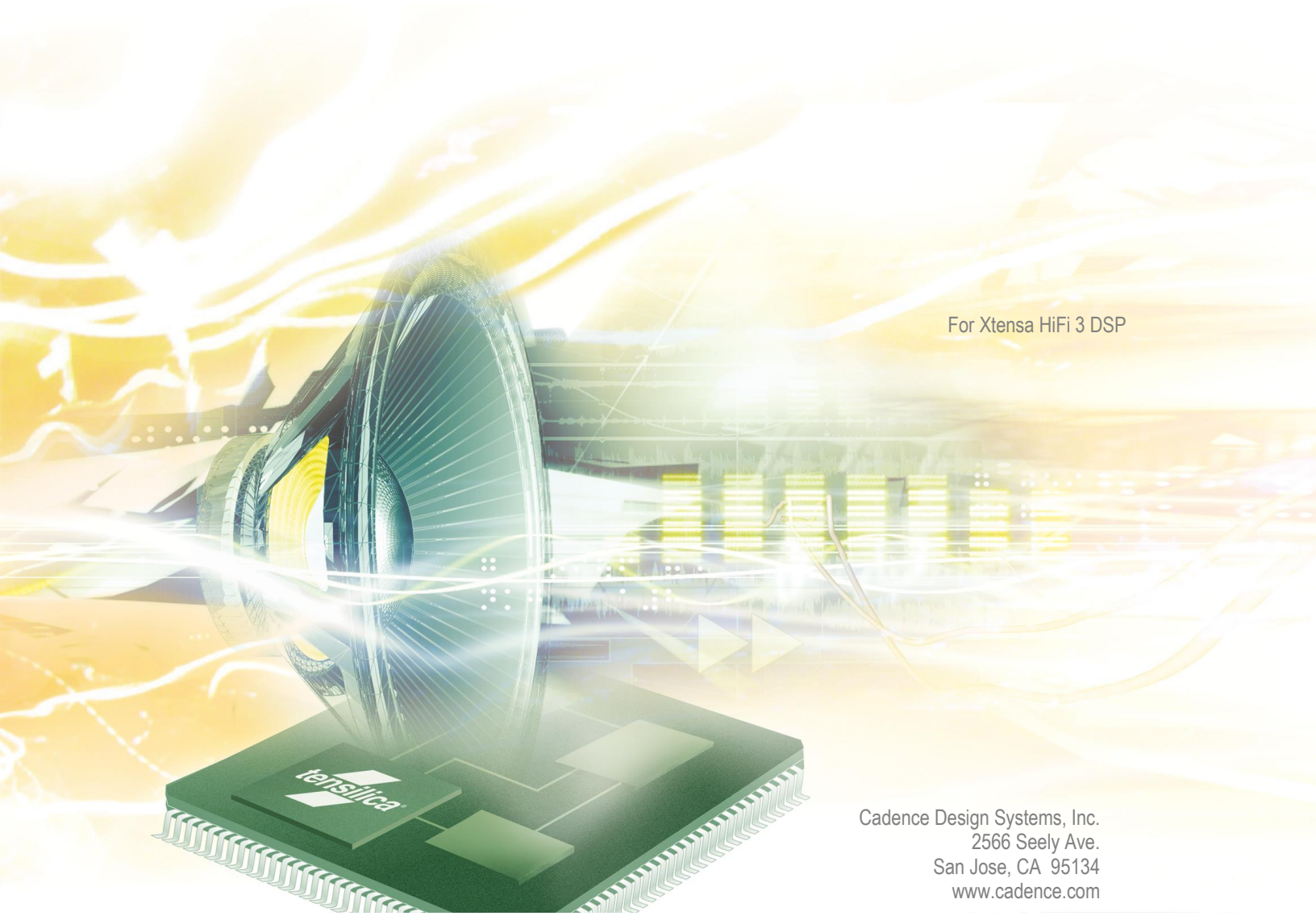
cadence®

HiFi 3 DSP

User's Guide

For Xtensa HiFi 3 DSP



Cadence Design Systems, Inc.
2566 Seely Ave.
San Jose, CA 95134
www.cadence.com

PD-17-8530-10-06
RG-2017.7
Issue Date: 08/2017

Contents

Figures

Tables

Document Change History

Version	Changes
RG.7	The following changes (denoted with change bars) were made to this document for the Cadence Tensilica RG-2017.7 release of the HiFi3 DSP: ■ Added information in section 2.4.2 that CBEGIN need not be less than CEND ■ Updated information in Chapter 6 HiFi 3 NatureDSP Signal Library
RG.5	The following changes (denoted with change bars) were made to this document for the Cadence Tensilica RG-2017.5 release of the HiFi3 DSP: ■ Enhanced the description of floating point operations ■ Corrected the intrinsic for AE_L16_IP ■ Corrected round instructions that were documented as returning integer types instead of fractional types ■ Corrected AE_PKSR24 as returning a fractional type instead of an integer type
RG.4	The following changes were made to this document for the Cadence Tensilica RG-2016.4 release of the HiFi3 DSP: ■ Minor corrections to the (NaN) description ■ Enhanced conversions between variables of different types in section 3.1. ■ Clarified the usage restrictions for AE_DIV64 ■ Clarified for converting HiFi 2 legacy types to and from HiFi 3 vector types ■ Clarified rules on conversions from float to and from ae_int32x2 ■ Corrected an inaccuracy in output type name for AE_MOVPA24x2
RG.3	■ Amended several instructions in Chapter 2, including: ▪ Included required alignments ▪ Amended the note for AE_MOVDA32X2 in Section2.11. ▪ Updated Section 2.16.1. ■ Added information about conversions being applied to intrinsic invocations in Section 3.1. ■ Updated Table 3-3 HiFi 3 C/C++ Operators.

1.2	■ Minor clarifications re AE_SA16x4, as well as in Table 3-2 HiFi 3 Display Types.
1.1	■ Added information regarding HiFi Mini in Section 3.8. ■ Corrected HiFi 3 Coprocessor number to 1 in Section 7.1
1.0	■ Initial customer release.

1. Introduction

Cadence's HiFi 3 DSP is a highly optimized audio processor geared for efficient execution of audio and voice codecs, and pre- and post-processing modules. It goes beyond the two MAC, two issue, HiFi 2/EP architecture with four multipliers, three VLIW slots, good support for 32x16-bit and 32x32-bit multiplication, a true 64-bit data path and native support for ITU-T/ETSI intrinsics. There is an optional floating point unit available, providing for a 2-way SIMD, single-precision IEEE floating point MAC or ALU operation every cycle. The extra resources provide for significant performance improvements compared to HiFi 2/EP, particularly on pre/post-processing algorithms as well as voice codecs. The support for 32-bit audio as well as ITU-T/ETSI intrinsics, including automatic vectorization, provides much better performance on out-of-the-box C programs and voice algorithms.

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

The HiFi 3 DSP is a configuration option that can be included with the Xtensa LX 4 (and later versions) processor. All HiFi 3 DSP operations can be used as intrinsics in standard C/C++ applications. In addition, when compiling with automatic vectorization or with the `-mcoproc` option, the compiler will automatically use HiFi 3 operations when compiling standard C code.

Cadence's HiFi 3 DSP consists of two main components: a DSP subsystem and a subsystem to assist with bitstream access, and variable-length (Huffman) encoding and decoding. These are covered in detail in Chapter 2.

1.1 Purpose of this User Guide

This guide provides an overview of the HiFi 3 architecture and its instruction set. It will help programmers using HiFi 3 by identifying some of the techniques that are commonly used to optimize algorithms. It provides guidelines to achieve improved performance by using HiFi 3's instructions, intrinsics, protos, and primitives. This guide also serves as a C/C++ usage reference for the appropriate way to use HiFi 3 features in a C/C++ software development. This guide will also assist Xtensa HiFi 3 users who wish to add additional instructions to the HiFi 3 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*.

1.1.1 Conventions

Throughout this guide, the symbol *<xtensa_root>* refers to the installation directory of a user's Xtensa configuration. For example, *<xtensa_root>* might refer to the directory `\usr\xtensa\XtDevTools\install\builds\RF-2015.2-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 3 configuration, follow the same procedures described in the *Xtensa Development Tools Installation Guide*. The HiFi 3 include 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_hifi3.h
```

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

For floating point usage with the optional floating point unit, include the following file:

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

1.3 HiFi 3 Architecture Overview

The HiFi 3 DSP, a SIMD (single-instruction/multiple-data) processor, can work in parallel on two 24/32-bit data items or four 16-bit data items. For example, it allows for one operation to perform two 32-bit additions in parallel, with each addition occupying half of a 64-bit AE_DR register. The HiFi 3 multipliers support multiplication of four 24-bit, or four 32x16-bit, or four 16x16-bit operands per cycle. They support two 32x32-bit multiplies per cycle. There are operations for single, dual, and quad multiplication. Single or dual multiply operations can be dual issued using VLIW instructions. Quad multiply operations cannot be issued together with other multiplies. The HiFi 3 DSP can only be configured to use a little-endian byte ordering.

With the optional floating point unit, HiFi 3 supports two IEEE-754 floating point MACs per cycle.

In general, 16-bit support is geared towards efficient support of the ITU-T/ETSI intrinsic model, while 32x16-bit and 24-bit support is provided for both integer and fixed-point computation.

HiFi 3 is a VLIW architecture, supporting the execution of three operations in parallel. DSP loads and stores, bitstream and Huffman operations, and core operations are available in slot 0 of a VLIW instruction. DSP MAC and ALU operations are typically available in slot 1

and slot 2. The optional floating point operations are generally available in slot 1 of a two-slot format.

HiFi 3 supports either caches or local memories with the full flexibility provided by Xtensa configurations. You can have either or both, and can make different choices for instruction and data. Audio packages supplied by Cadence do not use DMA. Hence, most customers either use caches or make local memories sufficiently large to cover desired applications.

The following diagram illustrates the main custom state, register file, and execution units added to an Xtensa LX processor by the HiFi 3 DSP.

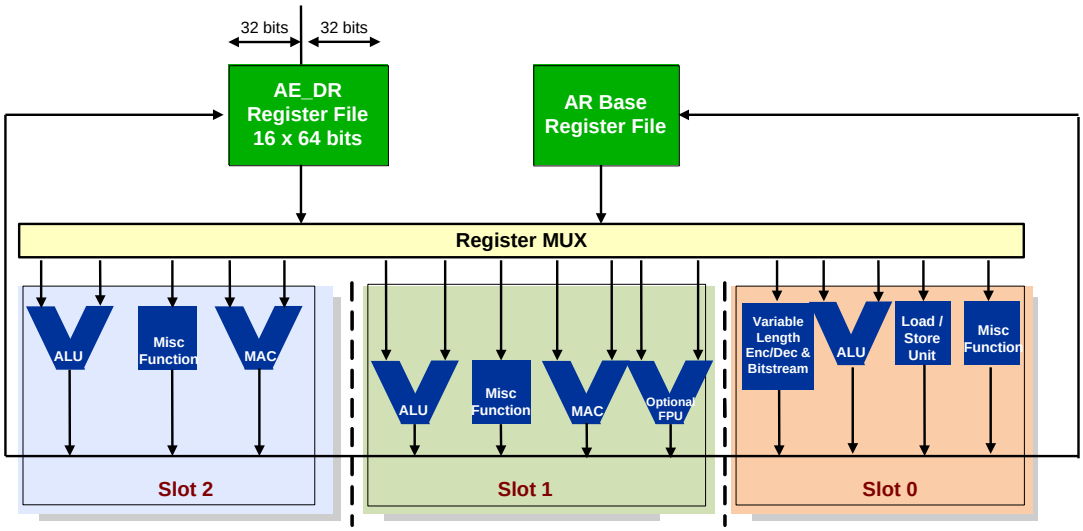


Figure 1-1 HiFi 3 DSP Components

The main hardware resources in the DSP subsystem are two 2-multiplier multiply/accumulate units, an option for a 2-way SIMD single-precision IEEE floating point unit, a 16-entry register file AE_DR to hold 64-bit, pairs of 32-bit or quads of 16-bit data items, an arithmetic/logic unit, and a shift unit to operate on the AE_DR values. The multiplier units support two 32x32-bit or 24x32-bit MACs or four 24x24, 16x32, or 16x16-bit MACs per cycle. The multiplies are supported through single-instruction quad multiplies or through parallel-issued dual or single multiply instructions.

The load/store unit is capable of loading or storing up to two 24-bit or 32-bit SIMD elements, four 16-bit SIMD elements, or single elements up to 64 bits in size. 24-bit data can either be contained inside 32-bit envelopes, or can be packed together into 24 bits of memory. Eight packed elements can be loaded or stored in three instructions. The load/store unit supports unaligned accesses, whereby a stream is first primed and afterwards 64 unaligned bits can be loaded or stored in every cycle.

The DSP subsystem can be issued in several VLIW formats—two 3-slot VLIW formats (ae_format and ae_format1), one 2-slot format (ae_format2), and one 2-slot mini format (ae_mini0). The operations for the 3-slot VLIW formats can be issued in one of the three slots. In each execution cycle, zero or one operation from each slot can be executed independently according to the static bundling expressed in the machine code. For example, load operations can execute concurrently with multiply/accumulate operations because loads

are in ae_slot0 and multiply/accumulate operations are in ae_slot1 or ae_slot2_0. The two slot format contains multiply instructions that produce more than one register result or use extra operands as well as some legacy operations. For better code size, many operations (not including integer or fixed point multiplies) are also available in single issue 16- and 24-bit formats. Most floating point operations are available in the 24-bit formats.

1.4 Prefetching

HiFi 3 includes a prefetch option geared for systems with long memory latency. When the HiFi 3 processor 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(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 Architectural Additions section of the Prefetch Unit Option chapter in the *Xtensa 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 Architectural Additions section of the Prefetch Unit Option chapter in the *Xtensa Microprocessor Data Book* for details.
  - `XTHAL_DCACHE_PREFETCH_L1_OFF` (prefetch data to prefetch buffers only)
  - `XTHAL_DCACHE_PREFETCH_L1` (on configurations that support it, prefetch directly to L1 data cache)

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

1.4.1 Software Prefetching

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

```
__builtin_prefetch(addr);
```

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

For configurations that do not prefetch into the cache, but instead use a small, 8- to 16-entry buffer outside of the cache, you must be careful not to prefetch too far ahead. Otherwise, the data will be overwritten before it is needed by the processor.

Consider a simple example that does an energy calculation. You might choose to place a few explicit prefetch instructions before the loop to seed the hardware prefetcher. Otherwise, depending on the mode, the hardware prefetch might delay prefetching until after the second miss.

```
__builtin_prefetch(&ap[0]);
__builtin_prefetch(&ap[XCHAL_DCACHE_LINESIZE]);
__builtin_prefetch(&ap[2*XCHAL_DCACHE_LINESIZE]);
```

```
for (i=0; i<n; i++) {
   sum += ap[i]*ap[i];
}
```

You might also want to put prefetch instructions directly inside the loop. Doing so allows you to prefetch more aggressively than the hardware prefetcher and allows you to prefetch patterns other than the stride-1 references that are detected by the hardware prefetcher. On the other hand, placing prefetch instructions inside the loop incurs instruction overhead whether or not the loop actually suffers from cache misses.

In general, given the effectiveness of the hardware prefetcher, software prefetches should be used judiciously. Carefully compare performance between using and not using software prefetching on a loop-by-loop basis.

1.5 HiFi 3 Instruction Set Overview

The HiFi 3 DSP is built on the baseline Xtensa RISC architecture, which implements a rich set of generic instructions optimized for efficient embedded processing. The power of HiFi 3 comes from a comprehensive DSP and audio instruction set. A wide variety of load/store operations support multiple addressing modes, with support for 16/24/32-bit scalar and vector data types together with 56/64-bit scalar. Vector data management is supported with select operations and shifting.

Multiply operations include 32x32-bit, 32x24-bit, 24x24-bit, 32x16-bit and 16x16-bit. Multiply operations come in fixed-point and integer variants. They come in high precision and low precision variants. High-precision multiplies utilize a 64-bit accumulator. Since an accumulator can only hold one result, HiFi 3 supports dual multiplies where the results of two multiplies are added or subtracted together before being added into the accumulator. For example, a single operation might compute the following operation where H and L refer to the high bits or low bits respectively of an operand.

```
acc = acc – d0.L*d1.L + d0.H*d1.H.
```

Low-precision multiplies accumulate in 32-bits. Since each register can hold two 32-bit accumulators, these instructions can perform two independent SIMD multiplies.

A set of bitstream and variable length instructions allow for efficient access of serial bitstreams, including Huffman encode and decode.

The optional floating point unit supports 2-way SIMD units of IEEE-754 single precision floating point operations. Refer to the *Xtensa Instruction Set Architecture (ISA) Reference Manua*l for more details about the core single precision floating point support, on which the 2-way SIMD units are based.

2. HiFi 3 Features

The HiFi 3 DSP contains a 16-entry, 64-bit register file, AE_DR. Each register can hold one or two, 24 or 32-bit operands, one or four 16-bit operands or one 56- or 64-bit operand as shown in Figure 2-1. 24-bit and 56-bit operands are sign extended to fill their 32- or 64-bit container. 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.

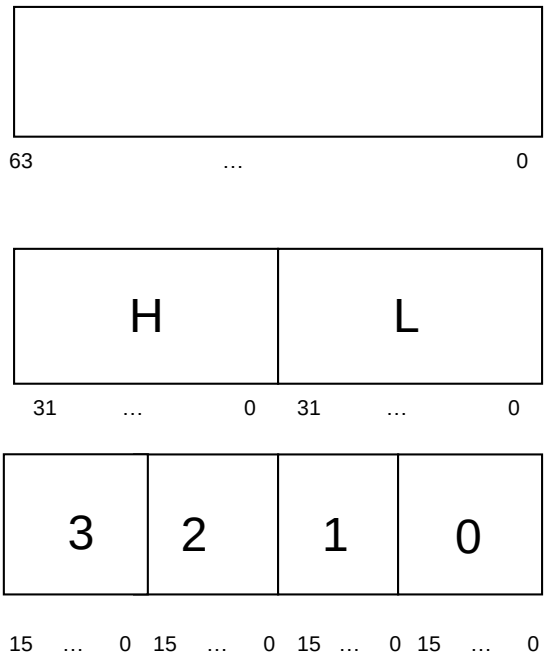


Figure 2-1 AE_DR Register

When a register is stored to memory, the high half of the register is always stored in the lower memory address. This enables the same source code to work on all configurations, including big-endian HiFi2 cores. 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.

For legacy HiFi 2/EP 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 3 supports a 4-entry, 64-bit alignment register, AE_VALIGN. Using this register allows the hardware to load or store a SIMD stream that is not 64-bit aligned at a rate of 64-bits per cycle. It also allows 24-bit data to be packed densely into 24-bit containers. These mechanisms are described in more detail In Section 2.4.1.

Table 2-1 lists the the TIE state registers in the HiFi 3 DSP.

Table 2-1 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	7	Contains the shift amount for various DSP shift operations.

The state registers listed in Table 2-2 pertain to the bitstream and variable-length encode/decode support subsystem of the HiFi 3 DSP. Programmers generally will not need to concern themselves with the details of how each of these state registers is used by the instructions. However, the state registers (understandable for those familiar with the variable-length encode/decode instructions) are documented here for completeness.

Table 2-2 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

State Register	Bit Size	Description
		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 following state registers 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 2-3 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_CWRAP	1	Indicates whether any circular buffer operation has wrapped around since the time when AE_CWRAP was last reset to zero.

The following state registers pertain to the optional floating point support.

Table 2-4 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.
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:

```
user_register AE_OVF_SAR 240 {  AE_SAR[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]
```

With the floating point option, use the following user register to control and detect rounding and exception behavior. Refer to 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 2-5.

Table 2-5 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 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

Instruction	Intrinsic	Description
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_GETBEGIN	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_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, and UnderflowFlag
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 state register AE_SAR
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
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.

Instruction	Intrinsic	Description
WUR.AE_CEND0	WUR_AE_CEND0, WAE_CEND0, AE_SETCEND0	Write state register AE_CEND0 AE_SETCEND0 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, and UnderflowFlag
AE_MOVFCRFSRV	AE_MOVFCRFSRV	Set user register FCR_FSR from a vector register which can be loaded from memory.

In the operation descriptions in Sections 2.4 through 2.17, 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 2-6.

Table 2-6 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	AE_DR	aed0 – aed15	aed2
b	BR	b0 – b15	b3
bhl	BR2	b0 – b14 (even)	b0
b3210	BR4	b0-b16 (multiple of 4)	b0
u	AE_VALIGN	u0-u3	u0

Table 2-7 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
i64pos	0..56	8
i	Operation-dependent	1

Each operation description is annotated with the name(s) of the slot(s) where that operation can be issued. 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 3 DSP is included in Chapter 3.

All HiFi 2 C types and intrinsics are available in HiFi 3 to ensure C/C++ source code portability. Notes on HiFi 2 code portability and matching intrinsics are included in the operation description for the relevant operations as well as in Chapter 3.

2.1 Instruction Naming Conventions

All HiFi 3 DSP operation mnemonics 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.

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

Table 2-8 Operation Mnemonics

Mnemonic	Meaning
`ASYM`	Denotes asymmetric rounding (*e.g.*, AE_ROUND32X2F64SASYM)
`F`	Denotes fractional arithmetic (*e.g.*, AE_MULZAAFD24.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_MULZAAFD24.HH.LL).
`0,1,2,3`	Combinations of `0`, `1`, `2` and `3` are used to refer to quarters of registers (*e.g.* AE_MULF32X16.L0)
`I`	Denotes use of an immediate operand (e.g., AE_SRAIP32)
`S`	Denotes saturating arithmetic (*e.g.*, AE_MULF32S.LL) or the use of the AE_SAR state register as a shift amount (*e.g.*, AE_SRASP32), depending on the context
`SYM`	Denotes symmetric rounding (*e.g.*, AE_ROUND32X2F64SSYM)
`T`	Denotes the value True in a conditional move (*e.g.*, AE_MOVT64)
`U`	Denotes unsigned arithmetic (*e.g.*, AE_MULS32U.LL)
`X`	Denotes use of an index register in an address computation (*e.g.*, AE_L64.XP)
`X2`	Denotes a 2-way SIMD operation in contexts (*e.g.*, AE_L32X2.I) where scalar operations are also available
`X4`	Denotes a four-way SIMD operation (*e.g.*, AE_L16X4.XC)

2.2 Fixed Point Values and Fixed Point Arithmetic

The HiFi 3 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.

2.2.1 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 2^n.

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

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)

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 3 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.

- **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 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 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 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.

2.2.2 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. For example, multiplying a 1.23 typed variable by a 1.23 typed variable generates a 2.46 typed variable. Since HiFi 3 supports the 17.47 data type, the fixed point multiply instructions shift the 2.46 result to the left by 1 bit and then sign extends it by 15 bits. In general, high-precision fixed-point multiplications shift their results to the left by 1 bit.

HiFi 3 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.

2.2.3 Other Fixed Point Representations

Programmers are free to use fixed-point representations other than the ones listed in Section 2.2.2. Most HiFi 3 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 left by one bit. So, if the input data is actually a 2.22 data type rather than a 1.23 data type, the 24-bit fixed point multiply instructions will correctly produce an 11.45 typed variable. The programmer is responsible for knowing what type of data is in what variables, and if manual conversions are needed, you can always use shift instructions.

2.3 VLIW Slots and Formats

HiFi 3 can issue up to three operations in a single 64-bit instruction bundle using Xtensa LX FLIX (VLIW) technology. HiFi 3 supports four different formats *ae_format*, *ae_format1*, *ae_format2* and *ae_mini0*. Every instruction belongs to one format, but different formats may pack different numbers of operations in a single instruction.

Formats *ae_format* and *ae_format1* both support three parallel operations. The two formats are logically equivalent, allowing the exact same operations in the first two slots and disjoint operations in the third. The reason for splitting the format in two is for encoding space. The first format is 60 bits while the second is 59 bits, allowing a total size of 60.5 bits, something that is not possible to attain with a single format. The rest of this guide treats the format as a single format containing the slots *ae_slot0*, *ae_slot1*, and *ae_slot2*.

Format *ae_format2* is a 2-slot format with slots *ae2_slot0* and *ae2_slot1*. Using this format allows for individual operations with more operands or larger immediates than can be used in the 3768 slot format.

Format *ae_mini0*, with slots *ae_minislot0* and *ae_minislot2*, is a specialized format that allows operations that read an AR register operand to execute in parallel with operations that have at most one AR read and one AR write operand. In particular, this format allows the parallel execution of some simple core operations such as MOVI and ADDI together with immediate loads and stores.

For the 3-slot format, the first slot contains all of the HiFi 3 load/store instructions and some miscellaneous operations. The second slot contains all of the regular multiply and DSP ALU operations. The third slot contains all the shifts and DSP ALU operations as well as a subset of the multiply operations.

A subset of the operations as well as all the bitstream operations are available in a single issue, 24-bit format called `Inst`. The compiler will automatically use the 24-bit format when it is not possible (or beneficial) to bundle a relevant operation together with an operation that can go in another slot. A subset of the core Xtensa operations is also available in the first VLIW slot, allowing some parallelism between DSP operations and core Xtensa operations.

For *ae_format2*, the first slot contains all of the load and store operations as well as many operations from the third slot of the 3-slot format. This slot also contains variants of the core

branch instructions with large immediates that do not fit into other formats. The second slot contains all of the multipliers including specialized multipliers that do not fit into the 3-slot format.

For the optional floating point unit, most floating point operations are available in the second slot of ae_format2, allowing the machine to issue, for example, one SIMD floating-point load in parallel with one, 2-way SIMD, multiply-accumulation operation.

Understanding the slotting is important when optimizing code for HiFi 3. 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) load or 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.

All HiFi 2 instructions available in the `Inst` slot share opcode space (but do not overlap) with the MAC16 Option.

The available slotting for the different operations are listed next to the operation descriptions in the remainder of this chapter.

2.4 Load and Store Operations

HiFi 3 supports loading and storing scalars or vectors of 16, 24, 32, and 64 bits. Each scalar load/store accesses 16, 24, 32, or 64 bits. Each vector accesses 64 bits or 48 bits for packed 24-bit data. For vector loads and stores, the high address in memory is always stored in the least significant bits in the register. This enables the same source code to work on both little and big endian systems. Reverse vector loads and stores reverse the elements in a register so that the 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.

Special support is provided for retaining full throughput when vectors of data are not aligned to 64-bits. HiFi 3 also supports a single circular buffer that can be used with either aligned or unaligned data.

2.4.1 Aligning Loads and Stores

HiFi 3 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 16-, 24-, and 32-bit data. The aligning vector load and store instructions use the HiFi 3 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;
  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;
  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 3 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` intrinsics 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;
  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 + 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.

2.4.2 Circular Buffer

HiFi 3 has support for a single circular buffer, which can be accessed in either the forward or the backward direction.

The circular buffer boundaries are specified through two 32-bit states:

Table 2-9 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.

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

```
void * AE_GETCBEGIN0 (void);
void * AE_GETCEND0 (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);
```

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 and must be less than or equal to the buffer byte size. 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` and `AE_CEND0` must be aligned to 64 bits. For aligned accesses, `AE_CBEGIN0` and `AE_CEND0` must be aligned to the size of the data being loaded or stored. Unaligned accesses use the alignment mechanism described in Section 2.4.1. Priming loads use the `PC` 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.

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 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));
  …
}
```

2.4.3 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 the following table.

Table 2-10 Load/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 24-bit	This operation accesses two of the size “24” above, occupying 48 bits in memory.
32X2	Vector of 32-bit	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 16 bit	This operation accesses four of the size “16” above. Some instructions need the quartet to be 64-bit aligned, while others do not.

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, or XC 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.

Table 2-11 Load/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.
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 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.
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.
RIP Reverse Post Update	Reg	Reg	The effective address is a base AR register. The base AR register is updated with the base AR register minus 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 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

Suffix & Definition	Effective Address	Base Reg Update	Description
			vector elements in the result register are also swapped.
PP Prime	See Instruc-tion	See Instruc-tion	This addressing mode is used for priming instructions which set up the beginning of an unaligned load sequence
PC Circular Prime	See Instruc-tion	See Instruc-tion	This addressing mode is used for priming instructions which set up the beginning of an unaligned load sequence in a circular buffer
FP Flush	See Instruc-tion	See Instruc-tion	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.

2.4.4 Load Operations

The following table 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 2-12 Load Overview

Instruction	Size <sz>	Suffix <adr>	Purpose
AE_L<sz>.<adr>	64, 32, 32F24, 16	I, X, IP, XP, XC	Aligned loads of scalars
AE_L<sz>.<adr>	32X2, 32X2F24, 16X4	I, X, IP, RIP, XP, XC, RIC	Aligned loads of vectors
AE_LA<sz>.<adr>	64	PP	Prime for Unaligned loads using IP
AE_LA<sz>POS.<adr>	32X2, 16X4, 24, 24X2	PC	Prime for Unaligned loads using IC with positive stride
AE_LA<sz>NEG.<adr>	32X2, 16X4, 24, 24X2	PC	Prime for Unaligned loads using IC with negative stride
AE_LA<sz>.<adr>	32X2, 32X2F24, 16X4, 24, 24X2	IP, IC	Unaligned Loads for accessing vectors of aligned scalars with positive update
AE_LA<sz>.<adr>	32X2, 32X2F24, 16X4, 24, 24X2	RIP, RIC	Unaligned Loads for accessing vectors of aligned scalars with negative update
AE_LALIGN64.I			Load of alignment register
AE_L<sz>M.<adr>	16X2, 32, 16	I, X, XC, IU, XU	Legacy Loads

AE_L64.I, AE_L64.IP, AE_L64.X Operations:

AE_L64.I	d, a, i64	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_L64.IP	d, a, i64	[ae_slot0, ae2_slot0, Inst]
AE_L64.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 8 bytes

Load a 64-bit value from memory into the AE_DR register `d`. See Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_LQ56_I (_X, _C, _IU, _XU) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_L64.I (.X, .XC, .I, .I), respectively.

C syntax:

```
ae_int64 AE_L64_I (const ae_int64 * a, immediate i64);
ae_int64 AE_L64_X (const ae_int64 * a, int ax);
void AE_L64_IP (ae_int64 d /*out*/,
      const ae_int64 *a /*inout*/, immediate i64);
void AE_L64_XP (ae_int64 d /*out*/,
      const ae_int64 *a /*inout*/, int ax);
void AE_L64_XC (ae_int64 d /*out*/,
      const ae_int64 *a /*inout */, int ax);
ae_q56s AE_LQ56_I (const ae_q56s * a, immediate i64);
void AE_LQ56_IU (ae_q56s d /*out*/,
      const ae_q56s * a /*inout*/, immediate i64);
ae_q56s AE_LQ56_X (const ae_q56s * a, int ax);
void AE_LQ56_XU (ae_q56s d /*out*/,
      const ae_q56s * a /*inout*/, int ax);
void AE_LQ56_C (ae_q56s d /*out*/,
      const ae_q56s * a /*inout*/, int ax);
```

AE_L32X2.I, AE_L32X2.IP, AE_L32X2.RIP, AE_L32X2.X Operations:

AE_L32X2.I	d, a, i64	[ae_slot0, ae2_slot0, Inst , ae_minslot0]
AE_L32X2.IP	d, a, i64pos	[ae_slot0, ae2_slot0, Inst]
AE_L32X2.RIP (.RIC)	d, a	[ae_slot0, ae2_slot0, Inst]
AE_L32X2.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 8 bytes

Load a pair of 32-bit values from memory into the AE_DR register `d`. Refer to Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_LP24X2_I (_X, _C, _IU, _XU) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_LP32X2.I (.X, .XC, .I, .I), respectively.

C syntax:

```
ae_int32x2 AE_L32X2_I (const ae_int32x2 * a, immediate i64);
ae_int32x2 AE_L32X2_X (const ae_int32x2 * a, int ax);
void AE_L32X2_IP (ae_int32x2 d /*out*/,
      const ae_int32x2 *a /*inout*/, immediate i64pos);
void AE_L32X2_XP (ae_int32x2 d /*out*/,
      const ae_int32x2 *a /*inout*/, int ax);
void AE_L32X2_XC (ae_int32x2 d /*out*/,
      const ae_int32x2 *a /*inout*/, int ax);
void AE_L32X2.RIP (ae_int32x2 d /*out*/,
      const ae_int32x2 *a /*inout*/);
void AE_L32X2.RIC (ae_int32x2 d /*out*/,
      const ae_int32x2 *a /*inout*/);
ae_p24x2s AE_LP24X2_I (const ae_p24x2s * a, immediate i64);
void AE_LP24X2_IU (ae_p24x2s d /*out*/,
      const ae_p24x2s * a /*inout*/, immediate i64);
ae_p24x2s AE_LP24X2_X (const ae_p24x2s * a, int ax);
void AE_LP24X2_XU (ae_p24x2s d /*out*/,
      const ae_p24x2s * a /*inout*/, int ax);
void AE_LP24X2_C (ae_p24x2s d /*out*/,
      const ae_p24x2s * a /*inout*/, int ax);
```

AE_L16X4.I, AE_L16X4.IP, AE_L16X4.RIP, AE_L16X4.X Operations:

AE_L16X4.I	d, a, i64	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_L16X4.IP	d, a, i64pos	[ae_slot0, ae2_slot0, Inst]
AE_L16X4.RIP (.RIC)	d, a	[ae_slot0, ae2_slot0, Inst]
AE_L16X4.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 8 bytes

Load four 16-bit values from memory into the AE_DR register `d`. See Table 2-3 for the meanings of the address mode suffixes.

C syntax:

```
ae_int16x4 AE_L16X4_I (const ae_int16x4 * a, immediate i64);
ae_int16x4 AE_L16X4_X (const ae_int16x4 * a, int ax);
void AE_L16X4_IP (ae_int16x4 d /*out*/,
                  const ae_int16x4 *a /*inout*/, immediate i64pos);
void AE_L16X4_XP (ae_int16x4 d /*out*/,
                  const ae_int16x4 *a /*inout*/, int ax);
void AE_L16X4_XC (ae_int16x4 d /*out*/,
                 const ae_int16x4 *a /*inout*/, int ax);
void AE_L16X4_RIP (ae_int16x4 d /*out*/,
                 const ae_int16x4 *a /*inout*/);
void AE_L16X4_RIC (ae_int16x4 d /*out*/,
                 const ae_int16x4 *a /*inout*/);
```

AE_L32X2F24.I, AE_L32X2F24.IP, AE_L32X2F24.RIP, AE_L32X2F24.X Operations:

AE_L32X2F24.I	d, a, i64	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_L32X2F24.IP	d, a, i64pos	[ae_slot0, ae2_slot0, Inst]
AE_L32X2F24.RIP (.RIC)	d, a	[ae_slot0, ae2_slot0, Inst]
AE_L32X2F24.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 8 bytes

Load a pair of 24-bit values, each from the most significant 24 bits of a 32-bit half of the 64 bits in memory, sign-extends them to 32 bits and stores the values into the two 32-bit elements of AE_DR register `d`. Refer to Table 2-3 for the meanings of the address mode suffixes. The intent here is that the values in memory represent 32-bit (1.31) fractions that get truncated and placed in the two elements of the AE_DR register as 9.23-bit fractions.

Note: C intrinsics AE_LP24X2F_I (_X, _C, _IU, _XU) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_LP32X2F24.I (.X, .XC, .I, .I), respectively.

C syntax:

```
ae_f24x2 AE_L32X2F24_I (const ae_f24x2 * a, immediate i64);
ae_f24x2 AE_L32X2F24_X (const ae_f24x2 *a, int ax);
void AE_L32X2F24_IP (ae_f24x2 d /*out*/,
             const ae_f24x2 * a /*inout*/, immediate
i64pos);
void AE_L32X2F24_XP (ae_f24x2 d /*out*/,
             const ae_f24x2 * a /*inout*/, int ax);
void AE_L32X2F24_XC (ae_f24x2 d /*out*/,
             const ae_f24x2 * a /*inout*/, int ax);
void AE_L32X2F24_RIP (ae_f24x2 d /*out*/,
             const ae_f24x2 *a /*inout*/);
void AE_L32X2F24_RIC (ae_f24x2 d /*out*/,
             const ae_f24x2 *a /*inout*/);
ae_p24x2s AE_LP24X2F_I (const ae_p24x2f * a, immediate i64);
void AE_LP24X2F_IU (ae_p24x2s d /*out*/,
             const ae_p24x2f * a /*inout*/, immediate i64);
ae_p24x2s AE_LP24X2F_X (const ae_p24x2f * a, int ax);
void AE_LP24X2F_XU (ae_p24x2s d /*out*/,
             const ae_p24x2f * a /*inout*/, int ax);
void AE_LP24X2F_C (ae_p24x2s d /*out*/,
             const ae_p24x2f * a /*inout*/, unsigned ax);
```

AE_L32.I, AE_L32.IP, AE_L32.X Operations:

AE_L32.I	d, a, i32	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_L32.IP	d, a, i32	[ae_slot0, ae2_slot0, Inst]
AE_L32.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 4 bytes

Load a 32-bit value from memory and replicate the value into the two elements of the AE_DR register `d`. See Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_LP24_I (_X, _C, _IU, _XU) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_L32.I (.X, .XC, .I, .I), respectively.

C syntax:

```
ae_int32x2 AE_L32_I (const ae_int32 * a, immediate i32);
ae_int32x2 AE_L32_X (const ae_int32 * a, int ax);
void AE_L32_IP(ae_int32x2 d /*out*/,
               const ae_int32 * a /*inout*/, immediate off);
void AE_L32_XP(ae_int32x2 d /*out*/,
               const ae_int32 * a /*inout*/, int ax);
void AE_L32_XC(ae_int32x2 d /*out*/,
               const ae_int32 * a /*inout*/, int ax);
ae_p24x2s AE_LP24_I (const ae_p24s * a, immediate i32);
void AE_LP24_IU (ae_p24x2s d /*out*/,
               const ae_p24s * a /*inout*/, immediate i32);
ae_p24x2s AE_LP24_X (const ae_p24s * a, int ax);
void AE_LP24_XU (ae_p24x2s d /*out*/,
               const ae_p24s * a /*inout*/, int ax);
void AE_LP24_C (ae_p24x2s d /*out*/,
               const ae_p24s * a /*inout*/, int ax);
```

AE_L32F24.I, AE_L32F24.IP, AE_L32F24.X Operations:

AE_L32F24.I	d, a, i32	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_L32F24.IP	d, a, i32	[ae_slot0, ae2_slot0, Inst]
AE_L32F24.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 4 bytes

Load a 24-bit value from the most significant 24 bits of the 32-bit word from memory, sign-extend to 32 bits and replicate the value into the two 32-bit elements of the AE_DR register `d`. See Table 2-3 for the meanings of the address mode suffixes. The intent here is that the value in memory represents a 32-bit (1.31) fraction that gets truncated and replicated into the two elements of d as 9.23-bit fractions.

Note: C intrinsics AE_LP24X2F_I (_X, _C, _IU, _XU) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_L32F24.I (.X, .XC, .I, .I), respectively.

C syntax:

```
ae_f24x2 AE_L32F24_I (const ae_f24 * a, immediate i32);
ae_p24s AE_L32F24_X (const ae_f24 * a, int ax);
void AE_L32F24_IP (ae_f24x2 d /*out*/,
                   const ae_f24 * a /*inout*/, immediate i32);
void AE_L32F24_XP (ae_f24x2 d /*out*/,
                   const ae_f24 * a /*inout*/, int ax);
void AE_L32F24_XC (ae_f24x2 d /*out*/,
                  const ae_f24 * a /*inout*/, int ax);
ae_p24x2s AE_LP24F_I (const ae_p24f * a, immediate i32);
void AE_LP24F_IU (ae_p24x2s d /*out*/,
                  const ae_p24f * a /*inout*/, immediate i32);
ae_p24x2s AE_LP24F_X (const ae_p24f * a, int ax);
void AE_LP24F_XU (ae_p24x2s d /*out*/,
                  const ae_p24f * a /*inout*/, int ax);
void AE_LP24F_C (ae_p24x2s d /*out*/,
                 const ae_p24f * a /*inout*/, int ax);
void AE_LP24X2_C(ae_p24x2s d /*out*/,
                 const ae_p24x2s * a /*inout*/, int ax);
void AE_LP24X2F_C(ae_p24x2s d /*out*/,
                 const ae_p24x2f * a /*inout*/, int ax);
```

AE_L16.I, AE_L16.IP, AE_L16.X Operations:

AE_L16.I	d, a, i16	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_L16.IP	d, a, i16	[ae_slot0, ae2_slot0, Inst]
AE_L16.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 2 bytes

Load a 16-bit value from memory and replicate the value into the four elements of AE_DR register `d`. Refer to Table 2-3 for the meanings of the address mode suffixes.

C syntax:

```
ae_int16x4 AE_L16_I (const ae_int16 * a, immediate i16);
ae_int16x4 AE_L16_X (const ae_int16 * a, int ax);
void AE_L16_IP (ae_int16x4 d /*out*/,
                const ae_int16 * a /*inout*/, immediate i16);
void AE_L16_XP (ae_int16x4 d /*out*/,
                const ae_int16 * a /*inout*/, int ax);
void AE_L16_XC (ae_int16x4 d /*out*/,
                const ae_int16 * a /*inout*/, int ax);
```

AE_LA64.PP Operation:

AE_LA64.PP	u, a	[ae_slot0, ae2_slot0]

Required alignment: 1 byte (but the following instructions have alignment requirements).

Load a 64-bit value from memory to AE_VALIGN register u. The effective address is (a & 0xFFFFFFF8). No update is done to the address register.

This instruction is used to prime the unaligned access stream for all AE_LA<size>.IP and AE_LA<size>.RIP instructions regardless of size or direction.

C syntax:

```
ae_valign AE_LA64_PP (void *a);
```

AE_LA32X2POS.PC, AE_LA32X2NEG.PC Operations:

AE_LA32X2POS.PC	u, a	[ae_slot0, ae2_slot0]
AE_LA32X2NEG.PC	u, a	[ae_slot0, ae2_slot0]

Required alignment: 4 bytes

This operation loads a 64-bit value from memory into AE_VALIGN register `u`. The effective address is (a & 0xFFFFFFF8).

This instruction AE_LA32X2POS.PC is used to prime the unaligned access stream for AE_LA32X2.IC and AE_LA32X2F24.IC instructions. The instruction AE_LA32X2NEG.PC is used to prime the unaligned access stream for AE_LA32X2.RIC and AE_LA32X2F24.RIC instructions.

Note: C intrinsic AE_LA32X2F24POS_PC is implemented using operation AE_LA32X2POS.PC. C intrinsic AE_LA32X2F24NEG_PC is implemented using operation AE_LA32X2NEG.PC.

C syntax:

```
void AE_LA32X2POS_PC (ae_valign u /*out*/, ae_int32x2 *a /*inout*/);
void AE_LA32X2F24POS_PC (ae_valign u /*out*/,ae_f24x2 *a /*inout*/);
void AE_LA32X2NEG_PC (ae_valign u /*out*/, ae_int32x2 *a /*inout*/);
void AE_LA32X2F24NEG_PC (ae_valign u/*out*/, ae_f24x2 *a /*inout*/);
```

AE_LA16X4POS.PC, AE_LA16X4NEG.PC Operations:

AE_LA16X4POS.PC	u, a	[ae_slot0, ae2_slot0]
AE_LA16X4NEG.PC	u, a	[ae_slot0, ae2_slot0]

Required alignment: 2 bytes

Load a 64-bit value from memory into AE_VALIGN register u. The effective address is (a & 0xFFFFFFF8).

The instruction AE_LA16X4POS.PC is used to prime the unaligned access stream for AE_LA16X4.IC instructions. The instruction AE_LA16X4NEG.PC is used to prime the unaligned access stream for AE_LA16X4.RIC instructions.

C syntax:

```
void AE_LA16X4POS_PC (ae_valign u /*out*/, ae_int16x4 *a /*inout*/);
void AE_LA16X4NEG_PC (ae_valign u /*out*/, ae_int16x4 *a /*inout*/);
```

AE_LA24POS.PC, AE_LA24NEG.PC Operations:

AE_LA24POS.PC	u, a	[ae_slot0, ae2_slot0]
AE_LA24NEG.PC	u, a	[ae_slot0, ae2_slot0]

Required alignment: 1 byte

Load a 64-bit value from memory to AE_VALIGN register `u`. The effective address is (a & 0xFFFFFFF8).

The instruction AE_LA24POS.PC is used to prime the unaligned access stream for AE_LA24.IC instructions. The instruction AE_LA24NEG.PC is used to prime the unaligned access stream for AE_LA24.RIC instructions.

C syntax:

```
void AE_LA24POS_PC (ae_valign u /*out*/, void *a /*inout*/);
void AE_LA24NEG_PC (ae_valign u /*out*/, void *a /*inout*/);
```

AE_LA24X2POS.PC, AE_LA24X2NEG.PC Operations:

AE_LA24X2POS.PC	u, a	[ae_slot0, ae2_slot0]
AE_LA24X2NEG.PC	u, a	[ae_slot0, ae2_slot0]

Required alignment: 1 byte

Load a 64-bit value from memory to AE_VALIGN register u. The effective address is (a & 0xFFFFFFF8).

The instruction AE_LA24X2POS.PC is used to prime the unaligned access stream for AE_LA24X2.IC instructions. The instruction AE_LA24X2NEG.PC is used to prime the unaligned access stream for AE_LA24X2.RIC instructions.

C syntax:

```
void AE_LA24X2POS_PC (ae_valign u /*out*/, void a */*inout*/);
void AE_LA24X2NEG_PC (ae_valign u /*out*/, void a */*inout*/);
```

AE_LA32X2.IP Operation:

AE_LA32X2.IP (.IC, .RIP, .RIC)	d, u, a	[ae_slot0, ae2_slot0]

Required alignment: 4 bytes

Load a pair of 32-bit values from effective address (a) in memory into the AE_DR register `d`. Instructions AE_LA32X2.IP (.IC) are used if the direction of the load operations is positive. Instructions AE_LA32X2.RIP (.RIC) are used if the direction of the load operations is negative.

C syntax:

```
void AE_LA32X2_IP (ae_int32x2 d /*out*/, ae_valign u /*inout*/,
                   ae_int32x2 *a /*inout*/);
void AE_LA32X2_IC (ae_int32x2 d /*out*/, ae_valign u /*inout*/,
                   ae_int32x2 *a /*inout*/);
void AE_LA32X2_RIP (ae_int32x2 d /*out*/, ae_valign u /*inout*/,
                   ae_int32x2 *a /*inout*/);
void AE_LA32X2_RIC (ae_int32x2 d /*out*/, ae_valign u /*inout*/,
                    ae_int32x2 *a /*inout*/);
```

AE_LA32X2F24.IP Operation:

AE_LA32X2F24.IP (.IC, .RIP, .RIC)	d, u, a	[ae_slot0, ae2_slot0]

Required alignment: 4 bytes

Load a pair of 24-bit values, each from the most significant 24 bits of a 32-bit half of the 64 bits in memory, sign-extend them to 32 bits and store the values into the two 32-bit elements of AE_DR register `d`. Instructions AE_LA32X2F24.IP (.IC) are used if the direction of the load operations is positive. Instructions AE_LA32X2F24.RIP (.RIC) are used if the direction of the load operations is negative.

C syntax:

```
void AE_LA32X2F24_IP (ae_f24x2 d /*out*/, ae_valign u /*inout*/,
                      ae_f24x2 *a /*inout*/);
void AE_LA32X2F24_IC (ae_f24x2 d /*out*/, ae_valign u /*inout*/,
                      ae_f24x2 *a /*inout*/);
void AE_LA32X2F24_RIP (ae_f24x2 d /*out*/, ae_valign u /*inout*/,
                      ae_f24x2 *a /*inout*/);
void AE_LA32X2F24_RIC (ae_f24x2 d /*out*/, ae_valign u /*inout*/,
                      ae_f24x2 *a /*inout*/);
```

AE_LA16X4.IP Operation:

AE_LA16X4.IP (.IC, .RIP, .RIC)	d, u, a	[ae_slot0, ae2_slot0]

Required alignment: 2 bytes

Load four 16-bit values from effective address (a) in memory into the AE_DR register `d`. Instructions AE_LA16X4.IP (.IC) are used if the direction of the load operations is positive. Instructions AE_LA16X4.RIP (.RIC) are used if the direction of the load operations is negative.

C syntax:

```
void AE_LA16X4_IP (ae_int16x4 d /*out*/, ae_valign u /*inout*/,
                   ae_int16x4 *a /*inout*/);
void AE_LA16X4_IC (ae_int16x4 d /*out*/, ae_valign u /*inout*/,
                   ae_int16x4 *a /*inout*/);
void AE_LA16X4_RIP (ae_int16x4 d /*out*/, ae_valign u /*inout*/,
                   ae_int16x4 *a /*inout*/);
void AE_LA16X4_RIC (ae_int16x4 d /*out*/, ae_valign u /*inout*/,
                   ae_int16x4 *a /*inout*/);
```

AE_LA24.IP Operation:

AE_LA24.IP (.IC, .RIP, .RIC)	d, u, a	[ae_slot0, ae2_slot0]

Required alignment: 1 byte

Load a 24-bit value from effective address (a) in memory into the AE_DR register `d`. Instructions AE_LA24.IP (.IC) are used if the direction of the load operations is positive. Instructions AE_LA24.RIP (.RIC) are used if the direction of the load operations is negative.

C syntax:

```
void AE_LA24_IP (ae_int24x2 d /*out*/, ae_valign u /*inout*/,
                 void *a /*inout*/);
void AE_LA24_IC (ae_int24x2 d /*out*/, ae_valign u /*inout*/,
                 void *a /*inout*/);
void AE_LA24_RIP (ae_int24x2 d /*out*/, ae_valign u /*inout*/,
                 void *a /*inout*/);
void AE_LA24_RIC (ae_int24x2 d /*out*/, ae_valign u /*inout*/,
                 void *a /*inout*/);
```

AE_LA24X2.IP Operation:

AE_LA24X2.IP (.IC, .RIP, .RIC)	d, u, a	[ae_slot0, ae2_slot0]

Required alignment: 1 byte

Load a pair of 24-bit values from effective address (a) in memory into the AE_DR register `d`. Instructions AE_LA24X2.IP (.IC) are used if the direction of the load operations is positive. Instructions AE_LA24X2.RIP (.RIC) are used if the direction of the load operations is negative.

C syntax:

```
void AE_LA24X2_IP (ae_int24x2 d /*out*/, ae_valign u /*inout*/,
                   void *a /*inout*/);
void AE_LA24X2_IC (ae_int24x2 d /*out*/, ae_valign u /*inout*/,
                   void *a /*inout*/);
void AE_LA24X2_RIP (ae_int24x2 d /*out*/, ae_valign u /*inout*/,
                   void *a /*inout*/);
void AE_LA24X2_RIC (ae_int24x2 d /*out*/, ae_valign u /*inout*/,
                   void *a /*inout*/);
```

AE_LALIGN64.I Operation:

AE_LALIGN64.I	u, a, imm	[ae_slot0, ae2_slot0]

Required alignment: 8 bytes

Load a 64-bit value from effective address (a + imm) in memory into the AE_VALIGN register `u`.

C syntax:

```
ae_valign AE_LALIGN64_I (void *a, immediate imm);
```

AE_L16X2M.I, AE_L16X2M.IU, AE_L16X2M.X Operations:

AE_L16X2M.I	d, a, i32	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_L16X2M.IU	d, a, i32	[ae_slot0, ae2_slot0, Inst]
AE_L16X2M.X (.XU, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 4 bytes

Load a pair of 16-bit values from memory, pad 8-bit zeroes at the low end and sign-extend to 32 bits and store the values into the two 32-bit elements of AE_DR register `d`. Refer to Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_LP16X2F_I (_IU, _X, _XU, _C) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_L16X2M.I (.IU, .X, .XU, and .XC), respectively.

C syntax:

```
ae_int32x2 AE_L16X2M_I (const ae_p16x2s * a, immediate i32);
void AE_L16X2M_IU (ae_int32x2 d /*out*/,
                   const ae_p16x2s * a /*inout*/, immediate i32);
      ae_int32x2 AE_L16X2M_X (const ae_p16x2s * a, int ax);
void AE_L16X2M_XU (ae_p16x2s d /*out*/,
                   const ae_p16x2s * a /*inout*/, int ax);
void AE_L16X2M_XC (ae_int32x2 d /*out*/,
                   const ae_p16x2s * a /*inout*/, int ax);
ae_p24x2s AE_LP16X2F_I (const ae_p16x2s * a, immediate i32);
void AE_LP16X2F_IU (ae_p24x2s d /*out*/,
                    const ae_p16x2s * a /*inout*/, immediate i32);
ae_p24x2s AE_LP16X2F_X (const ae_p16x2s * a, int ax);
void AE_LP16X2F_XU (ae_p24x2s d /*out*/,
                    const ae_p16x2s * a /*inout*/, int ax);
void AE_LP16x2F_C (ae_p24x2s d /*out*/,
                   const ae_p16x2s * a /*inout*/, int ax);
```

AE_L32M.I, AE_L32M.IU, AE_L32M.X Operations:

AE_L32M.I	d, a, i32	[ae_slot0, ae2_slot0, Inst , ae_minislot0]
AE_L32M.IU	d, a, i32	[ae_slot0, ae2_slot0, Inst]
AE_L32M.X (.XU, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 4 bytes

Load 32-bit values from memory, pad 16-bit zeroes at the low end and sign-extend to 64 bits and store the values into AE_DR register `d`. Refer to Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_LQ32F_I (_IU, _X, _XU, _C) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_L32M.I (.IU, .X, .XU, .XC), respectively.

C syntax:

```
ae_int64 AE_L32M_I (const ae_q32s * a, immediate i32);
void AE_L32M_IU (ae_int64 d /*out*/,
                 const ae_q32s * a /*inout*/, immediate i32);
ae_int64 AE_L32M_X (const ae_q32s * a, int ax);
void AE_L32M_XU (ae_int64 d /*out*/,
                 const ae_q32s * a /*inout*/, int ax);
void AE_L32M_XC (ae_int64 d /*out*/,
                const ae_q32s * a /*inout*/, int ax);
ae_p56s AE_LQ32F_I (const ae_q32s * a, immediate i32);
void AE_LQ32F_IU (ae_p56s d /*out*/,
                  const ae_q32s * a /*inout*/, immediate i32);
ae_p56s AE_LQ32F_X (const ae_q32s * a, int ax);
void AE_LQ32F_XU (ae_p56s d /*out*/,
                  const ae_q32s * a /*inout*/, int ax);
void AE_LQ32F_C (ae_p56s d /*out*/,
                 const ae_q32s * a /*inout*/, int ax);
```

AE_L16M.I, AE_L16M.IU, AE_L16M.X Operations:

AE_L16M.I	d, a, i16	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_L16M.IU	d, a, i16	[ae_slot0, ae2_slot0, Inst]
AE_L16M.X (.XU, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 2 bytes

Load a 16-bit value from memory, pad 8-bit zeroes at the low end and sign-extend to 32 bits and store the value into both halves of AE_DR register `d`. Refer to Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_LP16F_I (_IU, _X, _XU, _C) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_L16M.I (.IU, .X, .XU, .XC), respectively.

C syntax:

```
ae_int32x2 AE_L16M_I (const ae_p16s * a, immediate i16);
void AE_L16M_IU (ae_int32x2 d /*out*/,
                 const ae_p16s * a /*inout*/, immediate i16);
ae_int32x2 AE_L16M_X (const ae_p16s * a, int ax);
void AE_L16M_XU (ae_int32x2 d /*out*/,
                 const ae_p16s * a /*inout*/, int ax);
void AE_L16M_XC (ae_int32x2 d /*out*/,
                const ae_p16s * a /*inout*/, int ax);
ae_p24x2s AE_LP16F_I (const ae_p16s * a, immediate i16);
void AE_LP16F_IU (ae_p24x2s d /*out*/,
                  const ae_p16s * a /*inout*/, immediate i16);
ae_p24x2s AE_LP16F_X (const ae_p16s * a, int ax);
void AE_LP16F_XU (ae_p24x2s d /*out*/,
                  const ae_p16s * a /*inout*/, int ax);
void AE_LP16F_C (ae_p24x2s d /*out*/,
                 const ae_p16s * a /*inout*/, int ax);
```

2.4.5 Store Operations

The following table gives an overview of the various types of store instructions. The first column indicates a set of store instructions which include 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 2-13 Store Overview

Instruction	Size <sz>	Suffix <adr>	Purpose
AE_S<sz>.<adr>	64	I, X, IP, XP, XC	Aligned stores of scalars
AE_S<sz>.<adr>	32X2, 16X4, 32X2F24	I, X, IP, XP, XC, RIP, RIC	Aligned stores of vectors
AE_S<sz>.L.<adr>	32, 32F24, 16	I, X, IP, XP, XC	Aligned stores of scalars from the low part of a register
AE_S<sz>.<adr>	32RA64S, 24RA64S	I, IP, X, XP, XC	Aligned stores of scalars from the middle part of a register with rounding and saturation
AE_S<sz>.<adr>	32X2RA64S, 24X2RA64S	IP	Aligned Stores of two scalars from the middle part of a register with rounding and saturation
AE_SA<sz>.<adr>	32X2, 32X2F24, 16X4, 24, 24X2,	IP, IC, RIP, RIC	Unaligned stores for accessing vectors of aligned scalars
AE_SA64POS.FP			Flush after unaligned store with positive stride
AE_SA64NEG.FP			Flush after unaligned store with negative stride
AE_SALIGN64.I			Store of alignment register
AE_ZALIGN64			Zero alignment register
AE_S<sz>M.<adr>	16X2, 32, 16	I, X, XC, IU, XU	Legacy Stores

AE_S64.I, AE_S64.IP, AE_S64.X Operations:

AE_S64.I	d, a, i64	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_S64.IP	d, a, i64	[ae_slot0, ae2_slot0, Inst]
AE_S64.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 8 bytes

Store the 64 bits of the AE_DR register `d` to memory. See Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_SQ56S_I (_X, _C, _IU, _XU) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_SQ64.I (.X, .XC, .I, .I), respectively.

C syntax:

```
void AE_S64_I (ae_int64 d, ae_int64 * a, immediate i64);
void AE_S64_X (ae_int64 d, ae_int64 * a, int ax)
void AE_S64_IP (ae_int64 d, ae_int64 * a /*inout*/, immediate i64);
void AE_S64_XP (ae_int64 d, ae_int64 * a /*inout*/, int ax);
void AE_S64_XC (ae_int64 d, ae_int64 * a /*inout*/, int ax);
void AE_SQ56S_I (ae_q56s d, ae_q56s * a, immediate i64);
void AE_SQ56S_IU (ae_q56s d, ae_q56s * a /*inout*/, immediate i64);
void AE_SQ56S_X (ae_q56s d, ae_q56s * a, int ax)
void AE_SQ56S_XU (ae_q56s d, ae_q56s * a /*inout*/, int ax);
void AE_SQ56S_C (ae_q56s d, ae_q56s * a /*inout*/, int ax);
```

AE_S32X2.I, AE_S32X2.IP, AE_S32X2.RIP, AE_S32X2.X Operations:

AE_S32X2.I	d, a, i64	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_S32X2..IP	d, a, i64pos	[ae_slot0, ae2_slot0, Inst]
AE_S32X2.RIP (.RIC)	d, a	[ae_slot0, ae2_slot0, Inst]
AE_S32X2.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 8 bytes

Store a pair of 32-bit values from the AE_DR register `d` to memory. Refer to Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_SP24X2S_I (_X, _C, _IU, _XU) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_SP32X2.I (.X, .XC, .I, .I), respectively.

C syntax:

```
void AE_S32X2_I (ae_int32x2 d, ae_int32x2 * a, immediate i64);
void AE_S32X2_X (ae_int32x2 d, ae_int32x2 * a, int ax);
void AE_S32X2_IP (ae_int32x2 d,
                  ae_int32x2 * a /*inout*/, immediate i64);
void AE_S32X2_XP (ae_int32x2 d,
                  ae_int32x2 * a /*inout*/, int ax);
void AE_S32X2_XC (ae_int32x2 d,
                 ae_int32x2 * a /*inout*/, int ax);
void AE_S32X2_RIP (ae_int32x2 d, ae_int32x2 * a /*inout*/);
void AE_S32X2_RIC (ae_int32x2 d, ae_int32x2 * a /*inout*/);
void AE_SP24X2S_I (ae_p24x2s d, ae_p24x2s * a, immediate i64);
void AE_SP24X2S_IU (ae_p24x2s d,
                    ae_p24x2s * a /*inout*/, immediate i64);
void AE_SP24X2S_X (ae_p24x2s d, ae_p24x2s * a, int ax);
void AE_SP24X2S_XU (ae_p24x2s d,
                    ae_p24x2s * a /*inout*/, int ax);
void AE_SP24X2S_C (ae_p24x2s d,
                   ae_p24x2s * a /*inout*/, int ax);
```

AE_S16X4.I, AE_S16X4.IP, AE_S16X4.RIP, AE_S16X4.X Operations:

AE_S16X4.I	d, a, i64	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_S16X4.IP	d, a, i64pos	[ae_slot0, ae2_slot0, Inst]
AE_S16X4.RIP (.RIC)	d, a	[ae_slot0, ae2_slot0, Inst]
AE_S16X4.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 8 bytes

Store four 16-bit values from AE_DR register `d` to memory. Refer to Table 2-3 for the meanings of the address mode suffixes.

C syntax:

```
void AE_S16X4_I (ae_int16x4 d, ae_int16x4 * a, immediate i64);
void AE_S16X4_X (ae_int16x4 d, ae_int16x4 * a, int ax);
void AE_S16X4_IP (ae_int16x4 d,
                  ae_int16x4 * a /*inout*/, immediate i64);
      void AE_S16X4_RIP (ae_int16x4 d, ae_int16x4 * a /*inout*/);
      void AE_S16X4_RIC (ae_int16x4 d, ae_int16x4 * a /*inout*/);
void AE_S16X4_XP (ae_int16x4 d,
                  ae_int16x4 * a /*inout*/, int ax);
void AE_S16X4_XC (ae_int16x4 d,
                 ae_int16x4 * a /*inout*/, unsigned ax);
```

AE_S32X2F24.I, AE_S32X2F24.IP, AE_S32X2F24.RIP, AE_S32X2F24.X Operations:

AE_S32X2F24.I	d,a, 4	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_S32X2F24.IP	d, a, i64pos	[ae_slot0, ae2_slot0, Inst]
AE_S32X2F24.RIP (.RIC)	d, a	[ae_slot0, ae2_slot0, Inst]
AE_S32X2F24.X (.XU, .XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 8 bytes

Store the 24 LSBs of the two 32-bit elements of AE_DR register `d` with each value padded on the right with zeroes to 32 bits and placed in half of the 64 bits in memory. Refer to Table 2-3 for the meanings of the address mode suffixes. The intent here is that the values in the register `d` represent 9.23-bit values that get padded to a 1.31-bit memory representation.

Note: C intrinsics AE_SP24X2F_I (_X, _C, _IU, _XU) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_S32X2F24.I (.X, .XC, .I, .I), respectively.

C syntax:

```
void AE_S32X2F24_I (ae_f24x2 d, ae_f24x2 *a, immediate i64);
void AE_S32X2F24_X (ae_f24x2 d, ae_f24x2 * a, int ax);
void AE_S32X2F24_IP (ae_f24x2 d,
ae_f24x2 * a /*inout*/, immediate i64);
```

```
void AE_S32X2F24_RIP (ae_f24x2 d, ae_f24x2 * a /*inout*/);
void AE_S32X2F24_RIC (ae_f24x2 d, ae_f24x2 * a /*inout*/);
void AE_S32X2F24_XP (ae_f24x2 d,
                     ae_f24x2 * a /*inout*/, int ax);
void AE_S32X2F24_XC (ae_f24x2 d,
                    ae_f24x2 * a /*inout*/, int ax);
void AE_SP24X2F_I (ae_p24x2s d, ae_p24x2f * a, immediate i64);
void AE_SP24X2F_IU (ae_p24x2s d,
                    ae_p24x2f * a /*inout*/, immediate i64);
void AE_SP24X2F_X (ae_p24x2s d, ae_p24x2f * a, int ax);
void AE_SP24X2F_XU (ae_p24x2s d,
                    ae_p24x2f * a /*inout*/, int ax);
void AE_SP24X2F_C (ae_p24x2s d,
                   ae_p24x2f * a /*inout*/, int ax);
```

AE_S32.L.I, AE_S32.L.IP, AE_S32.L.X Operations:

AE_S32.L.I	d, a, i32	[ae_slot0, ae2_slot0, Inst ae_minislot0]
AE_S32.L..IP	d, a, i32	[ae_slot0, ae2_slot0, Inst]
AE_S32.L.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 4 bytes

Store the 32-bit `L` element of the AE_DR register `d` to memory. For operations with suffix .I, the effective address is (a + i32). Refer to Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_SP24S_L_I (_X, _C, _IU, _XU) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_S32.L.I (.X, .XC, .I, .I), respectively.

C syntax:

```
void AE_S32_L_I (ae_int32x2 d, ae_int32 * a, immediate i32);
void AE_S32_L_X (ae_int32x2 d, ae_int32 * a, int ax)
void AE_S32_L_IP (ae_int32x2 d,
                  ae_int32 * a /*inout*/, immediate i32);
void AE_S32_L_XP (ae_int32x2 d,
                  ae_int32 * a /*inout*/, int ax);
void AE_S32_L_XC (ae_int32x2 d,
                 ae_int32 * a /*inout*/, int ax);
void AE_SP24S_L_I (ae_p24x2s d, ae_p24s * a, immediate i32);
void AE_SP24S_L_IU (ae_p24x2s d,
                  ae_p24s * a /*inout*/, immediate i32);
void AE_SP24S_L_X (ae_p24x2s d, ae_p24s * a, int ax)
void AE_SP24S_L_XU (ae_p24x2s d,
                  ae_p24s * a /*inout*/, int ax);
void AE_SP24S_L_C (ae_p24x2s d,
                   ae_p24s * a /*inout*/, int ax);
```

AE_S32F24.L.I, AE_S32F24.L.IP, AE_S32F24.L.X Operations:

AE_S32F24.L.I	d, a, i32	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_S32F24.L.IP	d, a, i32	[ae_slot0, ae2_slot0, Inst]
AE_S32F24.L.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 4 bytes

Store the 24 LSBs from the `L` element of the AE_DR register `d`, padded with zeroes on the right, to the 32 bits in memory. Refer to Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_SP24F_L_I (_X, _C, _IU, _XU) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_S32F24.L.I (.X, .XC, .I, .I), respectively.

C syntax:

```
void AE_S32F24_L_I (ae_f24x2 d, ae_f24 * a, immediate i32);
void AE_S32F24_L X (ae_f24x2 d, ae_f24 * a, int ax);
void AE_S32F24_L IP (ae_f24x2 d,
                     ae_f24 * a /*inout*/, immediate i32);
void AE_S32F24_L_XP (ae_f24x2 d,
                     ae_f24 * a /*inout*/, int ax);
void AE_S32F24_L_XC (ae_f24x2 d,
                    ae_f24 * a /*inout*/, int ax);
void AE_SP24F_L_I (ae_p24x2s d, ae_p24f * a, immediate i32);
void AE_SP24F_L_IU (ae_p24x2s d,
                     ae_p24f * a /*inout*/, immediate i32);
void AE_SP24F_L_X (ae_p24x2s d, ae_p24f * a, int ax);
void AE_SP24F_L_XU (ae_p24x2s d,
                     ae_p24f * a /*inout*/, int ax);
void AE_SP24F_L_C (ae_p24x2s d,
                   ae_p24f * a /*inout*/, int ax);
```

AE_S16.0.I, AE_S16.0.IP, AE_S16.0.X Operations:

AE_S16.0.I	d, a, i16	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_S16.0.IP	d, a, i16	[ae_slot0, ae2_slot0, Inst]
AE_S16.0.X (.XP, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 2 bytes

Store the 16-bit `0` element of the AE_DR register `d` to memory. Refer to Table 2-3 for the meanings of the address mode suffixes.

C syntax:

```
void AE_S16_0_I (ae_int16x4 d, ae_int16 * a, immediate i16);
void AE_S16_0_X (ae_int16x4 d, ae_int16 * a, int ax);
void AE_S16_0_IP (ae_int16x4 d,
                  ae_int16 * a /*inout*/, immediate i16);
void AE_S16_0_XP (ae_int16x4 d, ae_int16 * a, int ax);
void AE_S16_0_XC (ae_int16x4 d, ae_int16 * a, int ax);
```

AE_SA16X4.IP Operation:

AE_SA16X4.IP (.IC, .RIP, .RIC)	d, u, a	[ae_slot0, ae2_slot0]

Required alignment: 2 bytes

Store four 16-bit values from AE_DR register `d` to memory with effective address (a). Instructions AE_SA16X4.IP (.IC) are used if the direction of the store operations is positive. Instructions AE_SA16X4.RIP (.RIC) are used if the direction of the store operations is negative.

C syntax:

```
void AE_SA16X4_IP (ae_int16x4 d, ae_valign u /*inout*/,
                   ae_int16x4 * a /*inout*/);
void AE_SA16X4_IC (ae_int16x4 d, ae_valign u /*inout*/,
                   ae_int16x4 * a /*inout*/);
void AE_SA16X4_RIP (ae_int16x4 d, ae_valign u /*inout*/,
                    ae_int16x4 * a /*inout*/);
void AE_SA16X4_RIC (ae_int16x4 d, ae_valign u /*inout*/,
                    ae_int16x4 * a /*inout*/);
```

AE_SA32X2.IP Operation:

AE_SA32X2.IP (.IC, .RIP, .RIC)	d, u, a	[ae_slot0, ae2_slot0]

Required alignment: 4 bytes

Store a pair of 32-bit values from AE_DR register `d` to memory with effective address (a). Instructions AE_SA32X2.IP (.IC) are used if the direction of the store operations is positive. Instructions AE_SA32X2.RIP (.RIC) are used if the direction of the store operations is negative.

C syntax:

```
void AE_SA32X2_IP (ae_int32x2 d, ae_valign u /*inout*/,
                   ae_int32x2 * a /*inout*/);
void AE_SA32X2_IC (ae_int32x2 d, ae_valign u /*inout*/,
                   ae_int32x2 * a /*inout*/);
void AE_SA32X2_RIP (ae_int32x2 d, ae_valign u /*inout*/,
                    ae_int32x2 * a /*inout*/);
void AE_SA32X2_RIC (ae_int32x2 d, ae_valign u /*inout*/,
                    ae_int32x2 * a /*inout*/);
```

AE_SA32X2F24.IP Operation:

AE_SA32X2F24.IP (.IC, .RIP, .RIC)	d, u, a	[ae_slot0, ae2_slot0]

Required alignment: 4 bytes

Store the 24 LSBs of the two 32-bit elements of AE_DR register `d`, with each value padded on the right with zeroes to 32 bits and placed in half of the 64 bits in memory with effective address (a). Instructions AE_SA32X2F24.IP (.IC) are used if the direction of the store operations is positive. Instructions AE_SA32X2F24.RIP (.RIC) are used if the direction of the store operations is negative.

C syntax:

```
void AE_SA32X2F24_IP (ae_f24x2 d, ae_valign u /*inout*/,
                      ae_ f24x2 * a /*inout*/);
void AE_SA32X2F24_IC (ae_ f24x2 d, ae_valign u /*inout*/,
                      ae_ f24x2 * a /*inout*/);
void AE_SA32X2F24_RIP (ae_ f24x2 d, ae_valign u /*inout*/,
                       ae_ f24x2 * a /*inout*/);
void AE_SA32X2F24_RIC (ae_ f24x2 d, ae_valign u /*inout*/,
                       ae_ f24x2 * a /*inout*/);
```

AE_SA24.L.IP Operation:

AE_SA24.L.IP (.IC, .RIP, .RIC)	d, u, a	[ae_slot0, ae2_slot0]

Required alignment: 1 byte

Store the 24 LSBs of AE_DR register `d` to 24 bits in memory with effective address (a). Instructions AE_SA24.IP (.IC) are used if the direction of the store operations is positive. Instructions AE_SA24.RIP (.RIC) are used if the direction of the store operations is negative.

C syntax:

```
void AE_SA24_L_IP (ae_int24x2 d, ae_valign u /*inout*/,
                   void * a /*inout*/);
void AE_SA24_L_IC (ae_int24x2 d, ae_valign u /*inout*/,
                   void * a /*inout*/);
void AE_SA24_L_RIP (ae_int24x2 d, ae_valign u /*inout*/,
                    void * a /*inout*/);
void AE_SA24_L_RIC (ae_int24x2 d, ae_valign u /*inout*/,
                    void * a /*inout*/);
```

AE_SA24X2.IP Operation:

AE_SA24X2.IP (.IC, .RIP, .RIC)	d, u, a	[ae_slot0, ae2_slot0]

Required alignment: 1 byte

Store the 24 LSBs of the two 32-bit elements of AE_DR register `d` to 48 bits in memory with effective address (a). Instructions AE_SA24X2.IP (.IC) are used if the direction of the store operations is positive. Instructions AE_SA24X2.RIP (.RIC) are used if the direction of the store operations is negative.

C syntax:

```
void AE_SA24X2_IP (ae_int24x2 d, ae_valign u /*inout*/,
                   void * a /*inout*/);
void AE_SA24X2_IC (ae_int24x2 d, ae_valign u /*inout*/,
                   void * a /*inout*/);
void AE_SA24X2_RIP (ae_int24x2 d, ae_valign u /*inout*/,
                   void * a /*inout*/);
void AE_SA24X2_RIC (ae_int24x2 d, ae_valign u /*inout*/,
                   void * a /*inout*/);
```

AE_SALIGN64.I Operation:

AE_SALIGN64.I	u, a, imm	[ae_slot0, ae2_slot0]

Required alignment: 8 bytes

Stores a 64-bit value from AE_VALIGN register u to memory with effective address (a + imm).

C syntax:

```
void AE_LALIGN64_I (ae_valign u, void *a, immediate imm);
```

AE_SA64POS.FP Operation:

AE_SA64POS.FP	u, a	[ae_slot0, ae2_slot0]

Required alignment: varies depending on the data type in the AE_VALIGN register `u`.

Flushes the value in AE_VALIGN register `u` to memory with effective address (a). The AE_VALIGN register `u` is updated with a value of zero. This operation is used when the direction of the store operation is positive.

C syntax:

```
void AE_SA64POS_FP (ae_valign u /*inout*/, void *a);
void AE_SA64POS_FC (ae_valign u /*inout*/, void *a);
```

AE_SA64NEG.FP Operation:

AE_SA64NEG.FP	u, a	[ae_slot0, ae2_slot0]

Required alignment: varies depending on the data type in the AE_VALIGN register `u`.

Flushes the value in AE_VALIGN register `u` to memory with effective address (a). The AE_VALIGN register `u` is updated with a value of zero. This operation is used when the direction of the store operation is negative.

C syntax:

```
void AE_SA64NEG_FP (ae_valign u /*inout*/, void *a);
void AE_SA64NEG_FC (ae_valign u /*inout*/, void *a);
```

AE_ZALIGN64 Operation:

AE_ZALIGN64	u	[ae_slot0, ae2_slot0]

Initialize the AE_VALIGN register `u` with zero.

C syntax:

```
ae_valign AE_ZALIGN64 ();
```

AE_S16X2M.I, AE_S16X2M.X Operations:

AE_S16X2M.I (.IU)	d, a, i32	[ae_slot0, ae2_slot0, Inst]
AE_S16X2M.X (.XU, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 4 byte.

Store the middle 16-bit element of each 32-bit half of AE_DR register `d` into 32 bits in memory. Refer to Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_SP16X2F_I (_IU, _X, _XU, _C) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_S16X2M.I (.IU, .X, .XU, .XC), respectively.

C syntax:

```
void AE_S16X2M_I (ae_int32x2 d, ae_p16x2s *a, immediate i32);
void AE_S16X2M_IU (ae_int32x2 d, ae_p16x2s *a /*inout*/, immediate i32);
void AE_S16X2M_X (ae_int32x2 d, ae_p16x2s *a, int ax);
void AE_S16X2M_XU (ae_int32x2 d, ae_p16x2s *a /*inout*/, int ax);
void AE_S16X2M_XC (ae_int32x2 d, ae_p16x2s *a /*inout*/, int ax);
void AE_SP16X2F_I (ae_p24x2s d, ae_p16x2s *a, immediate i32);
void AE_SP16X2F_IU (ae_p24x2s d, ae_p16x2s *a /*inout*/, immediate i32);
void AE_SP16X2F_X (ae_p24x2s d, ae_p16x2s *a, int ax);
void AE_SP16X2F_XU (ae_p24x2s d, ae_p16x2s *a /*inout*/, int ax);
void AE_SP16X2F_C (ae_p24x2s d, ae_p16x2s *a /*inout*/, unsigned ax);
```

AE_S32M.I, AE_S32M.IU, AE_S32M.X Operations:

AE_S32M.I	d, a, i32	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_S32M.IU	d, a, i32	[ae_slot0, ae2_slot0, Inst]
AE_S32M.X (.XU, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 4 bytes

Store the middle 32-bit element of AE_DR register `d` into 32 bits in memory. See Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_SQ32F_I (_IU, _X, _XU, _C) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_S32M.I (.IU, .X, .XU, .XC), respectively.

C syntax:

```
void AE_S32M_I (ae_int64 d, ae_q32s *a, immediate i32);
void AE_S32M_IU (ae_ int64 d, ae_q32s *a /*inout*/, immediate i32);
void AE_S32M_X (ae_ int64 d, ae_q32s *a, int ax);
void AE_S32M_XU (ae_ int64 d, ae_q32s *a /*inout*/, int ax);
void AE_S32M_XC (ae_ int64 d, ae_q32s *a /*inout*/, int ax);
void AE_SQ32F_I (ae_q56s d, ae_q32s *a, immediate i32);
void AE_SQ32F_IU (ae_q56s d, ae_q32s *a /*inout*/, immediate i32);
void AE_SQ32F_X (ae_q56s d, ae_q32s *a, int ax);
void AE_SQ32F_XU (ae_q56s d, ae_q32s *a /*inout*/, int ax);
void AE_SQ32F_C (ae_q56s d, ae_q32s *a /*inout*/, int ax);
```

AE_S16M.L.I, AE_S16M.L.IU, AE_S16M.L.X Operations:

AE_S16M.L.I	d, a, i16	[ae_slot0, ae2_slot0, Inst, ae_minislot0]
AE_S16M.L.IU	d, a, i16	[ae_slot0, ae2_slot0, Inst]
AE_S16M.L.X (.XU, .XC)	d, a, ax	[ae_slot0, ae2_slot0, Inst]

Required alignment: 2 bytes

Store the middle 16-bit element of the low-order 32-bit element of AE_DR register `d` into 16 bits in memory. Refer to Table 2-3 for the meanings of the address mode suffixes.

Note: C intrinsics AE_SP16F_L_I (_IU, _X, _XU, _C) are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_S16M.L.I (.IU, .X, .XU, .XC), respectively.

C syntax:

```
void AE_S16M_L_I (ae_int32x2 d, ae_p16s *a, immediate i16);
void AE_S16M_L_IU (ae_int32x2 d, ae_p16s *a /*inout*/, immediate i16);
void AE_S16M_L_X (ae_int32x2 d, ae_p16s *a, int ax);
void AE_S16M_L_XU (ae_int32x2 d, ae_p16s *a /*inout*/, int ax);
void AE_S16M_L_XC (ae_int32x2 d, ae_p16s *a /*inout*/, int ax);
```

```
void AE_SP16F_L_I (ae_p24x2s d, ae_p16s *a, immediate i16);
void AE_SP16F_L_IU (ae_p24x2s d, ae_p16s *a /*inout*/, immediate i16);
void AE_SP16F_L_X (ae_p24x2s d, ae_p16s *a, int ax);
void AE_SP16F_L_XU (ae_p24x2s d, ae_p16s *a /*inout*/, int ax);
void AE_SP16F_L_C (ae_p24x2s d, ae_p16s *a /*inout*/, int ax);
```

2.5 Multiply and Accumulate Operations

The HiFi 3 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 3 supports four 24x24-bit, 32x16-bit, or 16x16-bit multiplies per cycle or two 32x24-bit or 32x32-bit multiplies per cycle. Individual operations perform one, two, or four multiplies. Single or dual-multiply operations can typically be dual-issued in a VLIW bundle.

HiFi 3 MAC operations are named using the following convention:

```
AE_MUL<accum_type>[F][DPC]<size>{R,RA}[S][U].specifier
```

The operations use a specifier of an `L` or `H` suffix to select input operands from the two 32-bit AE_DR elements or a 0, 1, 2, 3 suffix for 16-bit data.

The two and four MAC operations have two forms—dual MACs take the results of two MACs and add or subtract them together, as in the example below.

```
acc = acc – d0.L*d1.L + d0.H*d1.H.
```

SIMD MACs do not combine the results of different multiplies. They instead perform the sample multiply operation on different portions of the data, as in the example below.

```
acc.h = acc.h – d0.h*d1.h
acc.l = acc.l – d0.l*d1.l
```

The dual MACs use a `D` in the name. Most of the SIMD MACs pack their results into 32 or 16 bits and hence use a `P` in their name. By adding or subtracting two multiply results together, the dual MAC instructions are able to maintain high precision for their accumulation without needing to write multiple output registers.

Complex multiply operations are quad-MAC operations that pack their two results down to 32-bits after combining the two terms comprising each real and imaginary component. They are designated with a `C` rather than a `P`.

Among the single-multiply and SIMD multiply operations, each family of multiply/accumulate operations has a multiply-only variant, a multiply/add variant, and a multiply/subtract variant, denoted by having `accum_type` set to nothing, `A` or `S` respectively. With the `MUL` variant, the accumulator contents are overwritten with the result of the multiplication. With the `MULA` variant, the result of the multiplication is added to the accumulator contents and written back to the accumulator. With the `MULS` variant, the result of the multiplication is subtracted from the accumulator contents and written back to the accumulator.

Dual MAC operations with an `accum_type` starting with `Z` do their accumulation against zero; in other words, the initial contents of the accumulator are discarded. Operations without a `Z` accumulate against the initial contents of the accumulator. Following the optional `Z` there are two letters that indicate addition or subtraction, one for each of the two multiplication results.

HiFi 3 supports both integral and fractional multiplication. Fractional multiply instructions have an `F` immediately following the `accum_type`.

The `size` of a multiply instruction is 16, 24, 32, or 32X16 for 16 bit, 24 bit, 32 bit and 32 times 16 bit respectively. For SIMD multipliers, there is an `X2` or `X4` suffix added to the `size` to signify the number of SIMD elements.

Integral SIMD multiply instructions throw away the upper bits of their results, just like standard C/C++ multiplies. Fractional SIMD multiply instructions round away the lower bits using either a symmetric or asymmetric rounding. They are signified with `R` or `RA` in the name. With asymmetric rounding, halves are rounded upward, *i.e.*, 0.5 times the least significant result bit is rounded up to 1.0 and -0.5 times the least significant result bit is rounded up to 0. With symmetric rounding, halves are rounding away from zero, *i.e.*, -0.5 times the least significant result bit is rounded down to -1.0. In the instruction descriptions, symmetric rounds are referred to as *round* while asymmetric are referred to as *round*$^{+\infty}$.

MAC operations without guard bits, 1.31x1.31 into 1.63, 1.31x1.15 into 1.31 and 1.15x1.15 into 1.15 or 1.31, saturate their results. All other MAC operations have guard bits and do not saturate. Saturating multiplies have an `S` following the `size` or the rounding designation. Some 16x16-bit multipliers are designed to be bit exact with the ITU-T/ETSI intrinsics and therefore do multiple saturations in series. These instructions have `SS` in the name.

Unsigned multiplies have a `U` preceding the `specifier`.

All MAC operations appear in slot ae_slot1 or ae_slot2 of the 3-slot format or ae2_slot1 of the 2-slot format. Any multiply operation appearing in ae_slot2 will have a `_S2` suffix. The C/C++ programmer can ignore the suffix. The compiler will automatically convert a normal multiply into a `_S2` multiply when needed.

HiFi 2/EP had a different naming scheme for multipliers. Compatibility intrinsics are provided for all the old HiFi 2/EP intrinsics and are listed in the following sections.

2.5.1 24x24-bit Multiplication Operations

HiFi 3 supports dual and quad 24x24-bit multiplication operations. SIMD variants compute two or four products that are individually accumulated in 32-bit precision. Non-SIMD variants compute the sum or difference of two 48-bit products added or subtracted to a 64-bit accumulator. There is no support for single 24x24-bit multiplication; use 32x32-bit instructions instead. To ensure compatibility with HiFi 2 and consistency with the dual multiply instructions, 24-bit single multiplication intrinsics are provided. However, these intrinsics are implemented using the higher precision 32x32-bit multipliers.

AE_MULZAAFD24.HH.LL, AE_MULZSSFD24.HH.LL, AE_MULZASFD24.HH.LL, AE_MULZSAFD24.HH.LL, AE_MULAAFD24.HH.LL, AE_MULSSFD24.HH.LL, AE_MULASFD24.HH.LL, AE_MULSAFD24.HH.LL Operations:

AE_MULZAAFD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZSSFD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZASFD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZSAFD24.HH.LL	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAAFD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSSFD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULASFD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSAFD24.HH.LL	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Dual 1.23x1.23-bit into 17.47-bit signed MAC:

$$d \leftarrow [d_{17.47}] \pm d0.H_{1.23} \times d1.H_{1.23} \pm d0.L_{1.23} \times d1.L_{1.23}$$

Note: C intrinsics with ae_p24x2s input operand types and ae_q56s accumulator operand types are provided to ensure HiFi 2 code portability and are implemented through the operations above.

C syntax:

```
ae_f64 AE_MULZAAFD24_HH_LL (ae_f24x2 d0, ae_f24x2 d1);
void AE_MULAAFD24_HH_LL (ae_f64 d /*inout*/,
                         ae_f24x2 d0, ae_f24x2 d1);
ae_q56s AE_MULZAAFP24S_HH_LL (ae_p24x2s d0, ae_p24x2s d1);
void AE_MULAAFP24S_HH_LL (ae_q56s q /*inout*/,
                          ae_p24x2s d0, ae_p24x2s d1);
```

AE_MULFD24X2.FIR.H, AE_MULAFD24X2.FIR.H Operations:

AE_MULFD24X2.FIR.H (.L)	q0, q1, d0, d1, c	[ae2_slot1]
AE_MULAFD24X2.FIR.H (.L)	q0, q1, d0, d1, c	[ae2_slot1]

Quad 1.23x1.23-bit multiplications into two 17.47-bit signed MAC with operands selected to accelerate FIR computations.

For the .H version

$$q0 \leftarrow [q0_{17.47}] + d0.H_{1.23} \times c.H_{1.23} + d0.L_{1.23} \times c.L_{1.23}$$
$$q1 \leftarrow [q1_{17.47}] + d0.L_{1.23} \times c.H_{1.23} + d1.H_{1.23} \times c.L_{1.23}$$

For the .L version

$$q0 \leftarrow [q0_{17.47}] + d0.L_{1.23} \times c.H_{1.23} + d1.H_{1.23} \times c.L_{1.23}$$
$$q1 \leftarrow [q1_{17.47}] + d1.H_{1.23} \times c.H_{1.23} + d1.L_{1.23} \times c.L_{1.23}$$

C syntax:

```
void AE_MULFD24X2_FIR_H (ae_f64 q0 /*out*/, ae_f64 q1 /*out*/,
                         ae_f24x2 d0,ae_f24x2 d1, ae_f24x2 c);
void AE_MULAFD24X2_FIR_H(ae_f64 q0 /*inout*/,
                         ae_f64 q1 /* inout*/
                         ae_f24x2 d0,ae_f24x2 d1,ae_f24x2 c);
```

AE_MULZAAD24.HH.LL, AE_MULZSSD24.HH.LL, AE_MULZASD24.HH.LL, AE_MULZSAD24.HH.LL, AE_MULAAD24.HH.LL, AE_MULSSD24.HH.LL, AE_MULASD24.HH.LL, AE_MULSAD24.HH.LL Operations:

AE_MULZAAD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZSSD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZASD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZSAD24.HH.LL	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAAD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSSD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULASD24.HH.LL (.HL.LH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSAD24.HH.LL	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Dual 24x24-bit into 64-bit signed integer MAC with no saturation:

$$d \leftarrow [d] \pm d0.H \times d1.H \pm d0.L \times d1.L$$

Note: C intrinsics with ae_p24x2s input operand types and ae_q56s accumulator operand types are provided to ensure HiFi 2 code portability and are implemented through the operations above.

C syntax:

```
ae_int64 AE_MULZAAD24_HH_LL (ae_int24x2 p0, ae_int24x2 p1);
void AE_MULAAD24_HH_LL (ae_int64 d /*inout*/,
                        ae_int24x2 p0, ae_int24x2 p1);
ae_q56s AE_MULZAAP24S_HH_LL (ae_p24x2s p0, ae_p24x2s p1);
void AE_MULAAP24S_HH_LL (ae_q56s q /*inout*/,
                         ae_p24x2s p0, ae_p24x2s p1);
```

AE_MULFC24RA, AE_MULAFC24RA Operations:

AE_MULFC24RA	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULAFC24RA	d, d0, d1	[ae_slot1, ae2_slot1]

Complex quad-mac 1.23x1.23-bit into 9.23-bit signed MAC with asymmetric rounding of the product.

$$d.H \leftarrow [d.H_{9.23} +] \mathit{round}^{+\infty}{}_{9.23}(d0.H_{1.23} \times d1.H_{1.23} - d0.L_{1.23} \times d1.L_{1.23})$$

$$d.L \leftarrow [d.L_{9.23} +] \mathit{round}^{+\infty}{}_{9.23}(d0.H_{1.23} \times d1.L_{1.23} + d0.L_{1.23} \times d1.H_{1.23})$$

C syntax:

```
ae_f32x2 AE_MULFC24RA (ae_f24x2 d0, ae_f24x2 d1);
void AE_MULAFC24RA (ae_f32x2 d /*inout*/,
                    ae_f24x2 d0, ae_f24x2 d1);
```

AE_MULC24, AE_MULAC24 Operations:

AE_MULC24	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULAC24	d, d0, d1	[ae_slot1, ae2_slot1]

Complex quad-mac 24x24 bit into 32-bit signed integer MAC with no saturation:

d.H ← [d.H +] d0.H × d1.H - d0.L × d1.L

d.L ← [d.L +] d0.H × d1.L + d0.L × d1.H

C syntax:

```
ae_int32x2 AE_MULC24 (ae_int24x2 d0, ae_int24x2 d1);
void AE_MULAC24 (ae_int32x2 d /*inout*/,
                 ae_int24x2 d0, ae_int24x2 d1);
```

AE_MULFP24X2R, AE_MULAFP24X2R, AE_MULSFP24X2R Operations:

AE_MULFP24X2R	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAFP24X2R	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSFP24X2R	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

2-way SIMD 1.23x1.23 bit into 9.23-bit signed MAC with symmetric (away from zero) rounding of the product.

$d.H \leftarrow [d.H_{9.23} \pm] \mathit{round}_{9.23}(d0.H_{1.23} \times d1.H_{1.23})$

$d.L \leftarrow [d.L_{9.23} \pm] \mathit{round}_{9.23}(d0.L_{1.23} \times d1.L_{1.23})$

C syntax:

```
ae_f32x2 AE_MULFP24X2R (ae_f24x2 d0, ae_f24x2 d1);
void AE_MULAFP24X2R (ae_f32x2 d /*inout*/,
                     ae_f24x2 d0, ae_f24x2 d1);
void AE_MULSFP24X2R (ae_f32x2 d /*inout*/,
                     ae_f24x2 d0, ae_f24x2 d1);
```

AE_MULFP24X2RA, AE_MULAFP24X2RA, AE_MULSFP24X2RA Operations:

AE_MULFP24X2RA	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAFP24X2RA	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSFP24X2RA	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

2-way SIMD 1.23x1.23-bit into 9.23-bit signed MAC with asymmetric rounding of the product.

$d.H \leftarrow [d.H_{9.23} \pm] \mathit{round}^{+\infty}{}_{9.23}(d0.H_{9.23} \times d1.H_{9.23})$

$d.L \leftarrow [d.L_{9.23} \pm] \mathit{round}^{+\infty}{}_{9.23}(d0.L_{9.23} \times d1.L_{9.23})$

C syntax:

```
ae_f32x2 AE_MULFP24X2RA (ae_f24x2 d0, ae_f24x2 d1);
void AE_MULAFP24X2RA (ae_f32x2 d /*inout*/,
                      ae_f24x2 d0, ae_f24x2 d1);
void AE_MULSFP24X2RA (ae_f32x2 d /*inout*/,
                      ae_f24x2 d0, ae_f24x2 d1);
```

AE_MULP24X2, AE_MULAP24X2, AE_MULSP24X2 Operations:

AE_MULP24X2	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAP24X2	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSP24X2	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

2-way SIMD 24x24-bit into 32-bit signed integer MAC with no saturation:

$d.H \leftarrow [d.H \pm] d0.H \times d1.H$

$d.L \leftarrow [d.L \pm] d0.L \times d1.L$

C syntax:

```
ae_int32x2 AE_MULP24X2 (ae_int24x2 d0, ae_int24x2 d1);
void AE_MULAP24X2 (ae_int32x2 d /*inout*/,
                   ae_int24x2 d0, ae_int24x2 d1);
void AE_MULSP24X2 (ae_int32x2 d /*inout*/,
                   ae_int24x2 d0, ae_int24x2 d1);
```

2.5.2 32x32-bit Multiplication Operations

HiFi 3 supports four 24x24 or 32x16-bit multiplications per cycle, but only two 32x32-bit ones. 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.

AE_MULF32S.LL, AE_MULAF32S.LL, AE_MULSF32S.LL Operations:

AE_MULF32S.LL (.LH .HH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAF32S.LL (.LH .HH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSF32S.LL (.LH .HH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Single 1.31x1.31-bit into 1.63-bit signed MAC with 64-bit saturation:

$$d \leftarrow saturate_{1.63}([d_{1.63} \pm] d0.L_{1.31} \times d1.L_{1.31})$$

Note: In ae_slot2, only AE_MULF32S.LL and AE_MULAF32S.LL are available.

Note: C intrinsics AE_MUL[AS]F32S_HL are provided and implemented through the .LH operations above. C intrinsics with ae_f24x2 input operands are implemented through the above operations. C intrinsics with ae_p24x2s input operand types and ae_q56s accumulator operand types are provided to ensure HiFi 2 code portability and are implemented through the operations above. The HiFi 2 intrinsics that perform 56-bit accumulator saturation (AE_MUL[AS]FS56*) instantiate an additional AE_SATQ56S operation.

C syntax:

```
ae_f64 AE_MULF32S_LL (ae_f32x2 d0, ae_f32x2 d1);
void AE_MULAF32S_LL (ae_f64 d /*inout*/,
                     ae_f32x2 d0, ae_f32x2 d1);
void AE_MULSF32S_LL (ae_f64 d /*inout*/,
                      ae_f32x2 d0, ae_f32x2 d1);
ae_f64 AE_MULF24S_LL (ae_f24x2 d0, ae_f24x2 d1);
void AE_MULAF24S_LL (ae_f64 d /*inout*/,
                     ae_f24x2 d0, ae_f24x2 d1);
void AE_MULSF24S_LL (ae_f64 d /*inout*/,
                      ae_f24x2 d0, ae_f24x2 d1);
ae_q56s AE_MULFP24S_LL (ae_p24x2s d0, ae_p24x2s d1);
void AE_MULAFP24S_LL (ae_q56s q /*inout*/, [TU]
                      ae_p24x2s d0, ae_p24x2s d1);
void AE_MULSFP24S_LL (ae_q56s q /*inout*/,
                      ae_p24x2s d0, ae_p24x2s d1);
void AE_MULAFS56P24S_LL (ae_q56s q /*inout*/,
                         ae_p24x2s d0, ae_p24x2s d1);
void AE_MULSFS56P24S_LL (ae_q56s q /*inout*/,
                         ae_p24x2s d0, ae_p24x2s d1);
```

AE_MUL32.LL, AE_MULA32.LL, AE_MULS32.LL Operations:

AE_MUL32.LL (.LH .HH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULA32.LL (.LH .HH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULS32.LL (.LH .HH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Single 32x32-bit into 64-bit signed integer MAC with no saturation:

$$d \leftarrow [d \pm] d0.L \times d1.L$$

Note: In ae_slot2, only AE_MUL32.LL and AE_MULA32.LL are available.

Note: C intrinsics AE_MUL[AS]32S_HL are provided and implemented through the .LH operations above. C intrinsics with ae_int24x2 input operands are implemented through the above operations. C intrinsics with ae_p24x2s input operand types and ae_q56s accumulator operand types are provided to ensure HiFi 2 code portability and are implemented through the operations above. The HiFi 2 intrinsics that perform 56-bit accumulator saturation instantiate an additional AE_SATQ56S operation.

C syntax:

```
ae_int64 AE_MUL32_LL (ae_int32x2 d0, ae_int32x2 d1);
void AE_MULA32_LL (ae_int64 d /*inout*/,
                   ae_int32x2 d0, ae_int32x2 d1);
void AE_MULS32_LL (ae_int64 d /*inout*/,
                   ae_int32x2 d0, ae_int32x2 d1);
ae_int64 AE_MUL24_LL (ae_int24x2 d0, ae_int24x2 d1);
void AE_MULA24_LL (ae_int64 d /*inout*/,
                   ae_int24x2 d0, ae_int24x2 d1);
void AE_MULS24_LL (ae_int64 d /*inout*/,
                   ae_int24x2 d0, ae_int24x2 d1);
ae_q56s AE_MULP24S_LL (ae_p24x2s d0, ae_p24x2s d1);
void AE_MULAP24S_LL (ae_q56s d /*inout*/,
                   ae_p24x2s d0, ae_p24x2s d1);
void AE_MULSP24S_LL (ae_q56s d /*inout*/,
                   ae_p24x2s d0, ae_p24x2s d1);
void AE_MULAS56P24S_LL (ae_q56s q /*inout*/,
                        ae_p24x2s d0, ae_p24x2s d1);
void AE_MULSS56P24S_LL (ae_q56s q /*inout*/,
                   ae_p24x2s d0, ae_p24x2s d1);
```

AE_MULF32R.LL, AE_MULAF32R.LL, AE_MULSF32R.LL Operations:

AE_MULF32R.LL (.LH .HH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAF32R.LL (.LH .HH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSF32R.LL (.LH .HH)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Single 1.31x1.31-bit into 17.47-bit signed MAC with symmetric (away from 0) rounding of the product:

$$d \leftarrow [d_{17.47} \pm] \mathit{round}_{17.47}(d0.L_{1.31} \times d1.L_{1.31})$$

Note: In ae_slot2, only the LL versions are available.

Note: C intrinsics AE_MUL[AS]F32R_HL and AE_MULF32R_HL are provided and implemented through the .LH operations above.

C syntax:

```
ae_f64 AE_MULF32R_LL (ae_f32x2 d0, ae_f32x2 d1);
void AE_MULAF32R_LL (ae_f64 d /*inout*/,
                     ae_f32x2 d0, ae_f32x2 d1);
void AE_MULSF32R_LL (ae_f64 d /*inout*/,
                     ae_f32x2 d0, ae_f32x2 d1);
```

AE_MUL32U.LL, AE_MULA32U.LL, AE_MULS32U.LL Operations:

AE_MUL32U.LL	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULA32U.LL	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULS32U.LL	d, d0, d1	[ae_slot1, ae2_slot1]

Single 32x32-bit into 64-bit unsigned integer MAC with no saturation:

$$d \leftarrow [d \pm] d0.L_u \times d1.L_u$$

C syntax:

```
ae_int64 AE_MUL32U_LL (ae_int32x2 d0, ae_int32x2 d1);
void AE_MULA32U_LL (ae_int64 d /*inout*/,
                    ae_int32x2 d0, ae_int32x2 d1);
void AE_MULS32U_LL (ae_int64 d /*inout*/,
                    ae_int32x2 d0, ae_int32x2 d1);
```

AE_MULFP32X2RS, AE_MULAFP32X2RS, AE_MULSFP32X2RS Operations:

AE_MULFP32X2RS	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULAFP32X2RS	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULSFP32X2RS	d, d0, d1	[ae_slot1, ae2_slot1]

2-way SIMD 1.31x1.31-bit into 1.31-bit signed MAC with symmetric (away from zero) rounding of the product and 32-bit saturation of the final result:

$$d.H \leftarrow saturate_{1.31}([d.H_{1.31} \pm] round_{1.31}(d0.H_{1.31} \times d1.H_{1.31}))$$

$$d.L \leftarrow saturate_{1.31}([d.L_{1.31} \pm] round_{1.31}(d0.L_{1.31} \times d1.L_{1.31}))$$

C syntax:

```
ae_f32x2 AE_MULFP32X2RS (ae_f32x2 d0, ae_f32x2 d1);
void AE_MULAFP32X2RS (ae_f32x2 d /*inout*/,
                      ae_f32x2 d0, ae_f32x2 d1);
void AE_MULSFP32X2RS (ae_f32x2 d /*inout*/,
                      ae_f32x2 d0, ae_f32x2 d1);
```

AE_MULFP32X2RAS, AE_MULAFP32X2RAS, AE_MULSFP32X2RAS Operations:

AE_MULFP32X2RAS	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULAFP32X2RAS	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULSFP32X2RAS	d, d0, d1	[ae_slot1, ae2_slot1]

2-way SIMD 1.31x1.31 bit into 1.31-bit signed MAC with asymmetric rounding of the product and 32-bit saturation of the final result:

$$d.H \leftarrow saturate_{1.31}([d.H_{1.31} \pm] round^{+\infty}{}_{1.31}(d0.H_{1.31} \times d1.H_{1.31}))$$

$$d.L \leftarrow saturate_{1.31}([d.L_{1.31} \pm] round^{+\infty}{}_{1.31}(d0.L_{1.31} \times d1.L_{1.31}))$$

C syntax:

```
ae_f32x2 AE_MULFP32X2RAS (ae_f32x2 d0, ae_f32x2 d1);
void AE_MULAFP32X2RAS (ae_f32x2 d /*inout*/,
                       ae_f32x2 d0, ae_f32x2 d1);
void AE_MULSFP32X2RAS (ae_f32x2 d /*inout*/,
                       ae_f32x2 d0, ae_f32x2 d1);
```

AE_MULP32X2, AE_MULAP32X2, AE_MULSP32X2 Operations:

AE_MULP32X2	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULAP32X2	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULSP32X2	d, d0, d1	[ae_slot1, ae2_slot1]

2-way SIMD 32x32-bit into 32-bit signed integer MAC with no saturation:

d.H ← [d.H ±] d0.H × d1.H

d.L ← [d.L ±] d0.L × d1.L

C syntax:

```
ae_int32x2 AE_MULP32X2 (ae_int32x2 d0, ae_int32x2 d1);
void AE_MULAP32X2 (ae_int32x2 d /*inout*/,
                   ae_int32x2 d0, ae_int32x2 d1);
void AE_MULSP32X2 (ae_int32x2 d /*inout*/,
                   ae_int32x2 d0, ae_int32x2 d1);
```

2.5.3 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.

AE_MULF32X16.L0, AE_MULAF32X16.L0, AE_MULSF32X16.L0 Operations:

AE_MULF32X16.L0 (.L1 .L2 .L3 .H0 .H1 .H2 .H3)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAF32X16.L0 (.L1 .L2 .L3 .H0 .H1 .H2 .H3)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSF32X16.L0 (.L1 .L2 .L3 .H0 .H1 .H2 .H3)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Single 1.31x1.15-bit into 17.47-bit signed MAC without saturation:

$d \leftarrow [d_{17.47} \pm] d0.L_{1.31} \times d1.0_{1.15}$

C syntax:

```
ae_f64 AE_MULF32X16_L0 (ae_f32x2 d0, ae_f16x4 d1);
void AE_MULAF32X16_L0 (ae_f64 d /*inout*/,
                       ae_f32x2 d0, ae_f16x4 d1);
void AE_MULSF32X16_L0 (ae_f64 d /*inout*/,
                       ae_f32x2 d0, ae_f16x4 d1);
```

AE_MULZAAFD32X16.H1.L0, AE_MULZASFD32X16.H1.L0, AE_MULZSAFD32X16.H1.L0, AE_MULZSSFD32X16.H1.L0, AE_MULAAFD32X16.H1.L0, AE_MULASFD32X16.H1.L0 , AE_MULSAFD32X16.H1.L0] AE_MULSSFD32X16.H1.L0 Operations:

AE_MULZAAFD32X16.H1.L0 (.H3.L2 .H2.L3 .H0.L1)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZASFD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZSAFD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZSSFD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAAFD32X16.H1.L0 (.H3.L2 .H2.L3 .H0.L1)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULASFD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSAFD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSSFD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Dual 1.31x1.15-bit into 17.47-bit signed MAC without saturation:

$$d \leftarrow [d_{17.47}] \pm d0.H_{1.31} \times d1.1_{1.15} \pm d0.L_{1.31} \times d1.0_{1.15}$$

The extra .H3.L2 and .H0.L1 specifiers are meant for computing half of a complex multiplication.

C syntax:

```
ae_f64 AE_MULZAAFD32X16_H1_L0 (ae_f32x2 d0, ae_f16x4 d1);
ae_f64 AE_MULZASFD32X16_H1_L0 (ae_f32x2 d0, ae_f16x4 d1);
ae_f64 AE_MULZSAFD32X16_H1_L0 (ae_f32x2 d0, ae_f16x4 d1);
ae_f64 AE_MULZSSFD32X16_H1_L0 (ae_f32x2 d0, ae_f16x4 d1);

void AE_MULAAFD32X16_H1_L0 (ae_f64 d /*inout*/,
                            ae_f32x2 d0, ae_f16x4 d1);
void AE_MULASFD32X16_H1_L0 (ae_f64 d /*inout*/,
                            ae_f32x2 d0, ae_f16x4 d1);
void AE_MULSAFD32X16_H1_L0 (ae_f64 d /*inout*/,
                            ae_f32x2 d0, ae_f16x4 d1);
void AE_MULSSFD32X16_H1_L0 (ae_f64 d /*inout*/,
                            ae_f32x2 d0, ae_f16x4 d1);
```

AE_MULFD32X16X2.FIR.LL, AE_MULAFD32X16X2.FIR.LL Operations:

AE_MULFD32X16X2.FIR.LL (.HH .HL .LH)	q0, q1, d0, d1, c	[ae2_slot1]
AE_MULAFD32X16X2.FIR.LL (.HH .HL .LH)	q0, q1, d0, d1, c	[ae2_slot1]

Quad 1.31x1.16-bit multiplications into two 17.47-bit signed MAC with operands selected to accelerate FIR computations.

For the .HH version

$$q0 \leftarrow [q0_{17.47} +] d0.H_{1.31} \times c.3_{1.15} + d0.L_{1.31} \times c.2_{1.15}$$
$$q1 \leftarrow [q1_{17.47} +] d0.L_{1.31} \times c.3_{1.15} + d1.H_{1.31} \times c.2_{1.15}$$

For the .HL version

$q0 \leftarrow [q0_{17.47} +] d0.H_{1.31} \times c.1_{1.15} + d0.L_{1.31} \times c.0_{1.15}$
$q1 \leftarrow [q1_{17.47} +] d0.L_{1.31} \times c.1_{1.15} + d1.H_{1.31} \times c.0_{1.15}$

For the .LH version

$q0 \leftarrow [q0_{17.47} +] d0.L_{1.31} \times c.3_{1.15} + d1.H_{1.31} \times c.2_{1.15}$
$q1 \leftarrow [q1_{17.47} +] d1.H_{1.31} \times c.3_{1.15} + d1.L_{1.31} \times c.2_{1.15}$

For the .LL version

$q0 \leftarrow [q0_{17.47} +] d0.L_{1.31} \times c.1_{1.15} + d1.H_{1.31} \times c.0_{1.15}$
$q1 \leftarrow [q1_{17.47} +] d1.H_{1.31} \times c.1_{1.15} + d1.L_{1.31} \times c.0_{1.15}$

C syntax:

```
void AE_MULFD32P16X2_FIR_H (ae_f64 q0 /*out*/, ae_f64 q1 /*out*/,
                            ae_f32x2 d0,ae_f32x2 d1, ae_f16x4 c);
void AE_MULAFD32X16X2_FIR_H(ae_f64 q0 /*inout*/,
                             ae_f64 q1 /* inout*/
                             ae_f32x2 d0,ae_f32x2 d1,ae_f16x4 c);
```

AE_MUL32X16.L0, AE_MULA32X16.L0, AE_MULS32X16.L0 Operations:

AE_MUL32X16.L0 (.L1 .L2 .L3 .H0 .H1 .H2 .H3)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULA32X16.L0 (.L1 .L2 .L3 .H0 .H1 .H2 .H3)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULS32X16.L0 (.L1 .L2 .L3 .H0 .H1 .H2 .H3)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Single 32x16-bit into 64-bit signed MAC without saturation:

$d \leftarrow [d] \pm d0.L \times d1.0$

C syntax:

```
ae_int64 AE_MUL32X16_L0 (ae_int32x2 d0, ae_f16x4 d1);
void AE_MULA32X16_L0 (ae_int64 d /*inout*/,
                      ae_int32x2 d0, ae_f16x4 d1);
void AE_MULS32X16_L0 (ae_int64 d /*inout*/,
                      ae_int32x2 d0, ae_int16x4 d1);
```

AE_MULZAAD32X16.H1.L0, AE_MULZASD32X16.H1.L0, AE_MULZSAD32X16.H1.L0, AE_MULZSSD32X16.H1.L0, AE_MULAAD32X16.H1.L0, AE_MULASD32X16.H1.L0, AE_MULSAD32X16.H1.L0, AE_MULSSD32X16.H1.L0 Operations:

AE_MULZAAD32X16.H1.L0 (.H3.L2 .H2.L3 .H0.L1)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZASD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZSAD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot2, ae2_slot1]
AE_MULZSSD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAAD32X16.H1.L0 (.H3.L2 .H2.L3 .H0.L1)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULASD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSAD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot2, ae2_slot1]
AE_MULSSD32X16.H1.L0 (.H3.L2)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Dual 32x16-bit into 64-bit signed MAC without saturation:

$$d \leftarrow [d] \pm d0.H \times d1.1 \pm d0.L \times d1.0$$

The extra .H3.L2 and .H0.L1 specifiers are meant for computing half of a complex multiplication.

C syntax:

```
ae_int64 AE_MULZAAD32X16_H1_L0 (ae_int32x2 d0, ae_int16x4 d1);
ae_int64 AE_MULZASD32X16_H1_L0 (ae_int32x2 d0, ae_int16x4 d1);
ae_int64 AE_MULZSAD32X16_H1_L0 (ae_int32x2 d0, ae_int16x4 d1);
ae_int64 AE_MULZSSD32X16_H1_L0 (ae_int32x2 d0, ae_int16x4 d1);

void AE_MULAAD32X16_H1_L0 (ae_int64 d /*inout*/,
                           ae_int32x2 d0, ae_int16x4 d1);
void AE_MULASD32X16_H1_L0 (ae_int64 d /*inout*/,
                           ae_int32x2 d0, ae_int16x4 d1);
void AE_MULSAD32X16_H1_L0 (ae_int64 d /*inout*/,
                           ae_int32x2 d0, ae_int16x4 d1);
void AE_MULSSD32X16_H1_L0 (ae_int64 d /*inout*/,
                           ae_int32x2 d0, ae_int16x4 d1);
```

AE_MULFP32X16X2RS.L, AE_MULAFP32X16X2RS.L, AE_MULSFP32X16X2RS.L Operations:

AE_MULFP32X16X2RS.L (.H)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAFP32X16X2RS.L (.H)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSFP32X16X2RS.L (.H)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

2-way SIMD 1.31x1.15-bit into 1.31-bit signed MAC with saturation and symmetric (away from zero) rounding of the product. When the suffix `.H` is specified, the upper two 16-bit elements of d1 are used. When the suffix `.L` is specified, the lower two 16-bit elements are used.

$$d.H \leftarrow saturate_{1.31}([d.H_{1.31} \pm] round_{1.31}(d0.H_{1.31} \times d1.1_{1.15}))$$

$$d.L \leftarrow saturate_{1.31}([d.L_{1.31} \pm] round_{1.31}(d0.L_{1.31} \times d1.0_{1.15}))$$

C syntax:

```
ae_f32x2 AE_MULFP32X16X2RS (ae_f32x2 d0, ae_f16x4 d1);
void AE_MULAFP32X16X2RS (ae_f32x2 d /*inout*/,
                         ae_f32x2 d0, ae_f16x4 d1);
void AE_MULSFP32X16X2RS (ae_f32x2 d /*inout*/,
                         ae_f32x2 d0, ae_f16x4 d1);
```

AE_MULFP32X16X2RAS.L, AE_MULAFP32X16X2RAS.L, AE_MULSFP32X16X2RAS.L Operations:

AE_MULFP32X16X2RAS.L (.H)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAFP32X16X2RAS.L (.H)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSFP32X16X2RAS.L (.H)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

2-way SIMD 1.31x1.15-bit into 1.31-bit signed MAC with saturation and asymmetric rounding of the product. When the suffix `.H` is specified, the upper two 16-bit elements of d1 are used. When the suffix `.L` is specified, the lower two 16-bit elements are used.

$$d.H \leftarrow saturate_{1.31}([d.H_{1.31} \pm] round^{+\infty}{}_{1.31}(d0.H_{1.31} \times d1.1_{1.15}))$$

$$d.L \leftarrow saturate_{1.31}([d.L_{1.31} \pm] round^{+\infty}{}_{1.31}(d0.L_{1.31} \times d1.0_{1.15}))$$

C syntax:

```
ae_f32x2 AE_MULFP32X16X2RAS_L (ae_f32x2 d0, ae_f16x4 d1);
void AE_MULAFP32X16X2RAS_L (ae_f32x2 d /*inout*/,
                            ae_f32x2 d0, ae_f16x4 d1);
void AE_MULSFP32X16X2RAS_L (ae_f32x2 d /*inout*/,
                            ae_f32x2 d0, ae_f16x4 d1);
```

AE_MULP32X16X2.L, AE_MULAP32X16X2.L, AE_MULSP32X16X2.L Operations:

AE_MULP32X16X2.L (.H)	d, d0, d1	[ae2_slot1]
AE_MULAP32X16X2.L (.H)	d, d0, d1	[ae2_slot1]
AE_MULSP32X16X2.L (.H)	d, d0, d1	[ae2_slot1]

2-way SIMD 32x16-bit into 32-bit signed MAC without saturation. When the suffix `.H` is specified, the upper two 16-bit elements of d1 are used. When the suffix `.L` is specified, the lower two 16-bit elements are used.

$$d.H \leftarrow [d.H \pm] d0.H_{1.31} \times d1.1$$

$$d.L \leftarrow [d.L \pm] d0.L_{1.31} \times d1.0$$

C syntax:

```
ae_int32x2 AE_MULP32X16X2_L (ae_int32x2 d0, ae_int16x4 d1);
void AE_MULAP32X16X2_L (ae_int32x2 d /*inout*/,
                        ae_int32x2 d0, ae_int16x4 d1);
void AE_MULSSP32P16X2_L (ae_int32x2 d /*inout*/,
                         ae_int32x2 d0, ae_int16x4 d1);
```

AE_MULFC32X16RAS.L, AE_MULAFC32X16RAS.L Operations:

AE_MULFC32X16RAS.L (.H)	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULAFC32X16RAS.L (.H)	d, d0, d1	[ae_slot1, ae2_slot1]

Complex quad-mac 1.31x1.15-bit into 1.31-bit signed MAC with asymmetric rounding of the product and 32-bit saturation of the final result. When the suffix `.H` is specified, the upper two 16-bit elements of d1 are used. When the suffix `.L` is specified, the lower two 16-bit elements are used.

$$d.H \leftarrow saturate_{1.31}([d.H_{1.31}+] round^{+\infty}{}_{3.31}(d0.H_{1.31} \times d1.1_{1.15} - d0.L_{1.31} \times d1.0_{1.15}))$$

$$d.L \leftarrow saturate_{1.31}([d.L_{1.31} +] round^{+\infty}{}_{3.31}(d0.H_{1.31} \times d1.0_{1.15} + d0.L_{1.31} \times d1.1_{1.15}))$$

C syntax:

```
ae_f32x2 AE_MULFC32X16RAS_L (ae_f32x2 d0, ae_f16x4 d1);
void AE_MULAFC32X16RAS_L (ae_f32x2 d /*inout*/,
                          ae_f32x2 d0, ae_f16x4 d1);
```

AE_MULC32X16.L, AE_MULAC32X16.L Operations:

AE_MULC32X16.L (.H)	d, d0, d1	[ae_slot1, ae2_slot1]
AE_MULAC32X16.L (.H)	d, d0, d1	[ae_slot1, ae2_slot1]

Complex quad-mac 32x16-bit into 32-bit signed integer MAC with no saturation. When the suffix `.H` is specified, the upper two 16-bit elements of d1 are used. When the suffix `.L` is specified, the lower two 16-bit elements are used.

d.H ← [d.H +] d0.H × d1.1 - d0.L × d1.0

d.L ← [d.L +] d0.H × d1.0 + d0.L × d1.1

C syntax:

```
ae_int32x2 AE_MULC32X16_L (ae_int32x2 d0, ae_f16x4 d1);
void AE_MULAC32X16_L (ae_int32x2 d /*inout*/,
                      ae_int32x2 d0, ae_f16x4 d1);
```

2.5.4 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.

AE_MULF16SS.00, AE_MULAF16SS.00, AE_MULSF16SS.00 Operations:

AE_MULF16SS.00 (.33 .22 .32 .21 .31 .30 .10 .20 .11)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAF16SS.00 (.33 .22 .32 .21 .31 .30 .10 .20 .11)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSF16SS.00 (.33 .22 .32 .21 .31 .30 .10 .20 .11)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Single 1.15x1.15-bit into 1.31-bit signed MAC with 32-bit intermediate product and accumulator saturation.

$d_{1.31} \leftarrow saturate_{1.31}([d_{1.31} \pm] saturate_{1.31}(d0.0_{1.15} \times d1.0_{1.15}))$

These MAC operations are bit-exact with the ITU-T L_mul, L_mac and L_msu basic primitives.

Note: In ae_slot1 and ae_slot2, only the .00 versions are available

C syntax:

```
ae_f32x2 AE_MULF16SS_00 (ae_f16x4 d0, ae_f16x4 d1);
void AE_MULAF16SS_00 (ae_f32x2 d /*inout*/,
                      ae_f16x4 d0, ae_f16x4 d1);
void AE_MULSF16SS_00 (ae_f32x2 d /*inout*/,
                      ae_f16x4 d0, ae_f16x4 d1);
```

AE_MULZAAFD16SS.11.00, AE_MULZSSFD16SS.11.00, AE_MULAAFD16SS.11.00, AE_MULSSFD16SS.11.00 Operations:

AE_MULZAAFD16SS.11.00 (.33.22 .13.02)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULZSSFD16SS.11.00 (.33.22 .13.02)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAAFD16SS.11.00 (.33.22 .13.02)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSSFD16SS.11.00 (.33.22 .13.02)	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Dual 1.15x1.15-bit into a single 1.31-bit signed MAC with 32-bit saturation after each product and after each accumulation.

$$tmp \leftarrow saturate_{1.31}([d_{1.31}] \pm saturate_{1.31}(d0.1_{1.15} \times d1.1_{1.15}))$$

$$d_{1.31} \leftarrow saturate_{1.31}(tmp \pm saturate_{1.31}(d0.0_{1.15} \times d1.0_{1.15}))$$

These MAC operations are bit-exact with a pair of ITU-T L_mul, L_mac and L_msu basic primitives.

C syntax:

```
ae_f32x2 AE_MULZAAFD16SS_11_00 (ae_f16x4 d0, ae_f16x4 d1);
ae_f32x2 AE_MULZSSFD16SS_11_00 (ae_f16x4 d0, ae_f16x4 d1);
void AE_MULAAFD16SS_11_00 (ae_f32x2 d /*inout*/,
                           ae_f16x4 d0, ae_f16x4 d1);
void AE_MULSSFD16SS_11_00 (ae_f32x2 d /*inout*/,
                           ae_f16x4 d0, ae_f16x4 d1);
```

AE_MULF16X4SS, AE_MULAF16X4SS, AE_MULSF16X4SS Operations:

AE_MULF16X4SS	d0, d1, d2, d3	[ae2_slot1]
AE_MULAF16X4SS	d0, d1, d2, d3	[ae2_slot1]
AE_MULSF16X4SS	d0, d1, d2, d3	[ae2_slot1]

Four way SIMD 1.15x1.15-bit into 1.31-bit signed MAC with 32-bit intermediate product and accumulator saturation.

$$d0.H \leftarrow saturate_{1.31}([d0.H_{1.31} \pm] saturate_{1.31}(d2.3_{1.15} \times d3.3_{1.15}))$$

$$d0.L \leftarrow saturate_{1.31}([d0.L_{1.31} \pm] saturate_{1.31}(d2.2_{1.15} \times d3.2_{1.15}))$$

$$d1.H \leftarrow saturate_{1.31}([d1.H_{1.31} \pm] saturate_{1.31}(d2.1_{1.15} \times d3.1_{1.15}))$$

$$d1.L \leftarrow saturate_{1.31}([d1.L_{1.31} \pm] saturate_{1.31}(d2.0_{1.15} \times d3.0_{1.15}))$$

These MAC operations are bit-exact with the ITU-T L_mul, L_mac and L_msu basic primitives.

C syntax:

```
void AE_MULF16X4SS (ae_f32x2 d0 /*out*/, ae_f32x2 d1 /*out*/
                    ae_f16x4 d2, ae_f16x4 d3);
void AE_MULAF16X4SS (ae_f32x2 d0 /*inout*/,
                     ae_f32x2 d1 /*inout*/,
                     ae_f16x4 d2, ae_f16x4 d3);
void AE_MULSF16X4SS (ae_f32x2 d0 /*inout*/,
                     ae_f32x2 d1 /*inout*/,
                     ae_f16x4 d2, ae_f16x4 d3);
```

AE_MUL16X4, AE_MULA16X4, AE_MULS16X4 Operations:

AE_MUL16X4	d0, d1, d2, d3	[ae2_slot1]
AE_MULA16X4	d0, d1, d2, d3	[ae2_slot1]
AE_MULS16X4	d0, d1, d2, d3	[ae2_slot1]

Four way SIMD 16x16-bit into 32-bit integer signed MAC without saturation.

d0.H ← [d0.H ±] d2.3 × d3.3

d0.L ← [d0.L ±] d2.2 × d3.2

d1.H ← [d1.H ±] d2..1 × d3.1

d1.L ← [d1.L ±] d2.0 × d3.0

C syntax:

```
void AE_MUL16X4 (ae_int32x2 d0 /*out*/, ae_int32x2 d1 /*out*/
                 ae_int16x4 d2, ae_int16x4 d3);
void AE_MULAA16X4 (ae_int32x2 d0 /*inout*/,
                   ae_int32x2 d1 /*inout*/,
                   ae_int16x4 d2, ae_int16x4 d3);
void AE_MULSS16X4 (ae_int32x2 d0 /*inout*/,
                   ae_int32x2 d1 /*inout*/,
                   ae_int16x4 d2, ae_int16x4 d3);
```

AE_MULFP16X4S Operation:

AE_MULFP16X4S	d, d0, d1	[ae2_slot1]

Four way SIMD multiply 1.15x1.15-bit into 1.15-bit signed multiply with saturation.

d.3 ← $saturate_{1.15}(d0.3_{1.15} \times d1.3_{1.15})$

d.2 ← $saturate_{1.15}(d0.2_{1.15} \times d1.2_{1.15})$

d.1 ← $saturate_{1.15}(d0.1_{1.15} \times d1.1_{1.15})$

d.0 ← $saturate_{1.15}(d0.0_{1.15} \times d1.0_{1.15})$

This operations is bit-exact with the ITU-T mult basic primitives.

C syntax:

```
ae_f16x4  AE_MULFP16X4S (ae_f16x4 d0, ae_f16x4 d1);
```

AE_MULFP16X4RAS Operation:

AE_MULFP16X4RAS	d, d0, d1	[ae2_slot1]

Four way SIMD 1.15x1.15-bit into 1.15-bit signed multiply with saturation and rounding.

$$d.3 \leftarrow saturate_{1.15}(round^{+\infty}_{2.15}(d0.3_{1.15} \times d1.3_{1.15}))$$

$$d.2 \leftarrow saturate_{1.15}(round^{+\infty}_{2.15}(d0.2_{1.15} \times d1.2_{1.15}))$$

$$d.1 \leftarrow saturate_{1.15}(round^{+\infty}_{2.15}(d0.1_{1.15} \times d1.1_{1.15}))$$

$$d.0 \leftarrow saturate_{1.15}(round^{+\infty}_{2.15}(d0.0_{1.15} \times d1.0_{1.15}))$$

The operation is bit-exact with the ITU-T mult_r basic primitives.

C syntax:

```
ae_f16x4 AE_MULFP16X4RAS (ae_f16x4 d0, ae_pf16x4 d1);
```

2.5.5 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.

AE_MULS32F48P16S.LL, AE_MULAS32F48P16S.LL, AE_MULSS32F48P16S.LL Operations:

AE_MULS32F48P16S.LL (.LH .HH)	q, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAS32F48P16S.LL (.LH .HH)	q, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSS32F48P16S.LL (.LH .HH)	q, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Single 1.15x1.15-bit into 1.31-bit signed MAC with 32-bit intermediate product and accumulator saturation. The input 32-bit AE_DR elements are treated as 9.23-bit values and the result is formatted as a 17.47-bit value.

$$q_{17.47} \leftarrow saturate_{1.31}([q_{17.47} \pm] saturate_{1.31}(d0.L[23:8]_{1.15} \times d1.L[23:8]_{1.15}))$$

These MAC operations are bit-exact with the ITU-T L_mul, L_mac and L_msu basic primitives.

Note: C intrinsics AE_MUL[AS]S32F48P16S_HL are provided and implemented through the .LH operations above. C intrinsics with ae_p24x2s input operand types and ae_q56s accumulator operand types are provided to ensure HiFi 2 code portability and are implemented through the operations above.

C syntax:

```
ae_q56s AE_MULS32F48P16S_LL (ae_p24x2s d0, ae_p24x2s d1);
void AE_MULAS32F48P16S_LL (ae_q56s q /*inout*/,
                           ae_p24x2s d0, ae_p24x2s d1);
void AE_MULSS32F48P16S_LL (ae_q56s q /*inout*/,
                           ae_p24x2s d0, ae_p24x2s d1);
ae_q56s AE_MULFS32P16S_LL (ae_p24x2s d0, ae_p24x2s d1);
void AE_MULAFS32P16S_LL (ae_q56s q /*inout*/,
                         ae_p24x2s d0, ae_p24x2s d1);
void AE_MULSSFS32P16S_LL (ae_q56s q /*inout*/,
                          ae_p24x2s d0, ae_p24x2s d1);
```

2.5.6 32x16-bit Legacy Multiplication Operations

HiFi 3 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 and are therefore less efficient than the HiFi 3-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 intrinsics are provided to ensure HiFi 2 code compatibility and are implemented through a sequence of one or more of the multiplication operations described in this section:

```
void AE_MULAFQ32SP16S_H (_L)      (ae_q56s q /* inout */,
  ae_q56s d0, ae_p24x2s d1);
void AE_MULAFQ32SP16U_H (_L) (ae_q56s q /* inout */,
  ae_q56s d0, ae_p24x2s d1);
void AE_MULAQ32SP16S_H (ae_q56s q /* inout */,
                        ae_q56s d0, ae_p24x2s d1);
void AE_MULAQ32SP16U_H (ae_q56s q /* inout */,
                        ae_q56s d0, ae_p24x2s d1);
ae_q56s AE_MULFQ32SP16S_H (_L) (ae_q56s d0, ae_p24x2s d1);
ae_q56s AE_MULFQ32SP16U_H (_L) (ae_q56s d0, ae_p24x2s d1);
ae_q56s AE_MULQ32SP16S_H (ae_q56s d0, ae_p24x2s d1);
ae_q56s AE_MULQ32SP16U_H (ae_q56s d0, ae_p24x2s d1);
void AE_MULSFQ32SP16S_H (_L) (ae_q56s q /* inout */,
  ae_q56s d0, ae_p24x2s d1);
void AE_MULSFQ32SP16U_H (_L) (ae_q56s q /* inout */,
  ae_q56s d0, ae_p24x2s d1);
void AE_MULSQ32SP16S_H (ae_q56s q /* inout */,
                        ae_q56s d0, ae_p24x2s d1);
void AE_MULSQ32SP16U_H (ae_q56s q /* inout */,
```

```
                                    ae_q56s d0, ae_p24x2s d1);

ae_q56s AE_MULZAAFQ32SP16S_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZAAFQ32SP16U_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZAAQ32SP16S_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZAAQ32SP16U_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZASFQ32SP16S_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZASFQ32SP16U_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZASQ32SP16S_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZASQ32SP16U_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZSAFQ32SP16S_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZSAFQ32SP16U_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZSAQ32SP16S_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZSAQ32SP16U_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZSSFQ32SP16S_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZSSFQ32SP16U_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                           ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZSSQ32SP16S_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
ae_q56s AE_MULZSSQ32SP16U_HH (_LH _LL) (ae_q56s q0, ae_p24x2s p0,
                                          ae_q56s q1, ae_p24x2s p1);
```

AE_MULF48Q32SP16S.L, AE_MULAF48Q32SP16S.L, AE_MULSF48Q32SP16S.L Operations:

AE_MULF48Q32SP16S.L	q, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAF48Q32SP16S.L	q, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSF48Q32SP16S.L	q, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Single 1.31x1.15-bit into 17.47-bit signed MAC without saturation:

$$q \leftarrow [q_{17.47} \pm] \; d0[47:16]_{1.31} \times d1[23:8]_{1.15}$$

Note: C intrinsic AE_MUL[AS]F48Q32SP16S.H are provided and implemented through the .L operations above.

C syntax:

```
ae_int64 AE_MULF48Q32SP16S_L (ae_int64 d0, ae_f32x2 d1);
void AE_MULAF48Q32SP16S_L (ae_int64 q /*inout*/,
                           ae_int64 d0, ae_f32x2 d1);
void AE_MULSF48Q32SP16S_L (ae_int64 q /*inout*/,
                           ae_int64 d0, ae_f32x2 d1);
```

AE_MULF48Q32SP16U.L, AE_MULAF48Q32SP16U.L, AE_MULSF48Q32SP16U.L Operations:

AE_MULF48Q32SP16U.L	qd, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAF48Q32SP16U.L	qd, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSF48Q32SP16U.L	qd, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Single 1.31x1.15$_u$-bit into 17.47-bit MAC without saturation. Note that the 32-bit operand is treated as a signed value while the 16-bit operand is treated as an unsigned value.

$$qd \leftarrow [qd_{17.47} \pm] d0[47:16]_{1.31} \times d1[23:8]_{1.15u}$$

C syntax:

```
ae_int64 AE_MULF48Q32SP16U_L (ae_int64 d0, ae_f32x2 d1);
void AE_MULAF48Q32SP16U_L (ae_int64 qd /*inout*/,
                           ae_int64 d0, ae_f32x2 d1);
void AE_MULSF48Q32SP16U_L (ae_int64 qd /*inout*/,
                           ae_int64 d0, ae_f32x2 d1);
```

AE_MULQ32SP16S.L, AE_MULAQ32SP16S.L, AE_MULSQ32SP16S.L Operations:

AE_MULQ32SP16S.L	q, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAQ32SP16S.L	q, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSQ32SP16S.L	q, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Single 32x16-bit into 64-bit signed integer MAC with no saturation:

$$q \leftarrow [q \pm] d0[47:16] \times d1[23:8]$$

C syntax:

```
ae_q56s AE_MULQ32SP16S_L (ae_q56s d0, ae_p24x2s d1);
void AE_MULAQ32SP16S_L (ae_q56s q /*inout*/,
                        ae_q56s d0, ae_p24x2s d1);
void AE_MULSQ32SP16S_L (ae_q56s q /*inout*/,
                        ae_q56s d0, ae_p24x2s d1);
```

AE_MULQ32SP16U.L, AE_MULAQ32SP16U.L, AE_MULSQ32SP16U.L Operations:

AE_MULQ32SP16U.L	qd, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAQ32SP16U.L	qd, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULSQ32SP16U.L	qd, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]

Single 32x16$_u$-bit into 64-bit integer MAC with no saturation. Note that the 32-bit operand is treated as a signed value, while the 16-bit operand is treated as an unsigned value.

qd ← [qd ±] d0[47:16] × d1[23:8]$_u$

C syntax:

```
ae_q56s AE_MULQ32SP16U_L (ae_q56s d0, ae_p24x2s d1);
void AE_MULAQ32SP16U_L (ae_q56s qd /*inout*/,
                        ae_q56s d0, ae_p24x2s d1);
void AE_MULSQ32SP16U_L (ae_q56s qd /*inout*/,
                        ae_q56s d0, ae_p24x2s d1);
```

2.5.7 HiFi 2 EP 32x24-bit Multiplication Operations

HiFi 3 provides a basic set of 32x24-bit MAC operations for efficient execution of HiFi 2 EP target code. The 32-bit input operand comes from bits 47 through 16 of the AE_DR register. The 24-bit input operand comes from the 24 LSBs of the `L` or `H` 32-bit AE_DR elements.

AE_MULFQ32SP24S.L, AE_MULAFQ32SP24S.L, AE_MULSFQ32SP24S.L Operations:

AE_MULFQ32SP24S.L (.H)	q, d0, d1	[ae_slot1, ae_slot2, ae2_slot1]
AE_MULAFQ32SP24S.L (.H)	q, d0, d1	[ae_slot2, ae2_slot1]
AE_MULSFQ32SP24S.L (.H)	q, d0, d1	[ae_slot2,ae2_slot1]

Single 1.31x1.23-bit to 3.47-bit signed MAC without saturation. The 56-bit (2.54) possibly negated product ± d0×d1.[LH] is truncated towards −∞ and sign-extended to 50 bits (3.47). The product is then written or added to the 50 LSBs of q, and the final 50-bit result is sign-extended to a 64-bit (17.47) fixed-point value.

q ← $sext_{17.47}$([$q_{3.47}$ ±] $truncate_{3.47}$(d0[47:16]$_{1.31}$ × d1[23:0]$_{1.23}$))

C syntax:

```
ae_q56s AE_MULFQ32SP24S_L (ae_q56s d0, ae_p24x2s d1);
void AE_MULAFQ32SP24S_L (ae_q56s q /*inout*/,
                         ae_q56s d0, ae_p24x2s d1);
void AE_MULSFQ32SP24S_L (ae_q56s q /*inout*/,
                         ae_q56s d0, ae_p24x2s d1);
```

AE_MULRFQ32SP24S.L, AE_MULARFQ32SP24S.L, AE_MULSRFQ32SP24S.L Operations:

AE_MULRFQ32SP24S.L (.H)	q, d0, d1	[ae_slot2, ae2_slot1]
AE_MULARFQ32SP24S.L (.H)	q, d0, d1	[ae_slot2, ae2_slot1]
AE_MULSRFQ32SP24S.L (.H)	q, d0, d1	[ae_slot1, ae_slot2]

Single 1.31x1.23-bit to 3.31-bit signed MAC with rounding and no saturation. The 56-bit (2.54) product d0×d1.[LH] is asymmetrically rounded, truncated and sign-extended to 34 bits (3.31). The product is then written to, added to or subtracted from q[49:16] (*i.e.*, the 17.47-bit fixed-point value in q is truncated to a 3.31-bit fixed-point value). The final 34-bit (3.31) result is sign-extended and padded with zeros to a 64-bit (17.47) fixed-point value.

$$q \leftarrow sext_{17.47}([q[49:16]_{3.31} \pm] round^{+\infty}_{3.31} (d0[47:16]_{1.31} \times d1[23:0]_{1.23}))$$

C syntax:

```
ae_q56s AE_MULRFQ32SP24S_L (ae_q56s d0, ae_p24x2s d1);
void AE_MULARFQ32SP24S_L (ae_q56s q /*inout*/,
                          ae_q56s d0, ae_p24x2s d1);
void AE_MULSRFQ32SP24S_L (ae_q56s q /*inout*/,
                          ae_q56s d0, ae_p24x2s d1);
```

2.6 Add, Subtract, and Compare Operations

AE_ADD32, AE_SUB32, AE_ADDSUB32, AE_SUBADD32 Operations:

AE_ADD32	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]
AE_SUB32	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]
AE_ADDSUB32	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]
AE_SUBADD32	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Add/subtract 32-bit elements of two AE_DR register d0 and d1 without saturation. The results are placed in `d`. For AE_ADDSUB32 the high half of each register is added together and the low half is subtracted. For AE_SUBADD32 the high half of each register is subtracted and the low half is added together.

$$d.H \leftarrow d0.H \pm d1.H$$

$$d.L \leftarrow d0.L \pm d1.L$$

Note: C intrinsics AE_ADDP24 and AE_SUBP24 are provided to ensure HiFi 2 code portability. They are implemented through operations AE_ADD32 and AE_SUB32, respectively.

C syntax:

```
ae_int32x2 AE_ADD32 (ae_int32x2 d0, ae_int32x2 d1);
ae_int32x2 AE_SUB32 (ae_int32x2 d0, ae_int32x2 d1);
ae_int32x2 AE_ADDSUB32 (ae_int32x2 d0, ae_int32x2 d1);
ae_int32x2 AE_SUBADD32 (ae_int32x2 d0, ae_int32x2 d1);
      ae_p24x2s AE_ADDP24 (ae_p24x2s d0, ae_p24x2s d1);
ae_p24x2s AE_SUBP24 (ae_p24x2s d0, ae_p24x2s d1);
```

AE_ADD32_HL_LH Operation:

AE_ADD32_HL_LH	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Generalized reduction add. Add 32-bit elements of two AE_DR register d0 and d1 without saturation. Add the low half of one register to the high half of the other.

d.H ← d0.H + d1.L

d.L ← d0.L + d1.H

C syntax:

```
ae_int32x2 AE_ADD32_HL_LH (ae_int32x2 d0, ae_int32x2 d1);
```

AE_ADD32S, AE_SUB32S, AE_ADDSUB32S, AE_SUBADD32S Operations:

AE_ADD32S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]
AE_SUB32S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]
AE_ADDSUB32S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]
AE_SUBADD32S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Add/subtract 32-bit elements signed, saturating two AE_DR registers d0 and d1. For AE_ADDSUB32S, the high half of each register is added together and the low half is subtracted. For AE_SUBADD32S, the high half of each register is subtracted and the low half is added together. The results are placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

d.H ← $saturate_{1.31}$(d0.H ± d1.H)

d.L ← $saturate_{1.31}$(d0.L ± d1.L)

C syntax:

```
ae_f32x2 AE_ADD32S (ae_f32x2 d0, ae_f32x2 d1);
ae_f32x2 AE_SUB32S (ae_f32x2 d0, ae_f32x2 d1);
ae_int32x2 AE_ADDSUB32S (ae_int32x2 d0, ae_int32x2 d1);
ae_int32x2 AE_SUBADD32S (ae_int32x2 d0, ae_int32x2 d1);
```

AE_ADD24S, AE_SUB24S Operations:

AE_ADD24S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]
AE_SUB24S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Add/subtract 32-bit elements with 24-bit (9.23) signed saturation of two AE_DR registers d0 and d1. The results are placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

$$d.H \leftarrow sext_{9.23}(saturate_{1.23}(d0.H_{9.23} \pm d1.H_{9.23}))$$

$$d.L \leftarrow sext_{9.23}(saturate_{1.23}(d0.L_{9.23} \pm d1.L_{9.23}))$$

Note: C intrinsics AE_ADDSP24S and AE_SUBSP24S are provided to ensure HiFi 2 code portability. They are implemented through operations AE_ADD24S and AE_SUB24S, respectively.

C syntax:

```
ae_f24x2 AE_ADD24S (ae_f24x2 d0, ae_f24x2 d1);
ae_f24x2 AE_SUB24S (ae_f24x2 d0, ae_f24x2 d1);
ae_p24x2s AE_ADDSP24S (ae_p24x2s d0, ae_int24x2 d1);
ae_p24x2s AE_SUBSP24S (ae_p24x2s d0, ae_int24x2 d1);
```

AE_ADD16, AE_SUB16 Operations:

AE_ADD16	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]
AE_SUB16	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Add/subtract signed 16-bit elements from two AE_DR registers d0 and d1.

C syntax:

```
ae_int16x4 AE_ADD16 (ae_int16x4 d0, ae_int16x4 d1);
ae_int16x4 AE_SUB16 (ae_int16x4 d0, ae_int16x4 d1);
```

AE_ADD16S, AE_SUB16S Operations:

AE_ADD16S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]
AE_SUB16S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Add/subtract signed 16-bit elements, saturating from two AE_DR registers d0 and d1. The results are placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_f16x4 AE_ADD16S (ae_f16x4 d0, ae_f16x4 d1);
ae_f16x4 AE_SUB16S (ae_f16x4 d0, ae_f16x4 d1);
```

AE_NEG32 Operation:

AE_NEG32	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Negate 32-bit elements of AE_DR register d0 without saturation, with result placed in `d`.

d.H ← −d0.H

d.L ← −d0.L

Note: C intrinsic AE_NEGP24 is provided to ensure HiFi 2 code portability. It is implemented through operation AE_NEG32.

C syntax:

```
ae_int32x2 AE_NEG32 (ae_int32x2 d0);
ae_p24x2s AE_NEGP24 (ae_p24x2s d0);
```

AE_NEG32S Operation:

AE_NEG32S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Negate, saturating. 32-bit element of an AE_DR register d0, with result placed in `d`.

d.H ← $saturate_{1.31}$(−d0.H)

d.L ← $saturate_{1.31}$(−d0.L)

C syntax:

```
ae_f32x2 AE_NEG32S (ae_f32x2 d0);
```

AE_NEG24S Operation:

AE_NEG24S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Negate 32-bit element with 24-bit (9.23) saturation of an AE_DR register d0, with result placed in d. In case of saturation, state AE_OVERFLOW is set to 1.

d.H ← $sext_{9.23}$($saturate_{1.23}$(−d0.$H_{9.23}$))

d.L ← $sext_{9.23}$($saturate_{1.23}$(−d0.$L_{9.23}$))

Note: C intrinsic AE_NEGSP24S is provided to ensure HiFi 2 code portability. It is implemented through operation AE_NEG24S.

C syntax:

```
ae_f24x2 AE_NEG24S (ae_f24x2 d0);
ae_p24x2s AE_NEGSP24S (ae_p24x2s d0);
```

AE_NEG16S Operation:

AE_NEG16S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Negate 16-bit, saturating, of an AE_DR register d0, with result placed in `d`.

C syntax:

```
ae_int16 AE_NEG16S (ae_int16 d0);
```

AE_ABS32 Operation:

AE_ABS32	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Absolute value of 32-bit element of an AE_DR register d0 without saturation, with result placed in `d`.

d.H ← |d0.H|

d.L ← |d0.L|

Note: C intrinsic AE_ABSP24 is provided to ensure HiFi 2 code portability. It is implemented through operation AE_ABS32.

C syntax:

```
ae_int32x2 AE_ABS32 (ae_int32x2 d0);
ae_p24x2s AE_ABSP24 (ae_p24x2s d0);
```

AE_ABS32S Operation:

AE_ABS32S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Absolute value, saturating, of a 32-bit element of an AE_DR register d0 with result placed in `d`.

d.H ← $saturate_{1.31}$(|d0.H|)

d.L ← $saturate_{1.31}$(|d0.L|)

C syntax:

```
ae_int32x2 AE_ABS32S (ae_int32x2 d0);
```

AE_ABS24S Operation:

AE_ABS24S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Absolute value, with 24-bit (9.23) saturation of a 32-bit element of an AE_DR register d0 with result placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

d.H ← $sext_{9.23}$($saturate_{1.23}$(|d0.$H_{9.23}$|))

d.L ← $sext_{9.23}$($saturate_{1.23}$(|d0.$L_{9.23}$|))

Note: C intrinsic AE_ABSSP24S is provided to ensure HiFi 2 code portability. It is implemented through operation AE_ABS24S.

C syntax:

```
ae_f24x2 AE_ABS24S (ae_f24x2 d0);
ae_p24x2s AE_ABSSP24S (ae_p24x2s d0);
```

AE_ABS16S Operation:

AE_ABS16S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Absolute value, saturating, element-wise of 16-bit elements of an AE_DR register d0 with result placed in `d`.

C syntax:

```
ae_f16x4 AE_ABS16S (ae_f16x4 d0);
```

AE_MAX32, AE_MIN32 Operations:

AE_MAX32	d, d0, d1	[ae_slot2, ae2_slot0]
AE_MIN32	d, d0, d1	[ae_slot2, ae2_slot0]

Get maximum/minimum of two 32-bit elements of AE_DR registers d0 and d1. The results are placed in `d`.

Maximum: d.H ← (d0.H > d1.H) ? d0.H : d1.H

d.L ← (d0.L > d1.L) ? d0.L : d1.L

Note: C intrinsics AE_MAXP24S and AE_MINP24S are provided to ensure HiFi 2 code portability. They are implemented through operations AE_MAX32 and AE_MIN32, respectively. C intrinsics AE_MAXB32/AE_MINB32 are implemented through a sequence of the AE_MAX32/AE_MIN32 and AE_LT32 operations and set the Boolean result only if the d0 element is greater/less than the d1 element. C intrinsics AE_MAXBP24S/AE_MINBP24S are implemented in a similar way and are provided to ensure HiFi 2 code portability.

C syntax:

```
ae_int32x2 AE_MAX32 (ae_int32x2 d0, ae_int32x2 d1);
ae_int32x2 AE_MIN32 (ae_int32x2 d0, ae_int32x2 d1);
ae_p24x2s AE_MAXP24 (ae_p24x2s d0, ae_p24x2x d1);
ae_p24x2s AE_MINP24 (ae_p24x2s d0, ae_p24x2s d1);
void AE_MAXB32 (ae_int32x2 d /* out */, ae_int32x2 d0,
                ae_int32x2 d1, xtbool2 bhl /* out */);
void AE_MINB32 (ae_int32x2 d /* out */, ae_int32x2 d0,
                ae_int32x2 d1, xtbool2 bhl/* out */);
void AE_MAXBP24S (ae_p24x2s d /* out */, ae_p24x2s d0,
                ae_p24x2s d1, xtbool2 bhl /* out */);
void AE_MINBP24S (ae_p24x2s d /* out */, ae_p24x2s d0,
                ae_p24x2s d1, xtbool2 bhl /* out */);
```

AE_MAXABS32S, AE_MINABS32S Operations:

AE_MAXABS32S	d, d0, d1	[ae_slot2, ae2_slot0, ae2_slot1]
AE_MINABS32S	d, d0, d1	[ae_slot2, ae2_slot0, ae2_slot1]

Get maximum/minimum of absolute value of two signed 32-bit elements of AE_DR registers d0 and d1. The two element-wise results are saturated to 32 bits and placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Maximum: d.H ← $saturate_{1.31}$(|d0.H| > |d1.H| ? |d0.H| : |d1.H|)

d.L ← $saturate_{1.31}$ (|d0.L| > |d1.L| ? |d0.L| : |d1.L|)

Note: C intrinsics AE_MAXBABSSP24S and AE_MINABSSP24S are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_MAXABS32S and AE_MINABS32S.

C syntax:

```
ae_f32x2 AE_MAXABS32S (ae_f32x2 d0, ae_f32x2 d1);
```

ae_f32x2 AE_MINABS32S (ae_f32x2 d0, ae_f32x2 d1);AE_LT32 Operation:

AE_LT32	bhl, d0, d1	[ae_slot2, ae2_slot0]

Compare, signed less-than, two 32-bit elements of AE_DR registers d0 and d1; results go to a pair bhl of adjacent Boolean registers.

bhl[1] ← (d0.H < d1.H) ? 1 : 0

bhl[0] ← (d0.L < d1.L) ? 1 : 0

Note: C intrinsic AE_LTP24S is provided to ensure HiFi 2 code portability. It is implemented through operation AE_LT32.

C syntax:

```
xtbool2 AE_LT32 (ae_int32x2 d0, ae_int32x2 d1);
xtbool2 AE_LTP24S (ae_p24x2s d0, ae_p24x2s d1);
```

AE_LE32 Operation:

AE_LE32	bhl, d0, d1	[ae_slot2, ae2_slot0]

Compare, less-than-or-equal, two 32-bit signed elements of AE_DR registers d0 and d1; results go to a pair bhl of adjacent Boolean registers.

bhl[1] ← (d0.H ≤ d1.H) ? 1 : 0

bhl[0] ← (d0.L ≤ d1.L) ? 1 : 0

Note: C intrinsic AE_LEP24S is provided to ensure HiFi 2 code portability. It is implemented through operation AE_LE32.

C syntax:

```
xtbool2 AE_LE32 (ae_int32x2 d0, ae_int32x2 d1);
xtbool2 AE_LEP24S (ae_p24x2s d0, ae_p24x2s d1);
```

AE_EQ32 Operation:

AE_EQ32	bhl, d0, d1	[ae_slot2, ae2_slot0]

Compare, equal, two 32-bit elements of AE_DR registers d0 and d1; results go to a pair bhl of adjacent Boolean registers.

bhl[1] ← (d0.H == d1.H) ? 1 : 0

bhl[0] ← (d0.L == d1.L) ? 1 : 0

Note: C intrinsic AE_EQP24 is provided to ensure HiFi 2 code portability. It is implemented through operation AE_EQ32.

C syntax:

```
xtbool2 AE_EQ32 (ae_int32x2 d0, ae_int32x2 d1);
xtbool2 AE_EQP24 (ae_p24x2s d0, ae_p24x2s d1);
```

AE_LT16 Operation:

AE_LT16	b321, d0, d1	[ae_slot2, ae2_slot0]

Compare, less-than, two 16-bit signed elements of AE_DR registers d0 and d1; results go to a four element Boolean register.

b3210[3] ← (d0.3 < d1.3) ? 1 : 0
b3210[2] ← (d0.2 < d1.2) ? 1 : 0
b3210[1] ← (d0.1 < d1.1) ? 1 : 0
b3210[0] ← (d0.0 < d1.0) ? 1 : 0

C syntax:

```
xtbool4 AE_LT16 (ae_int16x4 d0, ae_int16x4 d1);
```

AE_LE16 Operation:

AE_LE16	b3210, d0, d1	[ae_slot2, ae2_slot0]

Compare, less-than-or-equal, two 16-bit signed elements of AE_DR registers d0 and d1; results go to a four element Boolean register.

b3210[3] ← (d0.3 <= d1.3) ? 1 : 0

b3210[2] ← (d0.2 <= d1.2) ? 1 : 0

b3210[1] ← (d0.1 <= d1.1) ? 1 : 0

b3210[0] ← (d0.0 <= d1.0) ? 1 : 0

C syntax:

```
xtbool4 AE_LE16 (ae_int16x4 d0, ae_int16x4 d1);
```

AE_EQ16 Operation:

AE_EQ16	b3210, d0, d1	[ae_slot2, ae2_slot0]

Compare, equal, two AE_DR registers d0 and d1; results go to a four element Boolean register.

b321[3] ← (d0.3 == d1.3) ? 1 : 0

b321[2] ← (d0.2 == d1.2) ? 1 : 0

b321[1] ← (d0.1 == d1.1) ? 1 : 0

b321[0] ← (d0.0 == d1.0) ? 1 : 0

C syntax:

```
xtbool4 AE_EQ16 (ae_int16x4 d0, ae_int16x4 d1);
```

AE_ADD64, AE_SUB64 Operations

AE_ADD64	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0]
AE_SUB64	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0]

Add/Subtract two 64-bit AE_DR registers d0 and d1 without saturation, with result placed in `d`.

d ← d0 ± d1

Note: C intrinsics AE_ADDQ56 and AE_SUBQ56 are provided to ensure HiFi 2 code portability. They are implemented through operations AE_ADD64 and AE_SUB64, respectively.

C syntax:

```
ae_int64 AE_ADD64 (ae_int64 d0, ae_int64 d1);
ae_int64 AE_SUB64 (ae_int64 d0, ae_int64 d1);
ae_q56s AE_ADDQ56 (ae_q56s d0, ae_q56s d1);
ae_q56s AE_SUBQ56 (ae_q56s d0, ae_q56s d1);
```

AE_ADD64S, AE_SUB64S Operations:

AE_ADD64S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]
AE_SUB64S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Add/Subtract, saturating, two 64-bit signed AE_DR registers d0 and d1, with result placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

d ← $saturate_{1.63}$(d0 ± d1)

C syntax:

```
ae_f64 AE_ADD64S (ae_f64 d0, ae_f64 d1);
ae_f64 AE_SUB64S (ae_f64 d0, ae_f64 d1);
```

AE_ADDSQ56S, AE_SUBSQ56S Operations:

AE_ADDSQ56S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]
AE_SUBSQ56S	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Add/Subtract (56-bit (9.55) saturation), two 64-bit signed AE_DR registers d0 and d1, with the result placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

$$d \leftarrow sext_{9.55}((saturate_{1.55}(d0_{9.55} \pm d1_{9.55}))$$

Note: These are legacy instructions meant to support HiFi 2 code portability.

C syntax:

```
ae_q56s AE_ADDSQ56S (ae_q56s d0, ae_q56s d1);
ae_q56s AE_SUBSQ56S (ae_q56s d0, ae_q56s d1);
```

AE_NEG64 Operation:

AE_NEG64	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Negate 64-bit AE_DR register d0 without saturation, with result placed in `d`.

$$d \leftarrow -d0$$

Note: C intrinsic AE_NEGQ56 is provided to ensure HiFi 2 code portability. It is implemented through operation AE_NEG64.

C syntax:

```
ae_int64 AE_NEG64 (ae_int64 d0);
ae_q56s AE_NEGQ56 (ae_q56s d0);
```

AE_NEG64S Operation:

AE_NEG64S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Negate, saturating, 64-bit AE_DR register d0, with result placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

$$d \leftarrow saturate_{1.63}(-d0)$$

C syntax:

```
ae_f64 AE_NEG64S (ae_f64 d0);
```

AE_NEGSQ56S Operation:

AE_NEGSQ56S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Negate, with 56-bit (9.55) saturation, 64-bit AE_DR register d0, with result placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

$$d \leftarrow sext_{9.55}(saturate_{1.55}(-d0_{9.55}))$$

Note: These are legacy instructions meant to support HiFi 2 code portability.

C syntax:

```
ae_q56s AE_NEGSQ56S (ae_q56s d0);
```

AE_ABS64 Operation:

AE_ABS64	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Get absolute value of 64-bit AE_DR register d0 without saturation, with result placed in `d`.

$$d \leftarrow |d0|$$

Note: C intrinsic AE_ABSQ56 is provided to ensure HiFi 2 code portability. It is implemented through operation AE_ABS64.

C syntax:

```
ae_int64 AE_ABS64 (ae_int64 d0);
ae_q56s AE_ABSQ56 (ae_q56s d0);
```

AE_ABS64S Operation:

AE_ABS64S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Get absolute value, saturating, of 64-bit AE_DR register d0, with result placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

$$d \leftarrow saturate_{1.63}(|d0|)$$

C syntax:

```
ae_q64 AE_ABS64S (ae_q64 d0);
```

AE_ABSSQ56S Operation:

AE_ABSSQ56S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Get absolute value, with 56-bit (9.55) saturation of 64-bit AE_DR register d0, with result placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

$$d \leftarrow sext_{9.55}((saturate_{1.55}(|d0_{9.55}|))$$

Note: These are legacy instructions meant to support HiFi 2 code portability.

C syntax:

```
ae_q56s AE_ABSSQ56S (ae_q56s d0);
```

AE_MAX64, AE_MIN64 Operations:

AE_MAX64	d, d0, d1	[ae_slot2, ae2_slot0]
AE_MIN64	d, d0, d1	[ae_slot2, ae2_slot0]

Get maximum/minimum of two signed 64-bit AE_DR registers d0 and d1, with result placed in `d`.

Maximum: d ← (d0 > d1) ? d0 : d1

Note: C intrinsics AE_MAXQ56S and AE_MINQ56S are provided to ensure HiFi 2 code portability. They are implemented through operations AE_MAX64 and AE_MIN64, respectively. C intrinsics AE_MAXB64/AE_MINB64 are implemented through a sequence of the AE_MAX64/AE_MIN64 and AE_LT64 operations and set the Boolean result only if the d0 value is greater/less than the d1 value. C intrinsics AE_MAXBQ56S/AE_MINBQ56S are implemented in a similar way and are provided to ensure HiFi 2 code portability.

C syntax:

```
ae_int64 AE_MAX64 (ae_int64 d0, ae_int64 d1);
ae_int64 AE_MIN64 (ae_int64 d0, ae_int64 d1);
ae_q56s AE_MAXQ56S (ae_q56s d0, ae_q56s d1);
ae_q56s AE_MINQ56S (ae_q56s d0, ae_q56s d1);
void AE_MAXB64 (ae_int64 d /* out */, ae_int64 d0, ae_int64 d1,
                xtbool b /* out */);
void AE_MINB64 (ae_int64 d /* out */, ae_int64 d0, ae_int64 d1,
                xtbool b /* out */);
void AE_MAXBQ56S (ae_q56s d /* out */, ae_q56s d0, ae_q56s d1,
                xtbool b /* out */);
void AE_MINBQ56S (ae_q56s d /* out */, ae_q56s d0, ae_q56s d1,
                xtbool b /* out */);
```

AE_MAXABS64S, AE_MINABS64S Operations:

AE_MAXABS64S	d, d0, d1	[ae_slot2, ae2_slot0]
AE_MINABS64S	d, d0, d1	[ae_slot2, ae2_slot0]

Get maximum/minimum of absolute value of two 64-bit signed AE_DR registers d0 and d1. The result is saturated to 64 bits and placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Maximum: d ← $saturate_{1.63}$((|d0| > |d1|) ? |d0| : |d1|)

Note: C intrinsics AE_MAXBSSQ56S and AE_MINABSSQ56S are provided to ensure HiFi 2 EP code portability. They are implemented through operations AE_MAXABS64S and AE_MINABS64S.

C syntax:

```
ae_f64 AE_MAXABS64S (ae_f64 d0, ae_f64 d1);
ae_f64 AE_MINABS64S (ae_f64 d0, ae_f64 d1);
```

AE_LT64 Operation:

AE_LT64	b, d0, d1	[ae_slot2, ae2_slot0]

Compare, less-than, two signed 64-bit AE_DR registers d0 and d1; result goes to a Boolean register b.

b ← (d0 < d1) ? 1 : 0

Note: C intrinsic AE_LTQ56S is provided to ensure HiFi 2 code portability. It is implemented through operation AE_LT64.

C syntax:

```
xtbool AE_LT64 (ae_int64 d0, ae_int64 d1);
xtbool AE_LTQ56S (ae_q56s d0, ae_q56s d1);
```

AE_LE64 Operation:

AE_LE64	b, d0, d1	[ae_slot2, ae2_slot0]

Compare, less-than-or-equal, two 64-bit signed AE_DR registers d0 and d1; result goes to a Boolean register b.

b ← (d0 ≤ d1) ? 1 : 0

Note: C intrinsic AE_LEQ56S is provided to ensure HiFi 2 code portability. It is implemented through operation AE_LE64.

C syntax:

```
xtbool AE_LE64 (ae_int64 d0, ae_int64 d1);
xtbool AE_LEQ56S (ae_q56s d0, ae_q56s d1);
```

AE_EQ64 Operation:

AE_EQ64	b, d0, d1	[ae_slot2, ae2_slot0]

Compare, equal, two 64-bit AE_DR registers d0 and d1; result goes to a Boolean register b.

b ← (d0 == d1) ? 1 : 0

Note: C intrinsic AE_EQQ56 is provided to ensure HiFi 2 code portability. It is implemented through operation AE_EQQ64.

C syntax:

```
xtbool AE_EQ64 (ae_int64 d0, ae_int64 d1);
xtbool AE_EQQ56 (ae_q56s d0, ae_q56s d1);
```

2.7 Shift Operations

HiFi 3 comes with a large variety of shift operations, supporting 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, 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 hence can be issued in parallel with both a load and store and a multiply. Shift instructions using an AR register 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 `R` 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 is `I` for immediate shifts, `A` for AR shifts and `S` for AE_SAR shifts. Following is a number signifying the size of the element being shifted and an optional `R` for right shifts that round rather than truncate, and an optional S for left shifts that saturate.

AE_SRAI16 Operation:

AE_SRAI16	d, d0, i	[ae_slot2, ae2_slot0]

Shift right arithmetic (sign-extending), element-wise, 16-bit elements of AE_DR register d0 by immediate value, with result placed in `d`.

C syntax:

```
ae_int16x4 AE_SRAI16 (ae_int16x4 d0, immediate i);
```

AE_SRAI16R Operation:

AE_SRAI16R	d, d0, i	[ae_slot2, ae2_slot0]

Shift right arithmetic (sign-extending), element-wise, 16-bit elements of AE_DR register d0 by immediate, with result placed in `d`. Result is rounded corresponding to ITU intrinsic shr_r.

C syntax:

```
ae_int16x4 AE_SRAI16R (ae_int16x4 d0, immediate i);
```

AE_SRAA16RS Operation:

AE_SRAA16RS	d, d0, a0	[ae2_slot0, ae_minislot2]

Shift right or left arithmetic (sign-extending), saturating, element-wise, four 16-bit signed elements of AE_DR register d0 by AR register a0, with result placed in `d`. For a positive shift amount, the value is shifted to the right. For a negative shift amount, the value is shifted to the left. When shifted to the right, result is rounded corresponding to ITU intrinsic shr_r. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_f16x4 AE_SRAA16RS (ae_f6x4 d0, int32 a0);
```

AE_SRAA16S Operation:

AE_SRAA16S	d, d0, a0	[ae2_slot0,, ae_minislot2]

Shift right or left arithmetic, (sign-extending), saturating, element-wise, four 16-bit elements of AE_DR register d0 by AR register a0, with result placed in `d`. For a positive shift amount, the value is shifted to the right. For a negative shift amount, the value is shifted to the left. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_f16x4 AE_SRAA16S (ae_f16x4 d0, int32 a0);
```

AE_SLAI16S Operation:

AE_SLAI16S	d, d0, i	[ae_slot2, ae2_slot0]

Shift left arithmetic, saturating, element-wise, four 16-bit signed elements of AE_DR register d0 by immediate value, with result placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_f16x4 AE_SLAI16S (ae_f16x4 d0, immediate i);
```

AE_SLAA16S Operation:

AE_SLAA16S	d, d0, a0	[ae2_slot0, Inst ae_minislot2]

Shift left or right, saturating, element-wise, four 16-bit signed elements of AE_DR register by AR register a0, with result placed in `d`. For a positive shift amount, the value is shifted to the left. For a negative shift amount, the value is shifted to the right and sign-extended. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_f16x4 AE_SLAA16S (ae_f16x4 d0, int32 a);
```

AE_SLAI24 Operation:

AE_SLAI24	d, d0, i	[ae_slot2, ae2_slot0]

Shift left element-wise, two 24-bit elements of AE_DR register d0 by immediate value, with result placed in `d`.

d.L = $sext_{24}$(d0.L[23:0] << i);

d.H = $sext_{24}$(d0.H[23:0] << i).

Note: C intrinsic AE_SLLIP24 is implemented through operation AE_SLAI24.

C syntax:

```
ae_int24x2 AE_SLAI24 (ae_int24x2 d0, immediate i);
ae_p24x2s AE_SLLIP24 (ae_p24x2s d0, immediate i);
```

AE_SRLI24 Operation:

AE_SRLI24	d, d0, i	[ae_slot2, ae2_slot0]

Shift right logical (zero-extending), element-wise, two 24-bit elements of AE_DR register d0 by immediate, with result placed in `d`. Note that the sign of the result will be zero for any non-zero shift amount.

d.L = $sext_{24}$(d0.L[23:0] $>>_u$ i);

d.H = $sext_{24}$(d0.H[23:0] $>>_u$ i).

Note: C intrinsic AE_SRLIP24 is implemented through operation AE_SRLII24.

C syntax:

```
ae_int24x2 AE_SRLI24 (ae_int24x2 d0, immediate i);
ae_p24x2s AE_SRLIP24 (ae_p24x2s d0, immediate i);
```

AE_SRAI24 Operation:

AE_SRAI24	d, d0, i	[ae_slot2, ae2_slot0]

Shift right arithmetic (sign-extending), element-wise, two 24-bit elements of AE_DR register d0 by immediate value, with result placed in `d`.

d.L = $sext_{24}$(d0.L[23:0] $>>_s$ i);

d.H = $sext_{24}$(d0.H[23:0] $>>_s$ i).

Note: C intrinsic AE_SRAIP24 is implemented through operation AE_SRAII24.

C syntax:

```
ae_int24x2 AE_SRAI24 (ae_int24x2 p0, immediate i);
ae_p24x2s AE_SRAIP24 (ae_p24x2s d0, immediate i);
```

AE_SLAI24S Operation:

AE_SLAI24S	d, d0, i	[ae_slot2, ae2_slot0]

Shift left, saturating, element-wise, two 24-bit signed elements of AE_DR register d0 by immediate, with result placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

d.L = $sext_{24}$($saturate_{24}$(d0.L[23:0] << i));

d.H = $sext_{24}$($saturate_{24}$(d0.H[23:0] << i)).

Note: C intrinsic AE_SLLISP24S is implemented through operation AE_SLAI24S.

C syntax:

```
ae_f24x2 AE_SLAI24S (ae_f24x2 d0, immediate i);
ae_p24x2s AE_SLLISP24S (ae_p24x2s d0, immediate i);
```

AE_SLAS24 Operation:

AE_SLAS24	d, d0	[ae_slot2, ae2_slot0]

Shift left or right arithmetic, (sign-extending), element-wise two 24-bit elements of AE_DR register d0 by shift amount register AE_SAR, with result placed in `d`. For a positive shift amount, the value is shifted to the left. For a negative shift amount, the value is shifted to the right and sign-extended. Note that in the case of a negative shift amount, this intrinsic performs an arithmetic right shift.

d.L = $sext_{24}$((SAR ≥ 0) ? (d0.L[23:0] << SAR) : (d0.L[23:0] $>>_s$ −SAR));

d.H = $sext_{24}$((SAR ≥ 0) ? d0.H[23:0] << SAR) : (d0.H[23:0] $>>_s$ −SAR)).

Note: C intrinsic AE_SLLSP24 is implemented through operation AE_SLAS24.

C syntax:

```
ae_int24x2 AE_SLAS24 (ae_int24x2 d0);
ae_p24x2s AE_SLLSP24 (ae_p24x2s d0);
```

AE_SRLS24 Operation:

AE_SRLS24	d, d0	[ae_slot2, ae2_slot0]

Shift right or left, logical (zero-extending), element-wise two 24-bit elements of AE_DR register d0 by shift amount register AE_SAR, with result placed in `d`. For a positive shift amount, the value is shifted to the right. In case of a negative shift amount, the value is shifted to the left.

Note: C intrinsic AE_SRLSP24 is implemented through operation AE_SRLS24.

d.L = $sext_{24}$((SAR ≥ 0) ? (d0.L[23:0] $>>_u$ SAR) : (d0.L[23:0] << −SAR));
d.H = $sext_{24}$((SAR ≥ 0) ? (d0.H[23:0] $>>_u$ SAR) : (d0.H[23:0] << −SAR)).

C syntax:

```
ae_int24x2 AE_SRLS24 (ae_int24x2 d0);
ae_p24x2s AE_SRLSP24 (ae_p24x2s d0);
```

AE_SRAS24 Operation:

AE_SRAS24	d, d0	[ae_slot2, ae2_slot0]

Shift right or left arithmetic (sign-extending), element-wise two 24-bit elements of AE_DR register d0 by shift amount register AE_SAR, with result placed in `d`. For a positive shift amount, the value is shifted to the right. In case of a negative shift amount, the value is shifted to the left.

d.L = $sext_{24}$((SAR ≥ 0) ? (d0.L[23:0] $>>_s$ SAR) : (d0.L[23:0] << −SAR));

d.H = $sext_{24}$((SAR ≥ 0) ? (d0.H[23:0] $>>_s$ SAR) : (d0.H[23:0] << −SAR)).

Note: C intrinsic AE_SRASP24 is implemented through operation AE_SRAS24.

C syntax:

```
ae_int24x2 AE_SRAS24 (ae_int24x2 d0);
ae_p24x2s AE_SRASP24 (ae_p24x2s d0);
```

AE_SLAS24S Operation:

AE_SLAS24S	d, d0	[ae_slot2, ae2_slot0]

Shift left or right, arithmetic (sign-extending), saturating, element-wise, two 24-bit elements of AE_DR register d0 by shift amount register AE_SAR, with result placed in `d`. For a positive shift amount, the value is shifted to the left. In case of a negative shift amount, the value is shifted to the right. In case of saturation, state AE_OVERFLOW is set to 1.

d.L = $sext_{24}$((SAR ≥ 0) ? $saturate_{24}$(d0.L[23:0] << SAR) : (d0.L[23:0] $>>_s$ −SAR));

d.H = $sext_{24}$((SAR ≥ 0) ? $saturate_{24}$(d0.H[23:0] << SAR) : (d0.L[23:0] $>>_s$ −SAR)).

Note: C intrinsic AE_SLLSSP24S is implemented through operation AE_SLAS24S. Note that in the case of a negative shift amount, this intrinsic performs an arithmetic right shift.

C syntax:

```
ae_f24x2 AE_SLAS24S (ae_f24x2 d0);
ae_p24x2s AE_SLLSSP24S (ae_p24x2s d0);
```

AE_SLAI32 Operation:

AE_SLAI32	d, d0, i	[ae_slot2, ae2_slot0]

Shift left, element-wise, two 32-bit elements of AE_DR register d0 by immediate value, with result placed in `d`.

d.L = d0.L << i;

d.H = d0.H << i.

C syntax:

```
ae_int32x2 AE_SLAI32 (ae_int32x2 d0, immediate i);
```

AE_SRLI32 Operation:

AE_SRLI32	d, d0, i	[ae_slot2, ae2_slot0]

Shift right logical (zero-extending), element-wise, two 32-bit elements of AE_DR register d0 by immediate value, with result placed in `d`.

d.L = d0.L $>>_u$ i;

d.H = d0.H $>>_u$ i.

C syntax:

```
ae_int32x2 AE_SRLI32 (ae_int32x2 d0, immediate i);
```

AE_SRAI32 Operation:

AE_SRAI32	d, d0, i	[ae_slot2, ae2_slot0]

Shift right arithmetic (sign-extending), element-wise, two 32-bit elements of AE_DR register d0 by immediate value, with result placed in `d`.

d.L = d0.L $>>_s$ i;

d.H = d0.H $>>_s$ i.

C syntax:

```
ae_int32x2 AE_SRAI32 (ae_int32x2 d0, immediate i);
```

AE_SRAI32R Operation:

AE_SRAI32R	d, d0, i	[ae_slot2, ae2_slot0]

Shift right arithmetic, (sign-extending), element-wise, two 32-bit elements of AE_DR register d0 by immediate, with result placed in `d`. Result is rounded corresponding to ITU intrinsic L_shr_r.

C syntax:

```
ae_int32x2 AE_SRAI32R (ae_int32x2 d0, immediate i);
```

AE_SLAI32S Operation:

AE_SLAI32S	d, d0, i	[ae_slot2, ae2_slot0]

Shift left, saturating, element-wise, two signed 32-bit elements of AE_DR register d0 by immediate value, with result placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

d.L = $saturate_{32}$(d0.L << i);

d.H = $saturate_{32}$(d0.H << i).

C syntax:

```
ae_f32x2 AE_SLAI32S (ae_f32x2 d0, immediate i);
```

AE_SLAA32 Operation:

AE_SLAA32	d, d0, a0	[ae2_slot0, Inst, ae_minislot2]

Shift left or right arithmetic (sign-extending), element-wise, two 32-bit elements of AE_DR register d0 by AR register a0, with result placed in `d`. For a positive shift amount, the value is shifted to the left. In case of a negative shift amount, the value is shifted to the right and sign-extended.

d.L = (a0 ≥ 0) ? (d0.L << a0) : (d0.L $>>_s$ −a0);

d.H = (a0 ≥ 0) ? (d0.H << a0) : (d0.H $>>_s$ −a0).

C syntax:

```
ae_int32x2 AE_SLAA32 (ae_int32x2 d0, int32 sa);
```

AE_SRLA32 Operation:

AE_SRLA32	d, d0, a0	[ae2_slot0]

Shift right or left logical (zero-extending), element-wise, two 32-bit elements of AE_DR register d0 by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. In case of a negative shift amount, the value is shifted to the left.

d.L = (a0 ≥ 0) ? (d0.L $>>_u$ a0) : (d0.L << −a0);

d.H = (a0 ≥ 0) ? (d0.H $>>_u$ a0) : (d0.H << −a0).

C syntax:

```
ae_int32x2 AE_SRLA32 (ae_int32x2 d0, int32 a0);
```

AE_SRAA32 Operation:

AE_SRAA32	d, d0, a0	[ae2_slot0, Inst, ae_minislot2]

Shift right or left arithmetic (sign-extending), element-wise, two 32-bit elements of AE_DR register d0 by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. In case of a negative shift amount, the value is shifted to the left.

d.L = (a0 ≥ 0) ? (d0.L $>>_s$ a0) : (d0.L << −a0);

d.H = (a0 ≥ 0) ? (d0.H $>>_s$ a0) : (d0.H << −a0).

C syntax:

```
ae_int32x2 AE_SRAA32 (ae_int32x2 d0, int32 sa);
```

AE_SLAA32S Operation:

AE_SLAA32S	d, d0, a0	[ae2_slot0, Inst, ae_minislot2]

Shift left or right arithmetic (sign-extending), saturating, element-wise, two 32-bit elements of AE_DR register by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the left. In case of a negative shift amount, the value is shifted to the right and sign-extended. In case of saturation, state AE_OVERFLOW is set to 1.

d.L = (a0 ≥ 0) ? $saturate_{32}$(d0.L << a0) : (d0.L $>>_s$ −a0);

d.H = (a0 ≥ 0) ? $saturate_{32}$(d0.H << a0) : (d0.H $>>_s$ −a0).

C syntax:

```
ae_f32x2 AE_SLAA32S (ae_f32x2 d0, int32 a0);
```

AE_SLAS32 Operation:

AE_SLAS32	d, d0	[ae_slot2, ae2_slot0]

Shift left or right arithmetic (sign-extending), element-wise, two 32-bit elements of AE_DR register d0 by the shift amount register AE_SAR, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. For a negative shift amount, the value is shifted to the right and sign-extended.

d.L = (SAR ≥ 0) ? (d0.L << SAR) : (d0.L $>>_s$ −SAR);

d.H = (SAR ≥ 0) ? (d0.H << SAR) : (d0.H $>>_s$ −SAR).

C syntax:

```
ae_int32x2 AE_SLAS32 (ae_int32x2 d0);
```

AE_SRAA32RS Operation:

AE_SRAA32RS	d, d0, a0	[ae2_slot0, ae_minislot2]

Shift right or left arithmetic (sign-extending), element-wise, 32-bit elements of AE_DR register d0 by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. For a negative shift amount, the value is shifted to the right and rounded corresponding to ITU intrinsic L_shr_r.

C syntax:

```
ae_f32x2 AE_SRAA32RS (ae_f32x2 d0, int32 a0);
```

AE_SRAA32S Operation:

AE_SRAA32S	d, d0, a0	[ae2_slot0, Inst, ae_minislot2]

Shift right arithmetic (sign-extending), saturating, element-wise, two 32-bit elements of AE_DR register d0 by AR register a0, with the result placed in `d` corresponding to ITU intrinsic L_shr.

C syntax:

```
ae_f32x2 AE_SRAAR32S (ae_f32x2 d0, int32 a0);
```

AE_SRLS32 Operation:

AE_SRLS32	d, d0	[ae_slot2, ae2_slot0]

Shift right or left logical (zero-extending), element-wise, two 32-bit elements AE_DR register d0 by the shift amount in register AE_SAR, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. For a negative shift amount, the value is shifted to the left.

d.L = (SAR ≥ 0) ? (d0.L $>>_u$ SAR) : (d0.L << −SAR);

d.H = (SAR ≥ 0) ? (d0.H $>>_u$ SAR) : (d0.H << −SAR).

C syntax:

```
ae_int32x2 AE_SRLS32 (ae_int32x2 d0);
```

AE_SRAS32 Operation:

AE_SRAS32	d, d0	[ae_slot2, ae2_slot0]

Shift right or left arithmetic (sign-extending), element-wise, two 32-bit elements of AE_DR register d0 by the shift amount register AE_SAR, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. In case of a negative shift amount, the value is shifted to the left.

d.L = (SAR ≥ 0) ? (d0.L $>>_s$ SAR) : (d0.L << −SAR);

d.H = (SAR ≥ 0) ? (d0.H $>>_s$ SAR) : (d0.H << −SAR).

C syntax:

```
ae_int32x2 AE_SRAS32 (ae_int32x2 d0);
```

AE_SLAS32S Operation:

AE_SLAS32S	d, d0	[ae_slot2, ae2_slot0]

Shift left or right arithmetic (sign-extending), saturating, element-wise, two 32-bit elements of AE_DR register d0 by the shift amount register AE_SAR, with the result placed in `d`. For a positive shift amount, the value is shifted to the left. For a negative shift amount, the value is shifted to the right and sign-extended. In case of saturation, state AE_OVERFLOW is set to 1.

d.L = (SAR ≥ 0) ? $saturate_{32}$(d0.L << SAR) : (d0.L $>>_s$ −SAR);

d.H = (SAR ≥ 0) ? $saturate_{32}$(d0.H << SAR) : (d0.H $>>_s$ −SAR).

C syntax:

```
ae_f32x2 AE_SLAS32S (ae_int32x2 d0);
```

AE_SLAI64 Operation:

AE_SLAI64	d, d0, i	[ae_slot2, ae2_slot0]

Shift left, 64-bit AE_DR register d0 by immediate value, with the result placed in `d`:

d = d0 << i

Note: C intrinsic AE_CVTQ56P32S_L converts a signed 1.31-bit value in d0.L to a 1.63-bit value in `d`. It is implemented through operation AE_SLAIQ64 with a shift amount of 32.

C syntax:

```
ae_int64 AE_SLAI64 (ae_int64 d0, immediate i);
ae_int64 AE_CVTQ56P32S_L (ae_int32x2 d0);
```

AE_SRLI64 Operation:

AE_SRLI64	d, d0, i	[ae_slot2, ae2_slot0]

Shift right, logical (zero-extending), 64-bit AE_DR register d0 by immediate value, with the result placed in `d`.

d = d0 $>>_u$ i

C syntax:

```
ae_int64 AE_SRLI64 (ae_int64 d0, immediate i);
```

AE_SRAI64 Operation:

AE_SRAI64	d, d0, i	[ae_slot2, ae2_slot0]

Shift right arithmetic (sign-extending), 64-bit AE_DR register d0 by immediate, with result placed in `d`.

$$d = d0 >>_s i$$

Note: C intrinsic AE_SRAIQ56 is provided to ensure HiFi 2 code portability. It is implemented through operation AE_SRAI64.

C syntax:

```
ae_int64 AE_SRAI64 (ae_int64 d0, immediate i);
ae_q56s AE_SRAIQ56 (ae_q56s d0, immediate i);
```

AE_SLAI64S Operation:

AE_SLAI64S	d, d0, i	[ae_slot2, ae2_slot0]

Shift left, saturating, 64-bit AE_DR register d0 by immediate value, with the result placed in `d`. In case of saturation, state AE_OVERFLOW is set to 1.

$$d = saturate_{64}(d0 << i)$$

C syntax:

```
ae_f64 AE_SLAI64S (ae_f64 d0, immediate i);
```

AE_SLAA64 Operation:

AE_SLAA64	d, d0, a0	[ae2_slot0, Inst, ae_minislot2]

Shift left or right arithmetic (sign-extending) 64-bit AE_DR register d0 by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the left. For a negative shift amount, the value is shifted to the right and sign-extended.

$$d = (a0 \geq 0) \ ? \ (d0 << a0) : (d0 >>_s -a0)$$

C syntax:

```
ae_int64 AE_SLAA64 (ae_int64 d0, int32 a0);
```

AE_SRLA64 Operation:

AE_SRLA64	d, d0, a0	[ae2_slot0]

Shift right or left, logical, (zero-extending), 64-bit AE_DR register d0 by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. For a negative shift amount, the value is shifted to the left.

$$d = (a0 \geq 0) \ ? \ (d0 >>_u a0) : (d0 << -a0)$$

C syntax:

```
ae_int64 AE_SRLA64 (ae_int64 d0, int32 a0);
```

AE_SRAA64 Operation:

AE_SRAA64	d, d0, a0	[ae2_slot0, ae_minislot2]

Shift right or left arithmetic (sign-extending) 64-bit AE_DR register d0 by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. For a negative shift amount, the value is shifted to the left.

$$d = (a0 \geq 0) \ ? \ (d0 >>_s a0) : (d0 << -a0)$$

C syntax:

```
ae_int64 AE_SRAA64 (ae_int64 d0, int32 a0);
```

AE_SLAA64S Operation:

AE_SLAA64S	d, d0, a0	[ae2_slot0, ae_minislot2]

Shift left or right, arithmetic (sign-extending), 64-bit AE_DR register d0 by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the left. In case of a negative shift amount, the value is shifted to the right and sign-extended. In case of saturation, state AE_OVERFLOW is set to 1.

$$d = (a0 \geq 0) \ ? \ saturate_{64}(d0 << a0) : (d0 >>_s -a0)$$

C syntax:

```
ae_f64 AE_SLAA64S (ae_f64 d0, int32 a0);
```

AE_SLAS64 Operation:

AE_SLAS64	d, d0	[ae_slot2, ae2_slot0]

Shift left or right arithmetic (sign-extending) the 64-bit AE_DR register d0 by the shift amount register AE_SAR, with the result placed in `d`. For a positive shift amount, the value is shifted to the left. For a negative shift amount, the value is shifted to the right and sign-extended.

$$d = (SAR \geq 0) \ ? \ (d0 << SAR) : (d0 >>_s -SAR)$$

C syntax:

```
ae_int64 AE_SLAS64 (ae_int64 d0);
```

AE_SRLQ64 Operation:

AE_SRLQ64	d, d0	[ae_slot2, ae2_slot0]

Shift right or left, logical (zero-extending) the 64-bit AE_DR register d0 by the shift amount register AE_SAR, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. For a negative shift amount, the value is shifted to the left.

$$d = (SAR \geq 0) \ ? \ (d0 >>_u SAR) : (d0 << -SAR)$$

C syntax:

```
ae_int64 AE_SRLS64 (ae_int64 d0);
```

AE_SRAS64 Operation:

AE_SRAS64	d, d0	[ae_slot2, ae2_slot0]

Shift right or left arithmetic (sign-extending) the 64-bit AE_DR register d0 by the shift amount register AE_SAR, with result placed in `d`. For a positive shift amount, the value is shifted to the right. For a negative shift amount, the value is shifted to the left.

$$d = (SAR \geq 0)\ ?\ (d0 >>_s SAR) : (d0 << -SAR)$$

C syntax:

```
ae_int64 AE_SRAS64 (ae_int64 d0);
```

AE_SLAS64S Operation:

AE_SLAS64S	d, d0	[ae_slot2, ae2_slot0]

Shift left or right, arithmetic (sign-extending), saturating, the 64-bit AE_DR register d0 by the shift amount register AE_SAR, with the result placed in `d`. For positive shift amount, the value is shifted to the left. For a negative shift amount, the value is shifted to the right and sign-extended. In case of saturation, state AE_OVERFLOW is set to 1.

$$d = (SAR \geq 0)\ ?\ saturate_{64}(d0 << SAR) : (d0 >>_s -SAR)$$

C syntax:

```
ae_f64 AE_SLAS64S (ae_f64 d0);
```

AE_SRA64_32 Operation:

AE_SRA64_32	d, s, sa	[ae_slot1, ae2_slot1]

Convert a 1.31 variable s into a 17.47 variable by shifting left by 16-bits, filling in the bottom 16-bits with 0 and sign extending the upper 16-bits. Take the 17.47 variable and shift right using the 4-bit shift amount in AR register sa.

C syntax:

```
ae_int64 AE_SRA64_32 (ae_int32x2 s, uint32 sa);
```

2.8 HiFi 2 Shift Operations

The HiFi 3 ISA provides a set of shift operations for efficient execution of HiFi 2 target code. The 56-bit HiFi shift operations process the 56 LSBs of the AE_DR register and sign-extend the 56-bit result to 64 bits.

AE_SLAISQ56S Operation:

AE_SLAISQ56S	d, d0, i	[ae_slot2, ae2_slot0]

Shift left arithmetic (sign-extending), saturating the 56-bit signed element of AE_DR register d0 by immediate, with the result placed in `d`. For a positive shift amount, the value is shifted to the left. For a negative shift amount, the value is shifted to the right. In case of saturation, state AE_OVERFLOW is set to 1.

$$d = sext_{56}(saturate_{56}(d0[55:0] << i)).$$

Note: C intrinsic AE_SLLISQ56S is implemented through operation AE_SLAISQ56S.

C syntax:

```
ae_q56s AE_SLAISQ56S (ae_q56s d0, immediate i);
ae_q56s AE_SLLISQ56S (ae_q56s d0, immediate i);
```

AE_SLAAQ56 Operation:

AE_SLAAQ56	d, d0, a0	[ae_slot0, ae2_slot0, ae_minislot2]

Shift left or right arithmetic (sign-extending), the 56-bit element of AE_DR register d0 by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the left. In case of a negative shift amount, the value is shifted to the right and sign-extended.

$$d = sext_{56}((a0 \geq 0)\ ?\ (d0[55:0] << a0)\ :\ (d0[55:0] >>_s -a0)).$$

Note: C intrinsics AE_SLAIQ56 and AE_SLLIQ56 are implemented through operation AE_SLAAQ56 by first assigning the immediate shift amount to an AR register. C intrinsic AE_SLLAQ56 is implemented through operation AE_SLAAQ56; note that in the case of a negative shift amount, this intrinsic performs an arithmetic right shift.

C syntax:

```
ae_q56s AE_SLAAQ56 (ae_q56s d0, int32 a0);
ae_q56s AE_SLAIQ56 (ae_q56s d0, immediate i);
ae_q56s AE_SLLAQ56 (ae_q56s d0, ae_int32 sa);
```

AE_SRLAQ56 Operation:

AE_SRLAQ56	d, d0, a0	[ae2_slot0, Inst]

Shift right or left logical (zero-extending) the 56-bit element of AE_DR register d0 by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. For a negative shift amount, the value is shifted to the left.

d = $sext_{56}$((a0 ≥ 0) ? (d0[55:0] $>>_u$ a0) : (d0[55:0] << −a0)).

Note: C intrinsic AE_SRLIQ56 is implemented through operation AE_SRLAQ56 by first assigning the immediate shift amount to an AR register.

C syntax:

```
ae_q56s AE_SRLAQ56 (ae_q56s d0, int32 a0);
ae_q56s AE_SRLIQ56 (ae_q56s d0, immediate i);
```

AE_SRAAQ56 Operation:

AE_SRAAQ56	d, d0, a0	[ae2_slot0, Inst, ae_minislot0]

Shift right or left arithmetic (sign-extending) the 56-bit element of AE_DR register d0 by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. In case of a negative shift amount, the value is shifted to the left.

d = $sext_{56}$((a0 ≥ 0) ? (d0[55:0] $>>_s$ a0) : (d0[55:0] << −a0)).

C syntax:

```
ae_q56s AE_SRAAQ56 (ae_q56s d0, int32 a0);
```

AE_SLAASQ56S Operation:

AE_SLAASQ56S	d, d0, a0	[ae2_slot0, Inst]

Shift left or right arithmetic (sign-extending), saturating the 56-bit element of AE_DR register d0 by AR register a0, with the result placed in `d`. For a positive shift amount, the value is shifted to the left. In case of a negative shift amount, the value is shifted to the right and sign-extended. In case of saturation, state AE_OVERFLOW is set to 1.

d = $sext_{56}$((a0 ≥ 0) ? $saturate_{56}$(d0[55:0] << a0) : (d0[55:0] $>>_s$ −a0)).

Note: C intrinsic AE_SLLASQ56S is implemented through operation AE_SLAASQ56S; note that in the case of a negative shift amount, this intrinsic performs an arithmetic right shift.

C syntax:

```
ae_q56s AE_SLAASQ56S (ae_q56s d0, int32 a0);
ae_q56s AE_SLLASQ56S (ae_q56s d0, int32 a0);
```

AE_SLASQ56 Operation:

AE_SLASQ56	d, d0	[ae_slot2, ae2_slot0]

Shift left or right arithmetic (sign-extending) the 56-bit element of AE_DR register d0 by shift amount register AE_SAR, with result placed in `d`. For a positive shift amount, the value is shifted to the left. In case of a negative shift amount, the value is shifted to the right and sign-extended.

$$d = sext_{56}((SAR \geq 0)\ ?\ (d0[55:0] << SAR) : (d0[55:0] >>_s -SAR)).$$

Note: C intrinsic AE_SLLSQ56 is implemented through operation AE_SLASQ56; note that in the case of a negative shift amount, this intrinsic performs an arithmetic right shift.

C syntax:

```
ae_q56s AE_SLASQ56 (ae_q56s d0);
ae_q56s AE_SLLSQ56 (ae_q56s d0);
```

AE_SRLSQ56 Operation:

AE_SRLSQ56	d, d0	[ae_slot2, ae2_slot0]

Shift right or left logical (zero-extending) the 56-bit element of AE_DR register d0 by shift amount register AE_SAR, with the result placed in `d`. For a positive shift amount, the value is shifted to the left. For a negative shift amount, the value is shifted to the left.

$$d = sext_{56}((SAR \geq 0)\ ?\ (d0[55:0] >>_u SAR) : (d0[55:0] << -SAR)).$$

C syntax:

```
ae_q56s AE_SRLSQ56 (ae_q56s q0);
```

AE_SRASQ56 Operation:

AE_SRASQ56	d, d0	[ae_slot2, ae2_slot0]

Shift right or left arithmetic (sign-extending) the 56-bit element of AE_DR register d0 by shift amount register AE_SAR, with the result placed in `d`. For a positive shift amount, the value is shifted to the right. For a negative shift amount, the value is shifted to the left.

$$d = sext_{56}((SAR \geq 0)\ ?\ (d0[55:0] >>_s SAR) : (d0[55:0] << -SAR)).$$

C syntax:

```
ae_q56s AE_SRASQ56 (ae_q56s d0);
```

AE_SLASSQ56S Operation:

AE_SLASSQ56S	d, d0	[ae_slot2, ae2_slot0]

Shift left or right arithmetic (sign-extending), saturating the 56-bit element of AE_DR register d0 by shift amount register AE_SAR, with the result placed in `d`. For a positive shift amount, the value is shifted to the left. In case of a negative shift amount, the value is shifted to the right and sign-extended. In case of saturation, AE_OVERFLOW is set to 1.

$$d = sext_{56}((SAR \geq 0)\ ?\ saturate_{56}(d0[55:0] << SAR) : (d0[55:0] >>_s SAR)).$$

Note: C intrinsic AE_SLLSSQ56S is implemented through operation AE_SLASSQ56S; note that in the case of a negative shift amount, this intrinsic performs an arithmetic right shift.

C syntax:

```
ae_q56s AE_SLASSQ56S (ae_q56s d0);
ae_q56s AE_SLLSSQ56S (ae_q56s d0);
```

2.9 Normalization

AE_NSA64 Operation:

AE_NSA64	a, d0	[ae_slot0, ae2_slot0]

Calculate the left shift amount that will normalize (maximize the value that can be represented without overflow) the two's complement contents of an AE_DR register and write the amount (in the range of 0 to 63) to AR register `a`. If d0 contains 0 or -1, return 63. To calculate the normalization exponent for a 9.55 fixed-point number, subtract 8 from the result. If the result is negative, a right shift is required for normalization.

Note: C intrinsic AE_NSAQ56S is provided to ensure HiFi 2 code portability. It is implemented by subtracting 8 from the result of operation AE_NSA64.

C syntax:

```
int AE_NSA64 (ae_int64 d0);
int AE_NSAQ56S (ae_q56s d0);
```

AE_NSAZ32.L Operation:

AE_NSAZ32.L	a, d0	[ae_slot2, ae2_slot0]

Calculate the left shift amount that will normalize the two's complement contents of the lower 32 bits of an AE_DR register and write the amount (in the range of 0 to 31) to AR register `a`. If d0 contains 0, return 0.

```
int AE_NSAZ32.L (ae_int32x2 d0);
```

AE_NSAZ16.0 Operation:

AE_NSAZ16.0	a, d0	[ae_slot2, ae2_slot0]

Calculate the left shift amount that will normalize the two's complement contents of the lower 16 bits of an AE_DR register and write the amount (in the range of 0 to 15) to AR register `a`. If d0 contains 0, return 0.

```
int AE_NSAZ16.0 (ae_int16x4 d0);
```

2.10 Divide Step Operation

AE_DIV64D32.L Operation:

AE_DIV64D32.L (.H)	d, d0	[ae_slot2, ae2_slot0]

Perform a 1-bit divide-step operation. Shift left the 64-bit input value d by 1. If the unsigned 32-bit `L` (`H`) element of AE_DR register d0 is greater than the unsigned value in the 32 MSBs of the shifted value, the shifted value is placed in AE_DR register `d`; otherwise, d0.L (d0.H) is subtracted from the 32-bit MSBs of the shifted value, the LSB of the shifted value is set to 1 and the result is placed in AE_DR register `d`.

d = { d[62:31], d[30:1], 1'b0 }, if d0.L $>_u$ d[62:31];

d = { d[62:31] – d0.L, d[30:1], 1'b1 }, otherwise.

Note: This instruction is designed to work only when d >=0, d0.L(.H) > 0 and d <= d0.L(.H)

C syntax:

```
void AE_DIV64D32_L (ae_int64 d, ae_int32x2 d0);
```

2.11 Truncate, Round, Saturate, Convert, and Move Operations

AE_TRUNCA32Q48 Operation:

AE_TRUNCA32Q48	a, d0	[ae_slot2, ae2_slot0]

Truncate a 17.47 AE_DR fraction d0 to a 32-bit (1.31) AR fraction in a. The 16 MSBs and the 16 LSBs of the input 64-bit value are discarded. This operation is provided to ensure HiFi 2 code compatibility.

C syntax:

```
int AE_TRUNCA32Q48 (ae_q56s d0);
```

AE_TRUNCI32X2F64S Operation:

AE_TRUNCI32X2F64S	d, dh, dl, i	[ae2_slot0]

Shift-left two signed 64-bit values from AE_DR registers dh and dl by immediate shift amount i, saturate each shifted value to 64 bits and store the 32 MSBs of each result in the two 32-bit elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_TRUNCI32F64S performs the same operation on a single input AE_DR register and replicates the result in the two 32-bit AE_DR elements. It is implemented through operation AE_TRUNCI32X2F64S.

C syntax:

```
ae_int32x2 AE_TRUNCI32X2F64S (ae_int64 dh, ae_int64 dl, immediate i);
ae_int32x2 AE_TRUNCI32F64S (ae_int64 d0, immediate i);
```

AE_TRUNCA32X2F64S Operation:

AE_TRUNCA32X2F64S	d, dh, dl, a	[ae2_slot0, ae_minislot2]

Shift left or right arithmetic (sign-extending), two signed 64-bit values from AE_DR registers dh and dl by AR register shift amount a; each shifted value is saturated 64 bits and the 32 MSBs of the two results are stored in the two 32-bit elements of AE_DR register `d`. For a positive shift amount, the value is shifted to the left. For a negative shift amount, the value is sign-extended and shifted to the right. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_TRUNCA32F64S performs the same operation on a single input AE_DR register and replicates the result in the two 32-bit AE_DR elements. It is implemented through operation AE_TRUNCA32X2F64S.

C syntax:

```
ae_int32x2 AE_TRUNCA32X2F64S (ae_int64 dh, ae_int64 dl, int a);
ae_int32x2 AE_TRUNCA32F64S (ae_int64 d0, int a);
```

AE_TRUNCI32F64S.L Operation:

AE_TRUNCI32F64S.L	d, d0, d1, i	[ae2_slot0]

Shift left the signed 64-bit AE_DR register value d1 by immediate shift amount i, saturate it to 64 bits and store the 32 MSBs of the result into the `L` element of AE_DR register `d`; store the `L` element of AE_DR register d0 into the `H` element of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_int32x2 AE_TRUNCI32F64S_L (ae_int32x2 d0, ae_int64 d1, immediate
i);
```

AE_TRUNCA32F64S.L Operation:

AE_TRUNCA32F64S.L	d, d0, d1, a	[ae2_slot0, ae_minislot2]

Shift left or right arithmetic (sign-extending), 64-bit AE_DR register value d1 by AR register shift amount a; the shifted value is saturated to 64 bits and the 32 MSBs of the results are stored into the `L` element of AE_DR register `d`; store the `L` element of AE_DR register d0 into the `H` element of AE_DR register `d`. For a positive shift amount, the value is shifted to the left. For a negative shift amount, the value is sign-extended and shifted to the right. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_int32x2 AE_TRUNCISP32X2Q64S_L (ae_int32x2 d0, ae_int64 d1,
immediate i);
```

AE_TRUNCP24Q48X2 Operation:

AE_TRUNCP24Q48X2	d, dh, dl	[ae_slot2, ae2_slot0]

Truncate two 17.47-bit fixed-point fractions from AE_DR registers dh and dl into two 1.23-bit fixed-point fractions, sign-extend them to 9.23-bit values and store in the two 32-bit elements of AE_DR register `d`. This operation is provided to ensure HiFi 2 code compatibility.

Note: C intrinsic AE_TRUNCP24Q48 truncates and replicates a single 17.47-bit fixed-point fraction in AE_DR to two 9.23-bit fixed-point fractions in AE_DR. It is implemented through operation AE_TRUNCP24Q48X2.

C syntax:

```
ae_p24x2s AE_TRUNCP24Q48X2 (ae_q56s dh, ae_q56s dl);
ae_p24x2s AE_TRUNCP24Q48 (ae_q56s d0);
```

AE_TRUNCP24A32X2 Operation:

AE_TRUNCP24A32X2	d, ah, al	[ae_slot2, ae2_slot0]

Truncate and sign-extend two 1.31-bit fixed-point fractions (1.31) from AR registers ah and al to two 9.23-bit fixed-point fraction elements in AE_DR register `d`. This operation is provided to ensure HiFi 2 code compatibility.

C syntax:

```
ae_int24x2 AE_TRUNCP24A32X2 (unsigned ah, unsigned al);
```

AE_TRUNCA16P24S.L Operation:

AE_TRUNCA16P24S.L (.H)	a, d0	[ae_slot2, ae2_slot0]

Truncate a 9.23-bit AE_DR fraction in d0.L (d0.H) to 1.15 bits, sign-extend it to 17.15 bits and store it into AR register `a`. The 8 MSBs and the 8 LSBs of the input value are discarded. This operation is provided to ensure HiFi 2 code compatibility.

C syntax:

```
int AE_TRUNCA16P24S_L (ae_int24x2 d0);
```

AE_TRUNCP16 Operation:

AE_TRUNCP16	d, d0	[ae_slot2, ae2_slot0]

Truncate two 9.23-bit fixed-point fractions in AE_DR register d0 to 1.15-bits, and sign-extend them to 9.23-bit fractions into AE_DR register `d`. The eight MSBs and the eight LSBs of the input value are discarded. The eight LSBs of the result are set to zero. This operation is provided to ensure HiFi 2 code compatibility.

C syntax:

```
ae_int24x2 AE_TRUNCP16 (ae_int24x2 d0);
```

AE_TRUNC16X4F32 Operation:

AE_TRUNC16X4F32	d, dh, dl	[ae_slot2, ae2_slot0]

Truncate the four 1.31-bit fixed-point fractions (1.31) from registers dh and dl to four 1.15-bit fixed-point fraction elements in AE_DR register `d`. This is an intrinsic implemented using the AE_SELI16 instruction.

C syntax:

```
ae_f16x4 AE_TRUNCP16Q32X4 (ae_f32x2 dh, ae_f32x2 dl);
```

AE_TRUNCQ32 Operation:

AE_TRUNCQ32	d, d0	[ae_slot2, ae2_slot0]

Truncate (set to zero) the 16 least significant bits of a 15.47-bit fixed-point number in AE_DR register d0 with the result placed into AE_DR register `d`. This operation is provided to ensure HiFi 2 code compatibility.

C syntax:

```
ae_q56s AE_TRUNCQ32 (ae_q56s d0);
```

AE_ROUND32X2F64SSYM Operation:

AE_ROUND32X2F64SSYM	d, dh, dl	[ae_slot2, ae2_slot0]

Round symmetrically (away from 0), saturate the 1.63-bit values from AE_DR registers dh and dl to 1.31-bit values, and store the results in the two elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUND32F64SSYM is implemented through operation AE_ROUND32X2F64SSYM; it rounds a single input AE_DR value and replicates the result in the two elements of the output AE_DR register.

C syntax:

```
ae_f32x2 AE_ROUND32X2F64SSYM (ae_f64 dh, ae_f64 dl);
ae_f32x2 AE_ROUND32F64SSYM (ae_f64 d0);
```

AE_ROUND32X2F64SASYM Operation:

AE_ROUND32X2F64SASYM	d, dh, dl	[ae_slot2, ae2_slot0]

Round asymmetrically, saturate the 1.63-bit values from AE_DR registers dh and dl to 1.31-bit values, and store the results in the two elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUND32F64SASYM is implemented through operation AE_ROUND32X2F64SASYM; it rounds a single input AE_DR value and replicates the result in the two elements of the output AE_DR register.

C syntax:

```
ae_f32x2 AE_ROUND32X2F64SASYM (ae_f64 dh, ae_f64 dl);
ae_f32x2 AE_ROUND32F64SASYM (ae_f64 d0);
```

AE_ROUNDSP16F24SYM Operation:

AE_ROUNDSP16F24SYM	d, d0	[ae_slot2, ae2_slot0]

Round symmetrically (away from 0), saturate each 9.23-bit element of AE_DR register d0 to a 1.15-bit value, sign-extend it and store the results as 9.23-bit values in the two elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUNDSP16SYM is implemented through operation AE_ROUNDSP16F24SYM and is provided to ensure HiFi 2 code portability.

C syntax:

```
ae_f32x2 AE_ROUNDSP16F24SYM (ae_f32x2 d0);
ae_int24x2s AE_ROUNDSP16SYM (ae_int24x2s d0);
```

AE_ROUNDSP16F24ASYM Operation:

AE_ROUNDSP16F24ASYM	d, d0	[ae_slot2, ae2_slot0]

Round asymmetrically, saturate the two 9.23-bit elements of AE_DR register d0 to 1.15-bit values, sign-extend it and store the results as 9.23-bit values in the two elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUNDSP16ASYM is implemented through operation AE_ROUNDSP16F24ASYM and is provided to ensure HiFi 2 code portability.

C syntax:

```
ae_f32x2 AE_ROUNDSP16F24ASYM (ae_f32x2 d0);
ae_int24x2s AE_ROUNDSP16ASYM (ae_int24x2s d0);
```

AE_ROUND32X2F48SSYM Operation:

AE_ROUND32X2F48SSYM	d, dh, dl	[ae_slot2, ae2_slot0]

Round symmetrically (away from 0), saturate the 17.47-bit values from AE_DR registers dh and dl to 1.31-bit values and stores the results into the two elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUND32F48SSYM is implemented through operation AE_ROUND32X2F48SSYM; it rounds a single input AE_DR value and replicates the result in the two elements of the output AE_DR register.

C syntax:

```
ae_f32x2 AE_ROUND32X2F48SSYM (ae_f64 dh, ae_f64 dl);
ae_f32x2 AE_ROUND32F48SSYM (ae_f64 d0);
```

AE_ROUND32X2F48SASYM Operation:

AE_ROUND32X2F48SASYM	d, dh, dl	[ae_slot2, ae2_slot0]

Round asymmetrically, saturate the 17.47-bit values from AE_DR registers dh and dl to 1.31-bit values, and store the results into the two elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUND32F48SASYM is implemented through operation AE_ROUND32X2F48SASYM; it rounds a single input AE_DR value and replicates the result in the two elements of the output AE_DR register.

C syntax:

```
ae_f32x2 AE_ROUND32X2F48SASYM (ae_f64 dh, ae_f64 dl);
ae_f32x2 AE_ROUND32F48SASYM (ae_f64 d0);
```

AE_ROUND24X2F48SSYM Operation:

AE_ROUND24X2F48SSYM	d, dh, dl	[ae_slot2, ae2_slot0]

Round symmetrically (away from 0), saturate the 17.47-bit values from AE_DR registers dh and dl to 1.23-bit values, sign-extend it and store the results as 9.23-bit values in the two elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUND24F48SSYM is implemented through operation AE_ROUND24X2F48SSYM; it rounds a single input AE_DR value and replicates the result in the two elements of the output AE_DR register.

C syntax:

```
ae_f24x2 AE_ROUND24X2F48SSYM (ae_f64 dh, ae_f64 dl);
ae_f24x2 AE_ROUND24F48SSYM (ae_f64 d0);
```

AE_ROUND24X2F48SASYM Operation:

AE_ROUND24X2F48SASYM	d, dh, dl	[ae_slot2, ae2_slot0]

Round asymmetrically, saturate the 17.47-bit values from AE_DR registers dh and dl to 1.23-bit values, sign-extend it, and store the results as 9.23-bit values in the two elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUND24F48SASYM is implemented through operation AE_ROUND24X2F48SASYM; it rounds a single input AE_DR value and replicates the result in the two elements of the output AE_DR register.

C syntax:

```
ae_f24x2 AE_ROUND24X2F48SASYM (ae_f64 dh, ae_f64 dl);
ae_f24x2 AE_ROUND24F48SASYM (ae_f64 d0);
```

AE_ROUNDSP16Q48X2SYM Operation:

AE_ROUNDSP16Q48X2SYM	d, dh, dl	[ae_slot2, ae2_slot0]

Round symmetrically (away from 0), saturate the 17.47-bit values from AE_DR registers dh and dl to 1.15-bit values, sign-extend it, and store the results as 9.23-bit values in the two elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUNDSP16Q48SYM is implemented through operation AE_ROUNDSP16Q48X2SYM; it rounds a single input AE_DR value and replicates the result in the two elements of the output AE_DR register.

C syntax:

```
ae_f24x2 AE_ROUNDSP16Q48X2SYM (ae_f64 dh, ae_f64 dl);
ae_f24x2 AE_ROUNDSP16Q48SYM (ae_f64 d0);
```

AE_ROUNDSP16Q48X2ASYM Operation:

AE_ROUNDSP16Q48X2ASYM	d, dh, dl	[ae_slot2, ae2_slot0]

Round asymmetrically, saturate the 17.47-bit values from AE_DR registers dh and dl to 1.15-bit values, sign-extend it, and store the results as 9.23-bit values in the two elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUNDSP16Q48ASYM is implemented through operation AE_ROUNDSP16Q48X2ASYM; it rounds a single input AE_DR value and replicates the result in the two elements of the output AE_DR register.

C syntax:

```
ae_f24x2 AE_ROUNDSP16Q48X2ASYM (ae_f64 dh, ae_f64 dl);
ae_f24x2 AE_ROUNDSP16Q48ASYM (ae_f64 d0);
```

AE_ROUND16X4F32SASYM Operation:

AE_ROUND16X4F32SASYM	d, dh, dl	[ae_slot2, ae2_slot0]

Round asymmetrically, saturate the 1.31-bit values from AE_DR registers dh and dl to 1.15-bit values, and store the results in the four elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_f16x4 AE_ROUND16X4F32SASYM (ae_f32x2 dh, ae_f32x2 dl);
```

AE_ROUND16X4F32SSYM Operation:

AE_ROUND16X4F32SSYM	d, dh, dl	[ae_slot2, ae2_slot0]

Round symmetrically, saturate the 1.31-bit values from AE_DR registers dh and dl to 1.15-bit values, and store the results in the four elements of AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_f16x4 AE_ROUND16X4F32SSYM (ae_f32x2 dh, ae_f32x2 dl);
```

AE_ROUNDSQ32F48SYM Operation:

AE_ROUNDSQ32F48SYM	d, d0	[ae_slot2, ae2_slot0]

Round symmetrically (away from 0), saturate the 17.47-bit value from AE_DR register d0 to a 1.31-bit value, sign-extend it and store the result as 17.47-bit value in AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUNDSQ32SYM is implemented through operation AE_ROUNDSQ32F48SYM and is provided to ensure HiFi 2 code portability.

C syntax:

```
ae_f64 AE_ROUNDSQ32F48SYM (ae_f64 d0);
ae_q56s AE_ROUNDSQ32SYM (ae_q56s d0);
```

AE_ROUNDSQ32F48ASYM Operation:

AE_ROUNDSQ32F48ASYM	d, p, i16	[ae_slot2, ae2_slot0]

Round asymmetrically, saturate the 17.47-bit value from AE_DR register d0 to a 1.31-bit value, sign-extend it, and store the result as 17.47-bit value in AE_DR register d. In case of saturation, state AE_OVERFLOW is set to 1.

Note: C intrinsic AE_ROUNDSQ32ASYM is implemented through operation AE_ROUNDSQ32F48ASYM and is provided to ensure HiFi 2 code portability.

C syntax:

```
ae_f64 AE_ROUNDSQ32F48ASYM (ae_f64 d0);
ae_q56s AE_ROUNDSQ32ASYM (ae_q56s d0);
```

AE_S24RA64S.I, AE_S24RA64S.IP, AE_S24RA64S.X Operations:

AE_S24RA64S.I	d, a, i16	[ae_slot0, ae2_slot0, ae_minislot0]
AE_S24RA64S.IP	d, a, i16	[ae_slot0, ae2_slot0]
AE_S24RA64S.X (.XP .XC)	d, a, x	[ae_slot0, ae2_slot0]

Round asymmetrically, saturate the 17.47-bit value from AE_DR register `d` to a 1.23-bit value and store the result to memory in the high 24 bits of a 32-bit bundle. In case of saturation, state AE_OVERFLOW is set to 1. This operation is equivalent to an AE_ROUND24F48SASYM followed by a store.

```
void AE_S24RA64S_I (ae_f64 d, ae_f24 *a, immediate i16);
void AE_S24RA64S_IP (ae_f64 d, ae_f24 *a, immediate i16);
void AE_S24RA64S_X (ae_f64 d, ae_f24 *a, int x);
void AE_S24RA64S_XP (ae_f64 d, ae_f24 *a, int x);
void AE_S24RA64S_XC (ae_f64 d, ae_f24 *a, int x);
```

AE_S32RA64S.I, AE_S32RA64S.IP, AE_S32RA64S.X Operations:

AE_S32RA64S.I	d, a, i16	[ae_sloto, ae2_slot0]
AE_S32RA64S.IP	d, a, i16	[ae_sloto, ae2_slot0, ae_minislot0]
AE_S32RA64S.X (.XP .XC)	d, a, x	[ae_sloto, ae2_slot0]

Round asymmetrically, saturate the 17.47-bit value from AE_DR register d to a 1.31-bit value, and store the result to memory. In case of saturation, state AE_OVERFLOW is set to 1. This operation is equivalent to an AE_ROUNDSQ32F48SYM followed by a store.

```
void AE_S32RA64S_I (ae_f64 d, ae_f32 *a, immediate i16);
void AE_S32RA64S_IP (ae_f64 d, ae_f32 *a, immediate i16);
void AE_S32RA64S_X (ae_f64 d, ae_f32 *a, int32 x);
void AE_S32RA64S_XP (ae_f64 d, ae_f32 *a, int32 x);
void AE_S32RA64S_XC (ae_f64 d, ae_f32 *a, int32 x);
```

AE_S24X2RA64S.IP Operation:

AE_S24X2RA64S.IP	d0, d1, a	[ae_slot2, ae2_slot0]

Round asymmetrically, saturate the two 17.47-bit values from AE_DR registers d0 and d1 to 1.23-bit values, and store the results to memory in the high 24 bits of two 32-bit bundles. In case of saturation, state AE_OVERFLOW is set to 1. This operation is equivalent to an AE_ROUND24F48SASYM followed by a store.

```
void AE_S24X2RA64S_IP(ae_f64 d0, ae_f64 d1, ae_f24x2 *a);
```

AE_S32X2RA64S.IP Operation:

AE_S32X2RA64S.IP	d0, d1, a	[ae_slot2, ae2_slot0]

Round asymmetrically, saturate the two 17.47-bit values from AE_DR registers d0 and d1 to 1.31-bit values, and store the result to memory. In case of saturation, state AE_OVERFLOW is set to 1. This operation is equivalent to an AE_ROUND32X2F48SASYM followed by a store.

```
void AE_S32X2RA64S_IP(ae_f64 d0, ae_f64 d1, ae_int32x2 *a);
```

AE_SATQ56S Operation:

AE_SATQ56S	d, d0	[ae_slot2, ae2_slot0]

Saturate the 9.55-bit value in AE_DR register d0 to a 1.55-bit value, sign-extend it, and store the result as a 9.55-bit value in AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_f64 AE_SATQ56S (ae_f64 d0);
```

AE_SAT48S Operation:

AE_SAT48S	d, d0	[ae_slot2, ae2_slot0]

Saturate the 17.47-bit value in AE_DR register d0 to a 1.47-bit value, sign-extend it, and store the result as a 17.47-bit value in AE_DR register `d`. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_f64 AE_SAT48S (ae_f64 d0);
ae_q56s AE_SATQ48S (ae_q56s d0);
```

AE_SAT24S Operation:

AE_SAT24S	d, d0	[ae_slot2, ae2_slot0]

Saturate the two 17.23 values in AE_DR register d0 into 1.23 values, and sign extend into 17.23. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_f24x2 AE_SAT24S (ae_int32x2 d0);
```

AE_SAT16X4 Operation:

AE_SAT16X4	d, d0,d1	[ae_slot2, ae2_slot0]

Saturate the four 32-bit integral values in AE_DR registers d0 and d1 to a 16-bit integral value. In case of saturation, state AE_OVERFLOW is set to 1.

C syntax:

```
ae_int16x4 AE_SAT16X4 (ae_int32x2 d0, ae_int32x2 d1);
```

AE_SEXT32 Operation:

AE_SEXT32	d, d0, i	

Sign-extend (SIMD). Takes the contents of each 32-bit element of register d0 and replicates the bit specified by its immediate operand (in the range 7 to 22) to the high bits and writes the results to register `d`.

C syntax:

```
ae_int32x2 AE_SEXT32 (ae_int32x2 d0, immediate i);
```

AE_SEXT32X2D16.32 Operation:

AE_SEXT32X2D16.32 {.10}	d, d0	[ae_slot0, ae_slot2, ae2_slot0]

Promote the two higher (or lower) 16-bit elements from register d0 and place into the lower 16-bit elements of each pair of AE_DR register `d`. The remaining upper 16 bits of each half are sign extended. These correspond to ITU intrinsics L_deposit_l.

C syntax:

```
ae_int32x2 AE_SEXT32X2D16_32(ae_int16x4 d);
```

AE_CVTP24A16X2.LL Operation:

AE_CVTP24A16X2.LL (.LH, .HL. HH)	d, ah, al	[ae_slot2, ae2_slot0]

Sign-extend and copy the 16 most (.HL, .HH) or least (.LL, .LH) significant bits from the AR register ah into the 24 most significant bits of d.H, and the 16 most (.LH, .HH) or least (.LL, .HL) significant bits from the AR register al into the 24 most significant bits of d.L. In other words, convert 1.15-bit values in AR to 9.23-bit values in AE_DR.

Note: C intrinsic AE_CVTP24A16X2 is equivalent to and implemented through operation AE_CVTP24A16X2.LL. C intrinsic AE_CVTP24A16 sign-extends and replicates the 16 least significant bits from an AR register into the 24 most significant bits of both elements of an AE_DR register. It is implemented through operation AE_CVTP24A16X2.LL.

C syntax:

```
ae_int24x2 AE_CVTP24A16X2_LL (unsigned ah, unsigned al);
ae_int24x2 AE_CVTP24A16X2 (unsigned ah, unsigned al);
ae_int24x2 AE_CVTP24A16 (unsigned a);
```

AE_CVT64A32 Operation:

AE_CVT64A32	d, a	[ae_slot2, ae2_slot0, ae_minislot2]

Convert a signed 1.31-bit value in AR register `a` to 1.63-bit value in AE_DR register `d`.

C syntax:

```
ae_f64 AE_CVT64A32 (unsigned a);
```

AE_CVTQ56A32 Operation:

AE_CVTQ56A32	d, a	[ae_slot2, ae2_slot0, ae_minslot2]

Convert a signed 1.31-bit value in an AR register `a` to a 9.55-bit value in AE_DR register `d`.

C syntax:

```
ae_q56s AE_CVTQ56A32S (unsigned a);
```

AE_CVT48A32 Operation:

AE_CVT48A32	d, a	[ae_slot2, ae2_slot0, ae_minislot2]

Convert a signed 1.31-bit value in an AR register `a` to a 17.47-bit value in AE_DR register `d`.

C syntax:

```
ae_f64 AE_CVT48A32 (unsigned a);
```

AE_CVT64F32.H Operation:

AE_CVT64F32.H	d, d0	[ae_slot2, ae2_slot0]

Convert a signed 1.31-bit value in d0.H to a 1.63-bit value in `d`.

C syntax:

```
ae_f64 AE_CVT64F32_H (ae_int32x2 d0);
ae_f64 AE_CVT64F32_L(ae_int32x2 p0);
```

AE_CVT56F32.L Operation:

AE_CVT56F32.L (.H)	d, d0	[ae_slot2, ae2_slot0]

Convert a signed 1.31-bit value in d0.L (d0.H) to a 9.55-bit value in `d`.

Note: C intrinsic AE_CVTQ48P24S_L (_H) is provided to ensure HiFi 2 code portability. It is implemented through operation AE_CVT56F32.L (.H).

C syntax:

```
ae_f64 AE_CVT56F32_L (ae_int32x2 d0);
ae_q56s AE_CVTQ48P24S_L (ae_p24x2s d0);
```

AE_CVTA32F24S.L Operation:

AE_CVTA32F24S.L (.H)	a, d0	[ae_slot2, ae2_slot0]

Convert a 9.23-bit value in d0.L (d0.H) to a 1.31-bit value in AR register `a`. The 8 MSBs of the input value are discarded.

C syntax:

```
int AE_CVTA32F24S_L (ae_int24x2 d0);
```

AE_CVT16X4 Operation:

AE_CVT16X4	d, dh, dl	[ae_slot2, ae2_slot0]

Convert/truncate the lower 16-bit elements from four 32-bit signed elements of registers dh and dl into four 16-bit integer elements in AE_DR register `d`.

C syntax:

```
ae_int16x4 AE_CVT16X4 (ae_int32x2 dh, ae_int32x2 dl);
```

AE_CVT32X2F16.32 Operation:

AE_CVT32X2F16.32 {.10 }	d, d0	[ae_slot0, ae_slot2, ae2_slot0]

Promote the two higher (or lower) 16-bit elements from register d0 and place into the higher 16-bit elements of each pair in AE_DR register `d`. The remaining lower 16 bits of each half of register d are filled with 0.

These correspond to ITU intrinsics L_deposit_h.

C syntax:

```
ae_f32x2 AE_CVT32X2F16_32(ae_f16x4 d);
```

AE_PKSR32 Operation:

AE_PKSR32	d, d0, i	[ae_slot1, ae2_slot1]

Move the low 32 bits of d into the high 32 bits of `d`. Sign extend the 17.47-bit value in d0 by three bits so that it becomes a 20.47-bit value. Logical Left Shift the result by 0 to 3 bits as encoded in the 2-bit immediate. Round the result, using an asymmetric round, to a 20.31-bit value, removing 16 bits from the low end. Saturate the result to a 1.31-bit value, removing 19 bits from the top. If saturation is needed, state AE_OVERFLOW is set to 1. Store the result in the low 32 bits of `d`. This instruction is useful for the acceleration of biquad filters.

C syntax:

```
void AE_PKSR32 (ae_f32x2 d /*inout*/, ae_f64 d0, immediate i);
```

AE_PKSR24 Operation:

AE_PKSR24	d, d0, i	[ae_slot1, ae2_slot1]

Move the low 32 bits of d into the high 32 bits of `d`. Sign extend the 17.47-bit value in d0 by three bits so that it becomes a 20.47-bit value. Logical Left Shift the result by 0 to 3 bits as encoded in the 2-bit immediate. Round the result, using an asymmetric round, to a 20.23-bit value, removing 24 bits from the low end. Saturate the result to a 1.23-bit value, removing 19 bits from the top. If saturation is needed, state AE_OVERFLOW is set to 1. Sign extend the result by 8 bits to a 9.23-bit value. Store the result in the low 32 bits of `d`. This instruction is useful for the acceleration of biquad filters.

C syntax:

```
void AE_PKSR24 (ae_f24x2 d /*inout*/, ae_f64 d0, immediate i );
```

AE_MOV Operation:

AE_MOVI	d, i	[ae_slot0, ae_slot1, ae2_slot0, ae2_slot1, ae_minislot0, ae_minislot2]

Copy and replicate the immediate (from -16 to 47) into the two halves of `d`.

C syntax:

```
ae_int32x2 AE_MOVI (immediate i);
```

AE_MOVDA32X2 Operation:

AE_MOVDA32X2	d, ah, al	[ae_slot0, ae2_slot0]

Copy the 32-bit contents of each of two AR registers, ah and al, into the two 32-bit elements of an AE_DR register `d`.

Note: AE_MOVPA24X2 and AE_MOVPA24 are provided to ensure HiFi 2 code portability.

C syntax:

```
ae_int32x2 AE_MOVDA32X2 (unsigned ah, unsigned al);
ae_p24x2s AE_MOVPA24X2 (unsigned ah, unsigned al);
ae_p24x2s AE_MOVPA24 (unsigned a);
```

AE_MOVDA32 Operation:

AE_MOVDA32	d, a	[ae_slot0, ae2_slot0, ae_minislot0, ae_minislot2]

Copy and replicate the 32-bit contents of AR register `a`, into the two 32-bit elements of an AE_DR register `d`.

C syntax:

```
ae_int32 AE_MOVDA32 (unsigned a);
```

AE_MOVDA16 Operation:

AE_MOVDA16	d, a	[ae2_slot0, ae_minislot0, ae_minislot2]

Copy the 16-bit contents of a into each of the four 16-bit elements of an AE_DR register d.

C syntax:

```
ae_int16x4 AE_MOVDA16 (unsigned a);
```

AE_MOVDA16X2 Operation:

AE_MOVDA16X2	d, a0, a1	[ae2_slot0]

Combine the 16-bit contents of a0 and a1 and copy into each of the two 32-bit elements of an AE_DR register d.

C syntax:

```
ae_int16x4 AE_MOVDA16X2 (unsigned a0, unsigned a1);
```

AE_MOVAD32.L Operation:

AE_MOVAD32.L (.H)	a, d0	[ae_slot0, ae2_slot0, ae_minislot2]

Copy the 32-bit contents of d0.L (d0.H) to an AR register a.

Note: C intrinsic AE_TRUNCA32Q64 is implemented through operation AE_MOVAD32.H. C intrinsic AE_MOVAP24S_L (_H) is implemented through operation AE_MOVAD32_L (_H) and is provided to ensure HiFi 2 code portability.

C syntax:

```
int AE_MOVAD32_L (ae_int32x2 d0);
int AE_MOVAP24S_L (ae_p24x2s d0);
int AE_TRUNCA32Q64 (ae_int64 d0);
```

AE_MOVAD16.0 Operation:

AE_MOVAD16.0 (.1, .2, .3)	a, d0	[ae2_slot0, ae_minislot2]

Copy and sign-extend the 16-bit contents of d0.0 (d0.1, d0.2, d0.3) to an AR register a.

C syntax:

```
int AE_MOVAD16_0 (ae_int16x4 d0);
```

AE_MOV Operation:

AE_MOV	d, d0	[ae_slot0, ae_slot1, ae_slot2, ae2_slot0, ae2_slot1, Inst]

Copy the 64-bit contents of AE_DR register d0 to AE_DR register d.

Note: C intrinsic AE_MOV32X2 (operating on C type ae_int32x2) is implemented through operation AE_MOV. C intrinsics AE_MOVQ56 (operating on C type ae_q56s) and AE_MOVP48 (operating on C type ae_int24x2) are implemented through operation AE_MOV and are provided to ensure HiFi 2 code portability.

C syntax:

```
ae_int64 AE_MOV64 (ae_int64 d0);
ae_int32x2 AE_MOV32X2 (ae_int32x2 d0);
ae_q56s AE_MOVQ56 (ae_q56s d0);
ae_p24x2s AE_MOVP48 (ae_p24x2s d0);
```

AE_MOVT32X2 Operation:

AE_MOVT32X2	d, d0, bhl	[ae_slot2, ae2_slot0]

If bhl[0] is set, copy the contents of d0.L to d.L;

if bhl[1] is set, copy the contents of d0.H to d.H.

Note: C intrinsic AE_MOVTP24X2 is implemented through operation AE_MOVT32X2 and is provided to ensure HiFi 2 code portability.

C syntax:

```
void AE_MOVT32X2 (ae_int32x2 d /*inout*/, ae_int32x2 d0,xtbool2 bhl);
void AE_MOVTP24X2 (ae_p24x2s d /*inout*/, ae_p24x2s d0,xtbool2 bhl);
```

AE_MOVF32X2 Operation:

AE_MOVF32X2	d, d0, bhl	[ae_slot2, ae2_slot0]

If bhl[0] is clear, copy the contents of d0.L to d.L;

if bhl[1] is clear, copy the contents of d0.H to d.H.

Note: C intrinsic AE_MOVFP24X2 is implemented through operation AE_MOVF32X2 and is provided to ensure HiFi 2 code portability.

C syntax:

```
void AE_MOVF32X2 (ae_int32x2 d /*inout*/, ae_int32x2 d0,xtbool2 bhl);
void AE_MOVFP24X2 (ae_p24x2s d /*inout*/, ae_p24x2s d0,xtbool2 bhl);
```

AE_MOVT16X4 Operation:

AE_MOVT16X4	d, d0, b3210	[ae_slot2, ae2_slot0]

If b3210[0] is set, copy the contents of d0.0 to d.0;

if b3210[1] is set, copy the contents of d0.1 to d.1.

If b3210[2] is set, copy the contents of d0.2 to d.2;

If b3210[3] is set, copy the contents of d0.3 to d.3.

C syntax:

```
void AE_MOVT16X4 (ae_int16x4 d /*inout*/, ae_int16x4 d0, xtbool4
b3210);
```

AE_MOVF16X4 Operation:

AE_MOVF16X4	d, d0, b3210	[ae_slot2, ae2_slot0]

If b3210[0] is clear, copy the contents of d0.0 to d.0;

if b3210[1] is clear, copy the contents of d0.1 to d.1.

If b3210[2] is clear, copy the contents of d0.2 to d.2;

if b3210[3] is clear, copy the contents of d0.3 to d.3.

C syntax:

```
void AE_MOVF16X4 (ae_int16x4 d /*inout*/, ae_int16x4 d0, xtbool4
b3210);
```

AE_MOVT64 Operation:

AE_MOVT64	d, d0, b	[ae_slot2, ae2_slot0]

If b is set, copy the contents of d0 to d.

Note: C intrinsics AE_MOVTQ56 and AE_MOVTP48 are implemented through operation AE_MOVT64 and are provided to ensure HiFi 2 code portability.

C syntax:

```
void AE_MOVT64 (ae_int64 d /*inout*/, ae_int64 d0, xtbool b);
void AE_MOVTQ56 (ae_q56s d /*inout*/, ae_q56s d0, xtbool b);
void AE_MOVTP48 (ae_p24x2s d /*inout*/, ae_p24x2s d0, xtbool b);
```

AE_MOVF64 Operation:

AE_MOVF64	d, d0, b	[ae_slot2, ae2_slot0]

If b is clear, copy the contents of d0 to d.

Note: C intrinsics AE_MOVFQ56 and AE_MOVFP48 are implemented through operation AE_MOVF64 and are provided to ensure HiFi 2 code portability.

C syntax:

```
void AE_MOVF64 (ae_int64 d /*inout*/, ae_int64 d0, xtbool b);
void AE_MOVFQ56 (ae_q56s d /*inout*/, ae_q56s d0, xtbool b);
void AE_MOVFP48 (ae_p24x2s d /*inout*/, ae_p24x2s d0, xtbool b);
```

AE_MOVALIGN Operation:

AE_MOVALIGN	u,v	[ae_slot0, ae2_slot0]

Copy the 64-bit contents of AE_VALIGN register v to AE_VALIGN register u.

C syntax:

```
ae_valign u =  AE_MOVALIGN (ae_valign v);
```

2.12 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.

AE_SEL32.LL Operation:

AE_SEL32.LL	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

d.H = d0.L;

d.L = d1.L.

Note: C intrinsic AE_SELP24_LL is implemented through proto AE_SEL32.LL and is provided to ensure HiFi 2 code portability.

C syntax:

```
ae_int32x2 AE_SEL32_LL (ae_int32x2 d0, ae_int32x2 d1);
ae_p24x2s AE_SELP24_LL (ae_p24x2s d0, ae_p24x2s d1);
```

AE_SEL32.LH Operation:

AE_SEL32.LH	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

d.H = d0.L;

d.L = d1.H.

Note: C intrinsic AE_SELP24_LH is implemented through operation AE_SEL32.LH and is provided to ensure HiFi 2 code portability.

C syntax:

```
ae_int32x2 AE_SEL32_LH (ae_int32x2 d0, ae_int32x2 d1);
ae_p24x2s AE_SELP24_LH (ae_p24x2s d0, ae_p24x2s d1);
```

AE_SEL32.HL Operation:

AE_SEL32.HL	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

d.H = d0.H;

d.L = d1.L.

Note: C intrinsic AE_SELP24_HL is implemented through operation AE_SEL32.HL and is provided to ensure HiFi 2 code portability.

C syntax:

```
ae_int32x2 AE_SEL32_HL (ae_int32x2 d0, ae_int32x2 d1);
ae_p24x2s AE_SELP24_HL (ae_p24x2s d0, ae_p24x2s d1);
```

AE_SEL32.HH Operation:

AE_SEL32.HH	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

d.H = d0.H;

d.L = d1.H.

Note: C intrinsic AE_SELP24_HH is implemented through operation AE_SEL32.HH and is provided to ensure HiFi 2 code portability.

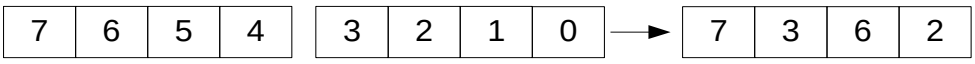
C syntax:

```
ae_int32x2 AE_SEL32_HH (ae_int32x2 d0, ae_int32x2 d1);
ae_p24x2s AE_SELP24_HH (ae_p24x2s d0, ae_p24x2s d1);
```

AE_SEL16.7362 Operation:

AE_SEL16.7362 (5146, 6543, 4321, 7520, 5410, 5432, 7610, 7632, 6420)		
d, d0, d1		[ae_slot2, ae2_slot0, ae2_slot1]

Combine 16-bit elements from d0 and d1 into d. Elements are numbered in order so that 7 corresponds to the highest significant 16 bits of input register d0 down to 0 which corresponds to the least significant 16 bits of register d1. For example, the diagram below shows the usage of AE_SEL16.7362 .



C syntax:

```
ae_int16x4 AE_SEL16.7362 (ae_int16x4 d0, ae_int16x4 d1);
```

AE_SEL16I Operation:

AE_SEL16I d, d0, d1, imm [ae_slot2, ae2_slot0, ae2_slot1]

Combine 16-bit elements from d0 and d1 into d. The AE_SEL16.xxxx operations above are actually implemented using this operation. Immediate field 'imm' is an encoded value choosing the permutations such as 5146 or 7520 using a 4-bit value. The following table shows the encoded value vs. the permutation.

Table 2-14 AE_SEL16 Operation Values

Immediate Field Value	Permutation
0	5432
1	7632
2	7610
3	5410
4	4321
5	6543
6	7520
7	Used for AE_TRUNC16X4F32 operation or equivalently 7531
8	6420
9	7362
10	5146
11 - 15	reserved

AE_SHORTSWAP Operation:

AE_SHORTSWAP v, v0 [ae_slot2, ae2_slot1]

v.3 = v.0;

v.2 = v.1.

v.1 = v.2;

v.0 = v.3.

C syntax:

```
ae_int16x4 AE_SHORTSWAP (ae_int16x4 d0);
```

2.13 Bitwise Logical Operations

The computations performed by these operations are implied by their opcode mnemonics and operands as given below.

AE_AND, AE_NAND, AE_OR, AE_XOR Operations:

AE_AND	d, d0, d1	[ae_slot2, ae2_slot0]
AE_NAND	d, d0, d1	[ae_slot2, ae2_slot0]
AE_OR	d, d0, d1	[ae_slot2, ae2_slot0]
AE_XOR	d, d0, d1	[ae_slot2, ae2_slot0]

Note: Type-specific C intrinsics are provided through the operations above. C intrinsics AE_ANDQ56, AE_NANDQ56, AE_ORQ56, AE_XORQ56 and AE_NOTQ56 (operating on the ae_q56s C type) and AE_ANDP48, AE_NANDP48, AE_ORP48, AE_XORP48 and AE_NOTP48 (operating on the ae_int24x2 C type) are provided to ensure HiFi 2 code portability and are implemented through the operations above.

C syntax:

```
ae_int64 AE_AND (ae_int64 d0, ae_int64 d1);
ae_int64 AE_NAND (ae_int64 d0, ae_int64 d1);
ae_int64 AE_OR (ae_int64 d0, ae_int64 d1);
ae_int64 AE_XOR (ae_int64 d0, ae_int64 d1);
ae_int64 AE_NOT (ae_int64 d0);

ae_int64 AE_AND64 (ae_int64 d0, ae_int64 d1);
ae_int64 AE_NAND64 (ae_int64 d0, ae_int64 d1);
ae_int64 AE_OR64 (ae_int64 d0, ae_int64 d1);
ae_int64 AE_XOR64 (ae_int64 d0, ae_int64 d1);
ae_int64 AE_NOT64 (ae_int64 d0);

ae_int32x2 AE_AND32 (ae_int32x2 d0, ae_int32x2 d1);
ae_int32x2 AE_NAND32 (ae_int32x2 d0, ae_int32x2 d1);
ae_int32x2 AE_OR32 (ae_int32x2 d0, ae_int32x2 d1);
ae_int32x2 AE_XOR32 (ae_int32x2 d0, ae_int32x2 d1);
ae_int32x2 AE_NOT32 (ae_int32x2 d0);

ae_int24x2 AE_AND24 (ae_int24x2 d0, ae_int24x2 d1);
ae_int24x2 AE_NAND24 (ae_int24x2 d0, ae_int24x2 d1);
ae_int24x2 AE_OR24 (ae_int24x2 d0, ae_int24x2 d1);
ae_int24x2 AE_XOR24 (ae_int24x2 d0, ae_int24x2 d1);
ae_int24x2 AE_NOT24 (ae_int24x2 d0);

ae_int16x4 AE_AND16 (ae_int16x4 d0, ae_int16x4 d1);
ae_int16x4 AE_NAND16 (ae_int16x4 d0, ae_int16x4 d1);
ae_int16x4 AE_OR16 (ae_int16x4 d0, ae_int16x4 d1);
```

```
ae_int16x4 AE_XOR16 (ae_int16x4 d0, ae_int16x4 d1);
ae_int16x4 AE_NOT16 (ae_int16x4 d0);

ae_q56s AE_ANDQ56 (ae_q56s d0, ae_q56s d1);
ae_q56s AE_NANDQ56 (ae_q56s d0, ae_q56s d1);
ae_q56s AE_ORQ56 (ae_q56s d0, ae_q56s d1);
ae_q56s AE_XORQ56 (ae_q56s d0, ae_q56s d1);
ae_q56s AE_NOTQ56 (ae_q56s d0);
ae_p24x2s AE_ANDP48 (ae_p24x2s d0, ae_p24x2s d1);
ae_p24x2s AE_NANDP48 (ae_p24x2s d0, ae_p24x2s d1);
ae_p24x2s AE_ORP48 (ae_p24x2s d0, ae_p24x2s d1);
ae_p24x2s AE_XORP48 (ae_p24x2s d0, ae_p24x2s d1);
ae_p24x2s AE_NOTP48 (ae_p24x2s d0);
```

2.14 Bit Reversal

AE_ADDBRBA32 Operation:

AE_ADDBRBA32	a, ab, ax	[slot0]

32-bit add to a bit-reversed base:

a ← $\text{bitrev}_{32}(\text{bitrev}_{32}(ab) + ax)$.

This helper operation may be used in combination with indexed loads and stores (.X) to perform bit-reversed addressing in optimized FFT implementations. For example, the C code below accesses through a set of 256 32-bit complex data elements in bit-reversed order:

```
/* The data elements will be accessed in the following order:
   0, 128, 64, 192, 32, 160, 96, 224, 16, 144, 80, …
   i.e., for i = 0…255, access element at index bitrev8(i). */
ae_int32x2 *buf = …;
unsigned int index = 0;
unsigned int stride =
  0x80000000U >> (8 /* log2256 */);
…
for (…) {
  …
  ae_int32x2 p = AE_L32X2_X(buf, index);
  index = AE_ADDBRBA32(index, stride);
  …
}
```

C syntax:

```
unsigned AE_ADDBRBA32 (unsigned ab, unsigned ax);
```

2.15 Zero Operation

AE_ZERO Operation:

AE_ZERO	d

Set all bits of an AE_DR register d to zero. This intrinsic is implemented in terms of the AE_MOVI instruction.

Note: Type specific C intrinsics are implemented through AE_ZERO64. C intrinsics AE_ZEROQ56 and AE_ZERO48 are provided to ensure HiFi 2 code portability and are implemented through operation AE_ZERO64.

C syntax:

```
ae_int64 AE_ZERO (void);
ae_int64 AE_ZERO64 (void);
ae_int32x2 AE_ZERO32 (void);
ae_int24x2 AE_ZERO24 (void);
ae_int16x4 AE_ZERO16 (void);
ae_q56s AE_ZEROQ56 (void);
ae_int24x2 AE_ZEROP48 (void);
```

2.16 Optional Floating Point Unit

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

Floating point operations typically have four cycles of latency, but are fully pipelined. Divide (IEEE754 exact), reciprocal, reciprocal square root, and square root (IEEE754 exact) operations are implemented using instruction sequences.

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

ADD.S Operation:

ADD.S	fr, fs, ft	[ae2_slot1, Inst]

Computes a 2-way SIMD IEEE754 single-precision sum of the contents of fs and ft. This operation rounds the result(s) to the destination format when necessary, according to the rounding mode in FCR.

fr.H ← fs.H + ft.H

fr.L ← fs.L + ft.L

C syntax:

```
float XT_ADD_S (float fs, float ft);
xtfloatx2 XT_ADD_SX2 (xtfloatx2 fs, xtfloatx2 ft);
```

SUB.S Operation:

SUB.S	fr, fs, ft	[ae2_slot1, Inst]

Computes a 2-way SIMD IEEE754 single-precision difference of the contents of fs and ft. This operation rounds the result(s) to the destination format when necessary, according to the rounding mode in FCR.

fr.H ← fs.H - ft.H

fr.L ← fs.L - ft.L

C syntax:

```
float XT_SUB_S (float fs, float ft);
xtfloatx2 XT_SUB_SX2 (xtfloatx2 fs, xtfloatx2 ft);
```

MUL.S Operation:

MUL.S	fr, fs, ft	

Computes a 2-way SIMD IEEE754 single-precision product of the contents of fs and ft. This operation rounds the result(s) to the destination format when necessary, according to the rounding mode in FCR.

fr.H ← fs.H * ft.H

fr.L ← fs.L * ft.L

C syntax:

```
float XT_MUL_S (float fs, float ft);
xtfloatx2 XT_MUL_SX2 (xtfloatx2 fs, xtfloatx2 ft);
```

MULC.S Operation:

MULC.S	fr, fs, ft	[ae2_slot1]

Computes a scalar, complex, SIMD IEEE754 single-precision product of the contents of fs and ft. This intrinsic is a proto implemented using a two instruction sequence.

fr ← fs * ft

C syntax:

```
xtfloatx2 XT_MULC_S (xtfloatx2 fs, xtfloatx2 ft);
```

MULMUX.S Operation:

MULMUX.S	fr, fs, ft, i	[ae2_slot1]

Helper instruction for scalar, complex, SIMD IEEE754 single-precision multiplication.

$(fr_{real}, fr_{imag}) \leftarrow (fs_{real} * ft_{real}, fs_{real} * ft_{imag})$ if i==0

$(fr_{real}, fr_{imag}) \leftarrow (fs_{real} * ft_{real}, -fs_{real} * ft_{imag})$ if i==1

C syntax:

```
xtfloatx2 XT_MULMUX_S (xtfloatx2 fs, xtfloatx2 ft, immediate
imm);
```

MULCCONJ.S Operation:

MULCCONJ.S	fr, fs, ft	[ae2_slot1]

Computes a scalar, complex, SIMD IEEE754 single-precision product of the contents of fs and the complex conjugate of ft. This intrinsic is a proto implemented using a two instruction sequence.

fr ← fs * ~ft

C syntax:

```
xtfloatx2 XT_MULCCONJ_S (xtfloatx2 fs, xtfloatx2 ft);
```

MADD.S Operation:

MADD.S	fr, fs, ft	[ae2_slot1, Inst]

MADD.S implements the IEEE754-2008 fusedMultiplyAdd in single precision (binary32). MADD.S multiplies the corresponding halves of data registers fs and ft, adds the products to the corresponding halves of data register fr, and then writes the sums back to the corresponding halves of data register fr. This operation rounds the sum to the destination format when necessary, according to the rounding mode in FCR. There is no rounding on the intermediate and precise product.

fr.H ←fr.H + fs.H * ft.H

fr.L ← fr.L + fs.L * ft.L

C syntax:

```
XT_MADD_S (float fr /* inout */,
              float fs, float ft);
XT_MADD_SX2 (xtfloatx2 fr /* inout */,
              xtfloatx2 fs, xtfloatx2 ft);
```

MADDC.S Operation:

MADDC.S	fr, fs, ft	[ae2_slot1]

Using IEEE754 single-precision arithmetic, MADDC.S does a complex multiplication of the contents of fs and ft and adds the product to complex floating point operand fr and then writes the sum back to complex floating-point operand fr. This intrinsic is a proto implemented using a two instruction sequence.

fr ←fr + fs * ft

C syntax:

```
XT_MADDC_S (xtfloatx2 fr /* inout */,
              xtfloatx2 fs, xtfloatx2 ft);
```

MADDMUX.S Operation:

MADDMUX.S	fr, fs, ft, i	[ae2_slot1]

Helper instruction for scalar, complex, SIMD IEEE754 single-precision multiply accumulation.

$(fr_{real}, fr_{imag}) \leftarrow (fr_{real}, fr_{imag}) + (fs_{real} * ft_{real}, fs_{real} * ft_{imag})$ if i==0

$(fr_{real}, fr_{imag}) \leftarrow (fr_{real}, fr_{imag}) + (-fs_{imag} * ft_{imag}, fs_{imag} * ft_{real})$ if i==1

$(fr_{real}, fr_{imag}) \leftarrow (fr_{real}, fr_{imag}) + (-fs_{real} * ft_{real}, - fs_{real} * ft_{imag})$ if i==2

$(fr_{real}, fr_{imag}) \leftarrow (fr_{real}, fr_{imag}) + (fs_{imag} * ft_{imag}, -fs_{imag} * ft_{real})$ if i==3

$(fr_{real}, fr_{imag}) \leftarrow (fr_{real}, fr_{imag}) + (fs_{real} * ft_{real}, -fs_{real} * ft_{imag})$ if i==4

$(fr_{real}, fr_{imag}) \leftarrow (fr_{real}, fr_{imag}) + (fs_{imag} * ft_{imag}, fs_{imag} * ft_{real})$ if i==5

$(fr_{real}, fr_{imag}) \leftarrow (fr_{real}, fr_{imag}) + (-fs_{real} * ft_{real}, fs_{real} * ft_{imag})$ if i==6

$(fr_{real}, fr_{imag}) \leftarrow (fr_{real}, fr_{imag}) + (-fs_{imag} * ft_{imag}, -fs_{imag} * ft_{real})$ if i==7

C syntax:

```
XT_MADDMUX_S (xtfloatx2 fr /* inout */, xtfloatx2 fs, xtfloatx2
ft, immediate imm);
```

MADDCCONJ.S Operation:

MADDCCONJ.S	fr, fs, ft	[ae2_slot1]

Using IEEE754 single-precision arithmetic, MADDCCONJ.S does a complex multiplication of the contents of fs and the complex conjugate of ft and adds the product to complex floating point operand fr, and then writes the sum back to complex floating-point operand fr. This intrinsic is a proto implemented using a two instruction sequence.

fr ←fr + fs * ~ft

C syntax:

```
XT_MADDCCONJ_S (xtfloatx2 fr /* inout */,
                xtfloatx2 fs, xtfloatx2 ft);
```

MSUB.S Operation:

MSUB.S	fr, fs, ft	[ae2_slot1, Inst]

MSUB.S implements the IEEE754-2008 fusedMultiplyAdd in single precision (binary32). This operation multiplies the corresponding halves of data registers fs and ft, subtracts the products from the corresponding halves of data register fr, and then writes the differences back to the corresponding halves of data register fr. This operation rounds the differences to the destination format when necessary, according to the rounding mode in FCR. There is no rounding on the intermediate and precise products.

fr.H ←fr.H - fs.H * ft.H

fr.L ← fr.L - fs.L * ft.L

C syntax:

```
XT_MSUB_S (float fr /* inout */,
              float fs, float ft);
XT_MSUB_SX2 (xtfloatx2 fr /* inout */,
              xtfloatx2 fs, xtfloatx2 ft);
```

MSUBC.S Operation:

MSUBC.S	fr, fs, ft	[ae2_slot1]

Using IEEE754 single-precision arithmetic, MSUBC.S does a complex multiplication of the contents of fs and ft and subtracts the product from complex floating point operand fr, and then writes the sum back to complex floating-point operand fr. This intrinsic is a proto implemented using a two instruction sequence.

fr ←fr - fs * ft

C syntax:

```
XT_MSUBC_S (xtfloatx2 fr /* inout */,
              xtfloatx2 fs, xtfloatx2 ft);
```

MSUBCCONJ.S Operation:

MSUBCCONJ.S	fr, fs, ft	[ae2_slot1]

Using IEEE754 single-precision arithmetic, MSUBCCONJ.S does a complex multiplication of the contents of fs and the complex conjugate of ft and subtracts the product from complex floating point operand fr, and then writes the sum back to complex floating-point operand fr. This intrinsic is a proto implemented using a two instruction sequence.

fr ←fr - fs * ~ft

C syntax:

```
XT_MSUBCCONJ_S (xtfloatx2 fr /* inout */,
                xtfloatx2 fs, xtfloatx2 ft);
```

MIN.S Operation:

MIN.S	fr, fs, ft	[ae2_slot1]

Computes a 2-way SIMD single-precision floating point minimum of the contents of fs and ft.

fr.H ← MIN(fs.H, ft.H)

fr.L ← MIN(fs.L, ft.L)

C syntax:

```
float XT_MIN_S (float fs, float ft);
xtfloatx2 XT_MIN_SX2 (xtfloatx2 fs, xtfloatx2 ft);
```

MAX.S Operation:

MAX.S	fr, fs, ft	[ae2_slot1]

Computes a 2-way SIMD single-precision floating point maximum of the contents of fs and ft.

fr.H ← MAX(fs.H, ft.H)

fr.L ← MAX(fs.L, ft.L)

C syntax:

```
float XT_MAX_S (float fs, float ft);
xtfloatx2 XT_MAX_SX2 (xtfloatx2 fs, xtfloatx2 ft);
```

DIV.S Operation:

DIV.S	fr, fs, ft

Computes a 2-way SIMD IEEE754 single-precision division of the contents of fs and ft using a sequence of instructions. Division will take approximately 15 cycles. Faster, but non-IEEE754 exact, results can be achieved using RECIP.S and MUL.S. This operation rounds the result(s) to the destination format when necessary, according to the rounding mode in FCR.

fr.H ← fs.H / ft.H

fr.L ← fs.L / ft.L

C syntax:

```
float XT_DIV_S (float fs, float ft);
xtfloatx2 XT_DIV_SX2 (xtfloatx2 fs, xtfloatx2 ft);
```

RECIP.S Operation:

RECIP.S	fr, fs

Computes a 2-way SIMD single-precision reciprocal of the contents of fs using a sequence of instructions taking approximately 5 cycles.

fr.H ← 1.0 / fs.H

fr.L ← 1.0 / fs.L

C syntax:

```
float XT_RECIP_S (float fs);
xtfloatx2 XT_RECIP_SX2 (xtfloatx2 fs);
```

SQRT.S Operation:

SQRT.S	fr, fs

Computes a 2-way SIMD IEEE754 single-precision sqrt using a sequence of instructions taking approximately 15 cycles. Faster, but non-IEEE754 exact, results can be achieved using RSQRT.S and MUL.S. This operation rounds the result(s) to the destination format when necessary, according to the rounding mode in FCR.

fr.H ← sqrtf(fs.H)

fr.L ← sqrtf(fs.L)

C syntax:

```
float XT_SQRT_S (float fs);
xtfloatx2 XT_SQRT_SX2 (xtfloatx2 fs);
```

RSQRT.S Operation:

RSQRT.S	fr, fs

Computes a 2-way SIMD single-precision reciprocal sqrt of the contents of fs using a sequence of instructions taking approximately 10 cycles.

fr.H ← 1.0 / sqrtf(fs.H)

fr.L ← 1.0 / sqrtf(fs.L)

C syntax:

```
float XT_RSQRT_S (float fs);
xtfloatx2 XT_RSQRT_SX2 (xtfloatx2 fs);
```

CONST.S Operation:

CONST.S	fr, i	[ae2_slot1, Inst]

Create a single-precision constant and places it in each half of floating-point register fr.

The constant is chosen by the value of the i field as shown in Table 2-15.

Table 2-15 CONST.S Immediates

Immediate "I"	Decimal Value	Hex Value
0	0.0	0x0000_0000
1	1.0	0x3F80_0000
2	2.0	0x4000_0000
3	0.5	0x3F00_0000

C syntax:

```
xtfloatx2 XT_CONST_S (immediate i);
```

MOVEQZ.S, MOVNEZ.S, MOVGEZ.S, MOVLTZ.S Operations:

MOVEQZ.S	fr, fs, art	[ae2_slot0, Inst]
MOVNEZ.S	fr, fs, art	[ae2_slot0, Inst]
MOVGEZ.S	fr, fs, art	[ae2_slot0, Inst]
MOVLTZ.S	fr, fs, art	[ae2_slot0, Inst]

Two way conditional move of data operand fs to fr based on integer condition in art.

EQZ: fr.H ← (art == 0) ? fs.H : fr.H; fr.L ← (art == 0) ? fs.L : fr.L;

NEZ: fr.H ← (art != 0) ? fs.H : fr.H; fr.L ← (art != 0) ? fs.L : fr.L;

GEZ: fr.H ← (art >= 0) ? fs.H : fr.H; fr.L ← (art >= 0) ? fs.L : fr.L;

LTZ: fr.H ← (art < 0) ? fs.H : fr.H; fr.L ← (art < 0) ? fs.L : fr.L;

C syntax:

```
void XT_MOVEQZ_S (float fr /* inout */,
                  float fs, int art);
void XT_MOVEQZ_SX2 (xtfloatx2 fr /* inout */,
                  xtfloatx2 fs, int art);
void XT_MOVNEZ_S (float fr /* inout */,
                  float fs, int art);
void XT_MOVNEZ_SX2 (xtfloatx2 fr /* inout */,
                  xtfloatx2 fs, int art);
void XT_MOVGEZ_S (float fr /* inout */,
                  float fs, int art);
void XT_MOVGEZ_SX2 (xtfloatx2 fr /* inout */,
```

```
                    xtfloatx2 fs, int art);
void XT_MOVLTZ_S (float fr /* inout */,
                        float fs, int art);
void XT_MOVLTZ_SX2 (xtfloatx2 fr /* inout */,
                    xtfloatx2 fs, int art);
```

MOVT.S, MOVF.S Operations:

MOVT.S	fr, fs, bt	[ae2_slot0, Inst]
MOVF.S	fr, fs, bt	[ae2_slot0, Inst]

Conditional move of data operand fs to fr based on scalar condition in xtbool bt

T: fr ← (bt==1) ? fs : fr

F: fr ← (bt==0) ? fs : fr

C syntax:

```
void XT_MOVT_S (float fr /* inout */,
                    float fs, xtbool b);
void XT_MOVF_S (float fr /* inout */,
                    float fs, xtbool b);
void XT_MOVT.SX2 (xtfloatx2 fr /* inout */,
                    xtfloatx2 fs, xtbool2 b);
void XT_MOVF.SX2 (xtfloatx2 fr /* inout */,
                    xtfloatx2 fs, xtbool2 b);
```

ABS.S Operation:

ABS.S	fr, fs	[ae2_slot1, Inst]

Computes a 2-way SIMD IEEE754 abs of the contents of fs.

fr.H ← abs(fs.H)

fr.L ← abs(fs.L)

C syntax:

```
float XT_ABS_S (float fs);
xtfloatx2 XT_ABS_SX2 (xtfloatx2 fs);
```

NEG.S Operation:

NEG.S	fr, fs	[ae2_slot1, Inst]

Computes a 2-way SIMD IEEE754 negation of the contents of fs.

fr.H ← -fs.H

fr.L ← -fs.L

C syntax:

```
float XT_NEG_S (float fs);
xtfloatx2 XT_NEG_SX2 (xtfloatx2 fs);
```

CONJC.S Operation:

CONJC.S	vtm, vr	[ae2_slot1]

Negate the sign bit of the imaginary lane, which is the lower-order element in the register and the higher-order element in memory, of the complex input operand vr.

vtm.H ← vr.H
vtm.L ← -vr.L

C syntax:

```
xtfloatx2 XT_CONJC_S (xtfloatx2 fs);
```

OLE.S Operation:

OLE.S	br2, fs, ft	[ae2_slot0, Inst]

Compares the single-precision values in each half of floating-point operands fs and ft. If the contents of a half of fs are ordered with,and less than or equal to the contents of the corresponding half of ft, then the corresponding half of xtbool2 br2 is set to 1; otherwise, the corresponding half of br2 is set to 0. According to IEEE754, +0 and -0 compare as equal. IEEE754 floating-point values are ordered if neither is a NaN (Not a Number).

C syntax:

```
xtbool2 XT_OLE_S (float fs, float ft);
xtbool2 XT_OLE_SX2 (xtfloatx2 fs, xtfloatx2 ft);
```

OLT.S Operation:

OLT.S	br2, fs, ft	[ae2_slot0, Inst]

Compares the single-precision values in each half of floating-point operands fs and ft. If the contents of a half of fs are ordered with, and less than the contents of the corresponding half of ft, then the corresponding half of xtbool2 br2 is set to 1; otherwise, the corresponding half of br2 is set to 0. According to IEEE754, +0 and -0 compare as equal. IEEE754 floating-point values are ordered if neither is a NaN.

C syntax:

```
xtbool2 XT_OLT_S (float fs, float ft);
xtbool2 XT_OLT_SX2 (xtfloatx2 fs, xtfloatx2 ft);
```

OEQ.S Operation:

OEQ.S	br2, fs, ft	[ae2_slot0, Inst]

Compares the single-precision values in each half of floating-point operands fs and ft. If the contents of a half of fs are ordered with, and equal to the contents of the corresponding half

of ft, then the corresponding half of xtbool2 br2 is set to 1; otherwise, the corresponding half of br2 is set to 0. According to IEEE754, +0 and -0 compare as equal. IEEE754 floating-point values are ordered if neither is a NaN.

C syntax:

```
xtbool2 XT_OEQ_S (float fs, float ft);
xtbool2 XT_OEQ_SX2 (xtfloatx2 fs, xtfloatx2 ft);
```

ULE.S Operation:

ULE.S	br2, fs, ft	[ae2_slot0, Inst]

Compares the single-precision values in each half of floating-point operands fs and ft. If the contents of a half of fs are are less than or equal to or unordered with the corresponding half of ft, then the corresponding half of xtbool2 br2 is set to 1; otherwise, the corresponding half of br2 is set to 0. According to IEEE754, +0 and -0 compare as equal. IEEE754 floating-point values are unordered if either is a NaN.

C syntax:

```
xtbool2 XT_ULE_S (float fs, float ft);
xtbool2 XT_ULE_SX2 (xtfloatx2 fs, xtfloatx2 ft);
```

ULT.S Operation:

ULT.S	br2, fs, ft	[ae2_slot0, Inst]

Compares the single-precision values in each half of floating-point operands fs and ft. If the contents of a half of fs are less than or unordered with the corresponding half of ft, then the corresponding half of xtbool2 br2 is set to 1; otherwise, the corresponding half of br2 is set to 0. According to IEEE754, +0 and -0 compare as equal. IEEE754 floating-point values are unordered if either is a NaN.

C syntax:

```
xtbool2 XT_ULT_S (float fs, float ft); xtbool2 XT_ULT_SX2
(xtfloatx2 fs, xtfloatx2 ft);
```

UEQ.S Operation:

UEQ.S	br2, fs, ft	[ae2_slot0, Inst]

Compares the single-precision values in each half of floating-point operands fs and ft. If the contents of a half of fs are equal to or unordered with the corresponding half of ft, then the corresponding half of xtbool2 br2 is set to 1; otherwise, the corresponding half of br2 is set to 0. According to IEEE754, +0 and -0 compare as equal. IEEE754 floating-point values are unordered if either is a NaN.

C syntax:

```
xtbool2 XT_UEQ_S (float fs, float ft); xtbool2 XT_UEQ_SX2
(xtfloatx2 fs, xtfloatx2 ft);
```

UN.S Operation:

UN.S	br2, fs, ft	[ae2_slot0, Inst]

Unordered compare. If the contents of a half of fs or half of ft are NaN, then the corresponding half of xtbool2 br2 is set to 1; otherwise, the corresponding half of br2 is set to 0.

C syntax:

```
xtbool2 XT_UN_S (float fs, float ft);
xtbool2 XT_UN_SX2 (xtfloatx2 fs, xtfloatx2 ft);
```

FLOAT.S Operation:

FLOAT.S	fr, ars, i	[ae2_slot0]

Converts and replicates the contents of integral operand ars from signed integer to single-precision format, rounding according to the current rounding mode. The converted integer value is then scaled by a power of two constant value encoded in the immediate field, with 0..31 representing 1.0, 0.5, 0.25, …, 1.0/2147483648.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for i=0 and moves to the left as i increases until for i=31 there are 31 fractional bits represented in the fixed point number. The result is replicated and written to each half of floating-point operand fr.

C syntax:

```
float XT_FLOAT_S (int ars, immediate i);
```

UFLOAT.S Operation:

UFLOAT.S	fr, ars, i	[ae2_slot0]

Converts and replicates the contents of integral operand ars from unsigned integer to single-precision format, rounding according to the current rounding mode. The converted integer value is then scaled by a power of two constant value encoded in the immediate field, with 0..31 representing 1.0, 0.5, 0.25, …, 1.0/2147483648.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for i=0 and moves to the left as i increases until for i=31 there are 31 fractional bits represented in the fixed point number. The result is replicated and written to each half of floating-point operand fr.

C syntax:

```
float XT_UFLOAT_S (unsigned int ars, immediate i);
```

FLOAT.SX2 Operation:

FLOAT.SX2	fr, fs, i	[ae2_slot0]

Computes a two-way SIMD conversion of the contents of integral operand fs from signed integer to single-precision format, rounding according to the current rounding mode. Each converted integer value is then scaled by a power of two constant value encoded in the immediate field, with 0...31 representing 1.0, 0.5, 0.25,...1.0/2147483648.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for i=0 and moves to the left as i increases until for i=31 there are 31 fractional bits represented in the fixed point number.

C syntax:

```
xtfloatx2 XT_FLOAT_SX2 (ae_int32x2 fs, immediate i);
```

UFLOAT.SX2 Operation:

UFLOAT.SX2	fr, fs, i	[ae2_slot0]

Computes a two-way SIMD conversion of the of integral operand fs from unsigned integer to single-precision format, rounding according to the current rounding mode. The converted integer values are then scaled by a power of two constant value encoded in the immediate field, with 0..31 representing 1.0, 0.5, 0.25,…, 1.0/2147483648.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for i=0 and moves to the left as i increases until for i=31 there are 31 fractional bits represented in the fixed point number.

C syntax:

```
xtfloatx2 XT_UFLOAT_SX2 (ae_int32x2 fs, immediate i);
```

FIROUND.S Operation:

FIROUND.S	vt, vr	[ae2_slot0]

Computes a 2-way SIMD round of the floating point values in vr. Each value is rounded to the nearest integral value but maintained in the same floating point format. When the fractional part of an input is exactly 1/2, the value is rounded away from 0.

C syntax:

```
float XT_FIROUND_S (float b);
xtfloatx2 XT_FIROUND_SX2 (xtfloatx2 b);
```

FIFLOOR.S Operation:

FIFLOOR.S	vt, vr	[ae2_slot0]

Computes a 2-way SIMD round of the floating point value in vr. Each value is rounded down to the nearest integral value but maintained in the same floating point format.

C syntax:

```
float XT_FIFLOOR_S (float b);
xtfloatx2 XT_FIFLOOR_SX2 (xtfloatx2 b);
```

FICEIL.S Operation:

FICEIL.S	vt, vr	[ae2_slot0]

Computes a 2-way SIMD round of the floating point values in vr. Each value is rounded up to the nearest integral value but maintained in the same floating point format.

C syntax:

```
float XT_FICEIL_S (float b);
xtfloatx2 XT_FICEIL_SX2 (xtfloatx2 b);
```

FITRUNC.S Operation:

FITRUNC.S	vt, vr	[ae2_slot0]

Computes a 2-way SIMD round of the floating point values in vr. Each value is rounded towards 0 but maintained in the same floating point format.

```
float XT_FITRUNC_S (float b);
xtfloatx2 XT_FITRUNC_SX2 (xtfloatx2 b);
```

FIRINT.S Operation:

FIRINT.S	vt, vr	[ae2_slot0]

Computes a 2-way SIMD round of the floating point values in vr, using the ROUND mode. The result is maintained in the same floating point format.

C syntax:

```
float XT_FIRINT_S (float b);
xtfloatx2 XT_FIRINT_SX2 (xtfloatx2 b);
```

TRUNC.S Operation:

TRUNC.S	arr, fs, i	[ae2_slot0]

Converts the contents of the lower 32 bits of floating-point operand fs from single-precision to signed integer format, rounding toward zero. The converted integer value is first scaled by a power of two constant value encoded in the immediate field, with 0..31 representing 1.0, 0.5, 0.25, …, 1.0/2147483648.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for i=0 and moves to the left as i increases until for i=31 there are 31 fractional bits represented in the fixed point number.

C syntax:

```
int XT_TRUNC_S (float fs, immediate i);
```

UTRUNC.S Operation:

UTRUNC.S	arr, fs, i	[ae2_slot0]

Converts the contents of the lower 32 bits of floating-point operand fs from single-precision to unsigned integer format, rounding toward zero.The converted unsigned integer value is first scaled by a power of two constant value encoded in the immediate field, with 0..31 representing 1.0, 0.5, 0.25, …, 1.0/2147483648.0 The scaling allows for a fixed point notation where the binary point is at the right end of the integer for i=0 and moves to the left as i increases until for i=31 there are 31 fractional bits represented in the fixed point number.

C syntax:

```
unsigned int XT_UTRUNC_S (float fs, immediate i);
```

TRUNC.SX2 Operation:

TRUNC.SX2	fr, fs, i	[ae2_slot0]

Computes a two-way SIMD conversion of floating-point operand fs from single-precision to signed integer format, rounding toward zero. Each converted integer value is first scaled by a power of two constant value encoded in the immediate field, with 0..31 representing 1.0, 0.5, 0.25, …, 1.0/2147483648.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for i=0 and moves to the left as i increases until for i=31 there are 31 fractional bits represented in the fixed point number.

C syntax:

```
ae_int32x2 XT_TRUNC_SX2 (xtfloatx2 fs, immediate i);
```

UTRUNC.SX2 Operation:

UTRUNC.SX2	fr, fs, i	[ae2_slot0]

Computes a two-way SIMD conversion of floating-point operand fs from single-precision to unsigned integer format, rounding toward zero.Each converted unsigned integer value is first scaled by a power of two constant value encoded in the immediate field, with 0..31 representing 1.0, 0.5, 0.25, …, 1.0/2147483648.0 The scaling allows for a fixed point notation where the binary point is at the right end of the integer for i=0 and moves to the left as i increases until for i=31 there are 31 fractional bits represented in the fixed point number.

C syntax:

```
ae_int32x2 XT_UTRUNC_SX2 (xtfloatx2 fs, immediate i);
```

CVTF16S.L Operation:

CVTF16S.L	vt, vr	[ae2_slot0]

Computes a two-way SIMD conversion of floating-point operand vr from single-precision to half-precision. The pair of 16-bit, half-precision results are written to the lower half of the output vt operand, while its upper half is zeroed.

C syntax:

```
xthalfx4 XT_CVTF16S_L (xtfloatx2 b);
```

CVTF16S.H Operation:

CVTF16S.H	vt, vr	[ae2_slot0]

Computes a two-way SIMD conversion of floating-point operand vr from single-precision to half-precision. The pair of 16-bit, half-precision results are written to the upper half of the output vt operand, while the original contents of its lower half is preserved.

C syntax:

```
xthalfx4 XT_CVTF16S_H (xtfloatx2 b);
```

CVTSF16.L Operation:

CVTSF16.L	vt, vr	[ae2_slot0]

Computes a two-way SIMD conversion of floating-point operand vr from half-precision to single-precision. The pair of 16-bit, half-precision inputs are read from the lower half of the input vr operand – bits [31:0], and their corresponding pair of 32-bit single-precision results are written to the output vt operand.

C syntax:

```
xtfloatx2 XT_CVTSF16_L (xthalfx4 b);
```

CVTSF16.H Operation:

CVTSF16.H	vt, vr	[ae2_slot0]

Computes a two-way SIMD conversion of floating-point operand vr from half-precision to single-precision. The pair of 16-bit, half-precision inputs are read from the upper half of the input vr operand – bits [63:32], and their corresponding pair of 32-bit single-precision results are written to the output vt operand.

C syntax:

```
xthalfx4 XT_CVTSF16_H (xtfloatx2 b);
```

RFR Operation:

RFR	art, vr	[Inst]

Copy the low 32 bits of vr into art.

C syntax:

```
unsigned int XT_RFR (float vr);
```

WFR Operation:

WFR	vt, art	[Inst]

Replicate art into each half of data register vt.

C syntax:

```
float XT_WFR (unsigned int vt);
```

Additional helper instructions exist that are used in compiler generated divide and sqrt sequences, which are not included in this document. See the generated HTML file available via the Xtensa Xplorer IDE for details.

2.16.1 Notes on 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.

2.16.2 Floating Point Intrinsics

HiFi floating point programs use the standard HiFi load, store and select operations. To ease programming, intrinsics using floating point types are provided that map into the core HiFi instructions. Refer to Sections 2.4 and 2.12 for more details on the instructions themselves.

LSX2I, LSX2IP, LSX2IRI, LSX2RIC, LSX2X Operations:

LSX2I	d, a, i64
LSX2IP	d, a, i64pos
LSX2RI (RIP)	d, a, i64
LSX2RIC	d, a, [i64neg]
LSX2X (XP,XC)	d, a, ax

Required alignment: 8 bytes

Load a pair of 32-bit values from memory into the AE_DR register `d`. See Table 2-3 for the meanings of the address mode suffixes.

Note: RI and RIP are intrinsics mapped to equivalent instructions.

C syntax:

```
xtfloatx2 XT_LSX2I (const xtfloatx2 * a, immediate i64);
xtfloatx2 XT_LSX2X (const xtfloatx2 * a, int ax);
void XT_LSX2IP (xtfloatx2 d /*out*/,
       const xtfloatx2 *a /*inout*/, immediate i64pos);
void XT_LSX2XP (xtfloatx2 d /*out*/,
            const xtfloatx2 *a /*inout*/, int ax);
void XT_LSX2XC  (xtfloatx2 d /*out*/,
            const xtfloatx2 *a /*inout*/, int ax);
xtfloatx2 XT_LSX2RI (const xtfloatx2 * a, immediate i64);
void XT_LSX2RIP (xtfloatx2 d /*out*/,
         const xtfloatx2 *a /*inout*/, immediate i64);
void XT_LSX2RIC (xtfloatx2 d /*out*/,
               const xt_floatx2 *a /*inout*/);
```

LSI, LSIP, LSX Operations:

LSI	d, a, i32
LSIP	d, a, i32
LSX (XP, XC)	d, a, ax

Required alignment: 4 bytes

Load a 32-bit value from memory and replicate the value into the two elements of the AE_DR register `d`. See Table 2-3 for the meanings of the address mode suffixes.

C syntax:

```
float XT_LSI (const float * a, immediate i32);
float XT_LSX (const float * a, int ax);
void XT_LSIP(float d /*out*/,
        const float * a /*inout*/, immediate off);
void XT_LSXP(float d /*out*/,
               const float * a /*inout*/, int ax);
void XT_LSXC (float d /*out*/,
              const float * a /*inout*/, int ax);
```

LASX2PP Operation:

LASX2PP	u, a

Required alignment: 1 byte (but following instructions have alignment requirements).

Load a 64-bit value from memory to AE_VALIGN register `u`. The effective address is (a & 0xFFFFFFF8). No update is done to the address register.

This instruction is used to prime the unaligned access stream for LASX2IP and LASX2RIP instructions regardless of size or direction.

C syntax:

```
ae_valign XT_LASX2PP (xtfloatx2 *a);
```

LASX2POSPC, LASX2NEGPC Operations:

LASX2POSPC	u, a
LASX2NEGPC	u, a

Required alignment: 4 bytes

This operation loads a 64-bit value from memory into AE_VALIGN register `u`. The effective address is (a & 0xFFFFFFF8).

The LASX2POSPC instruction is used to prime the unaligned access stream for LASX2IC instructions. The LASX2NEGPC instruction is used to prime the unaligned access stream for LASX2RIC instructions.

C syntax:

```
void XT_LASX2POSPC (ae_valign u /*out*/, xtfloatx2 *a /*inout*/);
void XT_LASX2NEGPC  (ae_valign u /*out*/, xtfloatx2 *a /*inout*/);
```

LASX2IP Operation:

LASX2IP (IC, RIP, RIC)	d, u, a

Required alignment: 4 bytes

Load a pair of 32-bit values from effective address (a) in memory into the AE_DR register `d`. Instructions LASX2IP (IC) are used if the direction of the load operations is positive. Instructions LASX2RIP (RIC) are used if the direction of the load operations is negative.

C syntax:

```
void XT_LA32X2IP (xtfloatx2 d /*out*/, ae_valign u /*inout*/,
                  xtfloatx2 *a /*inout*/);
void XT_LASX2IC (xtfloatx2 d /*out*/, ae_valign u /*inout*/,
                  xtfloatx2 *a /*inout*/);
void XT_LASX2RIP (xtfloatx2 d /*out*/, ae_valign u /*inout*/,
                   xtfloatx2 *a /*inout*/);
void XT_LASX2RIC (xtfloatx2 d /*out*/, ae_valign u /*inout*/,
                   xtfloatx2 *a /*inout*/);
```

SSX2I, SSX2IP, SSX2RI, SSX2RIC, SSX2X Operations:

SSX2I	d, a, i64
SSX2IP	d, a, i64pos
SSX2RI (RIP)	d, a, i64
SSX2RIC	d, a
SSX2X (XP, XC)	d, a, ax

Required alignment: 8 bytes

Store a pair of 32-bit values from the AE_DR register `d` to memory. Refer to Table 2-3 for meanings of the address mode suffixes.

Note: RI and RIP are intrinsics mapped to equivalent instructions.

C syntax:

```
void XT_SSX2I (xtfloatx2 d, xtfloatx2 * a, immediate i64);
void XT_SSX2X (xtfloatx2 d, xtfloatx2 * a, int ax);
void XT_SSX2IP (xtfloatx2 d,
                 xtfloatx2 * a /*inout*/, immediate i64);
void XT_SSX2XP (xtfloatx2 d,
                   xtfloatx2 * a /*inout*/, int ax);
void XT_SSX2XC (xtfloatx2 d,
                 xtfloatx2 * a /*inout*/, int ax);
void XT_SSX2RI (xtfloatx2 d, xtfloatx2 * a, immediate i64);
void XT_SSX2RIP (xtfloatx2 d, xtfloatx2 * a /*inout*/, immediate i64);
void XT_SSX2RIC (xtfloatx2 d, xtfloatx2 * a /*inout*/);
```

SSI , SSIP, SSIX Operations:

SSI	d, a, i32
SSIP	d, a, i32
SSIX (XP, XC)	d, a, ax

Required alignment: 4 bytes

Store the 32-bit `L` element of the AE_DR register `d` to memory. For operations with suffix I, the effective address is (a + i32). Refer to Table 2-3 for meanings of the address mode suffixes.

C syntax:

```
void XT_SSI (float d, float * a, immediate i32);
void XT_SSX (float d, float * a, int ax)
void XT_SSIP (float d,
                 float * a /*inout*/, immediate i32);
void XT_SSXP (float d,
                 float * a /*inout*/, int ax);
void XT_SSXC (float d,
                float * a /*inout*/, int ax);
```

SASX2IP Operation:

SASX2IP (IC, RIP, RIC)	d, u, a

Required alignment: 4 bytes

Store a pair of 32-bit values from AE_DR register `d` to memory with effective address (a). Instructions SASX2IP (IC, IC1) are used if the direction of the store operations is positive.

Instructions SASX2RIP (RIC, RIC1) are used if the direction of the store operations is negative.

C syntax:

```
void XT_SASX2IP (xtfloatx2 d, ae_valign u /*inout*/,
                 xtfloatx2 * a /*inout*/);
void XT_SASX2IC  (xtfloatx2 d, ae_valign u /*inout*/,
                 xtfloatx2 * a /*inout*/);
void XT_SASX2RIP (xtfloatx2 d, ae_valign u /*inout*/,
                  xtfloatx2 * a /*inout*/);
void XT_SASX2RIC  (xtfloatx2 d, ae_valign u /*inout*/,
                  xtfloatx2 * a /*inout*/);
```

SASX2POSFP Operation:

SASX2POSFP	u, a

Required alignment: varies depending on the data type in the AE_VALIGN register `u`.

Flushes the value in AE_VALIGN register `u` to memory with effective address (a). The AE_VALIGN register u is updated with value zero. This operation is used when the direction of the store operation is positive.

C syntax:

```
void XT_SASX2POSFP (ae_valign u /*inout*/, xtfloatx2 *a);
```

SASX2NEGFP Operation:

SASX2NEGFP	u, a

Required alignment: varies depending on the data type in the AE_VALIGN register `u`.

Flushes the value in AE_VALIGN register `u` to memory with effective address (a). The AE_VALIGN register u is updated with value zero. This operation is used when the direction of the store operation is negative.

C syntax:

```
void XT_SASX2NEGFP (ae_valign u /*inout*/, xtfloatx2 *a);
```

AE_ZALIGN64 Operation:

AE_ZALIGN64	u

Initialize the AE_VALIGN register u with zero.

C syntax:

```
ae_valign AE_ZALIGN64 ();
```

SEL32_LL.SX2 Operation:

SEL32_LL.SX2	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Concatenate the low floating point value in d0 (which becomes the high value in output d) with the low floating point value of d1 (which becomes the low value in output d).

d.H = d0.L;

d.L = d1.L.

C syntax:

```
xtfloatx2 XT_SEL32_LL_SX2 (xtfloatx2 d0, xtfloatx2 d1);
```

SEL32_LH.SX2 Operation:

SEL32_LH.SX2	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Concatenate the low floating point value in d0 (which becomes the high value in output d) with the high floating point value of d1 (which becomes the low value in output d).

d.H = d0.L;

d.L = d1.H.

C syntax:

```
xtfloatx2 XT_SEL32_LH_SX2 (xtfloatx2 d0, xtfloatx2 d1);
```

SEL32_HL.SX2 Operation:

SEL32_HL.SX2	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Concatenate the high floating point value in d0 (which becomes the high value in output d) with the low floating point value of d1 (which becomes the low value in output d).

d.H = d0.H;

d.L = d1.L.

C syntax:

```
xtfloatx2 XT_SEL32_HL_SX2 (xtfloatx2 d0, xtfloatx2 d1);
```

SEL32_HH.SX2 Operation:

SEL32_HH.SX2	d, d0, d1	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Concatenate the high floating point value in d0 (which becomes the high value in output d) with the high floating point value of d1 (which becomes the low value in output d).

d.H = d0.H;

d.L = d1.H.

C syntax:

```
xtfloatx2 XT_SEL32_HH_SX2 (floatx2 d0, floatx2 d1);
```

LOW.S Operation:

LOW.S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Extract the low floating point value of a SIMD (2-way) floating point pair.

d = d0.L.

C syntax:

```
float XT_LOW_S (xtfloatx2 d0);
```

HIGH.S Operation:

HIGH.S	d, d0	[ae_slot1, ae_slot2, ae2_slot0, ae2_slot1]

Extract the high floating point value of a SIMD (2-way) floating point pair.

d = d0.H;

C syntax:

```
float XT_HIGH_S (xtfloatx2 d0);
```

2.17 Bitstream and Variable-Length Encode and Decode Instructions

The instructions described in this section provide very efficient support for serial access to bitstreams, including variable-length (Huffman) encode and decode, which are stored in memory in serial byte order, with most significant bit first. Each individual instruction is described in this section. The formats of the variable-length encode and decode table entries are specified in Section 2.17.1.

For details on how the variable-length encode/decode instructions should be used, refer to Chapter 4.

All of the following are 24-bit instructions that issue in the Inst slot.

AE_SHA32 Operation:

AE_SHA32	a, a0	[Inst]

Swap 32 bits for half word access. This instruction is used to swap bytes in the two half words in an AR, typically for endianness change during a memcpy()-like operation. For example, if a0 contains 0x12345678 before the AE_SHA32 instruction executes, a contains 0x34127856 immediately afterward.

C syntax:

```
unsigned AE_SHA32 (unsigned a0);
```

AE_VLDL16T Operation:

AE_VLDL16T	b, a, a0	[Inst]

Required alignment: 2 bytes

16-bit table entry load for variable-length decode. Given a pointer a0 to a decoding table of 16-bit entries, an entry is loaded and parsed from a0[AE_NEXTOFFSET]. If the table entry loaded completes the current decoding operation, b is set to true and a is set to the decoded symbol value. Otherwise b is set to false.

C syntax:

```
void AE_VLDL16T (xtbool b /*out*/, unsigned a /*out*/,
                 const unsigned short * a0);
```

AE_VLDL32T Operation:

AE_VLDL32T	b, a, a0	[Inst]

Required alignment: 4 bytes

32-bit table entry load for variable-length decode. Given a pointer a0 to a decoding table of 32-bit entries, an entry is loaded and parsed from a0[AE_NEXTOFFSET]. If the table entry loaded completes the current decoding operation, b is set to true and a is set to the decoded symbol value. Otherwise b is set to false.

C syntax:

```
void AE_VLDL32T (xtbool b /*out*/, unsigned a /*out*/,
                 const unsigned * a0);
```

AE_VLDL16C Operation:

AE_VLDL16C	a	[Inst]

Required alignment: 2 bytes

16-bit conditional bitstream load for variable-length decode. 16 bits are loaded from the bitstream pointed to by (a+2) if they are needed to maintain the invariant that we have at least 16 bits of look ahead from the AE_BITPTR position in the AE_BITHEAD state register. In the event that a load occurs, a is advanced to refer to the next 16 bits in memory.

C syntax:

```
void AE_VLDL16C (const unsigned short * a /*inout*/);
```

AE_VLDL16C.IP Operation:

AE_VLDL16C.IP	a	[Inst]

Required alignment: 2 bytes

16-bit conditional bitstream load for variable-length decode. 16 bits are loaded from the bitstream pointed to by a if they are needed to maintain the invariant that we have at least 16 bits of look ahead from the AE_BITPTR position in the AE_BITHEAD state register. In the event that a load occurs, a is advanced to refer to the next 16 bits in memory.

C syntax:

```
void AE_VLDL16C.IP (const unsigned short * a /*inout*/);
```

AE_VLDL16C.IC Operation:

AE_VLDL16C.IC	a	[Inst]

Required alignment: 2 bytes

16-bit conditional bitstream load for variable-length decode. 16 bits are loaded from the bitstream pointed to by `a` if they are needed to maintain the invariant that we have at least 16 bits of look ahead from the AE_BITPTR position in the AE_BITHEAD state register. In the event that a load occurs, a is advanced using a circular wrap-around to refer to the next 16 bits in memory.

C syntax:

```
void AE_VLDL16C.IC (const unsigned short * a /*inout*/);
void AE_VLDL16C.XC (const unsigned short * a /*inout*/);
```

AE_VLDSHT Operation:

AE_VLDSHT	a	[Inst]

Set Huffman Table for variable-length decode. This instruction sets AE_NEXTOFFSET according to the current bits at the head of the bitstream and the table size specified by `a` for the next lookup that will take place via AE_VLDL16T or AE_VLDL32T.

C syntax:

```
void AE_VLDSHT (unsigned a);
```

AE_LB Operation:

AE_LB	a, a0	[Inst]

Look ahead in the bitstream. Return as few as 0 bits or as many as 16 bits from the head of the bitstream. The number of bits to return is given by the low five bits of a0, and must lie in the range [0..16]. No state is updated; this is a look ahead instruction. The bits from the bitstream are returned right-justified in `a`.

C syntax:

```
unsigned AE_LB (unsigned a0);
```

AE_LBI Operation:

AE_LBI	a, i	[Inst]

Look ahead in the bitstream. Return as few as 1 bit or as many as 16 bits from the head of the bitstream. The number of bits to return is given by the immediate value i, and must lie in the range [1..16]. No state is updated; this is a look ahead instruction. The bits from the bitstream are returned right-justified in `a`.

C syntax:

```
unsigned AE_LBI (immediate i);
```

AE_LBS Operation:

AE_LBS	a, a0	[Inst]

Signed look ahead in the bitstream. Return as few as 1 bit or as many as 16 bits from the head of the bitstream. The number of bits to return is given by the low five bits of a0, and must lie in the range [1..16]. No state is updated; this is a look ahead instruction. The bits from the bitstream are returned sign-extended, right-justified in `a`.

C syntax:

```
unsigned AE_LBS (unsigned a0);
```

AE_LBSI Operation:

AE_LBSI	a, i	[Inst]

Signed look ahead in the bitstream. Return as few as 1 bit or as many as 16 bits from the head of the bitstream. The number of bits to return is given by the immediate value i, and must lie in the range [1..16]. No state is updated; this is a look ahead instruction. The bits from the bitstream are returned sign-extended, right-justified in `a`.

C syntax:

```
unsigned AE_LBSI (immediate i);
```

AE_LBK Operation:

AE_LBK	a, a0, a1	[Inst]

Look ahead in the bitstream, keeping low bits of a0. Return as few as 0 bits or as many as 16 bits from the head of the bitstream in the low bits of `a`, with the remaining bits of `a` filled with low bits from a0. The number of bits to move from the stream to `a` is given by the low five bits of a1, and must lie in the range [0..16]. No state is updated; this is a look ahead instruction.

C syntax:

```
unsigned AE_LBK (unsigned a0, unsigned a1);
```

AE_LBKI Operation:

AE_LBKI	a, a0, i	[Inst]

Look ahead in the bitstream, keeping low bits of a0. Return as few as 1 bit or as many as 16 bits from the head of the bitstream in the low bits of `a`, with the remaining bits of `a` filled with low bits from a0. The number of bits to move from the stream to `a` is given by the immediate value i, and must lie in the range [1..16]. No state is updated; this is a look ahead instruction.

C syntax:

```
unsigned AE_LBKI (unsigned a0, immediate i);
```

AE_DB Operation:

AE_DB	a, a0	[Inst]

Required alignment: 2 bytes

Discard bits from the head of the input bitstream, refreshing the AE_BITHEAD state from address (a + 2) if necessary. `a` points to the current bitstream head address in memory, and a0 contains the number of bits to discard, which must lie in the range [0..16]. If a load from memory takes place to refresh the AE_BITHEAD state, `a` is advanced by two so it will refer automatically to the next 16-bit chunk of the bitstream.

C syntax:

```
void AE_DB (const unsigned short * a /*inout*/, unsigned a0);
```

AE_DB.IP Operation:

AE_DB.IP	a, a0	[Inst]

Required alignment: 2 bytes

Discard bits from the head of the input bitstream, refreshing AE_BITHEAD state from address `a` if necessary. `a` points to the current bitstream head address in memory, and a0 contains the number of bits to discard, which must lie in the range [0..16]. If a load from memory takes place to refresh the AE_BITHEAD state, `a` is advanced by two so it will refer automatically to the next 16-bit chunk of the bitstream.

C syntax:

```
void AE_DB_IP (const unsigned short * a /*inout*/, unsigned a0);
```

AE_DB.IC Operation:

AE_DB.IC	a, a0	[Inst]

Required alignment: 2 bytes

Discard bits from the head of the input bitstream, refreshing the AE_BITHEAD state from address `a` if necessary. `a` points to the current bitstream head address in memory, and a0 contains the number of bits to discard, which must lie in the range [0..16]. If a load from memory takes place to refresh the AE_BITHEAD state, `a` is advanced by two using a circular wrap-aound so that it will refer automatically to the next 16-bit chunk of the bitstream.

C syntax:

```
void AE_DB_IC (const unsigned short * a /*inout*/, unsigned a0);
void AE_DB_XC (const unsigned short * a /*inout*/, unsigned a0);
```

AE_DBI Operation:

AE_DBI	a, i	[Inst]

Required alignment: 2 bytes

Discard bits from the head of the input bitstream, refreshing AE_BITHEAD state from address (a + 2) if necessary. `a` points to the current bitstream head address in memory, and the immediate i gives the number of bits to discard, which must lie in the range [1..16]. If a load from memory takes place to refresh the AE_BITHEAD state, `a` is advanced by two so it will refer automatically to the next 16-bit chunk of the bitstream.

C syntax:

```
void AE_DBI (const unsigned short * a /*inout*/, immediate i);
```

AE_DBI.IP Operation:

AE_DBI.IP	a, i	[Inst]

Required alignment: 2 bytes

Discard bits from the head of the input bitstream, refreshing the AE_BITHEAD state from address `a` if necessary. `a` points to the current bitstream head address in memory, and the immediate i gives the number of bits to discard, which must lie in the range [1..16]. If a load from memory takes place to refresh the AE_BITHEAD state, `a` is advanced by two so it will refer automatically to the next 16-bit chunk of the bitstream.

C syntax:

```
void AE_DBI_IP (const unsigned short * a /*inout*/, immediate i);
```

AE_DBI.IC Operation:

AE_DBI.IC	a, i	[Inst]

Required alignment: 2 bytes

Discard bits from the head of the input bitstream, refreshing the AE_BITHEAD state from address `a` if necessary. `a` points to the current bitstream head address in memory, and the immediate i gives the number of bits to discard, which must lie in the range [1..16]. If a load from memory takes place to refresh the AE_BITHEAD state, `a` is advanced by two using a circular wraparound so that it will refer automatically to the next 16-bit chunk of the bitstream.

C syntax:

```
void AE_DBI_IC (const unsigned short * a /*inout*/, immediate i);
```

AE_VLEL16T Operation:

AE_VLEL16T	b, a, a0	[Inst]

Required alignment: 2 bytes

16-bit table entry load for variable-length encode. Given a pointer a0 to an encoding table of 16-bit entries, an entry is loaded and parsed from a0[a]. If the table entry loaded completes the current encoding operation, b is set to true, otherwise b is set to false and `a` is set to the appropriate index for the next lookup to continue the encoding operation. In either case, the appropriate codeword bits are pushed onto the output bitstream.

C syntax:

```
void AE_VLEL16T (xtbool b /*out*/, unsigned a /*inout*/,
                 const unsigned short * a0);
```

AE_VLEL32T Operation:

AE_VLEL32T	b, a, a0	[Inst]

Required alignment: 4 bytes

32-bit table entry load for variable-length encode. Given a pointer a0 to an encoding table of 32-bit entries, an entry is loaded and parsed from a0[a]. If the table entry loaded completes the current encoding operation, b is set to true, otherwise b is set to false and `a` is set to the appropriate index for the next lookup to continue the encoding operation. In either case, the appropriate codeword bits are pushed onto the output bitstream.

C syntax:

```
void AE_VLEL32T (xtbool b /*out*/, unsigned a /*inout*/,
                 const unsigned * a0);
```

AE_VLES16C Operation:

AE_VLES16C	a	[Inst]

Required alignment: 2 bytes

16-bit conditional bitstream store for variable-length encode. 16 bits are stored to the bitstream pointed to by (a+2) if doing so is needed to maintain the invariant that fewer than 16 bits are buffered in AE_BITHEAD.

C syntax:

```
void AE_VLES16C (unsigned short * a /*inout*/);
```

AE_VLES16C.IP Operation:

AE_VLES16C.IP	a	[Inst]

Required alignment: 2 bytes

16-bit conditional bitstream store for variable-length encode. 16 bits are stored to the bitstream pointed to by `a` if doing so is needed to maintain the invariant that fewer than 16 bits are buffered in AE_BITHEAD.

C syntax:

```
void AE_VLES16C_IP (unsigned short * a /*inout*/);
```

AE_VLES16C.IC Operation:

AE_VLES16C.IC	a	[Inst]

Required alignment: 2 bytes

16-bit conditional bitstream store for variable-length encode. 16 bits are stored to the bitstream pointed to by `a` if doing so is needed to maintain the invariant that fewer than 16 bits are buffered in AE_BITHEAD and a is advanced by two with a circular wrap-around.

C syntax:

```
void AE_VLES16C_IC (unsigned short * a /*inout*/);
```

AE_SB Operation:

AE_SB	a, a0	[Inst]

Required alignment: 2 bytes

Store low bits from a0 to a bitstream pointed to by (a + 2). The number of bits to store comes from the AE_BITSUSED state. If a store to memory is needed to maintain the invariant that fewer than 16 bits are buffered in AE_BITHEAD, `a` is updated for the next store-bits operation.

C syntax:

```
void AE_SB (unsigned short * a /*inout*/, unsigned a0);
```

AE_SB.IP Operation:

AE_SB.IP	a, a0	[Inst]

Required alignment: 2 bytes

Store low bits from a0 to a bitstream pointed to by `a`. The number of bits to store comes from the AE_BITSUSED state. If a store to memory is needed to maintain the invariant that fewer than 16 bits are buffered in AE_BITHEAD, `a` is updated for the next store-bits operation.

C syntax:

```
void AE_SB_IP (unsigned short * a /*inout*/, unsigned a0);
```

AE_SB.IC Operation:

AE_SB.IC	a, a0	[Inst]

Required alignment: 2 bytes

Store low bits from a0 to a bitstream pointed to by `a`. The number of bits to store comes from the AE_BITSUSED state. If a store to memory is needed to maintain the invariant that fewer than 16 bits are buffered in AE_BITHEAD, `a` is updated using a circular wrap-around for the next store-bits operation.

C syntax:

```
void AE_SB_IC (unsigned short * a /*inout*/, unsigned a0);
```

AE_SBI Operation:

AE_SBI	a, a0, i	[Inst]

Required alignment: 2 bytes

Store low bits from a0 to a bitstream pointed to by (a + 2). The number of bits to store comes from the immediate value i. If a store to memory is needed to maintain the invariant that fewer than 16 bits are buffered in AE_BITHEAD, `a` is updated for the next store-bits operation.

C syntax:

```
void AE_SBI (unsigned short *a /*inout*/, unsigned a0, immediate
i);
```

AE_SBI.IP Operation:

AE_SBI.IP	a, a0, i	[Inst]

Required alignment: 2 bytes

Store low bits from a0 to a bitstream pointed to by `a`. The number of bits to store comes from the immediate value i. If a store to memory is needed to maintain the invariant that fewer than 16 bits are buffered in AE_BITHEAD, `a` is updated for the next store-bits operation.

C syntax:

```
void AE_SBI_IP (unsigned short *a /*inout*/, unsigned a0, immediate i);
```

AE_SBI.IC Operation:

AE_SBI.IC	a, a0, i	[Inst]

Required alignment: 2 bytes

Store low bits from a0 to a bitstream pointed to by `a`. The number of bits to store comes from the immediate value i. If a store to memory is needed to maintain the invariant that fewer than 16 bits are buffered in AE_BITHEAD, `a` is updated using a circular wrap-around for the next store-bits operation.

C syntax:

```
void AE_SBI_IC (unsigned short *a /*inout*/, unsigned a0, immediate i);
```

AE_SBF Operation:

AE_SBF	a	[Inst]

Required alignment: 2 bytes

Flush any remaining bits from AE_BITHEAD to the stream in memory pointed to by (a + 2). After the instruction executes, the number of data bits stored (as opposed to padding) can be retrieved from the AE_BITPTR state register.

C syntax:

```
void AE_SBF (unsigned short * a /*inout*/);
```

AE_SBF.IP Operation:

AE_SBF.IP	a	[Inst]

Required alignment: 2 bytes

Flush any remaining bits from AE_BITHEAD to the stream in memory pointed to by `a`. After the instruction executes, the number of data bits stored (as opposed to padding) can be retrieved from the AE_BITPTR state register.

C syntax:

```
void AE_SBF_IP (unsigned short * a /*inout*/);
```

AE_SBF.IC Operation:

AE_SBF.IC	a	[Inst]

Required alignment: 2 bytes

Flush any remaining bits from AE_BITHEAD to the stream in memory pointed to by `a`. After the instruction executes, the number of data bits stored (as opposed to padding) can be retrieved from the AE_BITPTR state register.

C syntax:

```
void AE_SBF_IC (unsigned short * a /*inout*/);
```

2.17.1 Codebook Formats

The variable-length encode and decode instructions described in Section 3.1 use 16- and 32-bit variable-length encode and decode tables (codebooks). The structure of each codebook table entry is described below.

32-bit Variable-Length Decode Table Entry

Each 32-bit variable-length decode codebook table entry has the following format:

In this entry, F is a single bit that indicates whether the symbol has been found.

If F is set, the codeword is decoded and the symbol is found. N gives the number of bits consumed at the current (final) stage of the lookup, and S gives the 27-bit symbol value.

If F is clear, the codeword is only partly decoded and the symbol is not yet found. N is a 4-bit indication of the number of stream prefix bits used to perform the lookup in the next table, and S gives the 27-bit offset of the beginning of the next table. The number of bits consumed is implied by the size of the current sub-table.

16-bit Variable-Length Decode Table Entry

Each 16-bit variable-length decode codebook table entry has the following format:

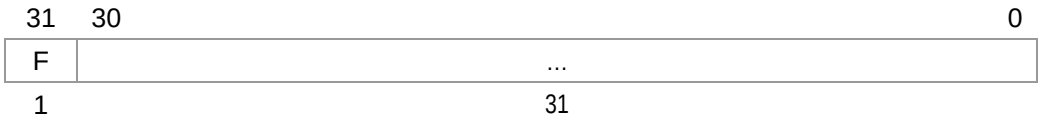
In this entry, F is a single bit that indicates whether the symbol has been found.

If F is set, the codeword is decoded and the symbol is found. N gives the number of bits consumed at the current (final) stage of the lookup, and S gives the 11-bit symbol value.

If F is clear, the codeword is only partly decoded and the symbol is not yet found. N is a 4-bit indication of the number of stream prefix bits used to perform the lookup in the next table, and S gives the 11-bit offset of the beginning of the next table. The number of bits consumed is implied by the size of the current sub-table.

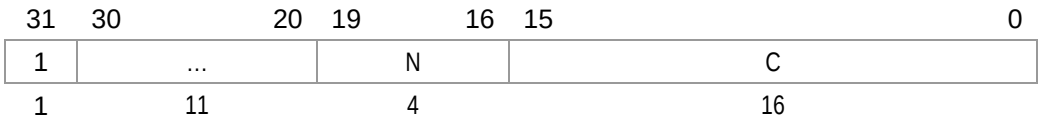
32-bit Variable-Length Encode Table Entry

Each 32-bit variable-length encode codebook table entry has the following format:



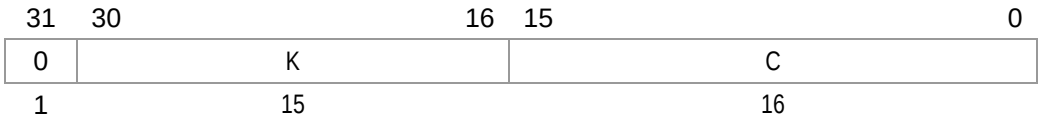
In this entry, F is a single bit that indicates whether the symbol has been completed.

If F is set, the symbol is encoded completely, and the rest of the table entry is interpreted as follows:



N is the codeword segment size in bits (N equal to zero means 16 bits). C contains the right-justified codeword segment. 11 bits of the 32-bit word are unused in this case.

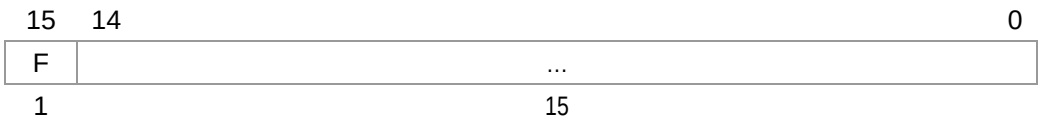
If F is clear, the symbol is only partly encoded, and the rest of the table entry is interpreted as follows:



K is the table entry index for the next encode lookup. C is a 16-bit segment of the codeword.

16-bit Variable-Length Encode Table Entry

Each 16-bit variable-length encode codebook table entry has the following format:



In this entry, F is a single bit that indicates whether the symbol has been completed.

If F is set, the symbol is encoded completely, and the rest of the table entry is interpreted as follows:

N is the codeword segment size in bits with valid values in the range from 1 to 11. C contains the right-justified codeword segment.

If F is clear, the symbol is only partly encoded, and the rest of the table entry is interpreted as follows:

K is the table entry index for the next encode lookup. C is a 6-bit segment of the codeword.

3. Programming the DSP

Cadence recommends the following two important Xtensa manuals that you should read and be familiar with before attempting to obtain optimal results by programming HiFi 3:

- *Xtensa C Application Programmer's Guide*
- *Xtensa C and C++ Compiler User's Guide*

Note that this chapter does not attempt to duplicate material in either of these guides.

HiFi 3 offers four MACs per cycle for 24x24-bit, 32x16-bit, and 16x16-bit audio and voice data and two MACs per cycle for 32x32-bit operations. 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 use HiFi 3's resources, applications that require fractional arithmetic or applications that require 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 then relying on built-in operator overloading, or they can involve using explicit intrinsics to express the exact operations desired. For 16-bit applications, the ITU-T/ETSI intrinsics are fully supported.

In essentially no cases is it required to resort to assembly. All the HiFi 3 instructions can be accessed from C/C++ level intrinsics. The XCC compiler will efficiently register allocate HiFi 3 variables and schedule HiFi 3 instructions, relieving the programmer from the hardest aspects of writing in assembly.

HiFi 3 is also a 2/4-way SIMD (Single Instruction/Multiple Data) architecture. Applications that do not take advantage of SIMD will run up to four times more slowly than applications that do. For 16- or 32-bit integer applications and for applications written using the ITU-T/ETSI intrinsics, the compiler is able to 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. It may only require recognizing that an existing implementation of an application is already in essentially the right form for vectorization, or it may require completely reordering the algorithm's computations to bring together those that can be done in parallel.

For 24-bit and 32x16-bit applications, 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 3 and illustrates them with some simple examples. The following chapter goes into more detail with more complicated examples.

To use the HiFi 3 data types and instruction intrinsics, the programmer must appropriately include:

```
#include <xtensa/tie/xt_hifi2.h>
```

in the C or C++ source code before referring to any of the data types or intrinsics. Note the use of `xt_hifi2.h` rather than `xt_hifi3.h`. This is to facilitate easy use of existing HiFi 2/EP applications.

For floating point intrinsics using the optional floating point unit:

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

3.1 Data Types

Several C data types are provided by the HiFi 3 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 Chapter 2.

HiFi 3 supports 16-, 24-, 32-, and 64-bit types. All types come in both integer and fractional versions. For intrinsic programmers using 16-, 32- and 64-bit types, the two types 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 where 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 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.

Assiging a low precision variable to a high precision variable in general sign extends the variable for signed types and zero extends for unsigned types. Assiging 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 3 supports a 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.

Conversions can also be implicitly applied to intrinsic invocations. For example, just as 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 the legacy HiFi 2 types are supported so that 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 3. 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 3-1 contains a complete list of the HiFi 3 DSP data types with a brief description of each.

Table 3-1 HiFi 3 C Types

Type	Description
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_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_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_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_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 format for this type is four elements stored in adjacent 16-bit words. In memory, this type is 8-byte aligned.
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_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_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.

Type	Description
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 four-byte aligned. This type is loaded and stored in a way that is equivalent to loading and storing the ae_int32 type.
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_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_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.
ae_int64	64-bit type representing the contents of an AE_DR register when the register entry holds a single integer element.
ae_f64	64-bit type representing the contents of an AE_DR register when the register entry holds a single fractional element.
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_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.
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 3, 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 the 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

Type	Description
	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 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 3, 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 3, this type is loaded and stored in a way that is equivalent to loading and storing the ae_p32s type.
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 3, the 1.31-bit value in memory is converted to a 9.23-bit value in 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 3, 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 eight 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 3, the 1.31-bit value in memory is converted to a 17.47-bit value in register.
xtfloatx2	For configurations with the optional SIMD IEEE floating point unit, a type containing two, 32-bit IEEE floating point values.
xthalfx4	For configurations with the optional SIMD IEEE floating point unit, a type containing four 16-bit IEEE half-precision floating point values.

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

The following examples demonstrate how to efficiently load, store, and convert various data types in C using HiFi 3. The examples do not enumerate all possible conversions between core C and HiFi 3 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 3 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 17.23-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);
```

3.1.2 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, you 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 3 supports conversion protos 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 wide variety of display formats, which makes use of these varied data types easier, and also makes it easier to debug. These 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 to any other format.

The display formats for the different types are shown in the following table.

Table 3-2 HiFi 3 Display Types

Type	Displays:
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_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.

Type	Displays:
.ae_f16	Hex and decimal assuming a 1.15 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.
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, programmers put in effort on their code to make it run efficiently on any fixed-point DSP. For example, if the reference code is floating point, the code must be converted into fixed point unless the optional floating point unit is utilized. Doing such conversions is beyond the scope of this guide. However, 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 3 intrinsics. When implementing such an intrinsic library, the programmer has the choice of using standard C/C++ data types as external interfaces or using the native HiFi 3 data types. If the body of a library is ported to use HiFi 3 intrinsics, but the interface remains standard C/C++, the implementation must convert to and from the HiFi 3 data types. The compiler can sometimes, but not always, eliminate these conversions. If instead, the interfaces of the libraries are changed to use HiFi 3 data types, performance will be better, but all the code that calls into the library must be changed to handle the HiFi 3 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 3, 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 3 data types and operator overloading
- Use of intrinsic functions for computation instruction along with HiFi 3 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 `xt-xcc` compiler provides several options and methods of analysis to assist in vectorization. These are discussed in more detail in the *Xtensa C and C++ Compiler User'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`, `-LNO:simd`, or by selecting the Enable Automatic Vectorization option in Xplorer. The `-LNO:simd_v` and `-keep` 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 XCC compiler will naturally align arrays to start on 8-byte boundaries. However 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)`
  - Compiling with `-LNO:aligned_pointers=on`
- 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.
- Compiler alignment options, such as `-LNO:aligned_pointers=on`, tell the compiler that it can assume data is always aligned.
- 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 3 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
        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
        ae_mula16x4     aed1,aed2,aed0,aed0
 }
 {
        ae_la16x4.ip    aed3,u0,a2
        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, simply 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 then 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, int 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
        ae_mulaf16ss.00 aed1,aed0,aed0
        nop
 }
L:
```

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

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

```
        }
        {
                nop
                ae_mulaafd16ss.33_22    aed1,aed0,aed0
        }
        {
                nop
                nop
                ae_mulaafd16ss.11_00_s2 aed1,aed0,aed0
        }
      L:
```

Looking at the loop, you might wonder why the compiler cannot generate a loop that executes four multiplies per cycle. The compiler, even when vectorizing, maintains bit-exactness with the reference intrinsics. These intrinsics require a saturation step after every multiplication and addition. HiFi 3 only supports a dual-mac instruction that does sequential saturations. It is not possible to issue two such dual-mac instructions in parallel as 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_int24x2 p0, p1;
ae_int24x2 p = p0 + p1;
```

Table 3-3 describes the supported operators. 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 3-3 HiFi 3 C/C++ Operators

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.

Operator	Operand Types	Operation	Description
*	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_AND64	Binary AND.
|	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_OR64	Binary OR.
^	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_XOR64	Binary Exclusive OR.
~	ae_f32, ae_f32x2, ae_f32x4, ae_int32, ae_int32x2, ae_int32x4, ae_f24, ae_f24x2, ae_int24, ae_int24x2,	AE_NAND64	Binary NOT.

Operator	Operand Types	Operation	Description
	ae_f64, ae_int64, ae_f16, ae_f16x4, ae_int16, ae_int16x4		
>>	ae_f32, ae_f32x2, ae_f32x4, ae_int32, ae_int32x2, ae_int32x4, ae_in24, ae_int24x2	AE_SRAI32	Signed arithmetic 32-bit right shift by an immediate shift amount.
>>	ae_f32, ae_f32x2, ae_f32x4, ae_int32, ae_int32x2, ae_int32x4 ae_int24, ae_int24x2	AE_SRAA32	Signed arithmetic 32-bit right shift by a variable shift amount.
<<	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_NEG32	Signed 32-bit negation.

Operator	Operand Types	Operation	Description
	ae_int24, ae_int24x2		
*	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
*	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.

Operator	Operand Types	Operation	Description
<<	ae_f24, ae_f24x2	AE_SLAS24S	Signed saturating 24-bit left shift by a variable shift amount.
*	ae_int24x2	AE_MULP24X2	Signed SIMD 24x24 into 32-bit multiplication with an ae_int32x2 return type.
*	ae_int24	AE_MULP24X2	Signed 24x24 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.
>>	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.

Operator	Operand Types	Operation	Description
+	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.
<<	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.

Operator	Operand Types	Operation	Description
>	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.

Table 3-4 describes the supported operators for the optional floating point unit. Scalar types should all be programmed using the standard C/C++ float support.

Table 3-4 HiFi 3 C/C++ Floating Point Operators

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 3-5 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 3.

Table 3-5 Legacy HiFi 2 C/C++ Operators

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.
*	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.

Operator	Operand Types	Operation	Description
<<	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.
>	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.

Operator	Operand Types	Operation	Description
>=	ae_q56s	AE_LEQ56S	Signed greater-than-or-equal comparison with an `xtbool` return type.
>	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 3 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 high-precision multiplies in HiFi 3, you must use intrinsics.

3.6.1 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 standard C reference code is shown below.

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

Assuming that we wish to use 24-bit arithmetic and can therefore throw away the bottom 8 bits of the input, the code can be converted into HiFi 3 code as follows.

```
ae_f24 *ap = (ae_f24 *) 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 24-bit, fixed point multiply. The `ae_f24` typed array is implicitly loaded, just like any standard C/C++ type. As part of the load, the bottom 8 bits of the 1.31 input array are discarded. The accumulator is of type `ae_f32`, giving 8 guard bits. The assignment of the result to an `int` does not change the bit pattern. Hence this routine returns a 9.23 value stored as an `int`.

The compiler generates the following inner loop.

```
loop a3, L
{
        ae_l32f24.ip    aed0,a2,4
        ae_mulafp24x2ra aed1,aed0,aed0
        nop
}
L:
```

HiFi 3 is able to issue a multiply and a load every cycle. Note that the compiler automatically generates the multiply-add instruction, `ae_mulafp24x2ra`. This instruction does a 24-bit multiplication with a 32-bit accumulation. The 32-bit accumulation does not saturate, so this code is only safe where a 32-bit overflow is not possible. If overflow is possible, compile with –mno-enable-non-exact-imaps. The compiler will leave the multiply and the addition as two separate instructions and will use a saturating add for the addition.

The inner loop is perfect, except that no SIMD is used. By changing `ae_f24` into `ae_f24x2f`, `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 two partial sums in parallel. After the loop, we must add together the two partial sums into a single sum using the `AE_ADD32_HL_LH` intrinsic.

```
ae_f24x2 *ap = (ae_f24x2 *) a;
ae_f32x2 s = 0;

for (i = 0; i < n>>1; i++)
{
        s = s +  ap[i]*ap[i];
}
return AE_ADD32_HL_LH(s,s);
```

The compiler generates the following inner loop.

```
loop a3, L
{
        ae_l32x2f24.ip  aed0,a2,8
        ae_mulafp24x2ra aed1,aed0,aed0
        nop
}
L:
```

The generated code is now able to do two multiplies every cycle with the speed limited by the load/store bandwidth of the machine.

Note that the optimized code assumes that `n` is a multiple of two. If that is not guaranteed, the last iteration of the loop must be conditionally peeled as follows.

```
ae_f24x2 *ap = (ae_f24x2 *) a;
ae_f32x2 s = 0;

for (i = 0; i < n>>1; i++)
{
        s = s +  ap[i]*ap[i];
}
ae_f32 last_prod = 0;
if (n%2) {
  ae_f24 *ap_scalar = (ae_f24 *) a;
  last_prod = ap_scalar[n-1]*ap_scalar[n-1];
}
return AE_ADD32_HL_LH(s,s) +
 AE_MOVINT32X2_FROMF32(last_prod);
```

If the total number of iterations dynamically turns out to be odd, the last iteration is executed separately, using scalar instructions. Note the use of the `AE_MOVINT32X2_FROMF32` intrinsic. The reduction add intrinsic returns an `ae_int32x2` type, therefore the product of the last iteration must be appropriately coerced.

This example code uses fixed point arithmetic. If instead, integral arithmetic is desired, simply use the integral rather than the fixed-point types.

```
ae_int24x2 *ap = (ae_int24x2 *) a;
ae_int32x2 s = 0;

for (i = 0; i < n>>1; i++)
{
        s = s +  ap[i]*ap[i];
}
ae_int32 last_prod = 0;
if (n%2) {
  ae_int24 *ap_scalar = (ae_int24 *) a;
  last_prod = ap_scalar[n-1]*ap_scalar[n-1];
}
return AE_ADD32_HL_LH(s,s)+last_prod;
```

3.6.2 Operator Overloading: 32X16-bit Dot Product Example

Consider now a scenario where we wish to do 32x16-bit multiplication, rather than 24-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, HiFi 3 defines an `ae_int32x4` (and an `ae_f32x4`) data type. These are structure data types that occupy two registers. Most operations defined on these types result in two instructions so are no faster than the 2-way SIMD types. However, their use is necessary when doing 32x16-bit multiplication using operator overloading. The example is shown below. Note that the result is reduced into a single `int` using the `AE_INT32X4_RADD` intrinsic. This is a convenience intrinsic, which translates into a 3-instruction sequence.

```
ae_int32x4 *ap = (ae_int32x4 *) a;
ae_int16x4 *bp = (ae_int16x4 *) b;
ae_int32x4 s = 0;

for (i = 0; i < n>>2; i++)
{
        s += bp[i]*ap[i];
}
return AE_INT32X4_RADD(s);
```

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 3 instruction can be directly accessed by an intrinsic of the same name (except that "." in instruction names get converted into "_" in intrinsic names). The prototypes of the supported intrinsics are listed along with the instruction descriptions in Chapter 2.

Consider a simple example that does a 24-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_f24x2 *ap = (ae_f24x2 *) a;
ae_f64 s = AE_ZERO64();

for (i = 0; i < n>>1; i++)
{
        AE_MULAAFD24_HH_LL(s, ap[i], ap[i]);
```

```
}
ae_f24 result =  AE_ROUND24F48SASYM(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 24-bits.

Following are some interesting 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 all 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_f24x2 tmp;
ae_f24x2 *ap = (ae_f24x2 *) a;
ae_f64 s = AE_ZERO64();

for (i = 0; i < n>>1; i++)
{
        AE_L32X2F24_XC(tmp, ap, 8);
        AE_MULAAFD24_HH_LL(s, tmp, tmp);
}
ae_f24 result =  AE_ROUND24F48SASYM(s);
```

The operand pointer is loaded using the updating, circular load intrinsic, `AE_L32X2F24_XC`. This example assumes that the boundaries of the circular buffer have been set elsewhere.

Chapter 5 provides more examples in detail.

3.8 HiFi 2 and HiFi Mini Code Portability

The HiFi 3 DSP implements all HiFi 2 C types, intrinsics and operator overloads to ensure that existing HiFi 2 C and C++ target source code can compile and run on a HiFi 3 processor. HiFi 2 assembly target code must be manually modified to build and run on a HiFi 3 processor. The HiFi 3 and HiFi 2 ISAs are not binary compatible—a binary compiled on a HiFi 2 processor will not execute correctly on a HiFi 3 processor.

The HiFi Mini DSP supports the 8-bit load instructions AE_LP8X2F.I and AE_LP8X2F.IU, which have no equivalent on HiFi3. 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 3.

Listed below are guidelines for porting HiFi 2 target code to HiFi 3:

- *Mapping:* Refer to the operations and intrinsics (C syntax) in Chapter 2 for notes on the HiFi 2 to HiFi 3 operation and intrinsic mapping
- *Precision:* To ensure efficient execution of existing HiFi 2 code on HiFi 3 as well as efficient HiFi 3 hardware implementation, some HiFi 2-specific intrinsics (DSP operations, loads, and stores) provide wider precision than the intrinsics available in the HiFi 2 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 3 the computation will maintain the extra precision in the 8 MSBs of each 32-bit AE_DR element. If a HiFi 2 application assumes wrap-around due to the limited register width, it may need to be fixed to ensure correct execution on HiFi 3
- *Performance:* To ensure efficient HiFi 3 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 3. For example, HiFi 2 intrinsic AE_MULZASFQ32SP16S_HH is implemented through a sequence of 4 operations in HiFi 3. If a HiFi 2 application relies on such intrinsics, it may need to be manually reoptimized to ensure efficient execution on HiFi 3. However, in many cases the additional VLIW slot, extra registers, and MACs provided by HiFi 3 will be sufficient to compensate.

3.9 Important Compiler Switches

The following compiler switches are important:

- `-mcoproc` – as discussed in the *Xtensa C Application Programmer's Guide* and *Xtensa C and C++ Compiler User's Guide.* In particular for HiFi 3, the use of this flag allows the compiler to emulate standard C/C++ operations using the HiFi 3 DSP instructions. In comparison to HiFi 2/EP, this flag may have a much larger impact on HiFi 3.
- 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. Variable Length Encode and Decode

The HiFi 3 Instruction Set Architecture includes a set of instructions that make it convenient for you to do variable-length encoding and decoding in your software routines for the HiFi 3 DSP. This chapter will help you become familiar with the HiFi 3 Huffman-related instructions and Huffman table formats.

4.1 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.

4.1.1 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.

4.2 Encoding

Since encoding usually has fewer worthwhile table-structure variants than decoding, we will describe the encode side first, and then move to the more complicated considerations around decoding.

Examples we'll look into (in Section 4.4) structure their tables in a couple of ways that are the most commonly used. You will certainly encounter cases, e.g., in at least one of WMA's codebooks, where you will want to implement a different structure for the tables.

For encoding, the usual technique is simple: Translate the symbol to be coded into a table index, and use that index to retrieve a sequence of codeword bits and a codeword length from a table or a pair of tables. Usually table entries for each codebook are just long enough to hold the longest codeword, but in the present mechanism we wanted to provide a way to keep the codeword length from being dependent on either the size of the table entries or on other aspects of the implementation. So in our scheme, depending on the length of the longest codeword, it might be that some codewords don't fit within a single table entry. When that happens, the first lookup in the encoding table provides not only a portion of the codeword, but also the index of the location in the table to look for the next codeword segment. Each lookup in the encoding table either completes the codeword or yields an index for the next lookup. 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.

4.2.1 What Encoding a Symbol Looks Like

Here's what the process of encoding one symbol looks like. In memory we have an encoding table indexed by symbol value, and each entry in the table is one of the 16- or 32-bit table entries we've been discussing. 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.

4.2.2 The 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`. `ae_vlel{16|32}t` 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).

4.3 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 2/EP 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.

4.3.1 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're using 32-bit table entries) and `ae_vldl16c`. `ae_vldl{16|32}t` 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).

4.4 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_vlel16tb0, 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_vldl16ca2
                                    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 need 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 3 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.

5. Audio DSP Examples

In this chapter, we cover a few examples in more detail, showing how to optimize the examples for the HiFi 3 DSP.

5.1 Correlation/Convolution/FIR Coding Example

The following example shows how to efficiently implement a correlation (or equivalently, a convolution or block FIR filter) using the HiFi 3 DSP.

We start with the following simple reference code, written using the scalar 24-bit HiFi 3 data types.

```
void fir_ref ( ae_f24 * __restrict y, ae_f24 *__restrict x,
ae_f24 *__restrict h, unsigned N, unsigned M)
    {
        int i, j;

        for (i = 0; i < N; i++) {
            ae_f64 y0 = AE_ZERO64();
            for(j = 0; j < M; j++) {
                AE_MULAF24S_LL(y0, x[i+j], h[j]);
            }
            y[i] = AE_ROUND24F48SASYM(y0);
        }
    }
```

We use a 64-bit accumulator for all the intermediate calculations. The accumulator is initialized using the `AE_ZERO64` intrinsic. When we have completed one output point, we round asymmetrically the 64-bits down to a 1.23 data type using the `AE_ROUND24F48SASYM` instruction.

Next, we utilize SIMD to perform two iterations of the outer loop in parallel, computing two output points together. We use the special FIR multiply instructions that can compute both `x[i+j] * h[j]` and `x[i+j+1]*h[j]` in a single instruction.

```
void fir_opt (ae_f24x2 *__restrict y, ae_f24x2 *__restrict x,
              ae_f24x2 *__restrict h, unsigned N, unsigned M)
{
        int i, j;

        for (i = 0; i < N/2; i++) {
            ae_f64 y0=AE_ZERO64(), y1=AE_ZERO64();
            for(j = 0; j < M/2; j++) {
                AE_MULAFD24X2_FIR_H(y0, y1,
            x[i+j], x[i+j+1], h[j]);
            }
            y[i] = AE_ROUND24X2F48SASYM(y0, y1);
        }
}
```

Note that we traverse both the coefficient array and the data array in the forward direction, while a typical formulation often accesses the data in the reverse direction (`x[M+N-1+i-j] * h[j]`). To use the reverse formulation, we must traverse the data array in the reverse direction using the `.RIP` instructions, and we must use the .L version of the FIR instructions. It is not possible to use the `.RIP` instructions with implicit loads; we must use explicit intrinsics as shown below.

```
        for (i = 0; i < N/2; i++) {
           ae_f64 y0=AE_ZERO64(), y1=AE_ZERO64();
           ae_f24x2 *xp = &x[M/2+N/2+i];
           ae_f24x2 ximj; // x[M/2+N/2 + i-j]
           ae_f24x2 ximjm1; // x[M/2+N/2 + i-j-1]
           AE_L32X2F24_RIP(ximj, xp);
           for(j = 0; j < M/2; j++) {
                AE_L32X2F24_RIP(ximjm1, xp);
                AE_MULAFD24X2_FIR_L(y1,y0,ximj,ximjm1,h[j]);
                ximj = ximjm1;
           }
           AE_S24X2RA64S_IP(y0, y1, y);
        }
```

As a minor enhancement, we have also replaced the round and the store with the compound operation `AE_S24X2RA64S_IP` that does both. The code is now nicely vectorized. However, if one looks at the code generated by the compiler for the inner loop, the code only does four multiplies every two cycles, half the peak multiplier speed of the machine.

```
 loop a4, L:
 {
       ae_l32x2f24.ip  aed0,a2,8
       ae_mulafd24x2.fir.l     aed2,aed3,aed4,aed1,aed0
 }
 {
       ae_l32x2f24.rip aed1,a3
       ae_mov64        aed4,aed1
       nop
 }
L:
```

Performance is limited by the load/store bandwidth of the machine. Note that the same coefficient values are read in every iteration of the outer loop `i`. Similarly, the data value being read in iteration `{i,j}` will be read again in iteration `{i+1, j+1}`. If we unroll the outer `i` loop further, we can eliminate the duplicate loads.

```
            for (i = 0; i < N/4; i++) {
               ae_f64 y0=AE_ZERO64(), y1=AE_ZERO64();
               ae_f64 y2=AE_ZERO64(), y3=AE_ZERO64();
               ae_f24x2 *xp = &x[M/2+N/2+2*i+1];
               ae_f24x2 x2imjp1; // x[M/2+N/2+2i-j+1]
               ae_f24x2 x2imj; // x[M/2+N/2+2i-j]
               ae_f24x2 x2imjm1; // x[M/2+N/2+2i-j-1]
               AE_L32X2F24_RIP(x2imjp1, xp);
               AE_L32X2F24_RIP(x2imj, xp);
               for(j = 0; j < M/2; j++) {
                   AE_L32X2F24_RIP(x2imjm1, xp);
                   AE_MULAFD24X2_FIR_L(y1,y0,x2imj,x2imjm1,h[j]);
                   AE_MULAFD24X2_FIR_L(y3,y2,x2imjp1,x2imj,h[j]);
                   x2imjp1 = x2imj;
                   x2imj = x2imjm1;
               }
                      AE_S24X2RA64S_IP(y0, y1, y);
AE_S24X2RA64S_IP(y2, y3, y);
            }
```

Our inner loop now contains two multiply instructions (each doing four multiplies) and two loads, and the compiler is able to schedule it perfectly to the multiply bound of the machine.

5.2 Floating Point FIR Example

Now we consider a floating point FIR on the optional floating point unit. The reference code is quite simple:

```
void fir_ref ( float * __restrict y, const   float *__restrict
x, const   float *__restrict h,
                    int n,   int m)
{
#pragma aligned(x,8)
#pragma aligned(h,8)
#pragma aligned(y,8)
        int i, j;
        for (i = 0; i < n; i++) {
            y[i] = 0;
            for(j = 0; j < m; j++) {
                y[i] += x[i+j]*h[j];
            }
        }
```

The code is all standard C, except that we use pragmas to tell the compiler that all the arrays are aligned on 8-byte boundaries. While those pragmas are not necessary, they allow the XCC vectorizer to generate somewhat more efficient code. When compiling with –O3 –LNO:simd, the XCC vectorizer vectorizes the outer *i* loop and further unrolls both it and the *j* loop by two. The resultant code performs four 2-way SIMD multiply-accumulate operations in every iteration and schedules in five cycles, which is close to the ideal limit of four.

To understand what the vectorizer does, and to try to get the ideal four cycle schedule, let us vectorize the code using intrinsics. First, let us vectorize the outer loop by two. Using pseudo code, the inner loop is computing y[i:i+1] += x[i+j:i+j+1]*h[j]. Note that in the first *j* iteration, we are using x[i:i+1], while in the second we are using overlapping data x[i+1:i+2]. We cannot simply utilize a SIMD load that will get the right data in both even and odd iterations. Instead, we also unroll the *j* loop by two. In the first unrolled iteration, we use x[i:i+1] and x[i+1:i+2]. In the next iteration, we use data that is exactly two elements ahead: x[i+2:i+3] and x[i+3:i+4]. The data can be loaded in one of two ways. First, we can use an aligned load to load the even iterations x[i+j:i+j+1] and an aligning load to load to odd iterations x[i+j+1:i+j+2]. Alternatively, we can use selects to extract x[i+j+1:i+j+2] from x[i+j:i+j+1] and x[i+j+2:i+j+3]. Unfortunately, selects cannot issue in parallel with multiply accumulate operations. Hence, the first approach of using two loads is better.

```
unsigned i, j;
xtfloatx2 *yx2p = (xtfloatx2 *) y;
for (i = 0; i < n/2; i++) {
    xtfloatx2 y0 = 0.0F;
    xtfloatx2 data0, data1;
    xtfloatx2 *x0 = (xtfloatx2 *) &x[2*i+0];
    xtfloatx2 *x1 = (xtfloatx2 *) &x[2*i+1];
    align = XT_LASX2PP(x1);
```

```
        for(j = 0; j < m/2; j++) {
            data0 = *x0++;
            XT_LASX2IP(data1,align,x1);
            XT_MADD_SX2(y0, data0,((xtfloatx2) h[2*j]));
            XT_MADD_SX2(y0, data1,((xtfloatx2) h[2*j+1]));
        }
        *yx2p++ = y0;
    }
```

Note that we have used explicit multiply-add intrinsics, `XT_MADD_SX2`, rather than just the standard C a = a + b*c. The HiFi floating point multiply add instruction does not round in between the multiply and the add and therefore is more accurate than the two-instruction sequence. When writing standard C code, the compiler will automatically infer the use of the instruction by default when compiling with –O3, but when writing using vector intrinsics, the compiler will not.

Note that MADD operations have four cycles of latency. In every iteration of the inner loop, we are accumulating into the same accumulator, causing us to spend eight cycles every iteration. We can double the performance by using separate accumulators, *y0* and *y1*, for each MACC and then add them together after the loop **yx2p++ = y0+y1*. Now we can achieve four cycles for two SIMD MADDs. This is still significantly slower than the original reference code since the vectorizer unrolled the outer loop by a further factor of two. We do the same below. Note that since we are reading data from *x[i+j],* iterations *i=1, j=0* is reading the same data that was read in *i=0, j=1*. Rather than reload the data, we simply copy the previously loaded data.

```
    for (i = 0; i < n/4; i++) {
        xtfloatx2 y0=0.0F, y1=0.0F, y2=0.0F, y3=0.0F;
        xtfloatx2 data0, data1, data2, data3;
        xtfloatx2 *x0 = (xtfloatx2 *) &x[4*i+0];
        data2 = *x0++;
        xtfloatx2 *x1 = (xtfloatx2 *) &x[4*i+1];
        align = XT_LASX2PP(x1);
        XT_LASX2IP(data3,align,x1);
        for(j = 0; j < m/2; j++) {
            data0 = data2;
            data1 = data3;
            data2 = *x0++;
            XT_LASX2IP(data3,align,x1);
            XT_MADD_SX2(y0, data0,((xtfloatx2) h[2*j]));
            XT_MADD_SX2(y1, data1,((xtfloatx2) h[2*j+1]));
            XT_MADD_SX2(y2, data2,((xtfloatx2) h[2*j]));
            XT_MADD_SX2(y3, data3,((xtfloatx2) h[2*j+1]));
        }
        *yx2p++ = y0+y1;
        *yx2p++ = y2+y3;
    }
```

Now the compiler is able to generate a perfect four cycle schedule for four, 2-way SIMD MADDs.

5.3 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. The following is a simple radix-2 decimation in frequency FFT implementation that assumes `n` is a power of 2 and the number of complex elements stored in the data array. In an efficient implementation, we would use a radix-4 implementation, but the radix-2 implementation is simpler to explain.

Consider the following simple reference implementation, where the data is 32-bits but the twiddle factors are 16-bit.

```
void fft_ref( int data[], const short twid[],
const unsigned int n)
{
  unsigned int i, j0, j1, k, b;
  int r = 0x4000;

  k = 0;
  for (b = n ; b > 2; b >>= 1) {
    unsigned int b2 = b >> 1;
    for (i = 0; i < b/4; i += 1) {
      short wr = twid[2 * k + 0];
      short wi = twid[2 * k + 1];
      for (j0 = j1 = i; j0 < n; ) {
        int d0r, d0i, d1r, d1i;
        int tr, ti;
        int r0r, r0i, r1r, r1i;

        d0r = data[2 * j0 + 0];
        d0i = data[2 * j0 + 1];
        j0 += b2;

        d1r = data[2 * j0 + 0];
        d1i = data[2 * j0 + 1];
        j0 += b2;

        r0r = d0r + d1r;
        r0i = d0i + d1i;
        tr = d0r - d1r;
        ti = d0i - d1i;

        r1r =((long long)wr*tr - (long long)wi*ti + r) >> 15;
        r1i =((long long)wr*ti + (long long)wi*tr + r) >> 15;

        data[2 * j1 + 0] = r0r;
        data[2 * j1 + 1] = r0i;
        j1 += b2;
```

```
            data[2 * j1 + 0] = r1r;
            data[2 * j1 + 1] = r1i;
            j1 += b2;
          }
          k += 1;
        }

      }
    }
```

The 2-way SIMD architecture of HiFi 3 maps nicely to this computation as a 64-bit register can hold a single, 32-bit complex data item. The 16-bit twiddle factors complicate the algorithm a bit since we can store four twiddle factors in one register. Successive iterations of `i` must access either the top two or the bottom two entries in the twiddle register. The simplest way to handle this is to load the twiddle array every other iteration, and use selects to copy the second pair of elements into position on the other iterations. Otherwise, the conversion to HiFi 3 is very simple, and the resultant code is actually simpler than the original. Note the use of the explicit complex multiply intrinsic `AE_MULFC32X16RAS_H`.

```
void fft_opt( ae_f32x2 data[], const ae_f16x4 twid[],
const unsigned int n)
{
  unsigned int i, j0, j1, k, b;

  k = 0;

  for (b = n ; b > 2; b >>= 1) {
    unsigned int b2 = b >> 1;
    ae_f16x4 w;
    for (i = 0; i < b/4; i += 1) {
      if (!(i%2)) w = twid[k];
      else {
        w = AE_SEL16_5410(w, w);
        k += 1;
      }
      for (j0 = j1 = i; j0 < n; ) {
        ae_f32x2 d0, d1;
        ae_f32x2 t;
        ae_f32x2 r0, r1;

        d0 = data[j0];
        j0 += b2;

        d1 = data[j0];
        j0 += b2;

        r0 = d0 + d1;
        t = d0 - d1;

        r1 = AE_MULFC32X16RAS_H(t, w);
```

```
            data[j1] = r0;
            j1 += b2;

            data[j1] = r1;
            j1 += b2;
          }
        }
      }
    }
```

Review the generated `.s` file in the *Xtensa C Application Programmer's Guide* to see how the compiler compiles the particular loop. Look at the code next to the inner loop and you will see the following message.

```
#<swpf>  complex trip count calculation
```

The compiler was not able to optimize the inner loop well because the compiler could not calculate the number of iterations in the inner loop. By rewriting the trip count calculation as follows, the compiler is able to better optimize the inner loop.

```
void fft_opt( ae_int32x2 data[], const ae_int16x4 twid[],
 const unsigned int n)
{
        unsigned int i, k=0, b,trip;
        unsigned lg2 = 31-AE_NSAZ32.L(n);
  for (b = n ; b > 2; b >>= 1) {
    lg2--;
    unsigned int b2 = b >> 1;
    ae_int16x4 w;
    for (i = 0; i < b/4; i += 1) {
      if (!(i%2)) w = twid[k];
      else {
        w = AE_SEL16_5410(w, w);
        k += 1;
      }
      for (trip=0; trip < (n-i+b-1)>>lg2; trip++) {
        ae_int32x2 d0, d1;
        ae_int32x2 t;
        ae_int32x2 r0, r1;

        d0 = data[i+2*b2*trip];
        d1 = data[i+2*b2*trip+b2];

        r0 = d0 + d1;
        t = d0 - d1;
        r1 = AE_MULFC32X16RAS_H(t, w);
```

```
            data[i+2*b2*trip] = r0;
            data[i+2*b2*trip+b2] = r1;
          }
        }
      }
    }
```

Performance is better but still not ideal. In order to achieve top performance, the compiler must software pipeline the loop and execute loads from iteration `trip+1` ahead of stores from the previous iteration `trip`. However, the compiler will not move the loads up because it does not know if the loads and stores access the same memory. Therefore, you must move the loads manually as follows.

```
void fft_opt( ae_int32x2 data[], const ae_int16x4 twid[],
const unsigned int n)
{
        unsigned int i, k=0, b,trip;
  unsigned lg2 = 31-AE_NSAZ32.L(n);

  for (b = n ; b > 2; b >>= 1) {
    lg2--;
    unsigned int b2 = b >> 1;
    ae_int16x4 w;
    for (i = 0; i < b/4; i += 1) {
      if (!(i%2)) w = twid[k];
      else {
        w = AE_SEL16_5410(w, w);
        k += 1;
      }
      if (((n-i+b-1)>>lg2) > 0) {
        ae_int32x2 d0, d1;
        ae_int32x2 t;
        ae_int32x2 r0, r1;

        d0 = data[i];
        d1 = data[i+b2];
        for (trip=0; trip < ((n-i+b-1)>>lg2)-1; trip++) {
          r0 = d0 + d1;
          t = d0 - d1;
          r1 = AE_MULFC32X16RAS_H(t, w);

          d0 = data[i+2*b2*(trip+1)];
          d1 = data[i+2*b2*(trip+1)+b2];

          data[i+2*b2*trip] = r0;
          data[i+2*b2*trip+b2] = r1;
        }
        r0 = d0 + d1;
```

```
            t = d0 - d1;
            r1 = AE_MULFC32X16RAS_H(t, w);

            data[i+2*b2*trip] = r0;
            data[i+2*b2*trip+b2] = r1;
          }
        }
      }
    }
```

Before the inner loop, we load the two elements from the first iteration. In the inner loop, we operate on the values loaded from the previous iteration and load the next iteration. After the inner loop, we complete the computation for the last iteration.

With these changes, the compiler is able to schedule the inner loop in four cycles per iteration, the minimum possible due to the load/store bandwidth of the machine.

Note that the same performance could have been achieved by using the `__restrict` attribute on two pointers for the input and output accesses of `data`, rather than manually software pipelining the loop. However, this attribute is only allowed to be used when pointers do not overlap, and the two pointers would in fact overlap.

6. HiFi 3 NatureDSP Signal Library

The HiFi 3 DSP has an associated generic DSP library that can be downloaded from the XPG. The library comes with two Xplorer projects, HiFi 3_NatureDSP_Signal, which is the library and HiFi3_Demo, which is a test application that exercises all the functions in the library. Both are delivered in source form, allowing you to use the library as is, or use it as a starting point for your own development. The library contains functions for FIR filters, IIR filters, basic math functions, matrix operations, and FFTs.

From the library project, under `doc`, is a reference manual, *NatureDSP Signal Library Reference for Tensilica HiFi 3 DSP*, which describes the library and the test program in depth.

7. Implementation Methodology

The HiFi 3 DSP is an optional coprocessor for the Xtensa LX4 (and later versions) core. HiFi 3 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 3 coprocessor.

The last section in this chapter discusses synthesis and place-and-route.

7.1 Configuring a HiFi 3

Configuring a HiFi 3 DSP in the XPG is done by selecting the relevant check box options in the Xplorer Configuration editor, in the Instructions window under the category HiFi Audio Coprocessor:

- HiFi 3 DSP coprocessor instruction family

Developers should also configure the Cache Prefetch Entries from the Interfaces window under the category PIF/Memory Interface Widths. Refer to Figure 7-1 Configuring Hardware Prefetch. A selection of 0 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. 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.

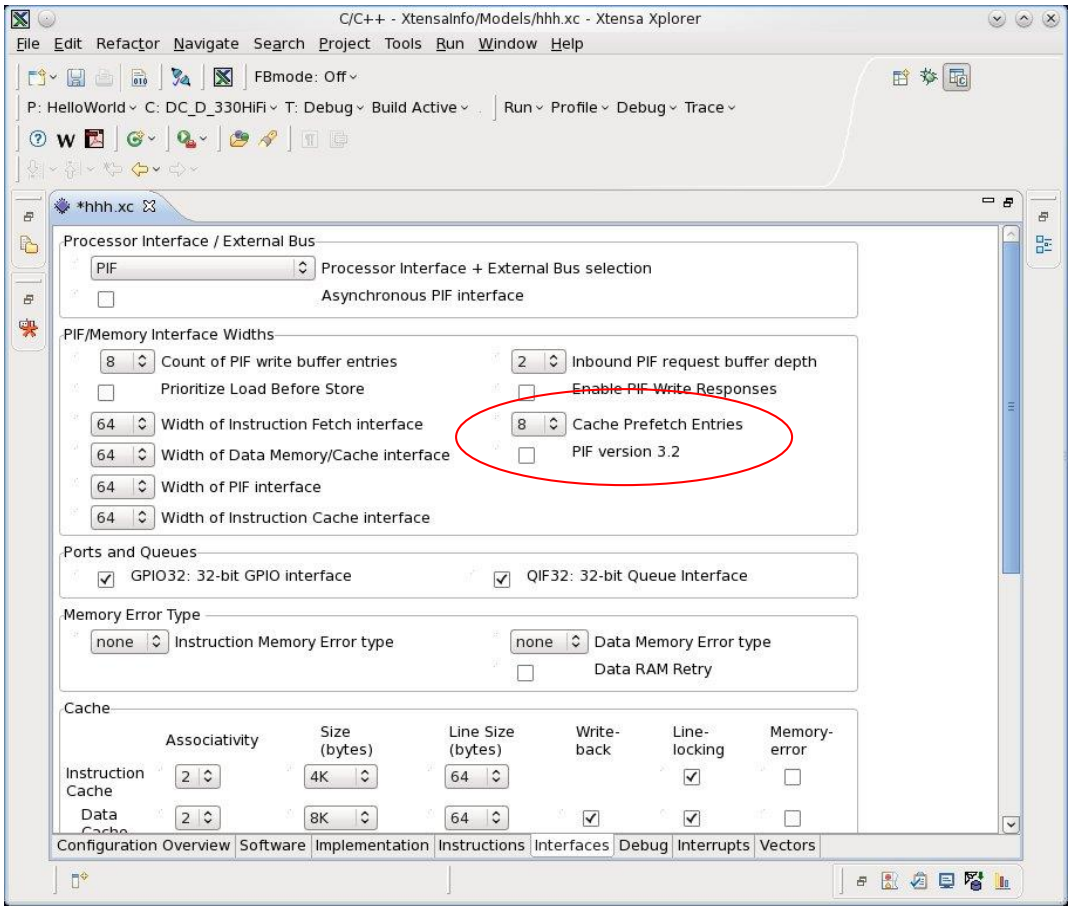


Figure 7-1 Configuring Hardware Prefetch

There are four optional configuration templates provided for HiFi 3, called hifi3_cache, hifi3_localmem, hifi3_5_bd, and hifi3_7_bd. They provide useful starting points for a developer who wishes to either use the configuration as described by the template, or create your own variation. It is useful, but not essential, to start with a HiFi 3 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**.

2. 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**.
3. Enter the configuration name and description. Select any LX processor version. Select one of the templates and click **Finish**.
4. 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 3 DSP as described in the *Xtensa Development Tools Installation Guide*. As you are customizing the processor, remember the following restrictions:

- The HiFi 3 option in the Instructions tab must be selected.
- As HiFi 3 is always coprocessor number 1, the Number of Coprocessors must be at least 2.
- HiFi 3 requires the selection of several options on the Instructions tab, as indicated below.
- The HiFi 3 option is incompatible with the other DSP families.
- Because the HiFi 3 DSP has some 64-bit instruction formats, the maximum instruction width must be 8 bytes and a 64-bit instruction fetch is required. The data interfaces to memory must be at least 64-bits.
- To select the optional floating point unit, check the Vector FP option, which is immediately below the HiFi 3 option.

Once a processor has been configured and downloaded, it can be exercised in simulation.

7.2 XPG Estimation for HiFi 3 Size, Performance and Power

When configuring a HiFi 3 configuration, the Estimation View provides an area, power and maximum MHz estimate for each potential configuration. While configuring a HiFi 3 configuration, keeping this view in mind allows the designer to estimate the hardware implications of their choices.

7.3 Basic HiFi 3 Characteristics

Some of the relevant configuration characteristics of the HiFi 3 coprocessor include:

- HiFi 3 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 8 bytes.
- Data memory interface of at least 8 bytes.
- Little-endian byte ordering (fixed).

7.4 Extending a HiFi 3 with User TIE

HiFi 3 can be extended with user-TIE defining new instructions. These can be assigned to the 24-bit regular instruction format, can use one of the existing 64-bit FLIX instruction formats, or can go in a new user-defined format. To use the existing formats, simply use the TIE `slot_opcode` statement to place the new operation in one of the following: `ae_slot0`, `ae_slot1`, `ae_slot2_0`, `ae_slot2_1`, `ae2_slot0`, `ae2_slot1`, `ae_minislot0`, or `ae_minislot2`. 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 3 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 3 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:

- The AR register file in HiFi 3 has two read ports and one write port, all of which are used in `ae_slot0`. Increasing the number of ports can have a large hardware impact. Creating an operation that requires more than two read or one write operation on the AR register file will increase the number of ports. Creating an operation in another slot of the two main formats that uses any AR register operand will increase the number of ports.
- The slots `ae_minislot0` and `ae_minislot2` are designed to have one AR read port, and one AR read and one AR write port respectively. New operations with those characteristics can benefit from being put in those slots.
- The AE_DR register file has seven read and three write ports. For the three slot format, the first slot utilizes one read and one write port, while the other two slots utilize three read ports and one write port each. Creating an operation that has more operands in either slot will increase the number of ports in the machine and will therefore have a large hardware impact. Such operations should instead be limited to one of the `ae2` slots.
- Single-cycle DSP instructions should read their AE_DR operands in Mstage and write them in Mstage. Two-cycle DSP instructions should ideally read their earliest AE_DR operands in Mstage and write their AE_DR operands in the Mstage+1 stage.

When creating a new FLIX format, consider the following. The existing HiFi 3 formats are defined as follows:

```
format ae_format ae_l64 { ae_slot0, ae_slot1, ae_slot2_0 }
{InstBuf[0] == 1'b1 }

format ae_format1 ae_l64 { ae_slot0, ae_slot1, ae_slot2_1 }
{ InstBuf[0] == 1'b0 && InstBuf[63] == 1'b1}

format ae_format2 ae_l64 { ae2_slot0, ae2_slot1 } {InstBuf[0] ==
1'b0 && InstBuf[63] == 1'b0 && InstBuf[62] == 1'b0 &&
InstBuf[61:60] == 2'b0}

format ae_mini0 ae_l64 {ae_minislot0, ae_minislot1,ae_minislot2}
{InstBuf[0] == 1'b0 && InstBuf[63] == 1'b0 && InstBuf[62] == 1'b1
&& InstBuf[61:41] == 21'b0}
```

Any new format must fit into the empty encoding space. For example, the following code will create a new 37-bit format. Larger formats can be created by leaving undefined some of bits 60 to 41.

```
format user_format ae_l64 {ae_user0, ae_user1} {InstBuf[0]== 1'b0
&&InstBuf[63] == 1'b0&&  InstBuf[62] == 1'b0&&  InstBuf[61] ==1'b1
&&InstBuf[60:41] == 21'b0}
```

7.4.1 Utilizing HiFi 3 Resources

New instructions may utilize existing HiFi 3/EP 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 3 states can be used by using the names listed in Table 2-1 DSP Subsystem State Registers or Table 2-2 Bitstream and Variable-length Encode/Decode Support Subsystem State Registers.

Existing instructions, either core or HiFi 3, can be placed in additional slots 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. Load and store instructions can be added to `ae_slot1` to double the memory bandwidth.

It is not currently possible to share existing HiFi 3 functional resources for new instructions. New multiplier instructions, for example, will have to use their own dedicated multipliers.

7.4.2 Name Space Restrictions for User TIE

All TIE constructs in HiFi 3 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.

When creating a new FLIX format, keep the following in mind. The existing HiFi 3 formats are defined as follows.

```
format ae_format ae_l64 { ae_slot0, ae_slot1, ae_slot2_0 }
{InstBuf[0] == 1'b1 }


format ae_format1 ae_l64 { ae_slot0, ae_slot1, ae_slot2_1 }
{ InstBuf[0] == 1'b0 && InstBuf[63] == 1'b1}


format ae_format2 ae_l64 { ae2_slot0, ae2_slot1 } {InstBuf[0] ==
1'b0 && InstBuf[63] == 1'b0 && InstBuf[62] == 1'b0 &&
InstBuf[61:60] == 2'b0}


format ae_mini0 ae_l64 {ae_minislot0, ae_minislot1,ae_minislot2}
{InstBuf[0] == 1'b0 && InstBuf[63] == 1'b0 && InstBuf[62] == 1'b1
&& InstBuf[61:41] == 21'b0}
```

Any new format must fit into the empty encoding space. For example, the following code will create a new 37-bit format. Larger formats can be created by leaving undefined some of bits 60 to 41.

```
format user_format ae_l64 {ae_user0, ae_user1} {InstBuf[0]==
1'b0&&InstBuf[63] == 1'b0&&  InstBuf[62] == 1'b0&&  InstBuf[61]
==1'b1&&InstBuf[60:41] == 21'b0}
```

Note: On inclusion of the optional floating point unit in HiFi 3, 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.

7.5 Optional Configuration Templates for HiFi 3

Four optional configuration templates are provided for HiFi 3. 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. The salient features of these templates are described briefly below. The optional floating point unit can be added to any of the templates. For more details, select the different templates in the XPG.

- Hifi3_localmem

  A modestly sized configuration with 128 KB each of local data and instruction memory, with no caches. Developers wishing to get better performance on non-tuned applications might consider adding a hardware divider and perhaps the MUL32 multiplier.

- Hifi3_cache

  A modestly sized configuration with an 8KB data cache and 4KB instruction cache. Developers wishing to get better performance on non-tuned applications might consider adding a hardware divider and perhaps the MUL32 multiplier.

- HiFi3_bd5

  A 5-stage HiFi 3 configuration suitable for use in a Blu-Ray disc application. This configuration contains a 32KB, 4-way set associative data cache with 128B lines and an 8KB, 2-way set associative instruction cache with 128B lines. For higher performance, consider increasing the size of the data cache. Developers wishing to minimize bus traffic might consider lowering the cache line size to 64B. Developers wishing to get better performance on non-tuned applications might consider adding a hardware divider and perhaps the MUL32 multiplier.

- HiFi3_bd7

  A 7-stage HiFi 3 configuration suitable for use in a Blu-Ray disc application. This configuration contains a 32KB, 4-way set associative data cache with 128B lines and an 8KB, 2-way set associative instruction cache with 128B lines. For a higher performance, consider increasing the size of the data cache. Developers wishing to minimize bus traffic might consider lowering the cache line size to 64B. Developers wishing to get better performance on non-tuned applications might consider adding a hardware divider and perhaps the MUL32 multiplier.

7.6 Synthesis and Place-and-Route

When the HiFi 3 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 Hardware User's Guide.*

For timing closure between synthesis and place-route, Cadence recommends using an appropriate conservative wire-load model. Because of the data path-intensive nature of the HiFi 3 DSP netlist, the wire-load model may need to be further tuned depending upon the process technology, the foundry, and the library vendor.