2, MULQ.H SUB.H, SUB.HX4X2

4.29.1 IEEE 754 Compliance Notes

The optional floating point unit supports single precision format and operations specified by IEEE-754-1985. Note that the single precision format is not changed in IEEE-754-2008.

The following table lists floating point arithmetic operations that are compliant with IEEE-754-1985.

Table 52: Arithmetic Operations Compliant with IEEE-754-1985

ABS.S	ADD.S	SUB.S
MUL.S	NEG.S	

The following table lists floating point conversion operations that are compliant with IEEE-754-1985.

Table 53: Conversion Operations Compliant with IEEE-754-1985

FLOAT.S	UFLOAT.S	FLOAT.SX2
UFLOAT.SX2	TRUNC.S	UTRUNC.S
TRUNC.SX2	UTRUNC.SX2	CVTF16S.L
CVTF16S.H	CVTSF16.L	CVTSF16.H

The following table lists floating point comparison operations that are compliant with IEEE-754-1985.

Table 54: Comparison Operations Compliant with IEEE-754-1985

OLE.S	OLT.S	OEQ.S
ULE.S	ULT.S	UEQ.S
UN.S		

The following table lists floating point arithmetic instruction sequences that are compliant with IEEE-754-1985.

Table 55: Arithmetic Instruction Sequences Compliant with IEEE-754-1985

MULC.S	MULMUX.S	MULCCONJ.S
DIV.S (with non-IEEE-754 compliant operations DIV0.S, ADDEXP.S, ADDEXPM.S, NEXP01.S, MKDADJ.S, MADDN.S, MSUBN.S, DIVN.S, CONST.S)	RECIP.S (with non-IEEE-754 compliant operations RECIP0.S, MADDN.S, CONST.S)	SQRT.S (with non-IEEE-754 compliant operations SQRT0.S, MADDN.S, MSUBN.S, NEXP01.S, ADDEXP.S, ADDEXPM.S, DIVN.S, CONST.S)
RSQRT.S (with non-IEEE-754 compliant operations RSQRT0.S, MADDN.S, CONST.S)		

The following table lists floating point operations that are not specified in IEEE-754-1985.

Table 56: Operations not Specified in IEEE-754-1985

MIN.S	MAX.S	MOVEQZ.S
MOVNEZ.S	MOVGEZ.S	MOVLTZ.S
MOVT.S	MOVF.S	CONJC.S

RFR	WFR	*DIV0.S
*ADDEXP.S	*ADDEXPM.S	*NEXP01.S
*MKDADJ.S	*MADDN.S	*MSUBN.S
*DIVN.S	*CONST.S	

*Not specified in IEEE-754-1985; however, these operations are used in the multi-instruction sequences that are IEEE-754-1985 compliant.

The following table lists floating point arithmetic instruction sequences that are not specified in IEEE-754-1985.

Table 57: Instruction Sequences not Specified in IEEE-754-1985

RECIP.S	RECIP.SX2	RSQRT.S
RSQRT.SX2		

The following table lists floating point Fused Multiply-Add operations that are not specified in IEEE-754-1985, but are specified in IEEE-754-2008.

Table 58: Fused Multiply-Add (FMA) Operations

MADD.S	MSUB.S	MADDMUX.S

The following table lists floating point Fused Multiply-Add instruction sequences that are not specified in either IEEE-754-1985 or IEEE-754-2008.

Table 59: Fused Multiply-Add (FMA) Operations not Specified in IEEE-754-1985 or IEEE-754-2008

MADDC.S	MADDCCONJ.S	MSUBC.S
MSUBCCONJ.S		

The following table lists floating point conversion to integral operations that are not specified in IEEE-754-1985, but in IEEE-754-2008.

Table 60: Conversion to Integral Operations

FIROUND.S	FIFLOOR.S	FICEIL.S
FITRUNC.S	FIRINT.S	

Status Flags and Handling

The floating point unit supports detection of five exception conditions defined by IEEE-754-1985:

- Invalid Operation
- Division by Zero
- Overflow
- Underflow
- Inexact

The status check of these exception conditions is performed by reading the state register FCR_FSR. The floating point unit does not take a trap upon the exception.

In summary, the floating point unit provides features following IEEE-754-1985. IEEE-754-2008 compliance is mentioned in those operations that are compatible (e.g., FMA operations).

4.30 Not a Number (NaN) Propagation

Some floating-point operations have a floating-point datum as an input operand or an output operand, but not both. Some other floating-point operations have both a floating-point input operand, and a floating-point output operand. Most of these floating-point operations, having floating-point data as both input and output operands, propagate a NaN as the output result if an input is a NaN, according to IEEE 754™ -2008. This propagation assists programmers to trace back to the origin of a numerical exception or NaN, usually an invalid operation such as inf - inf.

However, programmers are reminded not to depend on NaN propagation, payload, or the sign bit, since recompilation may cause the propagation to change or to cease.

4.31 ISA Enhancements to Support Activation Functions

Sigmoid and Tanh Functions

Algorithm description for Fixed-point Sigmoid/Tanh function. The instructions calculate the value of the sigmoid/tanh function of a signed input x, producing outputs based on the following equations:

- $sigmoid(x) = \frac{1}{1+e^{-x}}$
- $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

Activation Function Specific Operations

The following new operations have been added. These operations are 8-way SIMD operations which expect input in AE_DR registers and generate output in AE_DR registers of HiFi 5 DSP. These operations have 2 cycle latency; latencies are hidden when the compiler unrolls the loop and achieves the best software pipeline schedule.

1. AE_TANH16X4X2 (Out1, Out2, Inp1, Inp2)

   This is a 8-way SIMD, 16-bit operation that calculates the value of the tanh function for each lane of input vector registers Inp1 and Inp2. Both of these inputs contain four 16-bit input values. The result is written to output vector registers Out1 and Out2. Both the input and the output produced are in Q1.15 format.
2. AE_SIGMOID16X4X2 (Out1, Out2, Inp1, Inp2)

   This is a 8-way SIMD, 16-bit operation that calculates the value of the sigmoid function for each lane of input vector registers Inp1 and Inp2. Both of these inputs contain four 16-bit input values. The result is written to output vector registers Out1 and Out2. The input is in Q1.15 format and output produced is in Q0.16 format.

3. AE_SIGMOID8X8 (Out1, Inp1)

   This is a 8-way SIMD, 8-bit operation that calculates the value of the sigmoid function for each lane of input vector register Inp1. The result is written to output vector register Out1. The input is in Q1.7 format and output produced is in Q0.8 format.

4. AE_TANH8X8 (Out1, Inp1)

   This is a 8-way SIMD, 8-bit operation that calculates the value of the tanh function for each lane of input vector register Inp1. The result is written to output vector register Out1. Both the input and the output produced are in Q1.7 format.

Note: For the above four operations input can be in different Q-point format (like Q3.13). Shift value (to maintain input Q-point format) should be passed in shift amount register AE_SAR.

4.32 ISA enhancements Added in LX 7.1.9 (starting with RI-2022.9)

ISA enhancements in HiFi 5 DSP starting from HW version LX7.1.9 (RI-2022.9 release) include new operations to help:

- Simplify managing Boolean data types.
- Simplify native floating-point programming involving complex data and xtfloatx4 data type.
- Half/Single precision floating point operations.

Some helper instructions are also added to improve Out of Box performance.

All the newly added operations can be broadly categorised into the following groups.

Table 61: Boolean Operations

Operation	Description
AE_EXTRACTB1B2.H/L	Extracts Boolean 1-bit from high or low part of input xtbool2 Boolean register
AE_EXTRACTB2B4.H/L	Extracts Boolean 2-bit in xtbool2 register from high or low part of input xtbool4 Boolean register.
AE_EXTRACTB4B8.H/L	Extracts Boolean 4-bit in xtbool4 register from high or low part of input xtbool8 Boolean register.
AE_JOINB2B1	Combines two xtbool1 Boolean registers to form xtbool2 Boolean register.
AE_JOINB4B2	Combines two xtbool2 Boolean registers to form xtbool4 Boolean register.
AE_JOINB8B4	Combines two xtbool4 Boolean registers to form xtbool8 Boolean register.

These boolean operations are added to simplify managing Boolean data types, which is necessary for efficient implementation of some double-wide vector data types.

Table 62: New Load/Store and Arithmetic Operations

Operation	Description
AE_LAV32X2X2_XP	Variable length aligning load of upto 4 32-bit values into two AE_DR registers
AE_SAV32X2X2_XP	Variable length aligning store of upto 4 32-bit values from two AE_DR registers
AE_TRUNCAV32X2F64S	Two way bidirectional arithmetic shift by value from signed vector operand and truncate 1.63 bit to 1.31 bit
AE_SAT8X4X32_H	Four way saturation of signed 32-bit inputs into 8-bit values with the four 8b results places in the upper half of the DR register and zeroes in the lower half.
AE_SATU8X4X32_H	Four way saturation of 32-bit input into unsigned 8-bit values with the four 8b results places in the upper half of the DR register and zeroes in the lower half.

Table 63: New Half Precision and Single Precision Operations

Operation	Description
DIVN.HX4X2	8-way half-precision floating point divide operation to be used as the final-step of Newton Raphson divide or square root operations sequence

Operation	Description
MADDN.HX4X2	8-way half-precision floating point multiply and accumulate with rounding to nearest operation.
MSUBN.HX4X2	8-way half-precision floating point multiply and subtract with rounding to nearest operation.
DIVN.SX2X2	4-way single-precision floating point divide operation to be used as the final-step of Newton Raphson divide or square root operations sequence
MADDN.SX2X2	4-way single-precision floating point multiply and accumulate operation with rounding to nearest operation.
MSUBN.SX2X2	4-way single-precision floating point multiply and subtract operation with rounding to nearest operation.

Table 64: Helper Operations for Out of Box Performance

Operations	Description
AE_LTR4	Set Boolean register TRUE for up to 3-bits respectively controlled by index defined in Address Register based on the HiFi big-endian order of elements in the regfile.
AE_LTR8	Set Boolean register TRUE for up to 7-bits respectively controlled by index defined in Address Register based on the HiFi big-endian order of elements in the regfile
LOOP.W15	Wide loop operation
LOOPGTZ.W15	Wide loop operation
LOOPNEZ.W15	Wide loop operation

Change in **AE_SEL16I**

For this operation, the pattern encoded by immediate value “10” is changed from “5146” to “6521” based on usefulness for the various audio software. Intrinsic for pattern "5146" is provided. Hence, all the software which use AE_SEL16_5146 intrinsic will behave as before after re-compiling the code. If some legacy code is written using AE_SEL16I operation with 10 as immediate operand, it needs update to use the intrinsics "AE_SEL16_5146"

Change in syntax of **AE_ADDICIRC** intrinsic

This intrinsic is changed to accept first parameter as int* instead of int. This is done to keep the syntax for this operation consistent with other similar operations like AE_ADDCIRC_XC.

5. Variable-Length Encode and Decode

Topics:

- *Overview of Huffman Instructions*
- *Encoding*
- *Decoding*
- *Examples for Encode/ Decode*

The HiFi 5 DSP Instruction Set Architecture includes a set of instructions that allow you to perform variable-length Huffman encoding and decoding in your software routines for the HiFi 5 DSP.

This chapter gives an overview of the Huffman-related instructions and table formats, and encoding and decoding examples.

5.1 Overview of Huffman Instructions

This section will orient you to the instructions that support variable-length encoding and decoding, as well as raw bitstream reads and writes.

The instructions we supply to support variable-length (Huffman) encode/decode are very generic in the sense that they place only the most minimal restrictions on the kind of table hierarchies you use in your application. We expect every practical structure for variable-length encoding and decoding to be efficiently implementable using the instructions we supply. The programmer is free to choose the space/time tradeoffs that suit the application. One of the main goals of this discussion is to help you understand the mechanism well enough that you can make and exploit those choices.

In addition to the flexibility of table structure, we have the flexibility of instructions supporting both 16- and 32-bit table entries. 16-bit table entries are expected to be superior in most cases because they tend to save space over 32-bit entries. The option to use 32-bit entries is important however, because certain codebooks can make 16-bit table entries impossible to use: the smaller entries cannot represent large table indices like 32-bit entries can. While 16-bit table entries will also give slower encoding for long codewords, we don't expect this to be a major consideration because the difference is only a few cycles per symbol. In keeping with the versatility of the mechanism, it is possible to use hierarchical tables with 32-bit entries at some levels and 16-bit entries at others.

In the vast majority of implementations, 16-bit table entries will be the right choice. Nonetheless, the instructions for 32-bit entries are there when they are needed.

Reading and Writing a Sequence of Raw Bits

The instructions for variable-length encoding and decoding are part of a larger family of instructions designed to support highly efficient processing of bitstream input and output. In addition to the instructions for encoding and decoding, there are instructions to retrieve a sequence of raw bits from an input stream and there are instructions to write a sequence of raw bits to an output stream. There is one major restriction: Only one input bitstream or one output bitstream can be active at a given time without a significant sacrifice of efficiency. To explain, there is a single set of state registers that underpin the implementation of the whole family of instructions, and that collection of state pertains to a single stream. To switch from reading to writing, or even just to switch from one input (output) stream to another input (output) stream, all of the underlying state would typically need to be saved to memory and reinitialized. While this restriction is typically not a problem for audio and voice codec applications, programmers must nevertheless be aware of it.

5.2 Encoding

This section describes the encode side first followed by more complicated decoding.

The steps to perform encoding are:

- Translate the symbol to be coded into a table index
- Use this index to retrieve a sequence of codeword bits and a codeword length from a table or a pair of tables

Table entries for each codebook are long enough to hold the longest codeword, but in the present mechanism, the codeword length is not dependent on either the size of the table entries or other aspects of the implementation.

Depending on the length of the longest codeword some codewords may not fit within a single table entry. If the codeword does not fit in a single table entry, the first lookup in the encoding table provides a portion of the codeword, and the index of the location in the table of the next codeword segment.

In the case of 32-bit table entries, a second lookup is required only if the codeword exceeds 16 bits in length. In the case of 16-bit table entries, codewords longer than 11 bits will require a second and possibly subsequent lookups.

Refer to the *Examples for Encode/Decode* on page 140 for examples.

Encoding a Symbol

The memory has an encoding table indexed by symbol value. Each entry in the table is either 16-bit or 32-bit. The table is searched and the encoding entry corresponding to the symbol to be encoded, is retrieved.

We look in the table and retrieve the encoding entry corresponding to the symbol we want to encode. In that table entry we either find the entire codeword corresponding to our symbol, along with an indication of the codeword's length in bits, or we find that the entire codeword is too long to fit in the table entry. A bit in the table entry indicates which of the two cases occurred.

In the first case, we are finished encoding the present symbol once we push the found codeword bits onto the output bitstream.

The second case is a little more interesting. In the second case, we get some bits of the codeword from the table entry, and those are pushed onto the output stream, but there are more codeword bits still to come that cannot be accommodated in a single table entry. When this happens, the first table entry tells us the index of another table entry that will give us another segment of the codeword's bit sequence. Once we retrieve the second table entry based on the new index, we are back in the same situation: either this table entry completes the codeword, or yet another lookup is required. Table entries needed to support lookups beyond the first one for each symbol would generally appear at the end of the table, just beyond the symbol-indexed part.

The length of the codebook’s longest codeword and your decision about whether to use 16- or 32-bit table entries will bound the number of lookups required to encode a symbol. In practice three or more lookups per symbol will be rare with 32-bit table entries (Editor’s note:

we are not aware of any codebooks used in audio that would require three lookups for any symbol), and four or more will be rare with 16-bit entries.

Encoding Table Lookup Instruction Sequence

Each encoding table lookup operation consists of a sequence of two instructions: `ae_vlel16t` (or `ae_vlel32t` if you are using 32-bit table entries) and `ae_vles16c`. The `ae_vlel{16|32}t` instruction loads a table entry based on the current symbol value, and ae_vles16c pushes the segment of bits onto the bitstream being written, flushing 16 stream bits to memory (if that many are available).

The instruction mnemonics are as follows:

- Audio Engine Variable-Length Encode, Load {16|32}-bit Table entry
- Audio Engine Variable-Length Encode, Store 16 stream bits Conditional (reflecting the fact that the bitstream is stored to memory in 16-bit chunks)

5.3 Decoding

The decoding process is more complicated than encoding because codewords have variable lengths. If we could afford a huge table, we could just pad all the codewords out to the length of the longest codeword (with bits from the bitstream), and use the resulting string of bits as an index into a single giant table where we would find an entry telling us the symbol value and the number of bits in the codeword. Note that the lookup has to tell us the number of bits in the codeword so we know how many bits to discard from the head of the bitstream we are reading before doing the next decoding operation.

As with encoding, we look up entries for decoding in a table, but unlike encoding where the alphabet size determined the size of the initial table, the decoding process has power-of-two table sizes that are decided by you in accordance with the space/time tradeoffs you want to make. Decoding takes place through a hierarchy of tables where the size of each table in the hierarchy is up to you (within limits, of course). A table can have as few as two entries, in which case it is essentially a node in a binary tree where a single bit of the codeword guides the decoding process to the next step, or as many as 65536 entries, where a 16-bit chunk of the bitstream forms the table index.

Examples of Supported Decoding Structures

FAAD2, the freeware MPEG-AAC decoder, uses a binary tree as one of its basic table structures. Decoding begins with a 2-entry table at the root and proceeds one bit at a time to a new 2-entry table for each codeword bit, until the end of the codeword is reached and the table entry contains the symbol value.

FAAD2 uses a so-called 2-step table as the other of its basic table structures. K bits at the head of the stream are used to index into the first table. (Depending on the codebook, K is either 5 or 6.) The entry found in the first table gives an index into the second table, which essentially consists of consecutively placed subtables of various sizes. The index from the first table entry tells where the appropriate subtable begins. Each subtable in the second table corresponds to one or more K-bit combinations that might appear at the head of the bitstream. If the codeword is longer than K bits, the entry from the first table also tells how many bits are used to index into the subtable. If the codeword has K bits or fewer, the corresponding subtable has only one entry, so no additional bits are used as an index into it. The entry found in the second table by indexing using the appropriate number of bits off the base given in the first table entry gives the decoded symbol value and the codeword length. (This is not as complicated as it sounds.)

WMA uses a hierarchically-structured table consisting of 4-ary tree nodes and binary tree nodes. The eight levels closest to the root in the tree consist of 4-ary tree nodes, and the remaining six levels are binary.

Our decoding support essentially permits us to structure our decode according to any of those example schemes, or indeed according to a wide variety of other schemes as well. Our HiFi 5 DSP variable-length encoding and decoding instructions also permit us more efficient use of the bits in table entries than the generic-processor implementations, meaning that for a given table organization scheme, the tables to drive our instructions are smaller than those in the corresponding generic implementation. And, of course, our decoding operations are faster as well.

When we begin decoding a codeword, we start at the root of the decoding table hierarchy and use a prefix of the bitstream to look up a table entry. As mentioned before, the length of this prefix is determined when the table hierarchy is designed. Once we have a table entry, there are two cases much like there were for encoding, and again a bit in the table entry distinguishes between the two.

In the first case, the codeword is short enough that we are done decoding it and the table entry tells us the symbol corresponding to the codeword, along with the number of bits occupied by the codeword at the head of the stream. Note that the number of bits used to index into the table might be greater than the length of the codeword, in which case there are duplicate table entries, one for each combination of the “don't care” bits that follow the codeword in the stream.

In the second case, the codeword is longer than an index into the table. In this case, we have not found the symbol corresponding to our codeword yet (because we have not looked at all the codeword bits yet). In this case the table entry tells us where to find the next table and the number of bits to use as an index into that table. The bits we need to discard from the head of the stream are exactly those that we used as the table index, so the table entry itself need not have any direct indication of the number of bits to discard. Upon knowing the base of the next table in the hierarchy for this codeword and discarding the bits that made up the index we used for the first table, we are back in the same situation as when we began decoding: We have a table into which we will index according to a set number of bits at the head of the bitstream. The process repeats until we find ourselves in the first case with our symbol in hand.

The Decoding Table Lookup Instruction Sequence

Each decoding table lookup operation consists of a sequence of two instructions, `ae_vldl16t` (or `ae_vldl32t` if you are using 32-bit table entries) and `ae_vldl16c`. The `ae_vldl{16|32}t` instruction loads a table entry based on the bits currently at the head of the bitstream, and `ae_vldl16c` refreshes the head of the bitstream from memory if necessary.

The instruction mnemonics are as follows:

- DSP Variable-Length Decode, Load {16|32}-bit Table entry;
- DSP Variable-Length Decode, Load 16 stream bits Conditional (reflecting the fact that the bitstream is refreshed from memory in 16-bit chunks).

5.4 Examples for Encode/Decode

Within a C routine that uses encoding, a speed-optimized encoding sequence would look like this:

```
  xtbool          complete;
  unsigned int    symbol;
  unsigned short *table;
  ...
 not_done:
  AE_VLEL16T(complete, symbol, table);
  AE_VLES16C(stream);
  if (!complete) {
#pragma frequency_hint NEVER
    goto not_done;
  }
  ...
```

With the above sequence, the Xtensa C compiler generates assembly code like the following:

```
not_done:
    ae_vlel16t    b0, a3, a9    /* First lookup likely to succeed. */
    ae_vles16c    a2
    bf        b0, not_done /* Avoid branch delay in common case. */
done_encoding:
    ...
```

For example, if you know that your encoding table structure is only one layer deep, you can optimize the code more.

For decoding, the optimal code implementation will depend on the structure of your tables, although it is possible to build a single routine that works very fast with all the possible structures. A single decoding step might be enough most of the time if your top-level table uses a 5-bit index. In such a case, the best way to decode is the simplest, and is exactly analogous to the encoding code above:

```
  xtbool          complete;
  unsigned int    symbol;
  unsigned short *table;
  ...
 not_done:
  AE_VLDL16T(complete, symbol, table);
  AE_VLDL16C(stream);
  if (!complete) {
#pragma frequency_hint NEVER
    goto not_done;
  }
  ...
```

The above sequence in C should yield assembly code like the following:

```
    ...
not_done:
    ae_vldl16t    b0, a9, a4
    ae_vldl16c    a2
    bf        b0, not_done
done_decoding:
    ...
```

On the other hand, if you build your tables as a binary tree, you're unlikely to find any symbols within a single decoding step. In this case, if you have to have every last bit of decoding speed, you can use something like the following example, which is a fast, generic implementation that handles lookups deep in the table hierarchy with fewer branch delays than the simple loop above:

```
    ...
    ae_vldl16t    b0, a9, a4
    ae_vldl16c    a2
    bf        b0, not_done
done_decoding:
    ...

not_done:
    loopnez        a0, .Loopend    /* use stack pointer as while (1) loop counter */
    ae_vldl16t    b0, a9, a4
    ae_vldl16c    a2
    bt        b0, done_decoding
.Loopend:
    j        not_done    /* more lookup iterations than the stack pointer?!? */
```

In conclusion, the HiFi 5 DSP supplies a generic set of instructions to support variable-length (Huffman) encode/decode. They place only minimal restrictions on the kind of table hierarchies you use in your application.

6. Audio DSP Examples

Topics:

- *Complex FIR example*
- *Complex FFT Example*
- *User Defined Emulated Float Example*

This chapter describes audio DSP examples and shows how to optimize them for the HiFi 5 DSP.

6.1 Complex FIR example

HiFI 5 ISA has improved support for 16- and 32-bit complex operations, including 32x32/32x16/16x16 MAC operations which accumulate in different precisions like 16/32/17.47/64 bit.

HiFI 5 ISA provides two 32-bit complex MACs per cycle throughput. Complex and complex conjugate MAC variants would be useful in various DSP kernels. Along with complex MAC support, various helper ISA are provided to enhance the processing of complex data types.

Complex FFT and complex FIR are some of the important kernels in front end signal processing. In this chapter, we have explained HiFi 5 based efficient implementation of these kernels.

Complex FIR

The following code snippet shows FIR filtering of complex data using complex coefficients. We will show how this code can be efficiently implemented using HiFi 5 ISA.

```
typedef struct
{
    int re, im;
} cplx32;
void cplx_fir_ref(cplx32 * __restrict Y, const cplx32 *__restrict X, const cplx32 *__restrict
H, cplx32 *__restrict XDly,int *delay_idx, int N, int M)
{
int i, j, d;
int k=*delay_idx;
for (i = 0; i < N; i++)
{
    y[i].re = 0;
    y[i].im = 0;
    XDly[k] = X[i]; /* Update the delay buffer */
    k=(k++ % M);    /* Update the Delay buffer write index */
    d=k;            /* Set delay buffer read index */
    for (j = 0; j < M; j++)
    {
        y[i].re += ( XDly[(d + j) % M].re * H[j].re ) - ( XDly[(d + j) % M].im * H[j].im ) ;
        y[i].im += ( XDly[(d + j) % M].re * H[j].im ) + ( XDly[(d + j) % M].im * H[j].re ) ;
    }
}
*delay_idx = k;
```

In this code:

x[] : 32 bit complex input array where real and imaginary parts are interleaved.

XDly[] : 32 bit complex delay line array.

y[] : 32 bit complex output array

h[] : 32 bit complex coefficient array.

n : Number of samples to be processed.

M : Number of FIR filter taps.

For efficient implementation, the following assumptions are made in optimized code.

M and N are multiple of 4

x[], y[], and h[] arrays are 16 byte aligned and do not overlap.

The complex FIR filter calculates N output samples using M coefficients, requires last M-1 samples in the delay line which is updated in circular manner for each new sample. Real and imaginary parts are interleaved and real parts go first (at even indexes).

As in the case of any FIR function, the inner loop computes complex MAC operations between input from the delay line and filter coefficients. The outer loop has code to update the input samples in the delay line and also to store the results in the output buffer. For better software pipe-lining in HiFi 5 optimized code, two outputs are calculated at a time and inner loop processes two taps at a time. Using complex MAC operations in two slots provides a throughput of eight MACs/cycle (two complex MACs/cycle)

The following code snippet shows a hand-optimized code of complex FIR filter.

HiFi 5 optimized code

```
// Process 2 output samples at a time.
for (n = 0; n < (N >> 1); n++)
{
    // Load 2 complex input samples. // Q31
    AE_L32X2X2_IP(t0, t1, (ae_int32x4 *)X, +16);

    // Store 2 input complex samples to the delay line buffer with circular address update.
    AE_S32X2X2_XC(t0, t1, (ae_int32x4 *)D_wr, +16);

    // Circular buffer pointer update
    D_rd0 = D_wr;

    // Load two oldest samples
    AE_L32X2X2_XC(t_dummy, t0, (ae_int32x4 *)D_rd0, +16);

    // Zero the accumulators.
    q0r = z; q1r = z;
    q0i = z; q1i = z;

    // Inner loop: process 2 taps for 2 complex accumulators on each iterations.
    for (m = 0; m < (M >> 1); m++)
    {
        // Load next two samples
        AE_L32X2X2_XC(t1, t2, (ae_int32x4 *)D_rd0, +16);

        // Load the next 2 tap coefficients.
        AE_L32X2X2_XC1(c0, c1, (ae_int32x4 *)H, +16);

        // Complex MUls : // Q17.47 <- QQ17.47 + ([ Q31*Q31 ] >> 15) with asymmetric rounding
        AE_MULAFC32RA(q0r, q0i, t0, c0);
        AE_MULAFC32RA(q0r, q0i, t1, c1);
        AE_MULAFC32RA(q1r, q1i, t1, c0);
        AE_MULAFC32RA(q1r, q1i, t2, c1);

        t0 = t2;
```

```
    }
    // Convert and save 2 complex outputs.
    AE_S32X2RA64S_IP(q0r, q0i, (ae_int32x2 *)Y, +16);
    AE_S32X2RA64S_IP(q1r, q1i, (ae_int32x2 *)Y, +16);
}
```

Table 65: Useful HiFi 5 DSP Instructions for Complex FIR Implementation

Instruction	
AE_L32X2X2_IP	Loads two complex numbers (32x2) samples in two separate registers.
AE_L32X2X2_XC	Loads two complex numbers from the delay line stored in a circular buffer.
AE_L32X2X2_XC1	Loads two complex tap values from circular buffer.
AE_MUL(A)FC32RA	Complex 1.31 X 1.31 MAC operations that put results in 17.47 format after asymmetric rounding. The complex multiply operations also can be implemented using operations such as AE_MULASF2D32RA_HH_LL and AE_MULAAF2D32RA_HL_LH.
AE_S32X2RA64S_IP	Round asymmetrically two 17.47-bit fractions to 1.31-bits and store.

6.2 Complex FFT Example

The Fast Fourier Transform (FFT) algorithm is an optimized implementation of the Discrete Fourier Transform, which is common in digital signal processing applications.

Complex FFT algorithm is a fundamental block of any signal processing algorithm working in frequency domain.

This section describes an implementation of Radix-4 based 'decimation in frequency' (DIF) FFT algorithm that assumes `N` is an even power of 2, where `N` is the number of complex elements stored in the data array.

This is a simple reference implementation where the data is 32 bits, and the twiddle factors are 32 bits. The 2-way SIMD architecture of HiFi 5 DSP maps nicely to the 32-bit complex data types, as a 64-bit register can hold a single, 32-bit complex data item.

The following diagram shows a basic building block of radix-4 FFT : FFT Butterfly operation.

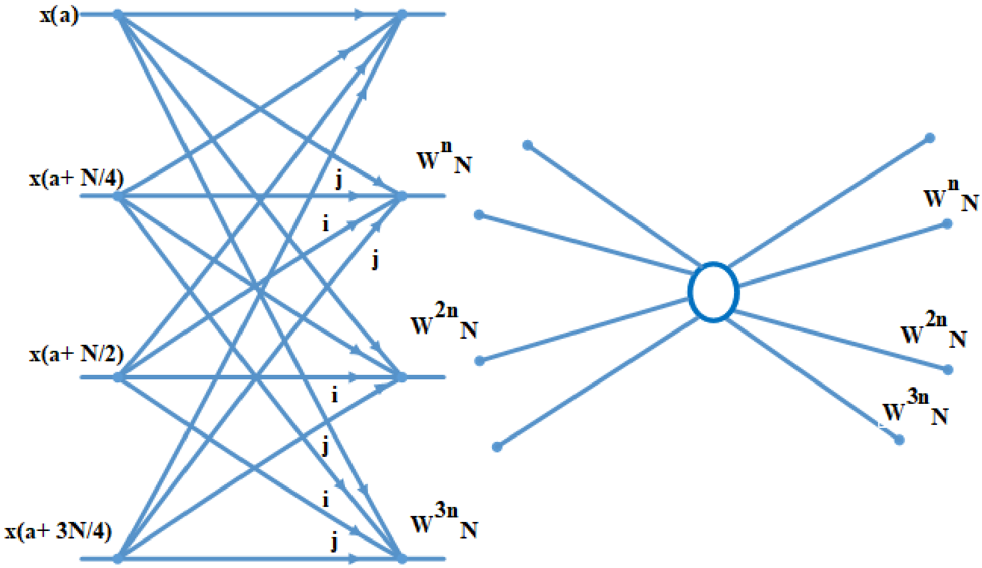


Figure 3: Radix-4 FFT : FFT Butterfly operation

Every Radix-4 butterfly stage processes 4 inputs with help of 3 twiddle coefficients, and generates 4 outputs.

If xn1, xn2, xn3 and xn4 are the four inputs and wn1, wn2 and wn3 are the twiddle coefficients, then outputs y0, y1, y2 and y3 can be calculated by the following equations:

```
y0 = xn1 + xn2 + xn3 + xn4
y1 = (xn1 - jxn2 - xn3 + jxn4) * wn1
y2 = (xn1 - xn2 + xn3 - xn4) * wn2
y3 = (xn1 + jxn2 - xn3 - jxn4) * wn3
```

As seen in above equations, each Radix-4 DIF butterfly requires:

- 4 complex inputs loads
- 3 complex twiddle loads
- 12 complex additions/subtractions
- 2 multiplications with 'j'
- 3 complex multiplication
- 4 complex output stores

This suggests approximately 28 operations per butterfly stage, including the complex loads and stores and multiplications.

Computations of Operations in N point FFT

`N` point FFT can be implemented as a series of `M` stages where each stage uses Radix-4 butterfly as described above, where `N=4^M`.

256-point FFT can be implemented with 4 stages. If `N` is not power of 4, then split-radix method can be used.

For `N` point FFT Radix-4 implementation, `(N/4)*M` butterfly operations are needed, where `M` is the total stages required for this `N` point FFT.

Total cycles required for `N` point FFT are mainly decided by the number of cycles consumed by single butterfly operation.

If `C` is the cycles consumed by butterfly computation,

Total cycles for `N` point FFT is equal to `C * ( (N/4)*M ) + Overhead cycles;`

Where `Overhead cycles` is other overhead that depends on architecture and the FFT implementation.

Thus efficiency of FFT implementation directly depends on efficiency of single Radix-4 butterfly computation.

HiFi 5 DSP ISA has instructions designed to optimize the operations in the core butterfly stage. One butterfly stage requires 28 operations; the aim is to efficiently implement these 28 operations so that the value of `C` is as small as possible.

The set of basic operations in butterfly stage described above suggests the need for better complex multiply, addition and subtraction and "multiply by j" operation.

HiFi 5 DSP instructions are designed to support 32X32, 32X16 and 16X16 FFT as well. It supports real and complex FFT/IFFT modules with various scaling options. Instructions designed for 32X32 FFT are described below in more detail along with the code snippet.

The following code snippet shows the 32X32 Radix-4 butterfly implementation using dynamic scaling. This code snippet is taken from the innermost loop of one of the intermediate stages.

```
// At entry of FFT function,
// Prescale the complex input so that minimum headroom is 3 ,
// otherwise set SAR register with 3 - number of minimum sign bits
// of input complex data
 WUR_AE_SAR( 0 );
-------
for (n = 0; n < LOOP_CNT; n++)
{
    // Load twiddle
    tw3 = AE_L32X2_I((ae_int32x2 *)TWD, 2 * 8); AE_L32X2X2_XP(tw1, tw2, (ae_int32x4*)TWD, (3 *
step * 8));

    // Load input
    AE_L32X2X2_IP(xn1, xn2, (ae_int32x4*)X0, 16);
    AE_L32X2X2_IP(xn3, xn4, (ae_int32x4*)X0, 16);

    // Core butterfly
```

```
    AE_ADDANDSUBRNG32_H(b0, b2, xn1, xn3); // result is down scaled by shift value which is
set in SAR register
    AE_ADDANDSUBRNG32_H(b1, b3, xn4, xn2); // result is down scaled by shift value which is
set in SAR register

    AE_ADDANDSUB32S(y0, y2, b0, b1);
    AE_ADDANDSUB32JS(y1, y3, b2, b3);  // Multiplication by j is combined with ADD/SUB
operation.

    y1 = AE_MULFC32RAS(y1, tw1);
    y2 = AE_MULFC32RAS(y2, tw2);
    y3 = AE_MULFC32RAS(y3, tw3);

    // Store o/p
    AE_S32X2X2RNG_IP(y0, y1, (ae_int32x4*)Y0, 16);
    AE_S32X2X2RNG_IP(y2, y3, (ae_int32x4*)Y0, 16);

    // next set of butterfly
    ----------
}
------
// Calculate the minimum headroom and update SAR register
AE_CALCRNG32(shift, shift_dummy, 0,3);
```

The instructions for FFT dynamic range detection and application are listed in section *FFT (Fast Fourier Transform) Operations* on page 121.

6.3 User Defined Emulated Float Example

In some of the DSP kernels (such as correlation, convolution, and energy computation), the dynamic range of the input signal/data is high and may not fit in fixed point format. To implement such algorithms on a DSP architecture that does not have floating point instructions, the programmers may convert those data into mantissa and exponent format and process using fixed point DSPs.

The mantissa in general uses Fixed Point Q formats explained in *Fixed Point Values and Fixed Point Arithmetic* on page 43 and exponents are integers. This kind of representation does not require complex hardware, but gives much better dynamic range than the standard fixed point representation. However, there is no standard way of representing and programmers choose the format that is convenient for the algorithm, hence it is called as User Defined Emulated Float. HiFi 5 comes with a set of instructions that helps to implement this kind of algorithm.

For example, the following code snippet shows addition of elements of two arrays that have mantissa and exponent representation:

```
void vecadd_mantexp (
        int *    p_mant1,           /* Mantissa array of first vector  */
        short *  p_exp1,            /* Exponent array of first vector  */
        int *    p_mant2,           /* Mantissa array of second vector */
        short *  p_exp2,            /* Exponent array of second vector */
        int *    p_result_mant,     /* Mantissa array of result        */
        short *  p_result_exp,      /* Exponent array of result        */
```

```
        short   length )            /* length of an array              */
{
    int res_mant, mant1, mant2;
    short res_exp, shift, i, exp1, exp2;

    for (i=0; i &lt; length; i+=1)
    {
        /* Load mantissa and exponents values of both vectors to be added */
        exp1 = p_exp1[i];
        exp2 = p_exp2[i];
        mant1 = p_mant1[i];
        mant2 = p_mant2[i];

        if (!mant1)
            exp1 = exp2;
        if (!mant2)
            exp2 = exp1;

        shift = (exp1 - exp2);
        shift = max(-31, shift);
        shift = min(31, shift);
        if (shift &lt; 0)
        {
            /* Exponent of mant2 is greater than exponent of mant1, perform right shift on
mant1 */
            mant1 = (mant1 >> (-shift));
        }
        if (shift > 0)
        {
            /* Exponent of mant1 is greater than exponent of mant2, perform right shift on
mant2 */
            mant2 = (mant2 >> shift);
        }

        res_mant = (mant1 >> 1) + (mant2 >> 1);  // Addition of two mantissa. Right shift
input by 1 to avoid saturation.
        res_exp = max(exp1,exp2) + 1;            // Increment result exponent by 1 to
compensate
        // for right shift in addition operation.
        shift = norm(res_mant);                  // norm function returns headroom available
in mantissa result

        /* Once the addition is done, the mantissa and exponent need to be updated so that the
mantissa is with same Q format */
        if (shift)
            res_mant = (res_mant >> shift);
        if (res_mant == 0)
            res_exp = 0;
        if (res_mant != 0)
            res_exp = res_exp-shift;

        // Store the result of addition
        p_result_exp[i] = res_exp;
        p_result_mant[i] = res_mant;
    }
}
```

The steps described above involve many conditional checks. HiFi 5 DSP provides intrinsics to implement this in an optimal way, as follows.

```
void vecadd_mantexp
(ae_int32x4   *__restrict p_mant1,       /* Mantissa array of first vector  */
 ae_int16x4   *__restrict p_exp1,        /* Exponent array of first vector  */
 ae_int32x4   *__restrict p_mant2,       /* Mantissa array of second vector */
 ae_int16x4   *__restrict p_exp2,        /* Exponent array of second vector */
 ae_int32x4   *__restrict p_result_mant, /* Mantissa array of result        */
 ae_int16x4   *__restrict p_result_exp,  /* Exponent array of result        */
 short   length )                        /* length of an array              */
{
    short i;
    ae_int16x4 maxval, zero, mone, diff1, diff2, exp1, exp2, shift;
    ae_int32x2 sum1, sum2, mant1_01, mant2_01, mant1_23, mant2_23, norm1, norm2;
    mone = AE_MOVDA16(-1);
    zero = AE_ZERO16();

  for (i=0; i<length>>2; i++)
  {
        /* Load mantissa and exponents values of both vectors to be added */
         AE_L16X4_IP( exp2, p_exp2, +8 );
         AE_L16X4_IP( exp1, p_exp1, +8 );
         AE_L32X2X2_IP( mant1_01, mant1_23, p_mant1, +16 );
         AE_L32X2X2_IP( mant2_01, mant2_23, p_mant2, +16 );

        /* Here we set exponent to SHRT_MIN if the mantissa is zero.
           When the mantissa value is zero, the exponent value should be set to a
deterministic value,
           we set this to SHRT_MIN, AE_EXPADD16 helps here      */
        AE_EXPADD16_H(exp1, zero, mant1_01);
        AE_EXPADD16_L(exp1, zero, mant1_01);
        AE_EXPADD16_H(exp2, zero, mant2_23);
        AE_EXPADD16_L(exp2, zero, mant2_23);

       /* Before adding mantissa, the exponent values must be same,
          Detecting the difference of exponents to perform right shift on corresponding
mantissa values */
        maxval = AE_MAX16(exp1, exp2);
        diff1 = AE_SUB16S(maxval, exp1);
        diff2 = AE_SUB16S(maxval, exp2);

        /* Performing right shift on mantissa. This is for compensating updated common value
of exponent */
        mant1_01 = AE_SRAV32RS(mant1_01, AE_SEXT32X2D16_32(diff1));
        mant1_23 = AE_SRAV32RS(mant1_23, AE_SEXT32X2D16_10(diff1));
        mant2_01 = AE_SRAV32RS(mant2_01, AE_SEXT32X2D16_32(diff2));
        mant2_23 = AE_SRAV32RS(mant2_23, AE_SEXT32X2D16_10(diff2));

        /* At this point, both the numbers have same exponents, hence mantissa can be added
directly */
        AE_ADDCEXP32_H(sum1, maxval, mant1_01, mant2_01);
        AE_ADDCEXP32_L(sum2, maxval, mant1_23, mant2_23);

        /* Once the addition is done, the exponent need to be updated so that the mantissa is
with same Q format */
        shift = AE_NSAZ32X4(sum1, sum2);
        AE_MUL16X4(norm1, norm2, shift, mone);
        sum1 = AE_SRAV32RS(sum1, norm1);
        sum2 = AE_SRAV32RS(sum2, norm2);

        /* Updating exponent for mantissa represented in same Q format */
```

```
        AE_EXPSUB16_H(maxval, shift, sum1);
        AE_EXPSUB16_L(maxval, shift, sum2);

        /* Store mantissa and exponent of result */
        AE_S32X2X2_IP( sum1, sum2, p_result_mant, +16 );
        AE_S16X4_IP( maxval, p_result_exp, +8 );
  }
}
```

The following intrinsics are useful to implement these types of emulated float arithmetic.

Table 66: Intrinsics for User Defined Emulated Float

Operation	Description
AE_ADDCEXP32_[H/L]	Adds two numbers (32 bit mantissa) if there is an overflow, the result is divided by 2 so that there is no overflow and exponent is incremented by 1 to compensate for the division.
AE_EXPSUB16_[H/L]	Subtracts two exponents if corresponding mantissa is non-zero. In case of zero mantissa, the exponent is set to SHRT_MIN as defined in C
AE_EXPADD16_[H/L]	Adds two exponents if corresponding mantissa is non-zero. In case of zero mantissa, the exponent is set to SHRT_MIN as defined in C
AE_SRAV32RS	Performs 32-bit variable arithmetic shift right or left, (sign-extending) on .H/.L parts of register holding mantissa values. The shift amount is difference of exponents.
AE_NSAZ32X4	Finds headroom for four 32 bit mantissa and returns the values in a 16x4 register. This is useful for normalizing the operands before performing arithmetic operation.

7. Neural Network (NN) Examples

Topics:

This chapter describes examples for neural network and shows the ways to optimize them for the HiFi 5 DSP.

7.1 Matrix 8bx16b Multiplication Example

The following example shows how to efficiently implement a matrix multiplication using the HiFi 5 DSP, for cases where one matrix has 8-bit data and the other has 16-bit data.

Following is a simple C reference code. The first input matrix is of 8-bit values stored row wise and the second input matrix is of 16-bit data stored column-wise. The output is stored as 64-bit values.

```
void mtx_vecmpy_8x16_64_ref

        (long long * __restrict__ p_out /* Output */
        ,const signed char * __restrict__ p_mat /* Input matrix */
        ,const short * __restrict__ p_vec /* Input vectors */
        ,int rows /* Number of rows in  input matrix */
        ,int cols /* Number of cols in  input matrix */
        ,int row_offset /* Offset to next row of  input matrix */
        ,int vec_count /* Number of input vectors*/
        ,int vec_offset /* Offset to next vector */
        ,int out_offset /* Offset to next row of output */
        )
{
    int x, y, i;
    for(y = 0; y < rows; y++)
   {
       const short *p_vec1 = p_vec;
       for(x = 0; x < vec_count; x++)
     {
       long long acc = 0;
       for(i = 0; i < cols; i++)
       {
            acc += (int)p_mat[i] * (int)p_vec1[i];
            }
          p_out[x] = acc;
           p_vec1 += vec_offset;
        }
        p_mat += row_offset;
        p_out += out_offset;
    }
}
```

The following code shows how the matrix multiplication can be implemented using the AE_MULAAAA2Q16 intrinsic.

For simplicity we assume that the matrix dimensions are multiple of 8 and the data is 16-byte aligned. We process two input rows at a time. The 8-bit input is loaded in the AE_DR registers as four 16-bit values (LSB aligned and sign extended) and the AE_MULAAAA2Q16 intrinsic performs eight 16x16 multiplications, four for each row, storing the result in two accumulators.

```
void mtx_vecmpy_8x16_64_opt

        (long long * __restrict__ p_out /* Output */
        ,const signed char * __restrict__ p_mat /* Input matrix */
```

```
        ,const short * __restrict__ p_vec /* Input vectors */
        ,int rows /* Number of rows in input matrix */
        ,int cols /* Number of cols in input matrix */
        ,int row_offset /* Offset to next row of input matrix */
        ,int vec_count /* Number of input vectors, columns of second matrix */
        ,int vec_offset /* Offset to next column of second input matrix */
        ,int out_offset /* Offset to next row of output */
        )
{
   int row = 0, col = 0, vec = 0;
    for(vec = 0; vec < vec_count; vec++)
    {
        ae_int64 *p_dst1 = (ae_int64 *)&p_out[vec];
        for (row = 0; row < rows; row += 2)
        {
            const signed char *p_mat1 = &p_mat[row * row_offset];
            const signed char *p_mat2 = &p_mat[(row + 1) * row_offset];
            ae_int16x4 *p_vec1 = (ae_int16x4 *)&p_vec[vec * vec_offset];
            ae_int64 accu1 = AE_ZERO64();
            ae_int64 accu2 = AE_ZERO64();

           for (col = 0; col < cols>>2; col++)
            {
                ae_int16x4 vec1, mat1, mat2;
                AE_L8X4S_IP(mat1, p_mat1, 4);
                AE_L8X4S_IP(mat2, p_mat2, 4);
                AE_L16X4_IP(vec1, p_vec1, 8);
               AE_MULAAAA2Q16(accu1, accu2, mat1, mat2, vec1, vec1);
           }
           p_dst1[row * out_offset] = accu1;
            p_dst1[(row + 1) * out_offset] = accu2;
       }
   }
}
```

If the Neural Network Extension is available, the performance can be further improved using the AE_MULA8QW8X16 matrix multiplication intrinsic. Here we process four rows at a time doing eight multiplications per row.

```
void mtx_vecmpy_8x16_64_opt

        (long long * __restrict__ p_out /* Output */
        ,const signed char * __restrict__ p_mat /* Input matrix */
        ,const short * __restrict__ p_vec /* Input vectors */
        ,int rows /* Number of rows in input matrix */
        ,int cols /* Number of cols in input matrix */
        ,int row_offset /* Offset to next row of input matrix */
        ,int vec_count /* Number of input vectors, columns of second matrix */
        ,int vec_offset /* Offset to next column of second input matrix */
        ,int out_offset /* Offset to next row of output */
        )
{
    int row = 0, col = 0, vec = 0;
    for(vec = 0; vec < vec_count; vec++)
    {
        ae_int64 *p_dst1 = (ae_int64 *)&p_out[vec];
       for (row = 0; row < rows; row += 4)
        {
            ae_int16x8 *p_vec1 = (ae_int16x8 *)&p_vec[vec * vec_offset];
            ae_int8x8 *p_mat1 = (ae_int8x8*)&p_mat[row * row_offset];
            ae_int8x8 *p_mat2 = (ae_int8x8*)&p_mat[(row + 1)* row_offset];
```

```
             ae_int8x8 *p_mat3 = (ae_int8x8*)&p_mat[(row + 2) * row_offset];
             ae_int8x8 *p_mat4 = (ae_int8x8*)&p_mat[(row + 3) * row_offset];

            ae_int64 accu1 = AE_ZERO64();
             ae_int64 accu2 = AE_ZERO64();
             ae_int64 accu3 = AE_ZERO64();
             ae_int64 accu4 = AE_ZERO64();

            for (col = 0; col < cols>>3; col++)
            {
                 ae_int16x4 vec1, vec2;
                ae_int8x8 mat1, mat2, mat3, mat4;
                 AE_L8X8_IP(mat1, p_mat1, 8);
                 AE_L8X8_IP(mat2, p_mat2, 8);
                 AE_L8X8_IP(mat3, p_mat3, 8);
                 AE_L8X8_IP(mat4, p_mat4, 8);
                 AE_L16X4X2_IP(vec1, vec2, p_vec1, 16);
                 AE_MULA8QW8X16(accu1, accu2, accu3, accu4, mat1, mat2, mat3, mat4, vec1, vec2);
             }
            p_dst1[row * out_offset] = accu1;
             p_dst1[(row + 1) * out_offset] = accu2;
             p_dst1[(row + 2) * out_offset] = accu3;
             p_dst1[(row + 3) * out_offset] = accu4;
        }
    }
```

The implementation can be further improved by processing multiple vectors at a time and using 16-byte loads for vectors. Operations useful for Matrix Vector Multiplication are listed in *Neural Networks Multiplication Operations* on page 106

7.2 Convolution 8bx8b Example

The following example shows ways to optimize the implementation of a 2D correlation (or equivalently a 2D convolution) for 8 bit input and 8 bit kernel using NN convolution MAC instructions on HiFi 5 DSP. We explain convolution assuming that kernels are already flipped.

The 2D convolution of 12x12 8 bit kernel with an 8 bit input matrix of size IHxIW (where 'IH >=12' and 'IW >12 and multiple of 8') produces output matrix of size (IH-12+1)x(IW-12+1). Here we assume IW is a multiple of 8, which ensures that if the first row of input matrix starts at an 8 byte aligned address (which can b controlled), all the subsequent rows of matrix also start at an 8 byte aligned addresses.

For example, if the size of the kernel is 12x12 and size of the input matrix is 24x24, then the size of output matrix will be 13x13.

The following C code implements core loop required for convolution of 12x12 kernel with input matrix by hovering it to right by just 8 elements.

```
void conv2d_8_8x8_k12x12(int *acc, Int8 *p_inp, Int8 *p_ker, int IW)
{
    int i, j;
    int count;
    for(count = 0; count < 8; count++)
    {
```

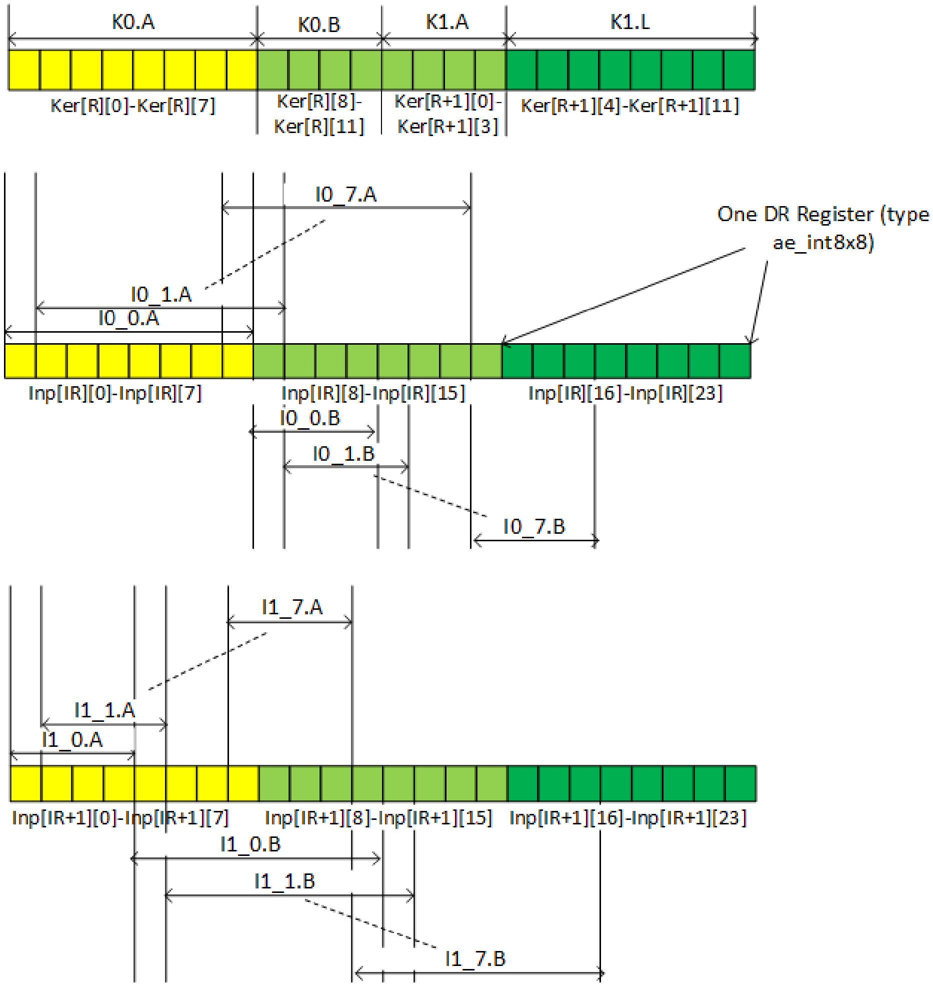
```
        acc[count] = 0;
    }
   for(i = 0; i < 12; i++)
    {
        for(j = 0; j < 12; j++)
        {
            for(count = 0; count < 8; count++)
            {
                acc[count] += ((int)p_inp[IW*i+j+count])*p_ker[12*i+j];
            }
        }
    }
}
```

This code example shows that generating one output sample involves 144 MACs, and uses a 32-bit accumulator. The code can be reorganized in the following manner.

```
void conv2d_8_8x8_k12x12(int *acc, Int8 *p_inp, Int8 *p_ker, int IW)
{
    int i, j;
    int count;
   for(count = 0; count < 8; count++)
    {
        acc[count] = 0;
    }
   for(i = 0; i < 6; i++)
    {
        /* Loop 1 */
        for(j = 0; j < 8; j++)
        {
            for(count = 0; count < 8; count++)
            {
                acc[count] += ((int)p_inp[IW*(2*i+0)+j+count])*p_ker[12*(2*i+0)+j];
            }
        }
        /* Loop 2 */
        for(j = 0; j < 4; j++)
        {
            for(count = 0; count < 8; count++)
            {
                acc[count] += ((int)p_inp[IW*(2*i+0)+8+j+count])*p_ker[12*(2*i+0)+8+j];
                acc[count] += ((int)p_inp[IW*(2*i+1)+j+count])*p_ker[12*(2*i+1)+j];
            }
        }
        /* Loop 3 */
        for(j = 0; j < 8; j++)
        {
            for(count = 0; count < 8; count++)
            {
                acc[count] += ((int)p_inp[IW*(2*i+1)+4+j+count])*p_ker[12*(2*i+1)+4+j];
            }
       }
   }
}
```

The loop running over kernel rows is unrolled by two, and the loop running over elements of rows is broken in three loops. The 2x12 convolution is broken in three parts. Now the row loop is processing two rows in one iteration, Loop 1 convolves the first 8 elements of kernel with first row of input (for 8 outputs), Loop 2 convolves the last 4 elements of first row of kernel with first row of input and first 4 elements of second row of kernel with second row of input, and Loop 3 convolves last 8 elements of second row of kernel with second row of input. This reorganized code can be optimized using HiFi 5 DSP NN convolution MAC instructions as follows:

```
void conv2d_8_8x8_k12x12(int *acc, Int8 *p_inp, Int8 *p_ker, int IW)
{
    int i, j, IW_3;
    ae_int8x8 *p_inp8x8, *p_ker8x8;
    ae_int32x2 acc0, acc1, acc2, acc3;

    acc0 = AE_ZERO32();
    acc1 = AE_ZERO32();
    acc2 = AE_ZERO32();
    acc3 = AE_ZERO32();

    IW_3 = IW>>3;    /* Divide by 8 required while indexing ae_int8x8 type pointer */
    p_inp8x8 = (ae_int8x8 *)p_inp;
    p_ker8x8 = (ae_int8x8 *)p_ker;
    for(i = 0; i < 6; i++)
    {
        /* Replaces the Loop 1 */
        AE_MULA8Q8X8CNV_H(acc0, acc1, p_ker8x8[3*i+0],
                          p_inp8x8[IW_3*(2*i+0)], p_inp8x8[IW_3*(2*i+0)+1]);

        AE_MULA8Q8X8CNV_L(acc2, acc3, p_ker8x8[3*i+0],
                          p_inp8x8[IW_3*(2*i+0)], p_inp8x8[IW_3*(2*i+0)+1]);

        /* Replaces the Loop 2 */
        AE_MULA2X4Q8X8CNV_H(acc0, acc1, p_ker8x8[3*i+1],
                             p_inp8x8[IW_3*(2*i+0)+1], p_inp8x8[IW_3*(2*i+1)+0]);

        AE_MULA2X4Q8X8CNV_L(acc2, acc3, p_ker8x8[3*i+1], p_inp8x8[IW_3*(2*i+0)+1],
                             p_inp8x8[IW_3*(2*i+0)+2], p_inp8x8[IW_3*(2*i+1)+0],
                             p_inp8x8[IW_3*(2*i+1)+1]);

        /* Replaces the Loop 3 */
        AE_MULA8Q8X8CNV_L(acc0, acc1, p_ker8x8[3*i+2],
                            p_inp8x8[IW_3*(2*i+1)+0], p_inp8x8[IW_3*(2*i+1)+1]);
        AE_MULA8Q8X8CNV_H(acc2, acc3, p_ker8x8[3*i+2],
                            p_inp8x8[IW_3*(2*i+0)+1], p_inp8x8[IW_3*(2*i+0)+2]);
    }
}
```



R – Current kernel row

IR – Current input row

K0 – 1x12 Vector which is current row of kernel. .A part is 1x8, .B part is 1x4.

I0_N – 1x12 Vectors from current row of input (there are 8, I0_0 to I0_7). .A part is 1x8, .B part is 1x4.

I1_N – 1X12 Vectors from next row of input (there are 8, I1_0 to I1_7). .A part is 1x4, .B part is 1x8.

OUT[N] = OUT[N].A + OUT[N].B + OUT[N].C;
N = 0-7

OUT[N].A = I0_N.A dot K0.A
OUT[N].B = I0_N.B dot K0.B + I1_N.A dot K1.A
OUT[N].C = I1_N.B dot K1.B

The dot '.' refers to the dot product.
OUT[0].A to OUT[7].A are generated by two 1x8 convolution instructions.
OUT[0].B to OUT[7].B are generated by two 2x4 convolution instructions.
OUT[0].C to OUT[7].C are generated by two 1x8 convolution instructions.

Figure 4: HiFi 5 DSP NN Convolution Example

Figure 4: HiFi 5 DSP NN Convolution Example on page 159 shows the processing performed by a single iteration of an optimized loop accumulating 8 outputs by convolving 2 rows of kernels with 2 rows of inputs.

After optimizing using the HiFi 5 DSP NN convolution MAC instruction, there are nine 8x8 loads, and six convolution MAC instructions. With software pipeline, this single loop goes to 6 cycle, resulting in a total 6*6 = 36 cycles for generating 8 outputs. This involves 144*8 = 1152 MACs, which gives us 32 MACs per cycle throughput.

Similar to `AE_8Q8X8CNV_H` and `AE_MUL[A]2X4Q8X8CNV_H`, HiFi 5 DSP has `AE_MUL[A]8Q8X16CNV, AE_MUL[A]2X4Q8X16CNV` (for convolution of 8 bit kernels and 16 bit inputs). _L variants are not applicable for 8X16 because one DR register holds only four 16-bit value, there are corresponding 4X16 precision instructions also.

Refer to the example code for full implementation of 2D convolution of 8 bit input matrix with 8 bit 12x12 kernel. 2D convolution with 11x11 kernel can be optimized in the same way by converting 11x11 kernel to 12x12 or 11x12 kernel by zero padding. In case of 11x12, the kernel row loop will process 10 rows (5 iterations) and the last row must be processed outside the loop.

The Multiplication and Multiplication-Accumulation from Neural Network Extension is listed in *Neural Networks Multiplication Operations* on page 106

7.3 Sparse Matrix and 8b Asymmetric Quantization Support

This section describes the sparse matrix and 8-bit asymmetric quantization support on the HiFi 5 DSP Neural Network Extension.

Support for sparse matrix multiplication

Neural Network algorithms predominantly use matrix - vector multiplications with large dimensions where matrix generally holds weights or coefficients, and vector holds the input data. Generally in Neural Network algorithms, these weight matrices are sparse (i.e., many matrix elements are zeros). A matrix with 75% zero elements is called 75% sparse matrix. Special instructions are available on Neural Network Extension of HiFi 5 DSP so as to save on storage memory and load bandwidth of such sparse matrices.

A compression scheme that works efficiently on the HiFi 5 DSP is proposed for storing sparse matrices. In this compression scheme, only non-zero matrix elements are stored in memory and a bit mask array of matrix dimensions length (1 bit per matrix element) is stored in memory. One bit value in bit mask array indicates if the corresponding matrix element is non-zero (1) or zero (0). Following is the compression scheme example for 75% sparse, 16x32, 8-bit matrix, which saves 62.5% on storage memory and load bandwidth.

```
char coeff[] = {

 0,     a0,    0,     0,    a1,    a2,    0,     0,     0,    a3,    0,     0,     0,     0,     0,
```

```
         0,    0,   a4,    0,   a5,    0,    0,    0,   a6,    0,    0,    0,    0,   a7,
         0,    0,    0,    0,    0,   a8,   a9,    0,  a10,  a11,    0,    0,    0,  a12,
         0,  a13,    0,    0,    0,  a14,    0,    0,    0,    0,    0,    0,    0,    0,
         0,    0,    0,    0,    0,    0,  a15,


 a16,    0,    0,    0,    0,  a17,    0,    0,  a18,  a19,    0,    0,    0,  a20,    0,
         0,    0,    0,  a21,    0,    0,  a22,    0,  a23,    0,    0,    0,    0,    0,
         0,    0,    0,    0,    0,    0,  a24,    0,    0,    0,  a25,    0,    0,    0,
         0,  a26,    0,    0,  a27,    0,  a28,    0,    0,  a29,    0,    0,    0,    0,
         a30,  0,    0,    0,    0,    0,  a31,    0,  a32,  a33,    0,    0,    0,    0,
         0,    0,  a34,  a35,    0,    0,    0,    0,    0,    0,    0,    0,    0,  a36,
         0,    0,  a37,    0,    0,  a38,  a39,    0,    0,    0,    0,    0,    0,    0,
         0,    0,    0,  a40,    0,    0,    0,  a41,    0,  a42,  a43,    0,    0,    0,
         0,  a44,  a45,    0,  a46,    0,    0,  a47,    0,    0,    0,    0,    0,    0,
         0,

 a48,    0,  a49,    0,  a50,    0,  a51,    0,    0,    0,    0,  a52,    0,    0,  a53,
         0,    0,    0,    0,  a54,    0,    0,    0,    0,    0,    0,    0,    0,    0,
         0,  a55,    0,    0,    0,    0,    0,    0,    0,    0,    0,  a56,    0,    0,
       a57,    0,    0,    0,  a58,  a59,    0,    0,    0,  a60,    0,    0,    0,    0,
         0,    0,  a61,  a62,    0,  a63,    0,    0,    0,    0,  a64,    0,  a65,    0,
         a66,  0,    0,    0,    0,    0,    0,  a67,    0,    0,    0,  a68,    0,    0,
         a69,  0,    0,  a70,    0,    0,    0,    0,    0,  a71,    0,    0,    0,    0,
         0,    0,    0,  a72,    0,  a73,  a74,  a75,  a76,    0,    0,    0,    0,    0,
         0,    0,    0,    0,    0,  a77,    0,    0,    0,  a78,    0,    0,  a79,    0,
         0,

a80,     0,    0,    0,    0,    0,    0,    0,    0,    0,  a81,    0,  a82,    0,    0,
       a83,    0,  a84,    0,    0,  a85,  a86,    0,    0,    0,    0,    0,    0,
         0,    0,    0,  a87,    0,    0,    0,    0,  a88,    0,  a89,    0,    0,    0,
         0,  a90,  a91,  a92,    0,    0,  a93,    0,    0,    0,    0,    0,    0,    0,
         a94,  a95,    0,    0,  0,    0,    0,    0,    0,  a96,    0,  a97,    0,    0,
         0,  a98,    0,  a99,    0, a100,    0,    0,    0, a101,    0,    0, a102, a103,
         0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0, a104,
         0,    0,    0,    0,    0,    0,    0,    0,    0,    0, a105, a106, a107,    0,
         0,    0,    0,    0, a108,    0,    0,    0, a109,    0, a110, a111,    0,    0,
         0,    0,    0,    0,    0,    0, a112, a113,    0,    0, a114,    0,    0,    0,
         0,    0,    0,    0,    0,    0, a115,    0,    0,    0, a116,    0, a117,    0,
         0, a118,    0,    0, a119,    0, a120,    0,    0,    0,    0,    0, a121,    0,
         0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0, a122,    0, a123,
      a124,    0,    0, a125,    0,    0, a126, a127,    0,    0

         };
```

```
char mask[] =
{
    0x4C,0x40,0x51,0x08, //0100 1100 0100 0000 0101 0001 0000 1000
    0x36,0x28,0x80,0x01, //0011 0110 0010 1000 1000 0000 0000 0001
    0x84,0xC4,0x25,0x00, //1000 0100 1100 0100 0010 0101 0000 0000
    0x11,0x09,0x48,0x41, //0001 0001 0000 1001 0100 1000 0100 0001
    0x60,0x60,0x09,0x30, //0110 0000 0110 0000 0000 1001 0011 0000
    0x02,0x2C,0x34,0x80, //0000 0010 0010 1100 0011 0100 1000 0000
    0xAA,0x12,0x10,0x02, //1010 1010 0001 0010 0001 0000 0000 0010
    0x00,0x91,0x88,0x1A, //0000 0000 1001 0001 1000 1000 0001 1010
    0x15,0x02,0x24,0x82, //0001 0101 0000 0010 0010 0100 1000 0010
    0x02,0xF0,0x02,0x24, //0000 0010 1111 0000 0000 0010 0010 0100
    0x80,0x15,0x4C,0x01, //1000 0000 0010 1 01 0100 1100 0000 0001
    0x0A,0x1C,0x80,0xC0, //0000 1010 0001 1100 1000 0000 1100 0000
    0x51,0x51,0x30,0x00, //0101 0001 0101 0001 0011 0000 0000 0000
    0x40,0x0E,0x08,0x58, //0100 0000 0000 1110 0000 1000 1011 0000
    0x0C,0x80,0x22,0x92, //0000 1100 1000 0000 0010 0010 1001 0010
```

```
    0x42,0x00,0x16,0x4C  //1000 0010 0000 0000 0001 0110 0100 1100
};
char coeff_squeezed[] = {a0, a1, a2 … a127 }
```

Note, to process multiple rows together and reuse vector data for cycle performance, some rearrangement of matrix (e.g. interleaving of rows) is done.

The following special instructions are available on the Neural Network Extension to efficiently un-compress the complete matrix from stored bit mask array and non-zero matrix array. These instructions should be used along with shuffle instructions to un-compress the matrix.

Table 67: Special load instructions available as part of the Neural Network Extension to support Sparse Matrices

Special load intrinsics and their parameters	Description
`AE_LAVUNSQZ8X8_XP (ae_int8x8 d0_reg, ae_int8x8 d1_pat, ae_valign align /*inout*/, const ae_int8x16 * p_mat /*inout*/, int mask, immediate i);`	1. count = number of non-zero 8-bit elements using bit mask from d1_pat, imm (0, 1, 2 or 3) value indicates which byte in mask to use 2. Read 'count' number of 8-bit elements into do_reg using p_mat and align 3. Make the rest '8-count' number of 8-bit elements of d0_reg zeros 4. Increment p_mat by 'count' 5. Create shuffle pattern (that can be readily used by AE_SHFL8X8 instruction) to rearrange 'count' number of non-zero elements and zeros in d0_reg to create uncompressed eight 8-bit elements of matrix
`AE_LAVUNSQZ16X4_XP (ae_int16x4 d0_reg, ae_int16x4 d1_pat, ae_valign align /*inout*/, const ae_int16x8 * p_mat /*inout*/, int mask, immediate i);` `AE_LAVUNSQZHX4_XP (xthalfx4 d0_reg, xthalfx4 d1_pat, ae_valign align /*inout*/, const xthalfx8 * p_mat /*inout*/, int mask, immediate i);`	These two intrinsics work on the 16-bit integer and half-precision floating-point types respectively: 1. count = number of non-zero 16-bit elements using bit mask from d1_pat. imm (0, 1, 2, 3, 4, 5, 6 or 7) value indicates which nibble in mask to use 2. Read 'count' number of 16-bit elements into do_reg using p_mat and align 3. Make the rest '4-count' number of 16-bit elements of d0_reg zeros 4. Increment p_mat by 'count'

Special load intrinsics and their parameters	Description
	5. Create shuffle pattern (that can be readily used by AE_SHFL16X4 instruction) to rearrange 'count' number of non-zero elements and zeros in d0_reg to create uncompressed four 16-bit elements of matrix

The following code example implements a sparse matrix (weights or coefficients) multiplication with a vector (input data).

```
/
***********************************************************************************************
***************************
*
*  Sparse matrix compression scheme:
*  Only non-zero elements of sparse matrix and corresponding bit mask table is stored in
memory. Bit mask table
*  consists of 1 bit per element - 0 indicates element is zero and 1 indicates element has non-
zero value.
*  Original matrix rows can be formed runtime using non-zero elements and bit mask table using
HiFi5 special
*  instructions as demonstrated in the function below.
*
*  Further the sparse matrix elements are offline rearranged as below to suite HiFi5
optimizations.
*  The sparse matrix is split into two sub-matrices as below:
*  - p_mask0 contains bitmask for 8 interleaved elements of rows (4n+0) and rows (4n+1)
*  - p_mat_sp0 contains non-zero elements corresponding to non-zero bit of p_mask0
*  - p_mask1 contains bitmask for 8 interleaved elements of rows (4n+2) and rows (4n+3)
*  - p_mat_sp1 contains non-zero elements corresponding to non-zero bit of p_mask1
*
*  Rearranging sparse matrix into 2 sub-matrices facilitates parallel execution of two
unsqueeze instructions.
*  Interleaving 8 elements per row allows efficient processing of multiple rows without
worrying about row boundaries.
*
*  Function parameters:
*  p_out        - Output of matrix by vector multiplication
*  p_mat_sp0    -
*  p_mat_sp1    -
*  p_mask0      -
*  p_mask1      - Re-arranged matrix as described above
*  p_vec        - Input vector elements
*  rows         - Total number of rows in input matrix
*  cols         - Elements in single row of matrix
*  row_offset   - Offset in terms of number of elements for next row
*  vec_count    - Total number of vectors, each of size equal to cols
*  vec_offset   - Offset in terms of number of elements for next input vector
*  out_offset   - Offset in terms of number of elements for next output vector
*  b_accumulate - Flag to accumulate result of multiplication in output vector
*  lsh          - Desired left shift in result of multiplication
*
***********************************************************************************************
***************************/
```

```
#include "xtensa/tie/xt_hifi5.h"

#ifndef NULL
#define NULL (void *)0
#endif


int xa_nn_matXvec_8x8_32_sparse(
  int  * __restrict__ p_out,
  char * __restrict__ p_mat_sp0,
  char * __restrict__ p_mat_sp1,
  int  * __restrict__ p_mask0,
  int  * __restrict__ p_mask1,
  char * __restrict__ p_vec,
  int rows, int cols, int row_offset,
  int vec_count, int vec_offset, int out_offset,
  int b_accumulate, int lsh) {

  int row, col, vec=0;

  if ((NULL == p_out) || (NULL == p_mat_sp0) || (NULL == p_mat_sp1) || (NULL == p_vec) ||
(NULL == p_mask0) || (NULL == p_mask1)) {
    return -1;
  }

  //columns should be multiple of 16 and rows should be multiple of 4
  if ((0 >= rows ) || (0 >= cols ) || (cols & 0xf) || (rows & 0x3)) {
    return -2;
  }

  if (0 >= vec_count) return -3;

  WAE_SAR(lsh);

  for(vec = 0; vec < vec_count; vec++) {
    ae_int32x2 *p_dst0 = (ae_int32x2 *)&p_out[vec*(out_offset)];
    ae_int8x8 m_pat00, m_pat01, m_pat02, m_pat03, m_pat04, m_pat05, m_pat06, m_pat07;
    ae_int8x8 m_reg00, m_reg01, m_reg02, m_reg03, m_reg04, m_reg05, m_reg06, m_reg07;
    ae_int8x8 m_reg10, m_reg11, m_reg12, m_reg13, m_reg14, m_reg15, m_reg16, m_reg17;
    ae_int8x8 v_reg0, v_reg1;

    #define UNROLL  4 /// Optimal unroll

    row = 0;
    if (rows >= UNROLL) {
      ae_int8x16 *p_mat0_0 = (ae_int8x16 *)p_mat_sp0;
      ae_valign alignx0 = AE_LA64_PP(p_mat0_0);
      ae_int8x16 *p_mat1_0 = (ae_int8x16 *)p_mat_sp1;
      ae_valign alignx1 = AE_LA64_PP(p_mat1_0);

      for (row = 0; row < ( rows & ~(UNROLL-1)); row+=UNROLL) {
        int idx = (row>>1);
        int *mask0 = &p_mask0[(row>>2)*(row_offset>>4)];
        int *mask1 = &p_mask1[(row>>2)*(row_offset>>4)];
        ae_int8x16 *p_src0 = (ae_int8x16 *)&p_vec[vec * vec_offset];

        ae_int32x2 accu0_0, accu1_0, accu2_0, accu3_0;
        accu0_0 = AE_ZERO32(); accu2_0 = AE_ZERO32();
        accu1_0 = AE_ZERO32(); accu3_0 = AE_ZERO32();

        for (col = 0; col < cols>>4; col++) {
          AE_L8X8X2_IP(v_reg0, v_reg1, p_src0, 16);

          AE_LAVUNSQZ8X8_XP(m_reg00, m_pat00, alignx0, p_mat0_0, mask0[col], 3);
```

```
          AE_LAVUNSQZ8X8_XP(m_reg02, m_pat02, alignx1, p_mat1_0, mask1[col], 3);
          AE_LAVUNSQZ8X8_XP(m_reg01, m_pat01, alignx0, p_mat0_0, mask0[col], 2);
          AE_LAVUNSQZ8X8_XP(m_reg03, m_pat03, alignx1, p_mat1_0, mask1[col], 2);
          AE_LAVUNSQZ8X8_XP(m_reg04, m_pat04, alignx0, p_mat0_0, mask0[col], 1);
          AE_LAVUNSQZ8X8_XP(m_reg06, m_pat06, alignx1, p_mat1_0, mask1[col], 1);
          AE_LAVUNSQZ8X8_XP(m_reg05, m_pat05, alignx0, p_mat0_0, mask0[col], 0);
          AE_LAVUNSQZ8X8_XP(m_reg07, m_pat07, alignx1, p_mat1_0, mask1[col], 0);

          m_reg10 = AE_SHFL8X8(m_reg00, m_pat00);
          m_reg11 = AE_SHFL8X8(m_reg01, m_pat01);
          m_reg12 = AE_SHFL8X8(m_reg02, m_pat02);
          m_reg13 = AE_SHFL8X8(m_reg03, m_pat03);
          m_reg14 = AE_SHFL8X8(m_reg04, m_pat04);
          m_reg15 = AE_SHFL8X8(m_reg05, m_pat05);
          m_reg16 = AE_SHFL8X8(m_reg06, m_pat06);
          m_reg17 = AE_SHFL8X8(m_reg07, m_pat07);

          AE_MULA8Q8X8(accu0_0, accu1_0, m_reg10, m_reg11, m_reg12, m_reg13, v_reg0);
          AE_MULA8Q8X8(accu2_0, accu3_0, m_reg14, m_reg15, m_reg16, m_reg17, v_reg1);
        }

        accu0_0 = accu0_0 + accu2_0;
        accu1_0 = accu1_0 + accu3_0;

        accu0_0 = AE_SLAS32S(accu0_0);
        accu1_0 = AE_SLAS32S(accu1_0);

        if (b_accumulate) {
          p_dst0[idx] += accu0_0;
          p_dst0[idx+1] += accu1_0;
        } else {
          p_dst0[idx] = accu0_0;
          p_dst0[idx+1] = accu1_0;
        }
      }
    }
  }
  return 0;
}
```

Support for Asymmetric 8 bit Quantization:

The NN extension of HiFi 5 DSP supports multiplication operations with the asymmetric quantized 8-bit data types used in Android Neural Network (ANN) and TensorFlow Lite. ANN supports asymmetric 8-bit quantized data types as listed in table below. For the quantization scheme used in TensorFlow Lite, please refer to *https://www.tensorflow.org/lite/performance/quantization_spec* for details.

Table 68: Android Neural Network special 8-bit quantized data type

`ANEURALNETWORKS_TENSOR_QUANT8_ASYMM`	A tensor of 8-bit integers that represent real numbers. Attached to this tensor are two numbers that can be used to convert the 8-bit integer to the real value and vice versa. These two numbers are:

	- scale: a 32-bit floating point value greater than zero - zeroPoint: a 32-bit integer, in range [0, 255]. The formula is: `real_value = (integer_value - zeroPoint) * scale` For more details, refer to the Android Developer's site at: *https://developer.android.com/ndk/reference/group/neural-networks*
`ANEURALNETWORKS_TENSOR_QUANT8_ASYMM_SIGNED`	A tensor of 8-bit signed integers that represent real numbers. Attached to this tensor are two numbers that can be used to convert the 8-bit integer to the real value and vice versa. These two numbers are: - scale: a 32-bit floating point value greater than zero - zeroPoint: a 32-bit integer, in range [-128, 127]. The formula is: `real_value = (integer_value - zeroPoint) * scale` For more details, refer to the Android Developer's site at: *https://developer.android.com/ndk/reference/group/neural-networks*

Note, for fixed-point representation, the scale and the resulting real_value would be fixed point 32-bit values.

The following special instructions are available on the HiFi 5 DSP to efficiently perform matrix x vector multiplications with asymmetric 8-bit quantized data types.

```
dequantized_value = (quantized_value - zeroPoint) * scale
```

When multiplying or convolving two such quantized values, subtraction of the zero-point values from matrix and vector values is performed as a part of the special multiplication instructions. Zero-point values for co-efficients and data vectors are read from a pair of 8-bit architectural state registers `AE_ZBIASC8` and `AE_ZBIASV8` .

Table 69: Special Instructions on the HiFi 5 DSP for Matrix - Vector Multiplications with Asymmetric 8 bit Quantized Data Types

Instructions	Description

`AE_MULUUZB8Q8X8`
`AE_MULAUUZB8Q8X8`
`AE_MULZB8Q8X8`
`AE_MULAZB8Q8X8`
	Instruction multiplying 4x8 matrix with 8x1 vector to generate 4x1 vector

`AE_MULUUZB4O8X8`
`AE_MULAUUZB4O8X8`
`AE_MULZB4O8X8`
`AE_MULAZB4O8X8`
	SIMD instruction multiplying two 4x4 matrices with two respective 4x1 vectors to generate two 4x1 vectors

Note, with all AE_MULA variants above, result is accumulated to output register.

The following HiFi 5 DSPspecial instructions can be used to efficiently perform convolutions with asymmetric 8 bit quantized data types. These instructions subtract respective zero point values from filter and input data before multiplications. Zero point values are used from special state registers AE_ZBIASC8 and AE_ZBIASV8 .

Table 70: Special instructions on the HiFi 5 DSP for convolution operations with asymmetric 8 bit quantized data types

Instructions	Description
AE_MULUUZB8Q8X8CNV_[H|L] AE_MULAUUZB8Q8X8CNV_[H|L] AE_MULZB8Q8X8CNV_[H|L] AE_MULAZB8Q8X8CNV_[H|L]	Instructions computing convolution of 1x8 filter over 1x11 input to generate 1x4 output _[H|L] indicates if convolution begins with upper or lower four elements of first input operand
AE_MULUUZB4O8X8CNV_[H|L] AE_MULAUUZB4O8X8CNV_[H|L] AE_MULZB4O8X8CNV_[H|L] AE_MULAZB4O8X8CNV_[H|L]	Instructions computing convolution of 1x4 filter over 1x11 input to generate 1x8 output _[H|L] indicates if convolution uses upper or lower four elements of filter operand
AE_MULUUZB2X4Q8X8CNV_[H|L] AE_MULAUUZB2X4Q8X8CNV_[H|L] AE_MULZB2X4Q8X8CNV_[H|L] AE_MULAZB2X4Q8X8CNV_[H|L]	Instructions computing convolution of 2x4 filter over 2x7 input to generate 1x4 output _[H|L] indicates if convolution begins with upper or lower four elements of first input operand

Instructions	Description
AE_MULUUZB3X3O8X8 AE_MULZB3X3O8X8 AE_MULAUUZB3X3O8X8 AE_MULAZB3X3O8X8	Instructions computing 8-way SIMD multiplication of 1x3x8 (Height x Width x Depth) input and 1x3x8 filter to generate 1x8 output

Note, with all AE_MULA variants above, result is accumulated to output register.

The following helper instructions are available on the HiFi 5 DSP to move zero point values for matrix / filter coefficients and vector / input data into state registers AE_ZBIASC8 and AE_ZBIASV8, respectively.

Table 71: Helper Instructions on the HiFi 5 DSP

Instructions	Description
AE_MOVZBVCDR (in ae_int64 v)	This helper instruction does following: **1.** Copies 8 bits from v[7:0] to state register AE_ZBIASC8, it is used as zero point for matrix or coefficients **2.** Copies 8 bits from v[39:32] to state register AE_ZBIASV8, it is used as zero point for vector or input
AE_LZBIASVC {in uint32 *a, in immediate b}	This helper instruction effectively calls AE_MOVZBVCDR (in ae_int64 v) where v = a[b+1] | a[b+0], a must be 8 bytes aligned address

8. Implementation Methodology

Topics:

The HiFi 5 DSP is an optional coprocessor for the Xtensa LX7 (and later versions) core. HiFi 5 DSP is provided as a check box option in the Xplorer Processor Generator (XPG) interface in Xtensa Xplorer (XX). This section includes guidelines for using the XPG to configure a HiFi 5 DSP coprocessor.

The last section in this chapter discusses synthesis and place-and-route.

8.1 Configuring a HiFi 5 DSP

Configuring a HiFi 5 DSP in the XPG is done by selecting an appropriate template or by selecting the relevant check box options in the Xplorer Configuration editor, in the Processors window under the category HiFi Audio Coprocessor:

- HiFi 5 DSP coprocessor instruction family

Developers should also configure the Prefetch Options from the Interfaces window. A selection of 0 prefetch entries will eliminate hardware prefetching from the configuration. Otherwise, 8 or 16 entries are available. The latter provides a little higher performance at the cost of a little more area. In addition, you must decide if you want to enable prefetching directly to L1 and whether to enable block prefetches. Prefetching into L1 typically improves performance, minimally on configurations with very large delays to main memory and more significantly on systems with small delays to secondary or main memory, but at the cost of additional hardware. In addition, prefetching into L1 is a requirement for support block prefetches.

There are some optional configuration templates provided for HiFi 5 (such as hifi5_ss_spfpu_7, hifi5_tv_car_5, hifi5_ao_7). They provide useful starting points for a developer who wants to either use the configuration as described by the template, or create their own variation. It is useful, but not essential, to start with a HiFi 5 DSP template. Follow these steps to create your configuration in the Xtensa Xplorer IDE.

1. Click the **Xplorer Quickstart Wizard** button in the toolbar from any perspective. Then, select **Create a New Xtensa Configuration** and click **Next**.

Alternatively, from the System Overview pane in the C/C++ perspective, right-click on **Configurations** and select **New Configuration**.

1. Select **Create new configuration with a new core ISA**. Enter the customer\username and password. Click **Test XPG Access**. When the test succeeds, click **Next**.
2. Enter the configuration name and description. Select any LX processor version. Select one of the templates and click **Finish**.
3. On the Configuration Summary pane, click **Edit** from the Configuration row of the Workspace Config column.

You can now customize the processor containing the HiFi 5 DSP as described in the *Xtensa Development Tools Installation Guide*. As you are customizing the processor, remember the following restrictions:

- The HiFi 5 DSP option in the Processors tab must be selected.
- As the HiFi 5 DSP is always coprocessor number 1, the Number of Coprocessors must be at least 2.
- The HiFi 5 DSP, from the RI-2022.9 release (HW LX 7.1.9) and onwards, requires the selection of several options on the Instructions tab, for example MUL16, 32-bit integer divide, MUL32 with Pipelined Plus UH/SH.

- The HiFi 5 DSP option is incompatible with other DSP families.
- As the HiFi 5 DSP has some 128-bit instruction formats, the minimum instruction width must be 16 bytes and a 128-bit instruction fetch is required. The data interfaces to memory must be at least 128 bits.
- The HiFi 5 DSP has three optional functional units to accelerate specific applications. Users can select these options independent of each other. Following is a brief description of the functional units.
  - HiFi 5 Single Precision Vector FP: Single Precision floating-point Unit (SP FPU) which can do up to eight single precision IEEE-754 floating-point MACs per cycle for enhanced audio and voice processing.
  - HiFi 5 Half Precision Vector FP: HP FPU option provides up to sixteen half-precision IEEE-754 floating point MACs per cycle for accelerating floating-point speech networks.
  - HiFi 5 Neural Network Extension: Neural Network Extension enables the hardware to perform up to thirty-two 8x16, 4x16 and 8x8-bit MACs per cycle for supporting speech recognition algorithms.
  - Sigmoid/tanh activation TIE: This is a sub-option provided with Neural Network Extension which enables the hardware to perform 16/8 bit 8-way SIMD sigmoid or tanh activation function

Once a processor has been configured and downloaded, it can be exercised in simulation.

8.2 Basic HiFi 5 DSP Characteristics

Some of the relevant configuration characteristics of the HiFi 5 DSP coprocessor include:

- HiFi 5 DSP instruction set
- Boolean registers
- Sign extend to 32 bits
- TIE arbitrary byte enables
- Density instructions
- Zero overhead loop instructions. Note that this option is not strictly required. However, audio codecs licensed by Cadence are compiled using these instructions and not selecting these instructions can significantly increase the MCPS required by an application.
- 5- or 7-stage pipeline. Note that this choice has several implications. A 5- stage pipeline will result in a smaller configuration, but the maximum speed that it is possible to synthesize and layout will be less than is possible with a 7-stage pipeline. In addition, larger local memories (e.g., 32 KB or larger) may operate better with a 7-stage pipeline configuration that has extra memory access stages. Consider these trade-offs in regards to your application.
- Cache prefetch entries.
- Maximum instruction width equals 16 bytes.

- Data memory interface of at least 16 bytes.
- Little-endian byte ordering (fixed).
- Optional iDMA (integrated DMA) for transferring data between external memory and local data RAM independent of, and in parallel with, HiFi 5 DSP processor execution. Refer to the Xtensa LX7 Microprocessor Data Book for a complete description of the iDMA. iDMA transfers to/from local data RAM can be lower priority than HiFi 5 DSP local data RAM accesses. Still, in use cases with high load/store and DMA bandwidth requirements, a PING-PONG buffering scheme where processor and iDMA buffers are kept in different data RAMs is recommended. Thus, when iDMA is transferring data to/ from PING buffer in one data RAM, HiFi 5 DSP core is processing data from/into PONG buffer in another data RAM and vice-versa.

Note: iDMA programming is supported by the Integrated DMA Library (iDMAlib) software library described in detail in the Xtensa System Software Reference Manual.

8.3 Extending a HiFi 5 DSP with User TIE

HiFi 5 can be extended with user-TIE defining new instructions. These can be assigned to the 24-bit regular instruction format, or can use one of the existing 64-bit or 128-bit FLIX instruction formats. Users can also define additional 64-bit or 128-bit FLIX instruction formats, based on availability of encoding space. To use the existing formats, simply use the TIE `slot_opcode` statement to place the new operation in one or more of the following slots:

```
ae_slot0,     ae_slot1,   ae_slot2,   ae_slot3,
ae2_slot0,   ae2_slot1,  ae2_slot2,
ae3_slot0,   ae3_slot1,
ae4_slot0,   ae4_slot1,  ae4_slot2,  ae4_slot3, ae4_slot4
ae5_slot0,   ae5_slot2,
ae6_slot0,   ae6_slot1,  ae6_slot2,  ae6_slot3,
ae7_slot0,   ae7_slot1,  ae7_slot2,  ae7_slot3,
ae8_slot0,   ae8_slot1,  ae8_slot2,             // available only when 'Neural Network
Extension' is configured
ae9_slot0,   ae9_slot1,  ae9_slot2,  ae9_slot3, // available only when 'SP VFPU' and/or 'HP
VFPU' packages are configured
ae10_slot0, ae10_slot1, ae10_slot2, ae10_slot3  // available only when 'SP VFPU' and/or 'HP
VFPU' packages are configured
```

New operations meant to benefit from parallel execution should go in a FLIX format. Such operations that might be used in parallel with existing HiFi 5 DSP operations should go in one of the slots of the existing instruction formats. If you are creating a set of operations that are meant to be used in code separate from other HiFi 5 DSP code, it is better to put that code in a new format.

Following are some points to consider when creating new instructions to put in the existing formats:

- HiFi 5 DSP requires the following register read and write ports. Adding TIE that will increase the port requirements will have hardware costs.

- AE_DR: 12 read / 8 write
- AE_EP: 3 read / 2 write
- AR: 4 read / 2 write
- AE_VALIGN: 2 read / 1 write

Utilizing HiFi 5 DSP Resources

New instructions may utilize existing HiFi 5 DSP resources. For example, it is possible to create a new instruction that utilizes the AE_DR register file. Simply use the string AE_DR in your `operation` arguments. Similarly, existing HiFi 5 DSP states can be used by using the names listed in *Table 1: DSP Subsystem State Registers* on page 27 or *Table 2: Bitstream and Variable-length Encode/Decode Support Subsystem State Registers* on page 28.

Existing instructions, either core or HiFi 5 DSP, can be placed in additional slots in order to increase parallelism. As with custom TIE instructions, simply use the TIE `slot_opcode` statement to place the existing operation in one of the VLIW slots. It is not currently possible to share existing HiFi 5 DSP functional resources for new instructions. New multiplier instructions, for example, will have to use their own dedicated multipliers.

TIE Language Example

The process of defining user FLIX instruction format, and adding operations to it using two TIE examples, is illustrated here. The first TIE example defines a new FLIX format, consisting of five FLIX slots.

The following TIE template can be used to define a new 64-bit wide FLIX format:

```
// Define a new 64-bit wide FLIX format for user operations in HiFi 5 DSP
length lae64_user 64 {InstBuf[3:0] == 4'he}
format fmt_user_64b lae64_user {user_slot0, user_slot1, user_slot2, user_slot3}
{InstBuf[31:29] == 3'h3}

// Assign NOPs to the user slots
slot_opcodes user_slot0 {NOP}
slot_opcodes user_slot1 {NOP}
slot_opcodes user_slot2 {NOP}
slot_opcodes user_slot3 {NOP}
```

Based on the scheduling requirements in the target application, users may now add one or more operations

- Either from the HiFi 5 DSP ISA or
- User defined TIE operations - to the `slot_opcodes` defined.

Based on the amount of encoding space available, the TIE Compiler will attempt to encode the format and opcodes assigned to its one or more slots.

Name Space Restrictions for User TIE

All TIE constructs in HiFi 5 DSP are prefixed with “ae_” or “AE_”, except for RUR and WUR instructions and iclasses which place the “ae_” root after the “wur” or “rur”. Do not use these prefixes or roots in your TIE file.

Note: On inclusion of the optional floating point unit in HiFi 5 DSP, all the newly added TIE constructs are prefixed with the “fp_” or “vfpu2_” prefixes. Do not use either of these prefixes in your TIE file.

8.4 Optional Configuration Templates for HiFi 5 DSP

Several optional configuration templates are provided for the HiFi 5 DSP. They serve as useful starting points for those wanting to either use the configuration as described by the template, or to create their own variations.

Refer to the Xtensa Xplorer help topic "Using Configuration Templates" to see how to start a configuration based on a template.

HiFi 5 DSP configuration templates are available in XPG Configuration Editor through Xtensa Xplorer. Select and load one of the HiFi DSP templates available in the "Create a Xtensa Configuration" dialog or "Processor selection" page. Refer to configuration options and details for creating your own variations.

8.5 Synthesis and Place-and-Route

When the HiFi 5 DSP is included in an Xtensa processor configuration, the synthesis and place-and-route scripts that are included with the software build can be used with the usual methodology, which is outlined in the *Xtensa LX Hardware User’s Guide.*