cādence®

ConnX BBE64EP

User's Guide

For Xtensa® Tools Version 11.0

Contents

List of Tables

List of Figures

1. Introduction

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

The ConnX BBE family natively supports both real and complex arithmetic operations. For digital signal processing developers, this greatly simplifies development of algorithms dominated by complex arithmetic. In addition to having the SIMD/VLIW DSP core, the ConnX BBE64EP contains a 32-bit scalar processor, ideal for efficient execution of control code. This combined SIMD/VLIW/Scalar design makes the ConnX BBE64EP ideal for building real systems where high computational throughput is combined with complex decision making.

The ConnX BBE64EP is built around a core vector pipeline consisting of sixty-four 16bx16b MACs along with a set of versatile pipelined execution units. These units support flexible precision real and complex multiply-add; bit manipulation; data shift and normalization; data select, shuffle and interleave. The ConnX BBE64EP multipliers and its associated adder and multiplexer trees enable execution of complex multiply operations and signal processing filter structures in parallel. The results of these operations can be extended up to a precision of 40-bits per element, truncated/rounded/saturated or shifted/packed to meet the needs of different algorithms and implementations. The ConnX BBE64EP instruction set is optimized for several DSP kernel operations and matrix multiplies with added acceleration for a wide range of key wireless functions. In addition, the instruction set supports

signed*unsigned multiplies for emulation of 32-bit wide multiplication operations.

The ConnX BBE64EP supports programming in C/C++ with a vectorizing compiler. Automatic vectorization of scalar C and full support for vector data types allows software development of algorithms without the need for programming at assembly level. Native C operator overloading is supported for natural programming with standard C operators on real and complex vector data-types. The ConnX BBE64EP has a Boolean predication architecture that supports a large number of predicated operations. This enables the ConnX BBE64EP compiler to achieve a high vectorization throughput even with complicated functions that have conditional operations embedded in their inner loops.

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

1.1 Purpose of this User's Guide

The ConnX BBE64EP User's Guide provides an overview of the ConnX BBE64EP architecture and its instruction set. It will help ConnX BBE64EP programmers identify commonly used techniques to vectorize algorithms. It provides guidelines to improve software performance through the use of appropriate ConnX BBE64EP instructions, intrinsics, protos and primitives. It also serves as a reference for programming the ConnX BBE64EP in a C/C++ software development environment using the Xtensa Xplorer (XX) Integrated Development Environment (IDE). Additionally, this guide will assist those ConnX BBE64EP users who wish to add custom operations (more hardware) to the ConnX BBE64EP instruction set using Tensilica® Instruction Extension (TIE) language.

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

Throughout this guide, the symbol `<xtensa_root>` refers to the installation directory of the user's Xtensa configuration. For example, `<xtensa_root>` might refer to the directory `/usr/xtensa/<user>/<s1>` if `<user>` is the username and `<s1>` is the name of the user's Xtensa configuration. For all examples in this guide, replace `<xtensa_root>` with the path to the installation directory of the user's Xtensa distribution.

1.2 Installation Overview

To install a ConnX BBE64EP configuration, follow the same procedures described in the *Xtensa Development Tools Installation Guide*. The ConnX BBE64EP comes with a library of examples provided in the XX workspace called `bbe64ep_examples_re_v<version_num>.xws`.

The ConnX BBE64EP 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_bben.h`

👉 **Note:** There is an additional include header file for TI C6x code compatibility, which is discussed in *TI C6x Instrinsics Porting Assistance Library* on page 62. This include file maps TI C6x intrinsics into standard C code and is meant to assist porting only.

1.3 ConnX BBE64EP Architecture Overview

ConnX BBE64EP, a 32-way SIMD processor, has the ability to work on several data elements in parallel at the same time. The ConnX BBE64EP executes a single operation simultaneously across a stream of data elements by means of vector processing. For example, it allows for vector additions through a narrow vector ADD (sum of two 32-element

16-bits/element vectors) or wide vector (sum of two 32-element 40-bits/element vectors) ADD, in parallel. These operations include optimized instructions for complex multiplication and multiply-accumulation, matrix computation, vector division (optional), vector reciprocal & reciprocal square root (optional) and other performance critical kernels.

ConnX BBE64EP has a 5-slot VLIW architecture, in which up to five operations can be scheduled and dispatched in parallel every cycle. This allows the processor to support multiply-accumulate operations of two narrow vectors of sixteen 16-bit complex (32-bit real-imaginary pair) elements in parallel, equivalently sixty-four 16-bit real elements in total, with a load of sixteen complex operands and a store of sixteen complex results in every cycle. To sustain such high memory bandwidth requirements, the ConnX BBE64EP has two asymmetric Load/Store Units (LSUs) which can independently communicate with two local data memories.

For higher efficiency, the ConnX BBE64EP fetches instructions out of a 128-bit wide access to local instruction memory (IRAM). The instruction fetch interface supports a mix of 16/24-bit single instructions and 48/96-bit FLIX (up to 5-way) instructions. The processor can also read from and write to system memory and devices attached to the standard system buses. Other processors or DMA engines can transfer data in and out of the local memories in parallel with the processor pipeline. The processor can also have an instruction cache. The ConnX BBE64EP can also be supplemented with any number of wide, high-speed I/O interfaces (data cache, TIE ports and queues) to directly control devices or hardware blocks, and to move data directly into and out of the processor register files.
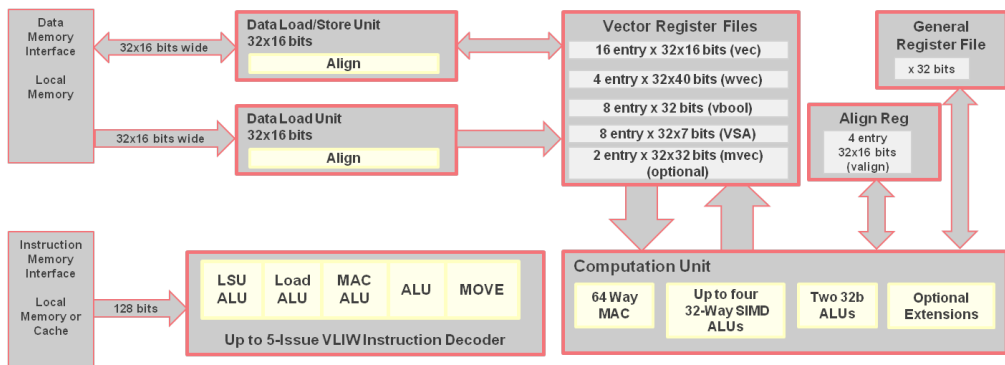
Figure 1: ConnX BBE64EP Architecture

The ConnX BBE64EP architecture uses variable length instructions, with encodings of 16/24-bits for its baseline Xtensa RISC instructions, and 48/96-bits for up to five operations in VLIW that may be issued in parallel. The Xtensa compiler schedules different operations into upto five VLIW slots available. The VLIW instruction Slot-0 is used to issue mostly loads and/or store operations. The instruction Slot-1 is used primarily for load operations using the second LSU while the instruction Slot-4 schedules most move operations. The instruction Slot-2 mostly allows ALU with multiply operations while the instruction Slot-3 is purely for ALU operations present in the ConnX BBE64EP instruction set. However, it is important to note

that the positions of these slots are all interleaved in an actual instruction word much differently than the software view.

The table below illustrates all the basic instruction formats supported by the different operation slots available in the ConnX BBE64EP VLIW architecture. The Xtensa C Compiler (XCC) automatically picks an instruction format that offers the best schedule for an application. When possible, XCC will attempt to pick a 48-bit FLIX format or 16/24-bit standard instruction format to reduce code size.

☞ **Note:**

- Users may optionally add additional 48/96-bit instructions formats as user TIE; see *Extending a ConnX BBE64EP with User TIE* on page 116.
- Format 8 is available only when the *Advanced Precision Multiply/Add* option is present in a ConnX BBE64EP configuration.

Table 1: Basic Instruction Formats

	Slot 0	Slot 1	Slot 2	Slot 3	Slot 4
Format0 - 96b	F0_S0_LdStALU	F0_S1_LdPk	F0_S2_Mul	F0_S3_ALU	-
Format1 - 96b	F1_S0_St	F1_S1_Base	F1_S2_WALUMul	F1_S3_ALU	F1_S4_Move
Format2 - 96b	F2_S0_LdSt	F2_S1_Ld	F2_S2_WALUMul	F2_S3_ALU	F2_S4_Move
Format3 - 96b	F3_S0_St	F3_S1_Ld	F3_S2_ALUMul	F3_S3_ALU	F3_S4_Move
Format4 - 96b	F4_S0_LdSt	F4_S1_LdPkDiv	F4_S2_Mul	F4_S3_ALU	-
Format5 - 96b	F5_S0_St	F5_S1_LdPk	F5_S2_Mul	F5_S3_ALU	F5_S4_Shfl
Format6 - 96b	F6_S0_St	F6_S1_LdPk	F6_S2_Mul	F6_S3_ALUFIRFFT	F6_S4_Move
Format7 - 96b	F7_S0_St	F7_S1_Base	F7_S2_ALUMul	F7_S3_ALU	F7_S4_Shfl
Format8* - 96b	F8_S0_LdSt	F8_S1_LdPk	F8_S2_MulM	F8_S3_ALU	F8_S4_Wacc
Format9 - 48b	F9_S0_LdStALU	F9_S1_None	F9_S2_None	F9_S3_ALU	-
Format10 - 48b	F10_S0_LdStALU	F10_S1_None	F10_S2_Mul	-	-
Format11 - 48b	F11_S0_LdStALU	F11_S1_LdPk	-	-	-
Format12 - 96b	F12_S0_St	F12_S1_Ld	F12_S2_Mul	F12_S3_ALUFIRFFT	F7_S4_Move

1.4 ConnX BBE64EP Instruction Set Overview

The ConnX BBE64EP is built around the baseline Xtensa RISC architecture which implements a rich set of generic instructions optimized for efficient embedded processing. The power of the ConnX BBE64EP comes from a comprehensive set of over 500 DSP and baseband optimized operations excluding the baseline Xtensa RISC operations. A variety of load/store operations support five basic and two special addressing modes for 16/32-bit scalar and 16-bit narrow vector data-types; see *Load & Store Operations* on page 105. A special addressing mode for circular addressing is also available. Additionally, the ConnX BBE64EP supports aligning load/store operations to deliver high bandwidth loads and stores for unaligned data.

Vector data management in the ConnX BBE64EP is supported through operations designed for element-level data selection, shuffle or shift. Further, to easily manage precision in vector data there are packing operations specific to each data-type supported. Additionally, there is an enhanced ISA support for predicated vector operations. Vector level predication allows a vectorizing compiler to exploit deeper levels of inherent parallelism in a program; see *Predicated Vector Operations* on page 101.

Multiply operations supported by the ConnX BBE64EP include real and complex 16bx16b multiply, multiply-round and multiply-add operations. Multiply operations for complex data provide support for conjugate arithmetic, full-precision arithmetic, magnitude computation with saturated/rounded outputs. The ConnX BBE64EP is capable of sixteen complex multiplies per cycle where each complex product involves four real multiplies. The architecture supports extended precision with guard bits on all 40-bit wide vector register data, full support for double precision data and 40-bit accumulation on all MAC operations without any performance penalty. A wide variety of arithmetic, logical, and shift operations are supported for up to thirty-two 40-bit data-words per cycle. The architecture provides special operations that assist matrix-multiply operations.

For algorithm and application specific acceleration, the ConnX BBE64EP can be configured with a number of options:

- FFT
- Symmetrical FIR
- Packed complex matrix multiply
- LFSR & convolutional encoding
- Linear block decoder
- 1D Despreader
- Soft-bit demapping
- Vector divide
- Fast reciprocal & reciprocal square root
- Advanced precision reciprocal & reciprocal square root
- Advanced precision multiply/add

- Inverse log-likelihood ratio (LLR)
- Single and dual peak search

As an example, by configuring a ConnX BBE64EP with the symmetric FIR option and using pairwise real multiply operations, the configured core offers very high performance for a wide range of FIR kernels. The performance, in terms of MACs/cycle, on a ConnX BBE64EP configured for FIR operations is highlighted in the table below.

Table 2: ConnX BBE64EP Multiply Performance

Data	Coefficients	Type	MACs/cycle
Complex	Real	Symmetric	128
Complex	Real	Asymmetric	64
Real	Real	Symmetric	128
Real	Real	Asymmetric	64

The ConnX BBE64EP instruction set is described in further detail later in *ConnX BBE64EP Features* on page 17.

1.5 Programming Model and XCC Vectorization

The ConnX BBE64EP supports a number of programming models -- including standard C/C++, the ConnX BBE64EP-specific integer and fixed-point data types with operator overloads and a level of automated vectorization, scalar intrinsics and vector intrinsics.

The ConnX BBE64EP contains integer and fixed-point data types that can be used explicitly by a programmer to write code. These data types can be used with built-in C/C++ operators or with protos (also called intrinsics) as described later in *Programming a ConnX BBE64EP* on page 45.

Vectorization, which can be manual or automatic, analyzes an application program for possible vector parallelism and restructures it to run efficiently on a given ConnX BBE64EP configuration. Manual vectorization using the ConnX BBE64EP data types and protos is discussed later in *Programming a ConnX BBE64EP* on page 45. The Xtensa C and C++ compiler (XCC) contains a feature to perform automatic vectorization on many ConnX BBE64EP supported data types. The compiler analyzes and vectorizes a program with little or no user intervention. It generates code for loops by using the ConnX BBE64EP operations. It also provides compiler flags and pragmas for users to guide this process. This feature and its related flags and pragmas are documented in the Xtensa C and C++ Compiler User's Guide.

The current generation ConnX cores introduce a new N-way programming model for vector processing using data-types in memory and registers. The N-way model consists of N-

element data groups to facilitate portable vector programming. N-way refers to the natural SIMD size of a ConnX machine. The ConnX BBE64EP supports N=32. A ConnX BBE64EP FLIX instruction can bundle up to five SIMD operations in parallel and each operation is capable of producing up to 'N' results in an independent FLIX lane. Programmers can adopt the N-way abstraction model by using Ctypes, operations and protos in their code that are either N-way or 'N/2'-way (denoted by 'N_2' in Ctypes, operation and proto names). N-way model on the ConnX BBE64EP supports N and N_2 as an abstract representation of 32 and 16 respectively. This makes code written using the N-way model easy to port to other architectures with a different SIMD size or example, other SIMD variants in the BBE EP Cores family -BBE32EP. Although not recommended, users can also use equivalent `ctype`s and protos whose names explicitly have '32' or '16' in place of 'N' or 'N_2' respectively.

For automatic vectorization in an N-way environment, however, it is required that all vector types inside a loop have the same SIMD width. In the ConnX BBE64EP, this exposes an important distinction between real and complex vector types. While the real vector types are supported by a SIMD width of N by the core, the complex vector types are supported by two natural SIMD widths – N (32) and 'N_2' (16). For instance, xb_vecN_2xc16 is an 16-way complex vector type stored in a single register while xb_vecNxc16 is a 32-way complex vector type stored in a pair of registers. It is recommended to use the xb_vecN_2xc16 type when programming only complex vector types, but when programming real and complex vector types together, the xb_vecNxc16 vector type is suggested for complex data.

2. ConnX BBE64EP Features

Topics:

- *ConnX BBE64EP Register Files*
- *ConnX BBE64EP Architecture Behavior*
- *Operation Naming Conventions*
- *Fixed Point Values and Fixed Point Arithmetic*
- *Data Types Mapped to the Vector Register File*
- *Data Typing*
- *Multiplication Operation*
- *Vector Select Operations*
- *Vector Shuffle Operations*
- *Block Floating Point*
- *Complex Conjugate Operations*
- *FLIX Slots and Formats*

2.1 ConnX BBE64EP Register Files

The ConnX BBE64EP has a partitioned set of register files to provide high bandwidth with less register bloat. Larger register files permit deeper software pipelining and reduced memory traffic. The first partition consists of a set of sixteen 512-bit general purpose narrow vector registers (vec) that can hold operands and results of SIMD operations. Each vec register can hold either thirty-two 16-bit real or sixteen 32-bit complex elements, depending on how the register is used by the software. The second partition consists of a set of four 1280-bit wide vector registers (wvec) each of which can hold thirty-two 40-bit elements. The wvec registers can hold either 32-bit elements with eight guard bits or 40-bit elements. The interface to each data memory is 512-bits wide and this makes all loads/stores for wide wvec registers take place through intermediate moves into the narrow 512-bit vec registers.

The ConnX BBE64EP register file organization also has four 512-bit Alignment registers (4x16N - where N = SIMD width; for ConnX BBE64EP, N = 32) and eight 224-bit specialized variable Shift/Select registers (8x7N) for use with the select category of operations that can manipulate the contents of the vec register file. There are also eight 32-bit Boolean registers (32xN) for flexible SIMD and VLIW predication.

Lastly, an optional register file - a two entry mvec - is added when the *Advanced Precision Multiply/Add* (advprec) option is configured in a ConnX BBE64EP core. The 32-way 32-bit mvec registers are used to hold results of only those operations belonging to the advprec option. Furthermore, floating-point operands (23-bit element vectors) for advanced precision operations are held as vectors of 7-bit exponents in vsa registers paired with 16-bit mantissa in narrow vec registers.

8 vsa
Registers
32x7b

16 vec
Registers
32x16b

4 w vec Registers 32x40b

2 mvec Registers 32x32b (Optional)

4 valign Registers 32x16b

8 vbool
Registers
16b

State registers

A/B/C/D: 32x16b, FFT / symmetric FIR option
MODE: 5b, RANGE: 4b, FFT option
BMUL: 32x32b, BMUL_ACC: 32x1b, LFSR option
PQUO0/1: 256b, PREM0/1: 256b, vector divide option
FLUSH_TO_ZERO: 1b, advanced precision mul/add option
CBEGIN/CEND: 32b
MAX[2]: 512b, MAXIDX[2]: 192b, IDX: 11b, peak search option

Figure 2: ConnX BBE64EP Register Files
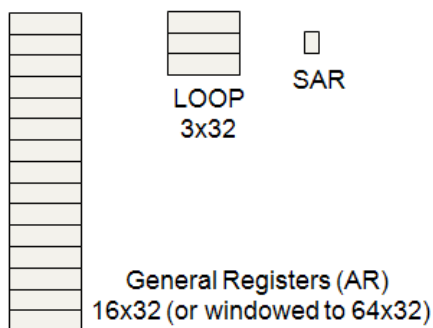
Figure 3: Base Xtensa ISA Register Files

On configuring a ConnX BBE64EP core with certain options, additional state registers are built into the processor core. These special state registers can also be used to hold contents of the vector registers temporarily without going to and from memory. The `RUR_<state_name>` and `WUR_<state_name>` operations are used to read from and write to these state registers. Once initialized, users are advised against using these states in order not to inadvertently overwrite the contents of these states. Following is a list of fixed and configuration dependent states in ConnX BBE64EP

- `BBE_STATE<A|B|C|D>` - 512-bit states added by the *FFT* or *symmetric FIR* options. And, they are shared when both the options are present in a ConnX BBE64EP configuration.
- `BBE_RANGE` - A 4-bit state added by the *FFT* option.
- `BBE_MODE` - A 5-bit state added by the *FFT* option.
- `BBE_BMUL_STATE` - A 1024-bit state added by the *LFSR & Convolutional Encoding* option.
- `BBE_BMUL_ACC` - A 32-bit state added by the *LFSR & Convolutional Encoding* option.
- `BBE_PQUO<0|1>` & `BBE_PREM<0|1>` - 256-bit states added by the *Vector Divide* option.
- `BBE_FLUSH_TO_ZERO` - A 1-bit state added by the *Advanced Precision Multiply/Add* option.
- `CBEGIN` & `CEND` - These two states are present in the base ConnX BBE64EP core to support circular addressing; they are used to initialize begin and end addresses in a circular buffer. More details about circular addressing are provided in *Load & Store Operations* on page 105.

More details on the configurable options for ConnX BBE64EP can be found in *Configurable Options* on page 67 and in the ISA HTML of the corresponding operations.

Since the ConnX BBE64EP supports 512-bit loads and stores between registers and memory, at 800MHz, this provides over 100GBps of combined data memory bandwidth between ConnX BBE64EP and the two load/store units.

For data transfers from memory to registers during Loads:

- 16-bit elements in memory are loaded as 16-bit elements in narrow `vec` registers

- 32-bit elements in memory are loaded to narrow `vec` registers first and then expanded to 40-bits in `wvec` wide registers after a move
- Signed loads sign-extend the elements while unsigned loads zero-fill inside vector registers

On the other hand, for data transfers from registers to memory during Stores:

- 16-bit elements in registers are stored as 16-bit elements in memory
- 40-bit elements in registers are saturated to 32-bit elements first and then moved to the narrow `vec` registers for a store
- Signed stores are saturated to signed 16/32-bit MAX and MIN, while unsigned stores are saturated to unsigned 16/32-bit MAX.

The ConnX BBE64EP provides protos, also called intrinsics, to store and restore register spills. The names of protos that store data element(s) to memory on register spills have a '_storei' suffix, for example xb_vecN_2xc40_storei, xb_vecNxcq9_30_storei, xb_c16_storei, etc. These protos use either appropriate store operations in the case of 16-bit data spills from narrow `vec` registers or use an appropriate combination of move and store operations in the case of 40-bit data spills from wide `wvec` registers or other special registers. The 40-bit data element(s) in the `wvec` registers are first zero-extended to 64-bit data element(s) before a move to the narrow `vec` register and then followed by four 16-bit stores, per data element, from the narrow `vec` register to memory.

Conversely, the ConnX BBE64EP has complementary protos to restore data elements from memory back into corresponding registers. The names of these protos have a '_loadi' suffix, for example, uint32_loadi, xb_vecN_2xcq19_20_loadi, xb_vecNx32U_loadi, etc.. To restore spills back into the narrow `vec` register, the restoring spill protos use appropriate load operations, for the 16-bit data-types. However, to restore spills back into the wide `wvec` register, the restoring spill protos use an appropriate combination of load and move operations, for the 40-bit data-types. In the latter case, the restoring process is through four loads of narrow vectors. Once loaded, these are concatenated, and for each element, 40-bits out of 64-bits are written into the wide `wvec` vector registers. It can be noticed that the previously zero-extended 40-bit elements are now truncated back from 64-bits prior to move from the narrow `vec` register to the wide `wvec` register. Note that In general, these special protos are meant to be used by the compiler, and not by regular programmers, who in general should not need them.

2.2 ConnX BBE64EP Architecture Behavior

The ConnX BBE64EP architecture provides guard bits in its data path and wide register file to avoid overflow on ALU and MAC operations. The typical data flow on this machine is:

1. Load data into a narrow unguarded vector register file.
2. Compute and accumulate into the wide vector register file with guard bits.

3. Store data in narrow format by packing down with truncation or saturation (via the `vec` register file).

The ConnX BBE64EP has thirty-two 16bx16b multipliers, each of which produces a 32-bit result to be stored in the guarded 40-bit register elements. This allows up to eight guard bits for accumulation in the 40-bit register elements. For real multiplication, only 32 multipliers are used while for complex multiplication, all 64 are used.

The first load/store unit supports all load and store operations present in the ConnX BBE64EP ISA and the baseline Xtensa RISC ISA. The second load unit supports a limited set of commonly used vector load and aligning vector load operations present in the ConnX BBE64EP ISA. This second load unit does not support any store operations.

Additionally, the ConnX BBE64EP offers operations that pack results. For instance, the 'PACK' category of operations extract 16-bits from 40-bit wide data in `wvec` register elements and saturate results to the range [-2^{15} .. 2^{15}-1]. Most 'MOVE' operations that move 16-bit or 32-bit results from high-precision to low-precision data widths saturate.

Unlike some architectures, the ConnX BBE64EP does not set flags or take exceptions when operations overflow their 40-bit range, or if saturation or truncation occurs on packing down from a 40-bit wide vector to a 16-bit vector in registers.

2.3 Operation Naming Conventions

The ConnX BBE64EP uses certain naming conventions for greater consistency and predictability in determining names of Ctypes, operations and protos. In general, ConnX BBE64EP operation names follow the pattern below:

`<prefix>_<op_class>[<num_SIMD>[X<element_width>]][<other_identifiers>]`

where,

- `prefix` := `BBE_`
- `op_class` := one or more letters identifying a simple or compound class of operation(s)
- `num_SIMD` := SIMD width of the core, here 'N' (=32) and 'N_2' (=16)
- `element_width` := bits in each SIMD element of an operand, here 8/16/32/40 bits
- `other_identifiers` := one or more letters specifying a sub-class of operation(s) in the context of the `op_class`

    - `C` := Complex or circular
    - `R` := Real
    - `U` := Unsigned
    - `S` := Signed or saturation
    - `I` := Immediate
    - `X` := Indexed
    - `P` := Post-increment or pairwise

- `T` := True (predicate)
- `F` := False (predicate)
- `J` := Conjugate
- `H` := High
- `L` := Low
- `B` := Boolean
- `U` := Unsigned
- `A` := Address register `ar`
- `V` := Narrow vector register `vec`
- `W` := Wide vector register `wvec`
- `BR` := Boolean register `br`
- `BV` := Boolean vector register `vbool`
- `VS` := Vector shift/select register `vsa`
- `SF` := Spread Factor
- `CS` := Code sets
- `INT` := Integer
- `ALIGN` := Vector alignment register `valign`

Following are a few specific examples to illustrate the naming conventions listed above.

BBE_LVNX16_I	**L**oads a **v**ector of thirty-two **16**-bit signed elements from memory into a narrow vector register. The base address used for the load is contained in address register ars. This base address is added to an offset. The _I extension represents an **i**mmediate offset such that the memory address is a multiple of 64 bytes.
BBE_MOVVA16C	This operation performs a single replicating **mov**e of a 32-bit **c**omplex element, consisting of two 16-bit data elements, from the address register **A**R to a narrow vector register **V**EC. The destination register, source register, size and nature (complex) of the data are identified by V, A, 16 and C respectively.
BBE_LSNX16_IP	This operation performs a 1-way signed **s**calar **l**oad of a **16**-bit element from memory into a narrow vector register. The rest of the output narrow vector register is zero filled. An address register (AR) holds the base address used for the load and this base address is updated using an immediate as an offset after the load is done. The _IP extension indicates a **p**ost-operation update for the address register after the sum of base address and the **i**mmediate offset.
BBE_MULNX16PACKL	32-way signed real **mul**tiply of two narrow **16**-bit vectors to produce a 512-bit combined narrow vector product. The PACKL variant of operations **pack** the thirty-two full-precision 32-bit results, which would otherwise be stored in a 32x40-bit wide vector register, back into thirty-two 16-bit integer results stored in a 512-bit narrow vector register. PACKL grabs the **l**ow order bits of the results — thus a presumed integer and truncates the high order bits of the result without using a vector shift/select (`vsa`) register. This kind of pack extracts the lowest 16-bits of each of the thirty-two

| | | | intermediate result elements and writes them out without shifting, rounding, or saturation. |

The following table broadly categorizes a set of commonly used ConnX BBE64EP operations. It also provides a brief description to show the specific naming conventions used in each of the categories.

Table 3: Sample Categories of Operations

Category	Mnemonic	Type	Description
LOAD	BBE_L	V	Load vector of 16b elements
		P	Load complex pair of 16b elements
		S	Load scalar 16b element
		A	Load unaligned vector of 16 elements
		B	Load vector of 1b elements
MOVE	BBE_MOV		Move between core, vector and state registers. Predicated versions available for some.
		A	Move to AR register
		BR	Move to BR (core boolean register)
		BV	Move to `vbool` (vector boolean register)
		IDX	Indexed move to vector register.
		PA	Move from AR to vector register as fractional Q5.10 with saturation and replication
		QA	Move from AR to vector register as fractional Q15 with saturation and replication
		PINT	Move immediate to vector register as fractional Q5.10 with saturation, replication
		QINT	Move immediate to vector register as fractional Q15 with saturation, replication
		QUO	Move quotients from input vector register to vector-divide quotient state-register
		REM	Move remainders from input vector register to vector-divide remainder state-register
		S	Move to state-register

Category	Mnemonic	Type	Description
		SV	Move to `vec` (narrow vector register) with saturation
		SW	Move to `wvec` (wide vector register) with sign-extension
		V	Move to `vec` (narrow vector register)
		VS	Move to vector shift/select register (`vsa`)
		W	Move to `wvec` (wide vector register)
	BBE_MALIGN		Move alignment register
	BBE_MB		Move between two `vbool` (vector boolean) registers
MULTIPLY	BBE_MUL		Multiply operation. Predicated versions available for some.
		A	Multiply-accumulate operation
		C	Multiply complex, high precision
		J	Multiply complex conjugate, high precision
		PACKQ	Multiply signed real, Q15 fractional results with high-order 16-bits, with saturation
		PACKP	Multiply signed real, results converted to Q5.10 fractional form, with saturation
		PACKL	Multiply signed real, integer results with lower-order 16-bits
		CPACKQ, CPACKP & CPACKL	Similar to PACKQ, PACKP & PACKL respectively, but with complex signed operands
		JCPACKQ, JCPACKP & JCPACKL	Similar to PACKQ, PACKP & PACKL respectively, but with complex and complex-conjugate signed operands
		PR	Multiply with pairwise sum and accumulation
		R	Multiply with variable round producing wide (40-bit) results with sign-extension
		SGN	Multiply multiplicand by the sign of the multiplier element-wise

Category	Mnemonic	Type	Description
		S	Multiply producing wide (40-bit) results with sign-extension
		UU	Multiply (unsigned*unsigned) producing wide (40-bit) results with sign-extension
		US	Multiply (unsigned(first input)*signed(second input)) producing wide (40-bit) results with sign-extension
PACK	BBE_PACK	L	Packs 40-bit signed real vector to 16-bit elements with truncation to keep lower bits
		P	Packs 40-bit Q19.20 signed fractional vector to 16-bit Q5.10 elements
		S	Packs low-precision integer vector
		Q	Packs vector of Q9.30 in `wvec` into Q15 results in `vec`
		V	Packs 40-bit `wvec` to low-precision `vec` based on signed shift amount in vector shift/select register (`vsa`)
	BBE_UNPK	P, Q, S, U	32-way unpack of 16-bit data in `vec` to 40-bit data in `wvec`
SELECT	BBE_SEL		Select thirty-two 16-bit elements from two input vectors using vector shift/select register (`vsa`)
		I	Select thirty-two 16/40-bit vector from two input vectors using an immediate value
		PR	Real element-wise right shift with shift count using an immediate value
		PC	Complex element-wise right shift with shift count using an immediate value
	BBE_SELS		Single real element select from `vec`/`wvec` register into element-0 of `vec`/`wvec` register
		C	Single complex element select from `vec`/`wvec` register into complex element-0 of `vec`/`wvec` register

Category	Mnemonic	Type	Description
	BBE_DSEL	I	Interleave or de-interleave real/complex elements from two input narrow vectors into two output narrow vector using an immediate value to specify a select pattern
SHUFF**L**E	BBE_SHFL		Shuffle 32-element narrow (16-bit) vector using vector selection register
		I	Shuffle 32-element 16/40-bit vector using an immediate value to specify a shuffle pattern
		VS	Shuffle elements from a vector shift/select (vsa) register to an output vsa register using an immediate value to specify a shuffle pattern
STORE	BBE_S	V	Store vector of 16b elements
		P	Store pair (complex) of 16b elements
		S	Store scalar 16b element
		A	Store and align vector of 16b elements
		B	Store vector of 1b Boolean elements
ABSOLUTE	BBE_ABS		Absolute value
ADD	BBE_ADD		Vector add
AND	BBE_AND		Vector bitwise Boolean AND
CONJUGATE	BBE_CONJ		Complex Conjugate
DIVISION (optional)	BBE_DIV	U	Unsigned vector divide
		S	Signed vector divide
SOFT-BIT **D**E**MAP** (optional)	BBE_SDMAP		3GPP and IEEE soft-constellation bit demap
EQUALITY	BBE_EQ		Vector equality check
	BBE_NEQ		Vector inequality check
EXTRACT	BBE_EXTR		Extract one real/complex element into AR (address register)
		B	Extract one real/complex element into BR (core Boolean register)

Category	Mnemonic	Type	Description
	BBE_EXTRACTB		Extract elements of a single `vbool` register into two `vbool` registers
FFT (optional)	BBE_FFT		FFT type operations
FLOATING-**P**OINT **RECIP**ROCAL (optional)	BBE_FP	RECIP	Signed 16-bit mantissa + 7-bit exponent pseudo-floating point reciprocal approximation
FLOATING-**P**OINT **R**ECIPROCAL **SQ**UARE-**R**OO**T** (optional)	BBE_FP	RSQRT	16-bit mantissa + 7-bit exponent pseudo-floating point reciprocal square-root approximation
NEGATE	BBE_NEG		Signed negate
		S	Signed saturating negate
INT**ER**L**EAV**E	BBE_ITLV		Bit-by-bit interleave of two narrow vectors into one narrow vector
JOIN	BBE_JOIN		Join boolean vectors
MAGNITUDE	BBE_MAGI		Interleaved magnitude of complex vectors high precision
		PACKQ	Interleaved magnitude of complex fractional vectors, Q15 results with the high-order 16-bits
		PACKL	Interleaved magnitude of complex integer vectors, integer results with the lower-order 16-bits
		PACKP	Interleaved magnitude of complex integer vectors, results packed to 16-bits in Q5.10 format after shifting
	BBE_MAGIA		Interleaved magnitude of complex integer vectors, result multiply-accumulated
	BBE_MAGIR		Interleaved magnitude of complex integer vectors, variable rounded results
MAX	BBE_MAX		Max of vector
	BBE_MAXU		Max of unsigned vector
	BBE_BMAX		Max of vector generating Boolean mask

Category	Mnemonic	Type	Description
MIN	BBE_MIN		Min of vector
	BBE_BMIN		Min of vector generating Boolean mask
NAND	BBE_NAND		NAND of `vec`/`wvec` vectors
NSA	BBE_NSA		Normalize shift amount
		E	Normalise shift amount truncated to even value
		C	Complex normalise shift amount
		U	Unsigned normalise shift amount
OR	BBE_OR		OR of `vec`/`wvec`/`vbool` vectors
POLYNOMIAL	BBE_POLY		Polynomial evaluation
RECIPROCAL (optional)	BBE_RECIP		16-bit vector reciprocal approximation
RECIPROCAL **SQ**UARE-**R**OO**T** (optional)	BBE_RSQRT		Compute normalization and table lookup factors for advanced reciprocal square root
REDUCTION	BBE_R	ADD	Vector sum reduction
	BBE_R	MAX, MIN	Vector signed reduction maximum/minimum
	BBE_RB	MAX, MIN	Vector signed reduction maximum/minimum along with boolean vector indicating location of maximum/minimum
REPLICATE	BBE_REP		Replicate elements
ROU**ND**	BBE_RND	ADJ	Rounding add, using variable round amounts from vector shift/select register (`vsa`)
		SADJ	Symmetric rounding add, using variable round amounts from vector shift/select register (`vsa`)
SATURATE	BBE_SAT	S	Saturate signed vector
		U	Saturate unsigned vector
SEQUENCE	BBE_SEQ		Create sequence of integer values from 0 to (64-1) in output `vec`/`wvec` register

Category	Mnemonic	Type	Description
SHIFT **L**EFT/ **R**IGHT	BBE_SLL		Logical left shift, amount of signed shift as input from `vsa` register
		I	Logical left shift, amount of unsigned shift as immediate input
	BBE_SLS		Saturating shift, amount of signed shift as input from `vsa` register
		I	Saturating shift, amount of unsigned shift as immediate input
	BBE_SLA		Arithmetic left shift, amount of signed shift as input from `vsa` register
	BBE_SRA		Arithmetic right shift, amount of signed shift as input from `vsa` register
		I	Arithmetic right shift, amount of unsigned shift as immediate input

Within each class, sub-conventions are used to describe types of operations, data type layouts, signed/unsigned, data formatting, addressing modes, etc., depending on the operation or its class. *Table 4: Types of Load/Store Operations* on page 30 has an abbreviated list of this information; for detailed information about the operations, see the HTML Instruction Set Architecture page. The easiest method to access this page is from the configuration overview in Xtensa Xplorer:

1. Double-click on the ConnX BBE64EP configuration in the System Overview to open the Configuration Summary window.
2. Click View Details button in the rightmost column for the installed build to open a Configuration Overview window.
3. Select All Instructions to open a complete list of instruction descriptions.

Table 4: Types of Load/Store Operations

Type	Operation Prefix	Proto Support for Ctypes	Addressing
Load **V**ector	BBE_LV	NX{16, 16U, Q15, Q5_10}	N_2XC{16, Q15, Q5_10}	(I|IP|X|XP| IC)
Load **A**ligning Vector	BBE_LA	NX{16, 16U, Q15, Q5_10}	N_2XC{16, Q15, Q5_10}	(IP|IC)
Load **A**ligning **V**ariable	BBE_LAV	NX{16, 16U, Q15, Q5_10}	N_2XC{16, Q15, Q5_10}	(XP)

Type	Operation Prefix	Proto Support for Ctypes	Addressing
Load **S**calar	BBE_LS	NX{16, 16U, Q15, Q5_10} | N_2XC{16, Q15, Q5_10}	(I|IP|X|XP)
Load **P**air	BBE_LP	NX{16, 16U, Q15, Q5_10} | N_2XC{16, Q15, Q5_10}	(I|IP|X|XP)
Load **B**oolean	BBE_LB	{N, N_2}	(I|IP)
Store **V**ector	BBE_SV	NX{16, 16U, Q15, Q5_10} | N_2XC{16, Q15, Q5_10}	(I|IP|X|XP|IC)
Store **A**ligning	BBE_SA	NX{16, 16U, Q15, Q5_10} | N_2XC{16, Q15, Q5_10}	(IP|IC)
Store **A**ligning Variable	BBE_SAV	NX{16, 16U, Q15, Q5_10} | N_2XC{16, Q15, Q5_10}	(XP)
Store **A**ligning **V**ariable - FFT **R**ange Update	BBE_SAVR	NX{16, 16U, Q15, Q5_10} | N_2XC{16, Q15, Q5_10}	(XP)
Store **S**calar	BBE_SS	NX{16, 16U, Q15, Q5_10} | N_2XC{16, Q15, Q5_10}	(I|IP|X|XP)
Store **P**air	BBE_SP	NX{16, 16U, Q15, Q5_10} | N_2XC{16, Q15, Q5_10}	(I|IP|X|XP)
Store **B**oolean	BBE_SB	{N, N_2}	(I|IP)

Assembly and C Naming Conventions

In the ConnX BBE64EP, all the operation and proto names (excluding the baseline Xtensa RISC ISA) begin with the prefix "BBE_". The full names are then constructed using a number of logical fields separated by underscores ("_"). Syntactically, operation names having either underscores ("_") or periods (".") may be used in an assembly-level program. However, the C language does not allow the use of periods (".") in the names of operation identifiers. Therefore, in referring to protos for base Xtensa operations in C, periods in names are substituted by underscores. This document uses the assembly-correct names for protos i.e. with underscores after a prefix and periods in the body of a proto name. However, note that all the programming examples in *Programming a ConnX BBE64EP* on page 45 list protos using the C-correct form with only underscores and no periods. Thus, in the online ISA HTML documentation (see *On-Line ISA, Protos and Configuration Information* on page 133), when searching for information about an operation or a proto having period(s) in its name, it is useful to search for a variation of the name with underscore(s) replacing period(s).

> **Note:** Only some base Xtensa operations have periods (".") in their names. ConnX BBE64EP operation names do not have any periods.

2.4 Fixed Point Values and Fixed Point Arithmetic

The ConnX BBE64EP contains operations for implementing fixed point arithmetic. This section describes the representation and interpretation of fixed point values as well as some operations on fixed point values.

Representation of Fixed Point Values

A fixed point data type Qm.n contains a sign bit, some number of bits m, 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+1 bits are the integer part expressed as a signed 2s complement number. If the binary value is interpreted as a 2s complement signed integer, converting from the binary value to a fixed point number requires dividing the integer by 2n.

Thus, for example, the 40-bit Q9.30 number 1.5 is represented as 0x00 6000 0000.

Bit Range (Field)	39 (Sign)	38 (Integer) 30	29 (Fraction) 0
Size	(1 bit)	(9 bits)	(30 bits)
Binary	0	0 0000 0001	10 0000 0000 0000 0000 0000 0000 0000
Hex	0x0	0x1	0x2000 0000

and the 16-bit Q15 number -0.5 is represented as 0xc000

Bit Range (Field)	15 (Sign)	14 (Fraction) 0
Size	(1 bit)	(15 bits)
Binary	1	100 0000 0000 0000
Hex	0x1	0x4000

When m = 0, we write Qn. When n=0, the data type is just a signed integer and we call the data type a signed m+1-bit integer or intm+1.

The ConnX BBE64EP operations use Q15, Q5.10, Q1.30, Q11.20, Q19.20 and Q9.30 data types, described in more detail, as follows:

- Q15 - 16-bit fixed point data type with 1 sign bit and 15 bits of fraction, to the right of the binary point. 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)
- Q5.10 - 16-bit fixed point data type with 1 sign bit, 5 integer bits to the left of the binary point and 10 bits of fraction to the right of the binary point.
- Q1.30 - 32-bit fixed point data type with 1 sign bit, 1 integer bit to the left of the binary point and 30 bits of fraction to the right of the binary point.
- Q11.20 - 32-bit fixed point data type with 1 sign bit, 11 integer bits to the left of the binary point and 20 bits of fraction to the right of the binary point.
- Q19.20 - 40-bit fixed point data type with 1 sign bit, 19 integer bits to the left of the binary point and 20 bits of fraction to the right of the binary point.
- Q9.30 - 40-bit fixed point data type with 1 sign bit, 9 integer bits to the left of the binary point and 30 bits of fraction to the right of the binary point.

Arithmetic with Fixed Point Values

When multiplying fixed point numbers Qm0.n0 * Qm1.n1, with a standard signed integer multiplier, the natural result of the multiple will be a Qm.n data type where n = n0+n1 and m = m0+m1+1. So multiplying a Q15 by a Q15 generates a Q1.30. Since the ConnX BBE64EP has 16-bit x 16-bit multipliers, it multiplies two Q15 values to produce a Q1.30 result sign extended to Q9.30 in a 40-bit register element. To convert Q1.30 to Q9.30 requires a sign extension that fills a 40-bit register. Similarly, a multiplication between two Q5.10 types produces a Q11.20 result that is sign extended to Q19.20 to fill a 40-bit register.

2.5 Data Types Mapped to the Vector Register File

A number of different data types are defined for the vector register files. These data types are also referred to as ctypes after the name of the TIE construct that creates them.

Scalar Data Types Mapped to the Vector Register File

The signed integer data types are:

- xb_int16 - A 16-bit signed integer stored in the 16-bit vector register element.
- xb_int32 - A 32-bit signed integer stored in the least significant 32 bits of a 40-bit vector register element. The upper 8 bits are sign extended from bit 31.
- xb_int40 - A 40-bit signed integer stored in a vector register element.

The complex integer data types are:

- xb_c16 - A signed complex integer value with 16-bit imaginary and 16-bit real parts. The real and imaginary pair is stored in two 16-bit elements of a vector register file, with the real part in the lower significant element.
- xb_c32 - A signed complex integer value with 32-bit imaginary and 32-bit real parts. The real and imaginary pair are stored in two 40-bit elements of a vector register file, with the real part in the less significant element. The values are sign extended from bit 31.
- xb_c40 - A signed complex integer value with 40-bit imaginary and 40-bit real parts. The real and imaginary pair occupies two 40-bit elements of a vector register file, with the real part in the lower significant element.

In addition to the integer data types, the ConnX BBE64EP also supports a programming model with explicit fixed-point (fractional) data types. The software programming model provides C intrinsics and operator overloading that use these data types. All the scalar ones that fit in a single vector register are listed below.

The six real fixed-point data types are:

- xb_q15 - A signed Q15 data type stored in a 16-bit vector register element.
- xb_q5_10 - A signed Q5.10 data type that occupies a 16-bit vector register element.
- xb_q1_30 - A signed Q1.30 data type is stored in the least significant 32-bits of a 40-bit vector register element. The rest of the bits are sign extended from bit 31.

- xb_q11_20 - A signed Q11.20 data type is stored in the least significant 32-bits of a 40-bit vector register element. The rest of the bits are sign extended from bit 31.
- xb_q19_20 - A signed Q19.20 data type that uses all 40 bits of a vector register element.
- xb_q9_30 - A signed Q9.30 data type that uses all 40 bits of a vector register element.

The six complex fixed-point data types are:

- xb_cq15 - A signed complex fixed-point value with Q15 imaginary and Q15 real parts. The real and imaginary pair is stored in two 16-bit elements of a vector register file, with the real part in the lower significant element.
- xb_cq5_10 - A signed complex fixed-point value with Q5.10 imaginary and Q5.10 real parts. The real and imaginary pair occupy two 16-bit elements of a vector register file, with the real part in the lower significant element.
- xb_cq1_30 - A signed complex fixed-point value with Q1.30 imaginary and Q1.30 real parts. The real and imaginary pair is stored in two 40-bit elements of a vector register file, with the real part in the lower significant element. The values are sign extended from bit 31.
- xb_cq11_20 - A signed complex fixed-point value with Q11.20 imaginary and Q11.20 real parts. The real and imaginary pair occupy two 40-bit elements of a vector register file, with the real part in the lower significant element. The values are sign extended from bit 31.
- xb_cq19_20 - A signed complex fixed-point value with Q19.20 imaginary and Q19.20 real parts. The real and imaginary pair occupies two 40-bit elements of a vector register file, with the real part in the lower significant element.
- xb_cq9_30 - A signed complex fixed-point value with Q9.30 imaginary and Q9.30 real parts. The real and imaginary pair occupy two 40-bit elements of a vector register file, with the real part in the lower significant element.

Vector Data Types Mapped to the Vector Register File

Following is a list of the vector register files and their corresponding vector data types. The double vector data types are physically stored in a pair of registers.

Table 5: Vector Data Types Mapped to Vector Register Files

Register Vector	Vector 16	Vector 40	Double Vector 16/40
Integer	xb_vecNx16	xb_vecNx40	xb_vecNx16
Q15	xb_vecNxq15	xb_vecNxq9_30	xb_vecNxcq15
Q5.10	xb_vecNxq5_10	xb_vecNxq19_20	xb_vecNxcq5_10
Complex Integer	xb_vecN_2xc16	xb_vecN_2xc40	xb_vecNxc40
Complex Q15	xb_vecN_2xcq15	xb_vecN_2xcq9_30	xb_vecNxcq9_30
Complex Q5.10	xb_vecN_2xcq5_10	xb_vecN_2xcq19_20	xb_vecNxcq19_20

Memory Vector	Vector 16	Vector 32	Double Vector 16/32
Integer	xb_vecNx16	xb_vecNx32	xb_vecNx16
Unsigned Integer	xb_vecNx16U	xb_vecNx32U	
Q15	xb_vecNxq15	xb_vecNxq1_30	xb_vecNxcq15
Q5.10	xb_vecNxq5_10	xb_vecNxq11_20	xb_vecNxcq5_10
Complex Integer	xb_vecN_2xc16	xb_vecN_2xc32	xb_vecNxc32
Complex Q15	xb_vecN_2xcq15	xb_vecN_2xcq1_30	xb_vecNxcq1_30
Complex Q5.10	xb_vecN_2xcq5_10	xb_vecN_2xcq11_20	xb_vecNxcq11_20

2.6 Data Typing

The following tables (Table 2 4 through Table 2 7) list the complete data type naming convention for both operations and protos. These basic data types are used to understand the naming convention. All operations can be accessed using intrinsics with the same name as the operation using one of the base data types xb_vecNx for full vector and xb_vecN_2x for half-vector (for use with complex types). Alternatively, intrinsics are provided to give the same functionality of each operation mapped appropriately to the set of data types.

The ConnX BBE64EP supports standard C data-types, referred to as memory data-types, for 16/32-bit scalar and vector data elements in memory. These scalar and vector data-elements are mapped to the ConnX BBE64EP register file as register data-types. While the 16-bit memory types are mapped to 16-bit register types, the 32-bit memory types are mapped to 40-bit register types along with 8 guard bits.

Following are scalar, mem vector, and vector register data type details defined for C.

Table 6: Scalar Memory Data Types

Scalar	Scalar 16	Scalar 32
Integer	xb_int16	xb_int32
Q15	xb_q15	xb_q1 _30
Q5.10	xb_q5 _10	xb_q11 _20
Complex int	xb_c16	xb_c32
Complex Q15	xb_cq15	xb_cq1 _30
Complex Q5.10	xb_cq5 _10	xb_cq11 _20

Table 7: Scalar Register Data Types

Scalar	Scalar 16	Scalar 40
Integer	xb_int16	xb_int40
Q15	xb_q15	xb_q9 _30
Q5.10	xb_q5 _10	xb_q19 _20
Complex int	xb_c16	xb_c40
Complex Q15	xb_cq15	xb_cq9 _30
Complex Q5.10	xb_cq5 _10	xb_cq19 _20

Table 8: Vector Memory Data Types

Memory Vector	Vector 16	Vector 32	Double Vector 16/32
Integer	xb_vecNx16	xb_vecNx32	xb_vecNxc16
Unsigned int	xb_vecNx16U	xb_vecNx32U	
Q15	xb_vecNxq15 (Q15)	xb_vecNxq1 _30 (Q1.30)	xb_vecNxcq15 (Q15)
Q5.10	xb_vecNxq5 _10 (Q5.10)	xb_vecNxq11 _20 (Q11.20)	xb_vecNxcq5 _10 (Q5.10)
Complex int	xb_vecN _2xc16	xb_vecN _2xc32	xb_vecNxc32
Complex Q15	xb_vecN _2xcq15 (Q15)	xb_vecN _2xcq1 _30 (Q1.30)	xb_vecNxcq1 _30 (Q1.30)
Complex Q5.10	xb_vecN _2xcq5 _10 (Q5.10)	xb_vecN _2xcq11 _20 (Q11.20)	xb_vecNxcq11 _20 (Q11.20)

Table 9: Vector Register Data Types

Temp Vector	Vector 16	Vector 40	Double Vector 40
Integer	xb_vecNx16	xb_vecNx40	xb_vecNxc16
Q15	xb_vecNxq15 (Q15)	xb_vecNxq9 _30 (Q9.30)	xb_vecNxcq15 (Q15)
Q5.10	xb_vecNxq5 _10 (Q5.10)	xb_vecNxq19 _20 (Q19.20)	xb_vecNxcq5 _10 (Q5.10)
Complex int	xb_vecN _2xc16	xb_vecN _2xc40	xb_vecNxc40

Temp Vector	Vector 16	Vector 40	Double Vector 40
Complex Q15	xb_vecN _2xcq15 (Q15)	xb_vecN _2xcq9 _30 (Q9.30)	xb_vecNxcq9 _30 (Q9.30)
Complex Q5.10	xb_vecN _2xcq5 _10 (Q5.10)	xb_vecN _2xcq19 _20 (Q19.20)	xb_vecNxcq19 _20 (Q19.20)

2.7 Multiplication Operation

The ConnX BBE64EP supports a variety of multiplication operations that use a set of sixty-four 16bx16b SIMD multipliers and associated adders provided as computational resources. These multiplier enabled operations can be broadly classified as – multiply, multiply-accumulate, multiply and round, multiply-subtract, unsigned multiply and signwise multiply operation.

The table below lists the various flavors of multiplication supported by the ConnX BBE64EP. Each of these operations may have multiple protos each of which support a specific data-type related to the operation. Refer to the ISA HTML of an operation to find all the protos using that operation.

Table 10: Types of ConnX BBE64EP Multiplication Operations

Type	Operation	Description
Real signed multiply	BBE_MULNX16	32 16-bit operands; 40-bit sign-extended results
Real signed multiply with low-precision results	BBE_MULNX16PACKL	32 16-bit operands; 16-bit low-precision integer results
	BBE_MULNX16PACKP	32 16-bit operands; 16-bit Q5.10 fractional results
	BBE_MULNX16PACKQ	32 16-bit operands; 16-bit Q15 fractional results
Complex signed multiply	BBE_MULNX16C	16 16-bit complex operands; 40-bit sign-extended results
Complex signed multiply with low-precision results	BBE_MULNX16CPACKL	16 16-bit complex operands; 16-bit low-precision integer results
	BBE_MULNX16CPACKP	16 16-bit complex operands;

Type	Operation	Description
		16-bit Q5.10 fractional results
	BBE_MULNX16CPACKQ	16 16-bit complex operands; 16-bit Q15 fractional results
Complex conjugate signed multiply	BBE_MULNX16J	16 16-bit complex operands; 40-bit sign-extended results
Complex conjugate signed multiply with low-precision results	BBE_MULNX16JPACKL	16 16-bit complex operands; 16-bit low-precision integer results
	BBE_MULNX16JPACKP	16 16-bit complex operands; 16-bit Q5.10 fractional results
	BBE_MULNX16JPACKQ	16 16-bit complex operands; 16-bit Q15 fractional results
Complex and complex conjugate signed multiply with low-precision results	BBE_MULNX16JCPACKL	16 16-bit complex operands; 16-bit low-precision integer results
	BBE_MULNX16JCPACKP	16 16-bit complex operands; 16-bit Q5.10 fractional results
	BBE_MULNX16JCPACKQ	16 16-bit complex operands; 16-bit Q15 fractional results
Special complex signed multiply with pairwise reduction add	BBE_MULNX16PC_0	Used to accelerate matrix multiplication. Refer to ISA HTML.
	BBE_MULNX16PC_1	
Real signed multiply-accumulate	BBE_MULANX16	32 16-bit operands; 40-bit sign-extended results
Complex signed multiply-accumulate	BBE_MULANX16C	16 16-bit complex operands; 40-bit sign-extended results
Complex conjugate signed multiply-accumulate	BBE_MULANX16J	16 16-bit complex operands; 40-bit sign-extended results

Type	Operation	Description
Special complex signed multiply accumulate with pairwise reduction add	BBE_MULANX16PC_0	Used to accelerate matrix multiplication.
	BBE_MULANX16PC_1	Refer to ISA HTML.
Real signed multiply and variable round	BBE_MULRNX16	32 16-bit operands; 40-bit sign-extended results
Complex signed multiply and variable round	BBE_MULRNX16C	16 16-bit complex operands; 40-bit sign-extended results
Complex conjugate signed multiply and variable round	BBE_MULRNX16J	16 16-bit complex operands; 40-bit sign-extended results
Special complex signed multiply with pairwise reduction add and rounding	BBE_MULRNX16PC_0	Used to accelerate matrix multiplication.
	BBE_MULRNX16PC_1	Refer to ISA HTML.
Real signed multiply-subtract	BBE_MULSNX16	32 16-bit operands; 40-bit sign-extended results
Complex signed multiply-subtract	BBE_MULSNX16C	16 16-bit complex operands; 40-bit sign-extended results
Complex conjugate signed multiply-subtract	BBE_MULSNX16J	16 16-bit complex operands; 40-bit sign-extended results
Unsigned with signed multiply	BBE_MULUSNX16	32 16-bit operands with first input unsigned and second input signed; 40-bit sign-extended results
Unsigned with signed multiply-accumulate	BBE_MULUSANX16	
Unsigned with signed multiply with rounding	BBE_MULUSRNX16	
Unsigned with unsigned multiply	BBE_MULUUNX16	
Unsigned with unsigned multiply-accumulate	BBE_MULUUANX16	
Unsigned with unsigned multiply with rounding	BBE_MULUURNX16	
Special signwise multiply	BBE_MULSGNNX16	Refer to ISA HTML.

2.8 Vector Select Operations

The vector select operations (BBE_SEL) allow elements from two source vectors to be selectively copied into a destination vector. With this general definition of selective element transfer, it is easy to implement replication, rotation, shift, extraction and interleaving with the same basic BBE_SEL type of operation.

To setup a selective transfer, the BBE_SELNX16 operation takes two 32x16-bit narrow source vectors and one 32x16-bit narrow target vector; along with the vector select register (vsa). The vector select register contains a user defined 32-valued pattern to define the required transfer. The 32 values in the vector select register can be [0,…,63] referring to the indices of the combination of the two source vectors.

On the other hand, the BBE_SELNX16I operation allows common selection patterns without having to set the vector select register that is required by the BBE_SELNX16 operation. The BBE_SELNX16I operation selects thirty-two 16-bit elements from a pair of narrow vector registers and produces a single narrow vector output. The nature of selection pattern is specified through an immediate value iSel. To use selection on 40-bit wide vector data-types, the BBE_SELNX40I operation provides support for a limited set of pre-defined selections. Eight preset selection patterns can be chosen with an appropriate immediate iSel to rotate or interleave wide vector data-types. Refer to the ISA HTML of BBE_SELNX40I for a list of all the available types and permitted immediate values iSel.

Vector Initialization and Some Additional Select Patterns

The ConnX BBE64EP also contains a special move operation, BBE_MOVVINX16, which is used for vector initialization based on an immediate value. The table below illustrates what is possible.

Table 11: Vector Initialization Operation Arguments

Symbolic Name	Immediate Argument	Value Assigned	Description
BBE_MOVVI_INT16_M1	-1	32'hFFFF_FFFF	int16 -1
BBE_MOVVI_ZERO	0	32'h0000_0000	zero
BBE_MOVVI_INT16_1	1	32'h0001_0001	int16 +1
BBE_MOVVI_INT16_MININT BBE_MOVVI_Q15_M1	2	32'h8000_8000	int16 MININT Q15 -1
BBE_MOVVI_INT16_MAXINT	3	32'h7FFF_7FFF	int16 MAXINT
BBE_MOVVI_Q5_10_1	4	32'h0400_0400	Q5.10 +1
BBE_MOVVI_Q5_10_M1	5	32'hFC00_FC00	Q5.10 -1

Symbolic Name	Immediate Argument	Value Assigned	Description
BBE_MOVVI_LOWER_CHAR	6	32'h00FF_00FF	Lower Char
BBE_MOVVI_UPPER_CHAR	7	32'hFF00_FF00	Upper Char
BBE_MOVVI_C16_1	8	32'h0000_0001	C16 +1
BBE_MOVVI_C16_I	9	32'h0001_0000	C16 +i
BBE_MOVVI_C16_M1 BBE_MOVVI_EVEN_SELECT	10	32'h0000_FFFF	C16 -1 even select
BBE_MOVVI_C16_MI BBE_MOVVI_ODD_SELECT	11	32'hFFFF_0000	C16 -i odd select
BBE_MOVVI_CQ15_M1	12	32'h0000_8000	CQ15 -1
BBE_MOVVI_CQ15_MI	13	32'h8000_0000	CQ15 -i
BBE_MOVVI_CQ5_10_1	14	32'h0000_0400	CQ5.10 +1
BBE_MOVVI_CQ5_10_I	15	32'h0400_0000	CQ5.10 +i
BBE_MOVVI_CQ5_10_M1	16	32'h0000_FC00	CQ5.10 -1
BBE_MOVVI_CQ5_10_MI	17	32'hFC00_0000	CQ5.10 -i

2.9 Vector Shuffle Operations

The vector shuffle operations (BBE_SHFL) in the ConnX BBE64EP can shuffle any 32-elements from a source vector register into a target vector register. These shuffle operations are often useful in performing vector compression, expansion, reordering and in preparation for matrix multiplication.

For shuffle-based operations, some examples of the support present in the ConnX BBE64EP are the ability to:

- perform a full SELECT/SHUFFLE and a specialized SELECT/SHUFFLE (pattern specified by an immediate value) in a single cycle
- perform one 2x2 interleave per cycle with two vector inputs and two vector outputs
- perform a 3x3 interleave in multiple steps
- support specialized shuffles for implementing FIR filters with two vector inputs and two vector outputs

The BBE_SHFLNX16 operation shuffles elements in a narrow source vector based on a shuffle pattern specified by the vector selection register `vsa`. The shuffle pattern is set by storing desired indices [0,…,N-1] in the N values in the `vsa` register.

Similar to the select operations, the BBE_SHFLNX16I and the BBE_SHFLNX40I operations use an immediate iSel to specify a shuffle pattern while not using the `vsa` register; and the

support for wide vector types are a limited set of shuffle patterns. Refer to the ISA HTML for a list of all available patterns and corresponding immediate values iSel.

2.10 Block Floating Point

When applications can operate across a wide numerical range, the programmer may wish to implement "block floating point", the adjustment of an entire data set based on determining the actual range of values, normalizing the data-set to maximize the number of bits of precision, and readjusting the range later in the computation. The operations BBE_NSANX16, BBE_NSANX16C, BBE_NSANX40, BBE_NSANX40C, BBE_NSAUNX16 and BBE_NSAUNX40 facilitate block floating point. These operations calculate the left shift amount required to normalize each element to a 16-bit or 40-bit value. The result is returned in another register, and can be used with the operation BBE_SLLNX16 and BBE_SLLNX40 to normalize the elements of a single vector.

The corresponding operations for xb_vecN_2x40 data are BBE_NSANX40 and BBE_NSAUNX40. To implement block floating point, BBE_NSANX16 or BBE_NSAUNX16 is applied to the entire data set, and the minimum normalization shift is calculated using BBE_MINNX16. Then the same shift is applied to the entire data set, typically using BBE_SLLNX16. This shift amount can be used at a later stage of the computation, with BBE_SRANX16 to return the data to non-normalized form.

2.11 Complex Conjugate Operations

A complex conjugate multiply of two complex numbers a and b is the multiplication of a by the complex conjugate of b and is defined as: result = (a.real + j a.imag) * (b.real - j b.imag) Where b = (b.real + j * b.imag) and complex conjugate of b = (b.real - j * b.imag). We use "J" in instruction names to indicate complex conjugate operations. And JC for regular multiply and complext conjugate multiply as a pair.

There are many variants of complex-conjugate multiply operations:

- BBE_MULNX16J – high-precision complex conjugate multiply
- BBE_MULNX16JCPACKL – low-precision integer result after complex and complex conjugate multiply
- BBE_MULNX16JCPACKP – low-precision fractional (Q5.10) result after complex and complex conjugate multiply
- BBE_MULNX16JCPACKQ – low-precision fractional (Q15) result after complex and complex conjugate multiply
- BBE_MULNX16JCPACKQ – low-precision fractional (Q15) result after complex and complex conjugate multiply
- BBE_MULNX16JPACKL - low-precision integer complex conjugate multiply
- BBE_MULNX16JPACKP - low-precision fractional (Q5.10) complex conjugate multiply

- BBE_MULNX16JPACKQ - low-precision fractional (Q15) complex conjugate multiply
- BBE_MULANX16J – high-precision complex conjugate multiply-add
- BBE_MULRNX16J – high-precision complex conjugate multiply and variable round
- BBE_MULSNX16J – high-precision complex conjugate multiply-subtract

For example, BBE_MULNX16J takes as input two 32X16-bit vectors, in which the real and imaginary portions of sixteen complex numbers are interleaved. It produces sixteen complex results in one 32X40-bit register where the real and imaginary parts the complex results are interleaved.

2.12 FLIX Slots and Formats

The ConnX BBE64EP can issue up to five operations in a single instruction bundle using Xtensa LX FLIX (VLIW) technology. It contains scalar and vector SIMD operations. The ConnX BBE64EP is implemented with a number of 48/96-bit formats in addition to the standard 24-bit and optional 16-bit instruction formats in the Xtensa LX architecture. Each basic 48/96-bit format can bundle five operations into its five separate FLIX slots respectively.

Instruction List – Showing Slot Assignments

The instruction slot assignment list, showing operations assigned to the various slots of the various formats, is automatically generated and available for use in the on-line configuration documentation for the ConnX BBE64EP. Consult ISA HTML for information about this on-line information. Since this list is automatically generated from the machine description, it is comprehensive and up to date including the user's choice of configuration options.

3. Programming a ConnX BBE64EP

Topics:

Cadence® recommends two important Xtensa manuals to read and become familiar with before attempting to obtain optimal results by programming the ConnX BBE64EP:

- 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.

The ConnX BBE64EP is based on SIMD (Single Instruction/Multiple Data) techniques for parallel processing. 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.

This chapter describes several approaches to programming the ConnX BBE64EP and explores the capabilities of automated instruction inference and vectorization, and cases where the use of intrinsic-based programming is appropriate.

To use the ConnX BBE64EP data types and intrinsics in C, please include the appropriate top-level header file by using the following preprocessor directive in the source file:

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

The xt_bben.h include file is auto-generated through the ConnX BBE64EP core build process and contains #defines for programmers to conditionalize code (see *Conditional Code* on page 59). Furthermore, xt_bben.h includes the lower-level include file (xt_bbe64.h) which has more core specific ISA information. It is worth noting that the header file – xt_bben_verification.h contains additional #defines but used only for ISA verification

purposes; and thus are not documented for programmers' use.

The ConnX BBE64EP processes both fixed-point and integer data. The basic data element, xb_vecNx16, is 16-bits wide, and a vector consists of thirty-two such 16-bit elements. Thus, an input vector in memory is 512-bits wide. The ConnX BBE64EP also supports xb_vecNx32 and xb_vecNx32U, a wider 32-bit data element with sixteen such elements stored in a 512-bit vector. This double-width data type is typically generated as the result of a multiply or multiply/accumulate operation on 16-bit data elements.

As described earlier, during a load operation, the ConnX BBE64EP loads 16-bit scalar values to 16-bits and it expands 32-bit scalar values to 40-bits. The ConnX BBE64EP supports both signed and unsigned data types in memory and provides a symmetrical set of load and store operations for both these data types. For signed data, a load results in sign extension from 32-bits to 40-bits. For unsigned data, loads result in zero extension.

The ConnX BBE64EP architecture philosophy follows the data flow of loading data at a lower precision into a narrow unguarded `vec` register file, computing and accumulating into the 40-bit/element guarded wide `wvec` register file (8-guard bits), and storing results back at a lower precision via narrow `vec` register file. The ConnX BBE64EP ALU/MAC operations wrap on overflow and either saturate going from 40-bits to 32-bits or PACK going from 32-bits to 16-bits.

> **Note:** There are no direct loads into or stores from the wide `wvec` register file. All `wvec` loads/stores take place through narrow `vec` registers.

Automatic type conversion is also supported between vector types and the associated scalar types, for example between xb_vecNx16 and short and between xb_vecNx40 and int. Converting from a scalar to a vector type replicates the scalar into each element of the vector. Converting from a vector to a scalar extracts the first element of the vector.

The C compiler also supports the xb_vecNx40 type, a vector of thirty-two, 40-bit elements. This data type is useful because logically the multiplication of two variables

of type xb_vecNx16 returns a variable of type xb_vecNx40. The xb_vecNx40 type is implemented using the 1280-bit wide register file.

Note that as well as the basic types discussed above; there are a number of other types, such as complex integer, fractional, complex fractional, etc. And as discussed in *ConnX BBE64EP Features* on page 17, there are alternative prototypes (protos) for the ConnX BBE64EP operations that deal with these types. This allows both a more intuitive programming style and richer capabilities for more automated compiler inference and vectorization using these additional data types.

Some examples of the various types are:

- xb_cq1_30: Memory scalar, one complex pair of Q1.30 type, 1-bit for integer and 30-bit fractional, sign bit implied, total 64 bits
- xb_vecN_2xcq9_30: Register vector, sixteen complex elements of Q9.30 type, each 9-bit integer and 30-bit fractional, sign bit implied, total 1280 bits
- xb_vecNx32U: Memory vector, thirty-two unsigned real elements of 32-bit integer type, total 1024 bits
- xb_vecN_2xcq15: Memory vector, sixteen complex elements of Q15 type, total 512 bits

3.1 Programming in Prototypes

As part of its programming model, the ConnX BBE64EP defines a number of operation protos (prototypes or intrinsics) for use by the compiler in code generation, for use by programmers with data types other than the generic operation support, and to provide compatibility with related programming models and DSPs. For programmers, the most important use of protos is to provide alternative operation mappings for various data types. If the data types are compatible, one operation can support several variations without any extra hardware cost. For example, protos allow vector operations to support all types of real and complex integer and fixed-point data types.

The ConnX BBE64EP contains several thousand protos. Some protos are meant for compiler usage only, although advanced programmers may find a use for them in certain algorithms. Many of these protos are called BBE_OPERATOR, and in general should not be used as manual intrinsics; look for the simpler protos for the function instead. The complete list of protos can be accessed via the ISA HTML.

In the ISA HTML package contained within the ConnX BBE64EP configuration, the proto_list.html page categorizes all the protos available for programmer use. The following categories of protos are listed in the easy-to-navigate HTML page:

- BBE PRIMARY PROTOS
- COMPILER USE PROTOS
- DATA MANAGEMENT PROTOS
- OPERATOR OVERLOAD PROTOS
- TYPE CONVERSION PROTOS
- TYPE-CASTING PROTOS
- USER REGISTER PROTOS
- ZERO ASSIGNMENT PROTOS

Extract Protos

Extract protos are used to convert between types in the ConnX BBE64EP programming model. The extract protos are used to produce new vectors from existing vectors. Note that their naming and prototype argument list follows certain conventions. The destination type follows "BBE_EXTRACT", which is followed by the source type. The destination argument is first in the argument list, followed by the source argument, and then any special arguments such as immediates.

There are some special destination-type names used that are not normal ConnX BBE64EP types. For example, "R" means real (extract the real parts from a vector of complex variables), and "I" means imaginary (extract the imaginary parts from a vector of complex variables).

The ConnX BBE64EP also provides protos to extract the real and imaginary parts of a complex vector - BBE_EXTRACTI_FROMC16 and BBE_EXTRACTR_FROMC16.

Example: The following code shows the use of an Extract proto; in this case, to extract the real values from a complex vector.

```
// Complex conjugate multiplies
xb_c40 cxprod = ycin[i] * ~ hcin[i];
xb_c40 hcxabs = hcin[i] * ~ hcin[i];
// Extract real magnitude squared for channel
xb_int40 habs = BBE_EXTRACTR_FROMC40(hcxabs);
```

Note that the ConnX BBE64EP scalar types are used here, as the compiler does automatic vectorization and inference.

Combine Protos

Combine protos are also used to convert between types in the ConnX BBE64EP programming model. Note that their naming and prototype argument list follows certain conventions. The destination type follows "BBE_COMBINE", which is followed by the source type. The destination arguments are first in the argument list, followed by the source arguments, and then any special arguments such as immediates. The "BBE_COMBINE" protos are used to produce a single integer or fixed-point complex vector from two non-complex input vectors of same types.

There are some special source type names used that are not normal ConnX BBE64EP types. For example, "I" means imaginary, "R" means real, and "Z" means zero. Thus, "IR" means imaginary-real (combine a vector of imaginary parts and a vector of real parts into a vector of interleaved complex numbers). "ZR" means zero-real — that is, all imaginary parts are zero, thus this will generate a vector of interleaved complex numbers whose imaginary parts are all zero.

Complementary to the BBE_EXTRACT set of protos, the ConnX BBE64EP provides a set of "BBE_JOIN" protos that can be used to produce a wide integer or fixed-point complex vector by joining two narrow complex vectors of same type. While the BBE_JOIN protos require narrow complex vectors as inputs, the BBE_COMBINE protos combine non-complex vectors into a complex vector.

Example: In the following code, a combine from a vector of real values and zeroes for the imaginaries is used to create two complex vectors.

```
// N-way inverse for scaling, note divide is integer division
// Also convert to complex multiplicand to use in later steps
xb_vecNx16 hinv = BBE_DIVNX32(numinv, numinv, hsabs);
xb_vecNxc16 hcxhi, hcxlo;
BBE_COMBINENXC16_FROMZR(hcxhi, hcxlo, hinv);
```

Move Protos

The ConnX BBE64EP additionally provides "BBE_MOV" category of protos used to cast between data-types. For vector inputs, the BBE_MOV protos cast from one data-type to

another related data-type without changing the size of the bit-stream. These protos don't have a cycle cost and are helpful in multi-type programming environments. While the data is interpreted differently, the length of the vector remains the same within the register file. This category of protos usually follows the format

```
BBE_MOV<after_type>FROM<before_type>
```

Examples: BBE_MOVNX16_FROMNXQ5_10, BBE_MOVNX40_FROMN_2XCQ9_30, etc.

Operator Overload Protos

The proto HTML documentation also includes the Operator protos for compiler usage. Note that Cadence® recommends usage by advanced programmers only.

When using intrinsics, you may pass variables of different types than expected as long as there is a defined conversion from the variable type to the type expected by the intrinsic.

Operator overloading is only supported if there is an intrinsic with types that exactly match the variables. Implicit conversion is not allowed since with operator overloading you are not specifying the intrinsic name, and the compiler does not guess which intrinsics might match. The resultant intrinsic is necessary for operator overloading, but there is no advantage in calling it directly.

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 easier to debug. These formats allow memory and vector register data contents to be displayed in a variety of formats. In addition, users can define their own display formats. Variables are displayed by default in a format matching their vector data types. Registers are by default always displayed as xb_vecNx16 types, but you can change the format to any other format.

Some examples of these display formats for a 512-bit variable are: xb_vecNx16m displays hex and decimal for each real element of vector

```
(720, -736, 721, -725, 739, 728, -725, -758, 727, 722, 724, -722, -720, -724, 722, 716, 720,
-736, 721, -725, 739, 728, -725, -758, 727, 722, 724, -722, -720, -724, 722, 716) = (0x02d0,
0xfd20, 0x02d1, 0xfd2b, 0x02e3, 0x02d8, 0xfd2b, 0xfd0a, 0x02d7, 0x02d2, 0x02d4, 0xfd2e,
0xfd30, 0xfd2c, 0x02d2, 0x02cc, 0x02d0, 0xfd20, 0x02d1, 0xfd2b, 0x02e3, 0x02d8, 0xfd2b,
0xfd0a, 0x02d7, 0x02d2, 0x02d4, 0xfd2e, 0xfd30, 0xfd2c, 0x02d2, 0x02cc)
```

xb_vecN_2xc16m displays above as sixteen complex numbers instead

```
(720i+-736, 721i+-725, 739i+728, -725i+-758, 727i+722, 724i+-722, -720i+-724, 722i+716, 720i
+-736, 721i+-725, 739i+728, -725i+-758, 727i+722, 724i+-722, -720i+-724, 722i+716) = (0x02d0i
+0xfd20, 0x02d1i+0xfd2b, 0x02e3i+0x02d8, 0xfd2bi+0xfd0a, 0x02d7i+0x02d2, 0x02d4i+0xfd2e,
```

```
0xfd30i+0xfd2c, 0x02d2i+0x02cc, 0x02d0i+0xfd20, 0x02d1i+0xfd2b, 0x02e3i+0x02d8, 0xfd2bi
+0xfd0a, 0x02d7i+0x02d2, 0x02d4i+0xfd2e, 0xfd30i+0xfd2c, 0x02d2i+0x02cc)
```

Note that the complex numbers are displayed as they are laid out in memories and registers, since the ordering of each pair is (imaginary, real).

3.3 Operator Overloading and Vectorization

Common ConnX BBE64EP operations can be accessed in C or C++ by applying standard C operators to the ConnX BBE64EP data types. There are many operator overloads defined for the various types, and the compiler will infer the correct ConnX BBE64EP vector operation in many cases depending upon the data-types used.

In addition, if the scalar types are used in loops, the compiler will often be able to automatically vectorize and infer or overload. That is, a loop using the ConnX BBE64EP scalar types may turn into a loop of the ConnX BBE64EP vector operations that is as tightly packed and efficient as manual code using ConnX BBE64EP intrinsics.

For operations that do not map to standard operators, intrinsics can be used. Several intrinsics can map to the same underlying operation by means of one or more operations. There are different intrinsics for different argument types. Intrinsics are particularly useful in special operations such as select and polynomials which cannot be easily inferred by the compiler. It's often best to use intrinsincs to select a specific load/store flavor for operations where the compiler may not always pick the right load/store operation for the best schedule. When using intrinsics, the compiler still handles type checking and data movement, and schedules operations into available slots of a FLIX format.

To understand the limits of compiler automatic inferencing and overloading and vectorization, the rest of this chapter discusses the various programming styles and provides several examples showing how the ConnX BBE64EP can be programmed, including the example results.

32-bit memory ctypes are treated equivalently to corresponding 40-bit register Ctypes. Even though operator overload protos are not explicity defined for 32-bit Cytpes, the equivalence lets programmers use the corresponding 40-bit ctype protos. This means xb_vecNx32 = xb_vecNx32 + xb_vecNx32 works the exact same way as xb_vecNx40 = xb_vecNx40 + xb_vecNx40. The following tables show the correspondance of 32/40-bit ctypes for the purpose of operator overload.

32-bit Ctype	40-bit Ctype
xb_q11_20	xb_q19_20
xb_vecNxq11_20	xb_vecNxq19_20
xb_cq11_20	xb_cq19_20
xb_vecN_2xcq11_20	xb_vecN_2xcq19_20

32-bit Ctype	40-bit Ctype
xb_vecNxcq11_20	xb_vecNxcq19_20
xb_c32	xb_c40
xb_cq1_30	xb_cq9_30
xb_vecNxc32	xb_vecNxc40
xb_vecNxcq1_30	xb_vecNxcq9_30
xb_vecN_2xc32	xb_vecN_2xc40
xb_vecN_2xcq1_30	xb_vecN_2xcq9_30
xb_vecNx32	xb_vecNx40
xb_vecNx32U	xb_vecNx40
xb_vecNxq1_30	xb_vecNxq9_30
xb_int32	xb_int40
xb_q1_30	xb_q9_30

3.4 Programming Styles

It is typical for programmers to have to put in some effort on their code, especially legacy code, to make it run efficiently on a vectorized DSP. For example, there may be changes required for automatic vectorization, or the algorithm may need some work to expose concurrency so vector instructions can be used manually as intrinsics. For efficient access to data items in parallel, or to avoid unaligned loads and stores, which are less efficient than aligned load/stores, some amount of data reorganization (data marshalling) may be necessary.

Four basic programming styles that can be used, in increasing order of manual effort, are:

- Auto-vectorizing scalar C code
- C code with vector data types (manually vectorized)
- Use of C instrinsic functions along with vector data types and manual vectorization
- Use of DSP Nature Library

One strategy is to start with legacy C code or to write the algorithm in a natural style using scalar types (possibly using the ConnX BBE64EP special scalars - xb_int16, xb_c16, xb_int40, xb_c40; e.g., for Q15 or complex or complex Q15 data). Once the correctness of a fixed-point code using the ConnX BBE64EP scalar data-types is determined the limits of what automatic/manual vectorization with operator overloading achieves can be investigated. By profiling the code, computationally intensive regions of code can be identified and the limits of automated vectorization determined.

These parts of code that could be vectorized further can then be modified manually to improve performance. Finally, the most computationally intensive parts of the code can be improved in performance through the use of C intrinsic functions.

At any point, if the performance goals for the code have been met, the optimization can cease. By starting with what automation can do 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 next sections.

Auto-Vectorization

Auto-vectorization of scalar C code using ConnX BBE64EP types 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. Tensilica® recommends 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.
- Data should be aligned to 64-byte boundaries because of the 512-bit load/store interface. The XCC compiler will naturally align arrays to start on 64-byte boundaries. But the compiler cannot assume that pointer arguments are aligned. The compiler needs to be told that data is aligned by one of the following methods:
  - Using global or local arrays rather than pointers
  - Using #pragma aligned(<pointer>, n)
  - 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 -LNO:simd_v feedback can assist 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.

Operator Overloading and Inferencing

Many basic C operators work in conjunction with both automatic and manual vectorization to infer the right intrinsic:

- + addition
- - subtraction: both unary (additive inverse) and binary
- * multiplication: real and complex
- & bitwise AND
- ^ bitwise XOR
- | bitwise OR
- << bitwise left shift
- >> bitwise right shift
- ~ for ConnX BBE64EP complex types, a complex conjugate operation is inferred; otherwise, bitwise NOT or one's complement operator
- < less than
- <= less than or equal to
- > greater than
- >= greater than or equal to
- == equal to

The next section illustrates how they work in conjunction.

Vectorization and Inferencing Examples

The examples provided in this section for the ConnX BBE64EP include code snippets which illustrate the extent of and limits to automatic compiler capabilities. This includes three very simple algorithms: vector add, vector dot product and matrix multiply, and several scalar data types: int, short, etc.

In all the following examples, the VEC_SIZE used is 1024 and ARRAY_SIZE was 16.

Almost all these examples vectorize. For example, int vector add:

```
int ai[VEC_SIZE], bi[VEC_SIZE], ci[VEC_SIZE];
void vec_add_int()
{
  int i;

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

```
    ci[i] = ai[i] + bi[i];
  }
}
```

On compilation, the following code is produced:

```
{ bbe_lvnx16_i    v0, a2, -64; bbe_lvnx16_xp    v1, a2, a6; bbe_movswv    wv0, v0, v1; nop }
{ bbe_svnx16_xp   v2, a4, a6; bbe_lvnx16_i_n    v2, a3, 0; bbe_addnx40    wv3, wv0, wv3; nop;
nop }
{ bbe_lvnx16_i    v0, a3, 64; bbe_movsvwl    v1, wv1; bbe_movswv    wv0, v1, v0; nop }
{ add    a5, a3, a6; bbe_lvnx16_i_n    v4, a2, -64; bbe_movsvwh    v3, wv1; nop; nop }
{ bbe_svnx16_i    v1, a4, -64; bbe_lvnx16_xp    v1, a2, a6; bbe_movsvwl    v5, wv3; nop; nop }
{ bbe_svnx16_xp   v3, a4, a6; bbe_lvnx16_i_n    v0, a5, 64; bbe_movswv    wv1, v0, v2; nop }
{ bbe_lvnx16_x    v1, a3, a6; bbe_movsvwh    v2, wv3; bbe_movswv    wv3, v1, v4; nop }
{ bbe_svnx16_i    v5, a4, -64; add    a3, a5, a6; bbe_addnx40    wv1, wv1, wv0; nop; nop }
```

Note the use of vectorized NX16 loads and stores, and vectorized NX40 adds. If we look at
vector add of complex fractional data:

```
xb_cq15 ac15[VEC_SIZE], bc15[VEC_SIZE], cc15[VEC_SIZE];
void vec_add_cq15()
{
  int i;

  for (i = 0; i < VEC_SIZE; i++)
  {
    cc15[i] = ac15[i] + bc15[i];
  }
}
```

and then look at the disassembly, we see the following:

```
{ bbe_lvnx16_ip    v0, a2, 128; bbe_lvnx16_ip    v1, a3, 128 }
{ bbe_svnx16_i     v5, a4, -64; bbe_lvnx16_i_n    v4, a3, -64; nop; bbe_addsnx16    v5, v4, v3 }
{ bbe_svnx16_ip    v2, a4, 128; bbe_lvnx16_i_n    v3, a2, -64; nop; bbe_addsnx16    v2, v1,
v0 }
```

Note the mapping of complex q15 types into vectorized NX16 loads and stores, and
vectorized NX16 adds.

A vector dot product short:

```
short vec_dot_short(short as[], short bs[])
{
    int i;
    int sum, sum1 = 0, sum2 = 0;
    short *ptr_as = &as[VEC_SIZE/2];
    short *ptr_bs = &bs[VEC_SIZE/2];
#pragma aligned(as, 64)
#pragma aligned(ptr_as, 64)
#pragma aligned(bs, 64)
#pragma aligned(ptr_bs, 64)

    for (i = 0; i < VEC_SIZE/2; i++)
```

```
    {
        sum1 += as[i] * bs[i];
        sum2 += ptr_as[i] * ptr_bs[i];
    }
    return sum = sum1 + sum2;
}
```

produces

```
loopgtz      a6, 40000a18 <vec_dot_short+0x48>
{ bbe_lvnx16_ip     v0, a2, 64; bbe_lvnx16_ip     v1, a3, 64; bbe_mulanx16     wv0, v1, v0; nop }
{ bbe_lvnx16_ip     v2, a4, 64; bbe_lvnx16_ip     v3, a5, 64; bbe_mulanx16     wv1, v3, v2; nop }

40000a18 <vec_dot_short+0x48>:
{ nop; nop; bbe_mulanx16     wv1, v3, v2 }
{ nop; nop; bbe_mulanx16     wv0, v1, v0 }
{ nop; nop; bbe_raddnx40     wv1, wv1 }
{ nop; nop; bbe_raddnx40     wv0, wv0 }
{ nop; bbe_movaw32     a4, wv1 }
{ nop; bbe_movaw32     a2, wv0 }
```

Note the automatic selection of the BBE_MULANX16 for the short, as well as appropriate loads and stores. In addition, the correct reduction-add is selected to return the result as a short.

Two variants that do not vectorize in these examples are vector dot product and matrix multiply of ints. This is for the simple reason that the ConnX BBE64EP offers 16-bit matrix multiplication (via its 32 16x16 bit multipliers), but not 32-bit matrix multiplication. In this case, ordinary scalar code and scalar instructions are used.

One interesting example of complex vectorization and inference starts with the following source code:

```
xb_cq15 test_arr_1[VEC_SIZE], test_arr_2[VEC_SIZE];
xb_cq9_30 test_global_red_0;
void test_MULAJ_xb_cq15_xb_cq9_30_xb_cq15()
{
  int i;
  xb_cq9_30 red_0 = test_global_red_0;

  for (i=0; i < VEC_SIZE; i++)
  {
    red_0 += (test_arr_1[i] * ~test_arr_2[i]);
  }
  test_global_red_0 = red_0;
}
```

This produces the disassembly for the loop:

```
loopgtz a4, 6000049b <test_MULAJ_xb_cq15_xb_cq9_30_xb_cq15+0x57>
{ bbe_lvnx16_ip v0, a2, 64; nop; bbe_mulanx16j wv0, v1, v0 }
{ bbe_lvnx16_ip v1, a3, 64; bbe_lvnx16_i_n v3, a3, 32; bbe_mulanx16j wv0, v3, v2; nop }
bbe_lvnx16_i_n v2, a2, -32
test_MULAJ_xb_cq15_xb_cq9_30_xb_cq15+0x57
```

```
{ nop; nop; bbe_mulanx16j wv0, v1, v0; nop }
{ nop; nop; bbe_mulanx16j wv0, v3, v2; nop }
{ nop; nop; bbe_raddnx40c wv0, wv0 }
{ nop; bbe_movvwll v5, wv0; nop; nop }
{ nop; nop; nop; bbe_shflnx16i v4, v5, 1 }
{ bbe_sv4x16_i v5, a6, 0; nop }
{ bbe_sv4x16_i v4, a6, 8; nop }
```

Note the automatic inference of a complex conjugate multiply BBE_MULANX16J from the construct in C X[i] * ~X[i], where X is defined as a complex Q15 type.

Manual Vectorization

Of course, even the best compiler cannot automatically vectorize all code and loop nests even if all the guidelines have been followed. In this case, the next step is to move from scalar ConnX BBE64EP types to vector types, and manually vectorize the loops. The compiler may still be able to infer the use of vector instrinsics by using standard C operators.

When you manually vectorize, you reduce the loop count size by the number of elements in the vector instructions, and replace scalar types with the corresponding vector type. For example, replacing shorts by xb_vecNx16 (memory variables or vector register variables). The compiler deals with the ConnX BBE64EP vector data types as it does with any other type.

Below is an example of manually vectorized code for a vector add function:

```
void vector_add (const short a[ ], const short b[ ],
short c[ ], unsigned len)
{
  int i;
  // Cast pointers to short into pointers to vectors
  const xb_vecNx16 *va = (xb_vecNx16*)a;
  const xb_vecNx16 *vb = (xb_vecNx16*)b;
  // Assume no pointer aliasing
  xb_vecNx16 * __restrict vc = (xb_vecNx16*)c;

  // Change loop count to work on vector types
  for (i = 0; i < len/XCHAL_BBEN_SIMD_WIDTH; i += 1)
  {
    vc[i] = va[i] + vb[i];
  }
}
```

Here we see the loop count divided by XCHAL_BBEN_SIMD_WIDTH, which is equal to 32 for ConnX BBE64EP, and the use of xb_vecNx16 vector variables. Also note the use of the __restrict attribute to allow efficient compilation by telling the compiler there is no pointer aliasing to array c.

The programmer casts short pointers (in this case, array references) to vector pointers. The compiler will automatically generate the correct loads and stores. Note that this example assumes that "len" is a multiple of BBEN SIMD WIDTH which is thirty-two. If it is not, then the programmer needs to write extra code for the more general situation. However, if the data is arranged to always be a size multiple of the normal vector size, then the result can be more

efficient even if a few unnecessary computations are included. Padding a data structure with a few zeroes to make it a multiple (of thirty-two in the case above) is also often easy to do.

C-Intrinsic-based Programming

The final programming style is to use explicit intrinsics. Interestingly, 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.

Here is a simple example adding up a vector:

```
short addemup(short a[ ], unsigned int n)
{
  int i;
  short sum = 0;

  for (i = 0; i < n; i += 1)
  {
    sum += a[i];
  }
  return sum;
}
```

Here is an optimized intrinsic-based version:

```
short addemup_v(short a[ ], unsigned int n)
{
  int i;
  // Set a vector pointer to array a
  xb_vecNx16 *pa = ((xb_vecNx16 *) a);
  // Declare sum as a vector type and initialize to zero
  xb_vecNx16 sum = 0;
  xb_vecNx16 avec;

  for (i = 0; i < n; i += XCHAL_BBEN_SIMD_WIDTH)
  {
    sum += (*pa++);
  }

  // Add vector of intermediate sums and return short result
  return BBE_RADDNX16(sum);
}
```

Following are several interesting points:

- There is no need to use explicit vector loads.
- Similarly, the efficient vector adds are inferred from the code, which is still "C-like".
- The only explicit intrinsic necessary is the BBE_RADDNX16.
- This is a simple evolution from a manually vectorized version of this code.

- "sum" is initialized by casting it to a short 0, which initializes the vector "sum" to 0 in each element.
- Note that intrinsics are not assembly operations. They need not be manually scheduled into FLIX bundles; the compiler takes care of all that. And the code still remains quite "C-like".
- Intrinsic based programming can make use of the rich set of the ConnX BBE64EP data types and the right proto can be chosen that maps the data type into the underlying base instruction. Protos are listed in detail in the ISA HTML.
- The compiler will automatically select load/store instructions, but programmers may be able to optimize results using their own selection, by using the correct intrinsic instead of leaving it to the compiler.

3.5 Conditional Code

Programmers are encouraged to use the N-way programming model in ConnX BBE64EP. N-way programming is a convenient way to write code that offers ease of portability across cores in the BBE family. However, the assembler still uses operations with specific SIMD number native to the core used, in this case N=16. N-way programming is supported by inclusion of the primary header file - `<xtensa/tie/xt_bben.h>`. This header file xt_bben.h includes a supplementary include file specific to the BBE machine being programmed, here `<xtensa/tie/xt_ bbe64.h>`. Within these include files, there are a number of #defines that define "XCHAL" variables describing the machine characteristics, such as:

- SIMD width: `#define XCHAL _BBEN _SIMD _WIDTH 32`
- FFT: `#define XCHAL_HAVE_BBEN_FFT 1`
- Advanced Precision Multiply/Add: `#define XCHAL_HAVE_BBEN_ADVPRECISION 1`

In addition to N-way programming, to make code robust to the wide configurability in ConnX BBE64EP, it is useful to be able to write *conditional* code so that if an optional package is present (say the FFT option), the ISA support from the package can be explicitly used. Otherwise, the code may use a slower emulation for that operation. This is particularly useful in a complicated codebase shared between many cores with no source differences, one can use an appropriate #define to write conditional code specific to the SIMD size or configuration options included, so that all variants can be located in a single source file. The #defines for external, user configurable options are separately listed as a section inside the primary header file (`<xtensa/tie/xt_bben.h>`).

The following example illustrates the usage of vector types, `XCHAL_BBEN _SIMD _WIDTH` and an intrinsic call based on the N-way programming model:

```
xb_vecNx16 vin;
xb_vecNx40 vout = 0;
for (i=0; i < N/XCHAL_BBEN_SIMD_WIDTH; i++)
vout += vin[i] * vin[i];
*out_p = BBE_RADDNX40(vout);
```

3.6 Using the Two Local Data RAMs and Two Load/Store Units

The ConnX BBE64EP has two load/store units, which are generally used with two local data RAMs. Effective use of these local memories and obtaining the best performance results may require experimentation with several options and pragmas in your program.

In addition, to correctly analyze your specific code performance, it is important to carry out profiling and performance analysis using the right ISS options.

You may have a "CBox" (Xtensa Connection Box) configured with your ConnX BBE64EP configuration in order to access the two local data RAMs. For example, the two default ConnX BBE64EP templates described in *Implementation Methodology* on page 113.

Using the CBox, if a single instruction issues two loads to the same local memory, the processor will stall for one cycle. Stores are buffered by the hardware so it can often sneak into a cycle that does not access the same memory. For example, use of a CBox with two local data RAMs may cause occasional access contention, depending on the data usage and access patterns of the code. This access contention is not modeled by the ISS unless you select the --mem_model simulation parameter. Thus, if your code uses the two local data RAMs and your configuration has a CBox, it is important to select memory modeling when studying the code performance.

If you are using the standard set of LSPs (Linker Support Packages) provided with your ConnX BBE64EP configuration, and do not have your own LSP, use of the "sim-local" LSP will automatically place compiled code and data into local instruction and data memories to the extent that this is possible. Thus, Tensilica® recommends the use of sim-local LSP or your own LSP for finer grained control.

Finer-grained control over the placement of data arrays and items into local memories and assigning specific items to specific data memories can be achieved through using attributes on data definitions. For example, the following declaration might be used in your source code:

```
short ar[NSAMPLES][ARRAY_SIZE][ARRAY_SIZE] __attribute__ (section(".dram1.data"));
```

This code declares a short 3-dimensional array ar, and places it in data RAM 1. The compiler automatically aligns it to a 64-byte boundary.

Once you have placed arrays into the specific data RAM you wish, there are two further things to control. The first is to tell the compiler that data items are distributed into the two data RAMs, which can be thought of as "X" and "Y" memory as is often discussed with DSPs. The second one is to tell the compiler you are using a CBox to access the two data RAMs. There are two controls, currently not documented in the *Xtensa® C Application Programmer's Guide* or *Xtensa C and C++ Compiler User's Guide*, that provide this further level of control.

These two controls are a compiler flag, -mcbox, and a compiler pragma (placed in your source code) called "ymemory".

The -mcbox compiler flag tells the compiler to never bundle two loads of "x" memory into the same instruction or two loads of "y" memory into the same instruction (stores are exempt as the hardware buffers them until a free slot into the appropriate memory bank is available). Anything marked with the ymemory will be viewed by the compiler as "y" memory. Everything else will be viewed as "x" memory.

There are some subtleties in using these two controls — when they should be used and how. Here are some guidelines:

- If your configuration does not have CBox, you should not use -mcbox as you are constraining the compiler to avoid an effect that does not apply.
- If you are simulating without --mem _model, -mcbox might seem to degrade performance as the simulator will not account for the bank stalls.
- If you have partitioned your memory into the two data RAMs, but you have not marked half the memory using the ymemory pragma, use of -mcbox may give worse performance. Without it, randomness will avoid half of all load-load stalls. With the flag, you will never get to issue two loads in the same instruction.
- However, also note that there are scenarios where -mcbox will help. If, for example, there are not many loads in the loop, it might be possible to go full speed without ever issuing two loads in one instruction. In that case, -mcbox will give perfect performance, while not having -mcbox might lead to random collisions.
- If you properly mark your dataram1 memory using ymemory, or if all your memory is in one of the data rams, -mcbox should always be used.
- Without any -mcbox flag, but with the ymemory pragma, the compiler will never bundle two "y" loads together but might still bundle together two "x" loads. With the --mcbox flag, it will also not bundle together two "x" loads.

Thus in general, the most effective strategy for optimal performance is to always analyze ISS results that have used memory modeling; to assign data items to the two local data memories using attributes when declaring them; to mark this using the ymemory pragma; and to use -mcbox assuming your configuration has a CBox.

Use of the ymemory pragma is illustrated in the following code:

```
complex a[ARRAY_SIZE][ARRAY_SIZE][NSAMPLES]__attribute__((aligned(64),section(".dram1.data")));

complex b[ARRAY_SIZE][ARRAY_SIZE][NSAMPLES]
__attribute__((aligned(64),section(".dram0.data")));

xb_vecN_2xc16 c_auto_opt[ARRAY_SIZE][ARRAY_SIZE][NSAMPLES/4]__attribute__
((aligned(64),section(".dram0.data")));

void mm_auto_opt_4x4_stream_complex (xb_vecNxc16 (*__restrict a)[ARRAY_SIZE][NSAMPLES/4],
xb_vecN_2xc16 (*__restrict b) [ARRAY_SIZE][NSAMPLES/4], xb_vecN_2xc16 (* __restrict cp)
[ARRAY_SIZE][NSAMPLES/4]){
  #pragma ymemory (a)
  int i,j,h;
```

```
   for (i=0; i < ARRAY_SIZE; i+=1)
   {
     for (j=0; j < ARRAY_SIZE; j+=1)
     {
       for (h=0; h < NSAMPLES/4; h++)
       {
         cp[i][j][h] = a[i][0][h] * b[0][j][h] + a[i][1][h] * b[1][j][h] + a[i][2][h] * b[2][j]
[h] + a[i][3][h] * b[3][j][h];
       }
     }
   }
}
.....
mm_auto_opt_4x4_stream_complex((xb_vecN_2xc16 (*) [ARRAY_SIZE][NSAMPLES/4])a, (xb_vecN_2xc16
(*) [ARRAY_SIZE][NSAMPLES/4])b, c_auto_opt);
.....
```

☞ **Note:** In this code that we place input array a in data RAM 1; b in data RAM 0; and the output array in data RAM 0. We tell the compiler with the ymemory pragma that the *a* array is in ymemory, which effectively tells it the other two arrays are in x memory. Finally, since this is run on a configuration with a cbox, we compile with the -mcbox option and run with the memory modeling enabled in the ISS. The combination of the ymemory pragma and the mcbox compiler directive produces better results than if only one was used.

3.7 Other Compiler Switches

The following two other compiler switches are important:

- *-mcoproc*: Discussed in the *Xtensa® C Application Programmer's Guide* and *Xtensa C and C++ Compiler User's Guide* may give better results to certain program code.
- *O3 and SIMD vectorization*: If you use intrinsic-based code and manually vectorize it, it may not be necessary to use O3 and SIMD options. In fact, this may produce code that takes longer to execute than using O2 (without SIMD, which only has effect at O3). However, if you are relying on the compiler to automatically vectorize, it is essential to use O3 and SIMD to see this happen. As is the case with all compiler controls and switches, experimenting with them is recommended. In general, -O3 (without SIMD) will still be better than -O2.

3.8 TI C6x Instrinsics Porting Assistance Library

Tensilica® provides the following include header file for the RF-2014.1 release:

```
<install_path>/XtDevTools/install/tools/<release>-linux/XtensaTools/xtensa-elf/include/xtensa/
c6x-compat.h
```

This file is included to help in porting code that uses TI C6x intrinsics to any Tensilica® Xtensa processor as it maps these intrinsics to standard C. Because it maps TI C6x intrinsics

to standard C, the performance of the code is not optimized for the ConnX BBE64EP; to optimize the code further, you need to manually modify it using either ConnX BBE64EP data types that the compiler can vectorize and infer from, and/or ConnX BBE64EP intrinsics.

> ☞ **Note:** The path to this header file has to be modified accordingly for a different release.

Thus, this header file is intended as a porting aid only. One recommended methodology is:

- Include this code in your source files that use TI C6x intrinsics and move them to ConnX BBE64EP. As it handles most intrinsics, the code should, with little manual effort, compile and execute successfully on ConnX BBE64EP.
- Using the command line or Xtensa Xplorer profiling capabilities, profile the code to determine those functions, loops and loop nests which take most of the cycles.
- Rewrite those computationally-intensive functions, loops or loop nests to use ConnX BBE64EP data types and compiler automatic vectorization, or ConnX BBE64EP intrinsics, to maximize application performance. You could substitute calls to the ConnX BBE64EP library functions in these places.

The TI C6X standard C intrinsics implement 122 of 131 TI C6x intrinsic functions. Those not implemented are: _gmpy, _gmpy4, _xormpy, _lssub, _cmpy, _cmpyr, _cmpyr1, _ddotpl2r, and _ddotph2r.

Porting TI C6X Code Examples

This section contains some simple examples for using the intrinsic porting assistance file to port TI C6X code.

The first example uses the TI C6X _mpy intrinsic.

Example 1: _mpy

```
int a[VEC_SIZE], b[VEC_SIZE], c[VEC_SIZE];
void test_mpy()
{
  int i;
  for (i = 0; i < VEC_SIZE; i++)
  {
    c[i] = _mpy(a[i], b[i]);
  }
}

int main()
{
  test_mpy();
}
```

The _mpy intrinsic, which is used when you include c6x-compat.h, is:

```
static inline int _mpy(int src1, int src2)
{
```

```
    return (short) src1 * (short) src2;
}
```

Note that the TI _mpy intrinsic is just mapped into a standard C multiply of two short variables. The inner-loop disassembly is:

```
{ mul16s a10, a8, a9; bbe_l16si_s1 a3, a2, 0 }
{ l16si a5, a4, 0; bbe_l16si_s1 a9, a4, 4 }
{ s32i.n a6, a7, 0; bbe_l16si_s1 a8, a2, 4 }
{ s32i a10, a7, 4; addi.n a4, a4, 8; nop; addi.n a2, a2, 8 }
{ mul16s a6, a3, a5; addi.n a7, a7, 8 }
```

However, for a few TI instructions, the compiler can do better with ConnX BBE64EP vectorization and automatic inference of intrinsics. The inner-loop disassembly is:

```
{ bbe_lvnx16_i    v1, a3, 0; bbe_lvnx16_i_n    v3, a3, 64; bbe_movswv    wv1, v3, v1;
bbe_packsnx40    v4, wv1 }
{ bbe_svnx16_i    v5, a4, -64; bbe_lvnx16_i_n    v0, a2, -64; bbe_movswv    wv0, v2, v0;
bbe_packsnx40    v2, wv0 }
{ bbe_svnx16_xp   v6, a4, a5; add    a3, a3, a5; bbe_movsvwl    v5, wv2; nop; nop }
{ bbe_lvnx16_xp   v2, a2, a5; bbe_movsvwh    v6, wv2; bbe_mulnx16    wv2, v4, v2; nop }
```

Note the compiler automatically uses 32-way multiplies, 32-way loads (for ints which are cast to shorts), packs (to convert 40-bit results to shorts, 32-way stores (for ints), etc. – all from standard C code.

Example 2: _add2

Some intrinsics may need some manual code modification in order to make use of compiler automated vectorization. TI has a number of intrinsics that unpack integers into two shorts, and repack shorts back into integers after computation, such as _add2:

```
void test_add2()
{
  int i;
  for (i = 0; i < VEC_SIZE; i++) {
    e[i] = _add2(a[i], b[i]);
  }
}
```

One approach is to do the unpacking and packing in separate loops and then transform _add2 into two routines, calc and merge, as follows:

```
void test_add2_transform_calc()
{
  int i;
  for (i = 0; i < VEC_SIZE; i++)
  {
    g1[i] = a[i] & 0xffff;
    g2[i] = a[i] >> 16;
    h1[i] = b[i] & 0xffff;
    h2[i] = b[i] >> 16;
```

```
  }
  for (i = 0; i < VEC_SIZE; i++)
  {
    r[i] = g1[i] + h1[i];
    s[i] = g2[i] + h2[i];
  }
}

void test_add2_transform_merge()
{
  int i;
  for (i = 0; i < VEC_SIZE; i++)
  {
    e[i] = ((unsigned int) r[i] << 16) | ((unsigned int) s[i]);
  }
}
```

The calc routine unpacks the operands and does the computation, using ordinary C code. The merge routine packs the two results back together in the form the TI intrinsic does.

The calc disassembly vectorizes:

```
 { bbe_lvnx16_i    v2, a3, 0; bbe_movsvwh    v4, wv0; nop; nop }
{ bbe_svnx16_xp    v3, a6, a10; bbe_lvnx16_i_n    v3, a3, 64; bbe_movsvwl    v5, wv1; nop;
nop }
{ bbe_lvnx16_i_n    v0, a2, -64; bbe_lvnx16_xp    v1, a2, a10 }
{ bbe_svnx16_i    v5, a7, -64; bbe_movsvwl    v2, wv2; bbe_movswv    wv1, v3, v2; nop }
{ bbe_svnx16_xp    v6, a7, a10; bbe_movsvwh    v3, wv2; bbe_movswv    wv0, v1, v0; nop }
{ bbe_svnx16_xp    v4, a8, a10; add    a3, a3, a10 }
{ bbe_svnx16_i_n    v2, a9, -64; nop; bbe_srainx40    wv2, wv1, 16 }
{ bbe_svnx16_xp    v3, a9, a10; bbe_packlnx40    v2, wv0 }
{ nop; nop; bbe_srainx40    wv0, wv0, 16 }
{ nop; bbe_movsvwh    v2, wv2; bbe_unpkunx16    wv3, v2; nop }
{ nop; bbe_packlnx40    v4, wv1 }
{ nop; bbe_movsvwl    v3, wv2; nop; nop }
{ nop; bbe_movsvwl    v4, wv0; bbe_unpkunx16    wv1, v4; nop }
{ nop; bbe_movsvwh    v5, wv0; nop; nop }
{ bbe_svnx16_i_n    v3, a5, -64; nop; bbe_addnx40    wv0, wv2, wv0 }
{ bbe_svnx16_i_n    v4, a4, -64; nop; bbe_addnx40    wv2, wv1, wv3 }
{ bbe_svnx16_xp    v2, a5, a10; bbe_movsvwl    v2, wv3; nop; nop }
{ bbe_svnx16_xp    v5, a4, a10; bbe_movsvwl    v4, wv0; nop; nop }
{ bbe_svnx16_i    v2, a6, -64; bbe_movsvwh    v6, wv1; nop; nop }
{ bbe_svnx16_i    v4, a8, -64; bbe_movsvwh    v3, wv3; nop; nop }
```

But the merge disassembly does not. Therefore, try to avoid transforming data to and from packed intrinsic forms in the loops that must be optimized.

Example 3: _add2 and _sub2

Suppose you had a short sequence of a TI _add2 and then a TI_sub2 intrinsics. To begin to optimize this sequence, we want to avoid the transformation of intermediates backed into packed form. This can be tried by creating a _add2_sub2_calc routine, followed by a merge routine, which does the pack back into the TI merged form.

If there is a long sequence of calculations, avoiding the packing back allows the xt-xcc compiler to vectorize and infer naturally, which will save considerable cycles.

Manual Vectorization

There may be times when manual vectorization and intrinsics may be necessary to achieve optimal results. Also, you will need to decide what to do about saturating operations. The ConnX BBE64EP does not saturate on most operations, nor does it have an overflow register. Instead, it uses 40-bit vector registers that includes 8 guard bits and 16-bit vector registers without guard bits. For 32/40-bit data contained in 40-bit vector registers, the data is either packed and saturated to 16-bits or saturated to 32-bits. In either case post-saturation, the data is moved into one or two narrow `vec` registers respectively before storing as 16-bit elements in memory. Keep this different model in mind when converting code.

If the code with intrinsics is in parts of code which do not take many cycles in execution, (for example, control code, not loop-intensive data code) then you may just leave the intrinsic conversion to standard C and divide the code into one of the following categories:

- Rarely executed, low-cycle count code (that is, do not optimize the code)
- Heavily-executed, high cycle-count (optimize manually where the compiler does not)
- "Middle ground" code, which you must decide whether to optimize based on time and performance goals

4. Configurable Options

Topics:

4.1 FFT

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

The ConnX BBE64EP FFT package provides special FFT instructions optimized to perform a range of DFT computations. These FFT instructions implement a generic Discrete Fourier Transform in a series of steps. Each step is a pass over the whole input.

The basic sequence for decimation in frequency FFT operation is as follows:

1. Load a vector of inputs,
2. Multiply by constants (also called "rotations"). These are constant throughout all FFT pass sizes and depend on the radix type (1, j, -1 and –j) in *Figure 4: Radix4 FFT Pass* on page 68. Effectively, these rotation constants only modify the sign of the inputs (real or imaginary) and thus paired with the next step.
3. Perform a radix add. The radix add instructions add the inputs to a radix block while applying the correct sign change coming from the rotation constants.
4. Complex multiply by constants (also called "twiddle factors"). These depend on the FFT size and each individual value going through the transform. They are marked as Wn in *Figure 4: Radix4 FFT Pass* on page 68.
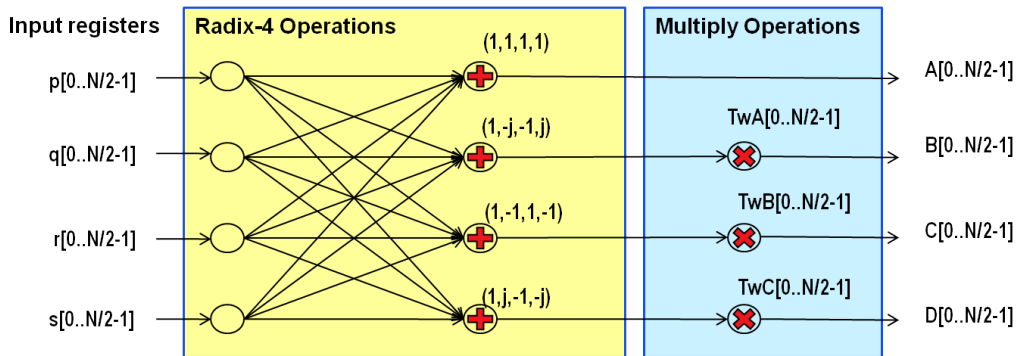
Figure 4: Radix4 FFT Pass

In the general case, the ConnX BBE64EP architecture allows a vector/state load, a vector store, a radix4 add, a vector-multiply and a shuffle to execute all in parallel. This produces to a full vector worth of radix4 FFT results as output per cycle. As long as the inputs for the butterfly are coming from a distance of at least one vector length away, the following sequence of basic instructions is followed:

1. Load four input vectors,
2. Use four appropriate FFT add instructions,
3. Apply the twiddle factors. The twiddle factor for one of the vectors is '1', the other three require a multiply (see *Figure 4: Radix4 FFT Pass* on page 68),

4. Store four output vectors.

> 👉 **Note:** In some stages, an additional interleave or shuffle step may be required.

Some FFT operations require control or monitoring of an instruction for, say, normalization and/or scaling. The BBE_MODE control register state is used to set appropriate mode for such operations.

Depending on the FFT size (N) the usage of FFT instructions in the ConnX BBE64EP can differ slightly. Consequently, the package has built-in support for radix-2, radix-3, radix-4 and radix-5 FFT implementations.

- When N is a power of '2'
- Even powers of '2' – radix4
- Odd powers of '2' – radix4 and radix2 (for only the last pass when N=8)
- When N is a non-power of '2'
- Radix3 and radix5

When performing an FFT computation, unless the results go into an inverse FFT (with some processing applied), the results are usually needed in natural order. For best performance it's recommended to use auto-sort FFT algorithms, for example Stockham FFT.

4.2 Symmetric FIR

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

Optimized ISA support to accelerate symmetric FIR operations is available as a separate configurable option. The symmetric FIR operations double the effective MACs per cycle. Symmetric FIR functions are supported by the following ConnX BBE64EP special operations:

- For real coefficients, real data

    - BBE_ADDPNX16RRU - State-Based Add with Shift Right for Real Data Symmetric FIR with Updates of States by Shifting Elements
    - BBE_ADDPNX16RRUMBC - State-Based Add with Shift Right for Real Data Symmetric FIR with Update of States Partially by Shifting Elements and Partially by using Input Vectors
    - BBE_ADDPNX16RRUMBCIAD - State-Based Add with Shift Right for Real Data Symmetric FIR with Update of States B and C by using Input Vectors

- For real coefficients, complex data

    - BBE_ADDPNX16RCU - State-Based Add with Shift Right for Complex Data Symmetric FIR by Shifting Elements
    - BBE_ADDPNX16RCUMBC - State-Based Add with Shift Right for Complex Data Symmetric FIR with Update of States Partially by Shifting Elements and Partially by using Input Vectors

- **BBE_ADDPNX16RCUMBCIAD** - State-Based Add with Shift Right for Complex Data Symmetric FIR with Update of States B and C by using Input Vectors

This package does not support (complex data, complex coefficients) symmetric FIR filters.

In ConnX BBE64EP, the doubling of effective acceleration of FIR filtering computations occurs when real coefficients are assumed to have a symmetric impulse response while the data can be real/complex. For real data, operation `BBE_ADDPNX16RRU` is first used for data summation to produce two results X, Y. Four special machine states `BBE_STATE{A,B,C,D}` are pre-loaded with Nx16 real data

1. `StateA` and `StateD` are first added into Nx16 result `X`
2. `{StateA, StateB}` concatenated (1:32) elements are added to `{StateD, StateC}` concatenated elements (31:62) for Nx16 result `Y`
3. Concatenated `{StateA, StateB}` are updated by shifting down 2 real elements and setting 0s at the top. `{StateD, StateC}` data is updated by shifting up 2 real elements and filling the bottom with the top 2 elements of state D
4. Use `BBE_ADDPNX16RRU{MBC, MBCIAD}` variants to input new vector data into states while rotating and shifting

For complex data, `BBE_ADDPNX16RCU` is first used for data summation to produce two results X, Y. Four special states `BBE_STATE{A,B,C,D}` are pre-loaded with N/2x16 complex data (see *Figure 5: 16-tap Real Symmetric FIR with Complex Data* on page 71)

1. StateA and StateD are first added into N/2x16 complex result X
2. `{StateA, StateB}` concatenated (1:16) complex elements are added to {State D, State C} concatenated complex elements (15:30) for complex N/2x16 result `Y`
3. `{StateA, StateB}` updated by down 2 complex element shift and 0 setting at the top; `{StateD, StateC}` shifted up 2 complex elements and filling of the bottom with the top 2 elements of `stateD`
4. Use `BBE_ADDPNX16RCU{MBC, MBCIAD}` variants to input new vector data into states while rotating and shifting

Use `BBE_EXTRNX16C` or `BBE_L32XP` operations to prepare pair of real 16-bit coefficients. Results X, Y are then multiplied with coefficients and pairwise added using special operation `BBE_MUL(A)NX16PR` as covered earlier

- `out = X * p + Y * q`, where `p`, and `q` are real 16-bit coefficients, and X, Y are the results found above
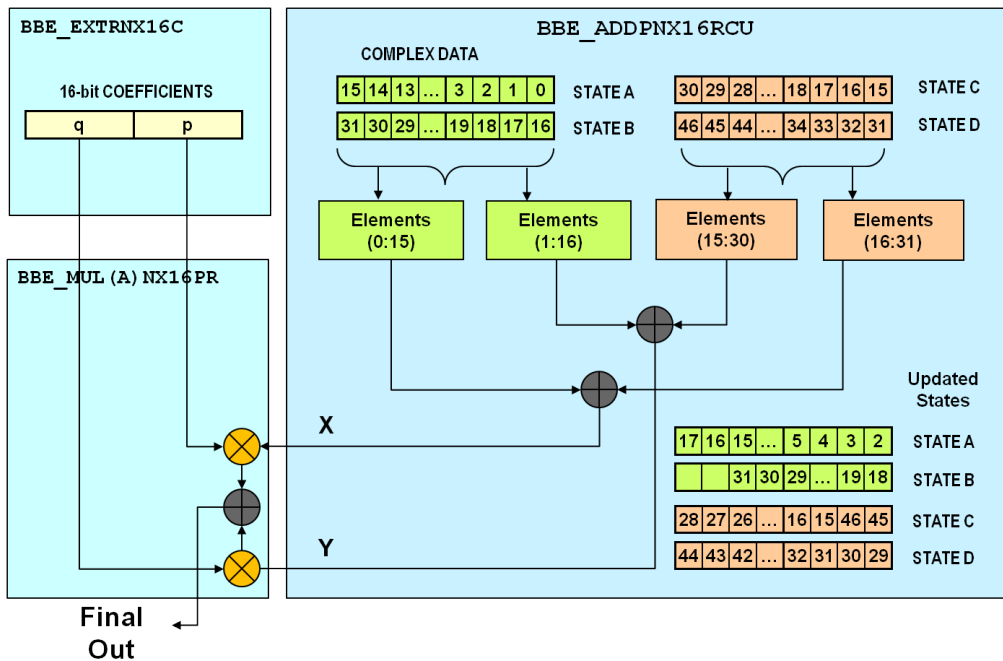
Figure 5: 16-tap Real Symmetric FIR with Complex Data

Refer to *Implementation Methodology* on page 113 on how to configure a ConnX BBE64EP with the symmetric FIR option. The ISA HTML for symmetric FIR instructions describes each operation's implementation and how the different operands are set up for a symmetric FIR operation.

☞ **Note:**

- *Symmetric FIR* option adds four states - `BBE_STATE{A,B,C,D}` - to a ConnX BBE64EP configuration which are otherwise not included.
- *FFT* & *Symmetric FIR* options share significant amounts of hardware. When configuring in one of these options in your ConnX BBE64EP core, it may be appropriate to add the other as the incremental cost is small.
- As a reference, several code examples are packaged with standard ConnX BBE64EP configurations in Xtensa Xplorer.

4.3 Packed Complex Matrix Multiply

This option adds support for vectors where the matrix elements are ordered by matrix rather than grouped by element.

ConnX BBE64EP has support for efficient computation of small complex packed matrix multiplies - 2x2, 4x4, and variants.

Complex packed matrix multiplies in ConnX BBE64EP are accomplished by decomposing the problem into smaller tasks:

1. Load matrix data into machine vector registers
2. Use *special shuffle instructions* in multi-step execution to reorganize order of data depending on matrix size
3. Perform multistep MAC operations and store result

  • For square matrices, use regular multiplies after shuffle
  • For rectangular matrices use special multiplies with replication

The *special shuffle instructions* (patterns for matrix element reordering) are available only as a configuration option to enhance regular shuffle operation BBE_SHFLNX16I and select operation BBE_SELNX16I. The shuffle/select patterns reorder N complex elements of two source vectors into a N/2 complex elements of destination output vector for processing.

The immediate value argument in these operations selects one of many patterns to select/ shuffle from inputs to outputs. As for notations, even though operations target complex values, elements are designated as real and imaginary pairs, so (0, 1, 2, 3,) elements means 0th real, 0th imaginary, 1st real, 1st imaginary elements for the first two complex numbers and so on.

Here's an example of a 2x2 complex packed matrix multiply routine in ConnX BBE64EP

```
// Using matrix multiply uses special data shuffles for complex 2x2*2x2 matrix multiply

void matmul_packed_complex_2x2_2x2(complex_short in1[][2][2], complex_short in2[][2][2],
complex_short out[][2][2], int n_vec)
{
    int i;
    xb_vecN_2xc16 * __restrict in1_p = (xb_vecN_2xc16 *) in1;
    xb_vecN_2xc16 * __restrict in2_p = (xb_vecN_2xc16 *) in2;
    xb_vecN_2xc16 * __restrict out_p = (xb_vecN_2xc16 *) out;
    xb_vecN_2xc40 vout;
    xb_vecN_2xc16 select_vec1, select_vec2;
    const vsaN_2C shft = 15; // variable pack shift amount
    for (i = 0; i < n_vec;++i)
    {
        select_vec1 = BBE_SHFLN_2XC16I(in1_p[i], BBE_SHFLI_MMC2X2X2X2_M1_STEP_1); // 1st
matrix, step 1
        select_vec2 = BBE_SHFLN_2XC16I(in2_p[i], BBE_SHFLI_MMC2X2X2X2_M2_STEP_1); // 2nd
matrix, step 1
        vout = select_vec1 * select_vec2; // multiply partial result
        select_vec1 = BBE_SHFLN_2XC16I(in1_p[i], BBE_SHFLI_MMC2X2X2X2_M1_STEP_2); // 1st
matrix, step 2
        select_vec2 = BBE_SHFLN_2XC16I(in2_p[i], BBE_SHFLI_MMC2X2X2X2_M2_STEP_2); // 2nd
matrix, step 2
        vout += (select_vec1 * select_vec2); // multiply complete result
        out_p[i] = BBE_PACKVN_2XC40(vout, shft); // variable pack and store
    }
    return;
}
```

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

4.4 LFSR and Convolutional Encoding

This option adds support for channel coding operations that involve LSFR code generation, such as scrambling and channel spreading. This option also provides support for convolutional encoding.

The LFSR option adds special operations to accelerate LFSR sequence generation up to 32-bits per cycle. This is done by emulating a 32x32-bit matrix by 32x1-bit multiplication. All the multiplication and reduction add operations in this option are in Galois Field (GF2). ConnX BBE64EP LFSR generation is executed in steps:

1. Initialize 32x32-bit matrix state to hold polynomial variants for 32 shifts. Use `BBE_MOVBMULSTATEV` operation to load state `BBE_BMUL_STATE`
2. Initialize 32-bit vector to hold initial shift register value. Use `BBE_MOVBMULACCA` operation to load state `BBE_BMUL_ACC`. The state is updated with result for the next issue.
3. Output 32-bit result at each issue and shift right old results using the `BBE_BMUL32A` operation. Set the second 32-bit register input to 0, not used for LFSR.

The above sequence can be modified to efficient implement various variants - generate Gold31 (3GPP) standard PRBS sequence. This can be done by using two sequence generators and XOR'ing the resulting sequences. You may also need to interleave parallel processed results to avoid interlocks. A similar methodology can be used for XRC processing too - use second input 32-bit register for updating the CRC state by XOR'ing input bits with output.

```
/* First block LFSR bits generated outside loop for x and y polynomials along with the current
initial states*/

  // Inner loop, generating the next nbits_512 - 1 blocks
  for(i=0;i < nbits_512;++i) {

    out_ptr[i] = v1^v2; // scrambler output as (x XOR y)
    BBE_MOVBMULACCA(x1); // initial condition for x

    // Compute next 512 LFSR bits, preparing x states
    S0 = x_mat_ptr[0];
    S1 = x_mat_ptr[1];
    BBE_MOVBMULSTATEV(S1,S0);

    // Two-vectors to avoid interlocks, 32 bits out each
    v1 = BBE_BMUL32A(v1, 0); v3 = BBE_BMUL32A(v3, 0);
    v1 = BBE_BMUL32A(v1, 0); v3 = BBE_BMUL32A(v3, 0);
    v1 = BBE_BMUL32A(v1, 0); v3 = BBE_BMUL32A(v3, 0);
    v1 = BBE_BMUL32A(v1, 0); v3 = BBE_BMUL32A(v3, 0);
    v1 = BBE_BMUL32A(v1, 0); v3 = BBE_BMUL32A(v3, 0);
    v1 = BBE_BMUL32A(v1, 0); v3 = BBE_BMUL32A(v3, 0);
    v1 = BBE_BMUL32A(v1, 0); v3 = BBE_BMUL32A(v3, 0);
    v1 = BBE_BMUL32A(v1, 0); v3 = BBE_BMUL32A(v3, 0);
```

```
    // Pick top half vectors
    v1 = BBE_SELNX16I(v3, v1, BBE_SELI_INTERLEAVE_2_HI);
    x1 = BBE_MOVABMULACC(); // keep x state

  // Repeat for vector y sequence

    // Set up y states
    BBE_MOVBMULACCA(x2);
    S0 = y_mat_ptr[0];
    S1 = y_mat_ptr[1];
    BBE_MOVBMULSTATEV(S1,S0);

    // Process
    v2 = BBE_BMUL32A(v2, 0); v4 = BBE_BMUL32A(v4, 0);
    v2 = BBE_BMUL32A(v2, 0); v4 = BBE_BMUL32A(v4, 0);
    v2 = BBE_BMUL32A(v2, 0); v4 = BBE_BMUL32A(v4, 0);
    v2 = BBE_BMUL32A(v2, 0); v4 = BBE_BMUL32A(v4, 0);
    v2 = BBE_BMUL32A(v2, 0); v4 = BBE_BMUL32A(v4, 0);
    v2 = BBE_BMUL32A(v2, 0); v4 = BBE_BMUL32A(v4, 0);
    v2 = BBE_BMUL32A(v2, 0); v4 = BBE_BMUL32A(v4, 0);
    v2 = BBE_BMUL32A(v2, 0); v4 = BBE_BMUL32A(v4, 0);

    // Pick top half vectors
    v2 = BBE_SELNX16I(v4, v2, BBE_SELI_INTERLEAVE_2_HI);
    x2 = BBE_MOVABMULACC(); // keep y state
}

// Process remaining bits
```

onvolutional coding is a form of forward error correction (FEC) coding based on finite state machines - input bit stream is augmented by adding patterns of redundancy data. ConnX BBE64EP version supports up to 16-bit polynomial encoder computing 64 encoded bits at each issue.

The optional `BBE_CC64` convolutional coding operation accepts the following inputs:

- An input Nx16-bit vector, 79 LSBs are used at each cycle (taken from the 5 LSB elements). A shuffle is needed to process 64 new bits at the LSB positions.
- Polynomial definition (16 LSB bits of AR register) to form register state definition
- Inout Nx16-bit vector (contains previous 64 generated bits placed at 4x16 LSB elements)

The operations returns 64 encoded bits placed at 4x16 MSB elements of inout register and shifts right old 64 processed bits to prepare for next cycle.

4.5 Linear Block Decoder

The support from this option is used for decoding block linear error-correction codes such as Hamming codes. In general, it can correlate a set of binary code vectors against a vector of real quantities.

The support to accelerate linear block decoding in the ConnX BBE64EP comes in the form of two optimized operations:

- BBE_DSPRMCNRNX16CS8 - 8-Codeset 32-Way 16-bit Real Coded Multiply and Reduce for Linear Block Decoding with No Rotation
- BBE_DSPRMCANX16CS8 - 8-Codeset 32-Way 16-bit Real Coded Multiply, Reduction and Accumulate for Linear Block Decoding with No Rotation

Both operations operate on 16-bit real data and perform a 32-way multiplication based on a vector of 1-bit codes. The operations support up to sixteen such sets of 1-bit code vectors. The ISA HTML for these two operations has more information on how a linear block decode operation is set up.

During a linear block decoding operation, reduction-add's between intermediate results are performed in full precision (32-bits wide) for each of the eight code-words. This is designed considering algorithms needs and hardware optimization. The result after a reduction-add is then sign-extended to 40-bits.

For the accumulating version of a linear block decoding operation, the accumulation is performed only after truncating the 40-bit results to 32-bits first, and then after the accumulation, the 32-bit sums are sign-extended back to 40-bit results.

4.6 1D Despread

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

The despread and descramble functions are needed at different sections of the 3G (WCDMA) receiver chain; these are listed below:

- 1D/2D single/multi-code 16-bit complex despread functions are used for S-SCH inner/ outer code correlations, for coarse frequency-offset estimation (based on P-SCH correlation) as well as coarse channel estimation (based on CPICH correlation) and for CCPCH channels dispreading (through 1D single-code 16-bit complex despread function)

Scrambling by itself, in 3G, only involves complex multiplication and since we usually need to work with a single (primary) scrambling code (as most channels are scrambled using a single scrambling code), 1D single-code type function is required for the descrambling operation. De-spread function also includes reduction-add following multiplication.

ConnX BBE64EP supports 1D despreading of 16-bit real or 8/16-bit complex input data. The 1D despread operations perform N-way element-wise signed, vector multiplication of real/ complex inputs with a vector of real/complex codes at full precision. If n is the spreading factor (SF), n consecutive products are consecutively added (2, 4, 8 or 16). A vector of N/n wide outputs, correspondingly real/complex, is produced every cycle. Say for SF=4, N-way real data and real codes

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

Inputs to the despreading operation are a Nx16-bit vector of real data to be multiplied by the codes, a Nx16-bit vector of code sets containing thirty-two Nx1b codes, immediate(s) that select(s) the code-sets to be processed each issue & the type of code. The accumulating versions of the despreading operations allow accumulating the reduction-add results with previous results into the output wide vector at 40-bit precision to obtain SF beyond 16.

☞ **Note:** Types of codes can be {1, -1} for real or {+/-1, +/-j, +/-1+/-j} for complex

The ISA support to accelerate applications performing 1D despreading comes in the form of the following special operations:

- BBE_DSPR1DANX16CSF8 - 16-way 16-bit Complex Coded Multiply, Reduction and Accumulate for Despreading with spreading factor 8
- BBE_DSPR1DANX16SF16 - 32-way 16-bit Real Coded Multiply, Reduction and Accumulate for Despreading with spreading factor 16
- BBE_DSPR1DANX8CSF16 - 32-way 8-bit Complex Coded Multiply, Reduction and Accumulate for Despreading with spreading factor 16
- BBE_DSPR1DNX16CSF4 - 16-way 16-bit Complex Coded Multiply and Reduction for Despreading with spreading factor 4
- BBE_DSPR1DNX16CSF8 - 16-way 16-bit Complex Coded Multiply and Reduction for Despreading with spreading factor 8
- BBE_DSPR1DNX16SF16 - 32-way 16-bit Real Coded Multiply and Reduction for Despreading with spreading factor 16
- BBE_DSPR1DNX16SF4 - 32-way 16-bit Real Coded Multiply and Reduction for Despreading with spreading factor 4
- BBE_DSPR1DNX16SF8 - 32-way 16-bit Real Coded Multiply and Reduction for Despreading with spreading factor 8
- BBE_DSPR1DNX8CSF16 - 32-way 8-bit Complex Coded Multiply and Reduction for Despreading with spreading factor 16
- BBE_DSPR1DNX8CSF4 - 32-way 8-bit Complex Coded Multiply and Reduction for Despreading with spreading factor 4
- BBE_DSPR1DNX8CSF8 - 32-way 8-bit Complex Coded Multiply and Reduction for Despreading with spreading factor 8

These operations are a part of the 1D despread option configurable in the ConnX BBE64EP. Refer to the operation's ISA HTML page for more details on its implementation and setup.

Table 12: Decoding Complex Codes for Despreading

Encoding (2b)	Code-type (1b immediate)	Decoded value	re_code (1b)	im_code (1b)
00	0	1+j	1	1
01	0	-1+j	-1	1

Encoding (2b)	Code-type (1b immediate)	Decoded value	re_code (1b)	im_code (1b)
10	0	1-j	1	-1
11	0	-1-j	-1	-1
00	1	1	1	0
01	1	-j	0	-1
10	1	-1	-1	0
11	1	j	0	-1

4.7 Soft-bit Demapping

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

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

The soft-bit demapper accepts as inputs complex-valued soft-symbol estimates x in addition to a scaling factor. Given these inputs, for each bit bi, it calculates the log-likelihood ratio

$$LLR(b_i) = \ln \frac{P(b_i = 1 \mid x)}{P(b_i = 0 \mid x)}$$

according to the mapping of bits to a constellation S. The LLR calculation uses a Max-Log approximation and assumes an unbiased symbol estimate with zero-mean additive white Gaussian noise (AWGN), i.e. x=s+w where s belongs to S and w is AWGN. Therefore, the SDMAP output is given by

$$LLR_{approx}(b_i) = (sign) \times (scaling\ factor) \times \left(min_{s_1 \in S \mid b_i = 1} |x - s_1|^2 - min_{s_0 \in S \mid b_i = 0} |x - s_0|^2 \right)$$

The scaling factor is used to account for signal-to-noise ratio and any other desired weighting adjustments. Users can negate the LLR values with an additional sign option.

Supported constellations and mappings are summarized in *Table 13: Set of Symbol Constellations Supported* on page 78. Symbol mappings for 3GPP and WiFi use different Gray Encoding formats, both supported by the soft-bit demapper operations.

Table 13: Set of Symbol Constellations Supported

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

ConnX BBE64EP implementation covers cases of 4/16/64/256-QAM soft-demodulation (`BBE_SDMAP256QAMNX16C, BBE_SDMAP64QAMNX16C, BBE_SDMAP16QAMNX16C, BBE_SDMAPQPSKNX16C`).

Inputs to the ConnX BBE64EP soft-bit demap operations are

- Complex constellation points
  - Assumed Q5.10 (16 bit resolution), not normalized
- Scale factors per point: 4-bit mantissa and 4-bit exponent in paired vector elements
  - Used for SNR and channel weighting adjustments
- Three immediate values to select between various modes
  - Pick upper or lower half of input complex vector to operate
  - Optionally negate soft-bit LLRs
  - Optionally interleave output soft-bit LLRs for real/imaginary parts (IEEE vs. 3GPP standard)

And, as for the outputs of the ConnX BBE64EP soft-bit demap operations,

- Up to 2N soft-bits per half complex vector are computed each cycle, scaled by the scaling factors, with rounding and saturation to 8-bit integer resolution at output.

Scaling before the soft demapper is needed to place onto an integer grid (assumed hardware implementation Q5.10 format). Scaling after the soft-demodulation is optionally applied by the operations.

4.8 Comparison of Divide Related Options

ISA support for divide related functions in the ConnX BBE64EP is offered as three configurable options:

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

Each option provides multiple operations (by adding more hardware) to the ConnX BBE64EP ISA to accelerate a specific divide related function. *Table 14: Comparison of Divide Related Options* on page 79) briefly outlines the intended use for each operation along with any functional overlap with other operation(s).

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

Table 14: Comparison of Divide Related Options

Class	Operations	Function	Overlaps functionally with	Configurable Option
Advanced Vector Reciprocal	BBE_RECIPLUNX40_0	High precision reciprocal approximation: ~23b mantissa accuracy	32b/16b divide and BBE_FPRECIP	Advanced Vector Reciprocal & Reciprocal Square Root
	BBE_RECIPLUNX40_1	High precision reciprocal approximation: ~23b mantissa accuracy	32b/16b divide and BBE_FPRECIP	
Advanced Vector Reciprocal Square Root	BBE_RSQRTLUNX40_0	High precision reciprocal square root approximation: ~24b mantissa accuracy	BBE_FPRSQRT	
	BBE_RSQRTLUNX40_1	High precision reciprocal square root approximation: ~24b mantissa accuracy	BBE_FPRSQRT	
Fast Vector Reciprocal	BBE_RECIPUNX16_0	Unsigned integer reciprocal approximation: results with approximately 15b precision	16b/16b unsigned divide	Fast Vector Reciprocal & Reciprocal Square Root
	BBE_RECIPUNX16_1	Unsigned integer reciprocal approximation: results with approximately 15b precision	16b/16b unsigned divide	
	BBE_RECIPNX16_0	Signed integer reciprocal approximation: results with approximately 15b precision	16b/16b signed divide	

	BBE_RECIPNX16_1	Signed integer reciprocal approximation: results with approximately 15b precision	16b/16b signed divide	
	BBE_FPRECIPNX16_0	Signed floating point reciprocal approximation: results with approximately 10b precision in the typical case, 7.5b in the worst case.	16b/6b signed divide, 32b/16b divide and advanced precision recip	
	BBE_FPRECIPNX16_1	Signed floating point reciprocal approximation: results with approximately 10b precision in the typical case, 7.5b in the worst case.	16b/6b signed divide, 32b/16b divide and advanced precision recip	
	BBE_DIVADJNX16	Signed divide adjust to allow C-exact 16b/16b integer divide - works with BBE_RECIPNX16*	16b/16b signed divide	
	BBE_DIVUADJNX16	Unsigned divide adjust to allow C-exact 16b/16b integer divide	16b/16b unsigned divide	
Fast Vector Reciprocal Square Root	BBE_FPRSQRTNX16_0	Signed floating point reciprocal square root approximation: results with approximately 9.7b precision in the typical case, 7.5b in the worst case.	High precision reciprocal square root	
	BBE_FPRSQRTNX16_1	Signed floating point eciprocol square root approximation: results with approximately 9.7b precision in the typical case, 7.5b in the worst case.	High precision reciprocal square root	
16-bit Vector Divide - First Step	BBE_DIVNX16S_5STEP0_0	Signed 16b/16b vector divide	BBE_RECIPNX16	Vector Divide
	BBE_DIVNX16S_5STEP0_1	Signed 16b/16b vector divide	BBE_RECIPNX16	
	BBE_DIVNX16U_4STEP0_0	Unsigned 16b/16b vector divide	BBE_RECIPUNX16	
	BBE_DIVNX16U_4STEP0_1	Unsigned 16b/16b vector divide	BBE_RECIPUNX16	
	BBE_DIVNX16Q_4STEP0_0	Unsigned fractional 16b/16b vector divide	BBE_FPRECIPNX16	

	BBE_DIVNX16Q_4STEP0_1	Unsigned fractional 16b/16b vector divide	BBE_FPRECIPNX16
Vector Divide - Other Steps	BBE_DIVNX16S_3STEPN_0	Last step of 32b/16b and 16b/16b signed vector divide	BBE_RECIPNX16
	BBE_DIVNX16S_3STEPN_1	Last step of 32b/16b and 16b/16b signed vector divide	BBE_RECIPNX16
	BBE_DIVNX16S_4STEP_0	Middle step of 32b/16b and 16b/16b signed vector divide	BBE_RECIPUNX16, high precision reciprocal approximation
	BBE_DIVNX16S_4STEP_1	Middle step of 32b/16b and 16b/16b signed vector divide	BBE_RECIPUNX16, high precision reciprocal approximation
	BBE_DIVNX16U_4STEP_0	Middle step of 32b/16b and 16b/16b unsigned vector divide	BBE_RECIPUNX16, high precision reciprocal approximation
	BBE_DIVNX16U_4STEP_1	Middle step of 32b/16b and 16b/16b unsigned vector divide	BBE_RECIPUNX16, high precision reciprocal approximation
	BBE_DIVNX16U_4STEPN_0	Last step of 32b/16b and 16b/16b unsigned vector divide	BBE_RECIPUNX16, high precision reciprocal approximation
	BBE_DIVNX16U_4STEPN_1	Last step of 32b/16b and 16b/16b unsigned vector divide	BBE_RECIPUNX16, high precision reciprocal approximation
32-bit Vector Divide - First Step	BBE_DIVNX32S_5STEP0_0	Signed 32b/16b vector divide	High precision reciprocal approximation
	BBE_DIVNX32S_5STEP0_1	Signed 32b/16b vector divide	High precision reciprocal approximation
	BBE_DIVNX32U_4STEP0_0	Unsigned 32b/16b vector divide	BBE_RECIPUNX16
	BBE_DIVNX32U_4STEP0_1	Unsigned 32b/16b vector divide	BBE_RECIPUNX16

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

4.8.1 Vector Divide

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

The support for 32-bit integer or scalar divide is an option available to supplement the base Xtensa ISA and turning it on upon configuring a ConnX BBE64EP provides users with the following protos:

- Signed integer divide – QUOS() and REMS()
- Unsigned integer divide – QUOU() and REMU()

More significantly, the ConnX BBE64EP can be configured with a vector divide option. Low-precision vector division operations in the ConnX BBE64EP are multiple cycle operations and are executed stepwise – four step operations for N/2 results of 16-bit precision and each step taking one cycle. These steps are pipelined by interleaving even and odd elements of the N-element vector. For convenience of programming, two sets of these four steps, one set each for even and odd elements are bundled into an intrinsic that produces N results of vector division in 8 cycles.

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

On inclusion of this option in a ConnX BBE64EP configuration, the core supports the following types of vector division:

- 16-bit by 16-bit unsigned vector divide

  - It takes a set of thirty-two 16-bit unsigned dividends and thirty-two 16-bit unsigned divisors from the `vec` register file and produces thirty-two unsigned quotients and thirty-two unsigned remainders, both with 16-bit precision and zero-extended to 16-bits per SIMD element.
  - Overflow and divide-by-zero conditions return 0x7FFF.
  - Protos used - BBE_DIVNX16U() / BBE_DIVNX16U()
- 16-bit by 16-bit signed vector divide

  - It takes a set of thirty-two 16-bit signed dividends and thirty-two 16-bit signed divisors from the `vec` register file and produces thirty-two signed quotients and thirty-two signed remainders, both with 16-bit precision and sign-extended to 16-bits per SIMD element.
  - Overflow returns 0x7FFF and underflow returns 0x8000 for negative. Similarly, for divide-by-zero, the operation returns either 0x7FFF or 0x8000 appropriately.
  - Protos used - BBE_DIVNX16() / BBE_DIVNX16()
- 32 -bit by 16-bit unsigned vector divide

  - It takes a set of thirty-two 32-bit dividends from the `wvec` register file and thirty-two 16-bit unsigned divisors from the `vec` register file and produces thirty-two unsigned quotients and thirty-two unsigned remainders, both with 16-bit precision and zero-extended to 16-bits per SIMD element.

- Overflow and divide-by-zero conditions return 0x7FFF.
- Protos used - BBE_DIVNX32U() / BBE_DIVNX32U()
- 32-bit by 16-bit signed vector divide

  - It takes a set of thirty-two 32-bit signed dividends from the `wvec` register file and thirty-two 16-bit signed divisors from the `vec` register file and produces thirty-two signed quotients and thirty-two signed remainders, both with 16-bit precision and sign-extended to 16-bits per SIMD element.
  - Overflow returns 0x7FFF and underflow returns 0x8000 for negative. Similarly, for divide-by-zero, the operation returns either 0x7FFF or 0x8000 appropriately.
  - Protos used - BBE_DIVNX32() / BBE_DIVNX32()

If users need to write scalar code with the use of scalar data-types like xb_int16, xb _int16U, xb _int40, xb _c16 and xb _c40, the Xtensa compiler can auto-vectorize the divide operations in the code by using the following set of 'compiler-assist' protos:

- Unsigned scalar divide – BBE_DIV16U() and BBE_DIV32U()
- Signed scalar divide – BBE_DIV16() and BBE_DIV32()

Refer to the description section of the various divide operations in the ISA HTML for further understanding of stepwise divide operations. Additionally, the vector_divide example included in the ConnX BBE64EP installation illustrates a sample usage of this option using a 16-bit by 16-bit signed vector divide example.

4.8.2 Fast Vector Reciprocal & Reciprocal Square Root

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

Refer to the ISA HTML of the following operations for more details on implementation and setup.

- Fast vector reciprocal

  - BBE_RECIPUNX16_0 - Unsigned 16-bit reciprocal approximation on even elements
  - BBE_RECIPUNX16_1 - Unsigned 16-bit reciprocal approximation on odd elements
  - BBE_RECIPNX16_0 - Signed 16-bit reciprocal approximation on even elements
  - BBE_RECIPNX16_1 - Signed 16-bit reciprocal approximation on odd elements
  - BBE_FPRECIPNX16_0 - Signed 16-bit + 7-bit pseudo-floating point reciprocal approximation on even elements
  - BBE_FPRECIPNX16_1 - Signed 16-bit + 7-bit pseudo-floating point reciprocal approximation on odd elements
  - BBE_DIVADJNX16 - Compute adjustment for correction of 16b signed divide based on fast reciprocal
  - BBE_DIVUADJNX16 - Compute adjustment for correction of 16b unsigned divide based on fast reciprocal
- Fast vector reciprocal square root

- BBE_FPRSQRTNX16_0 - 16-bit mantissa + 7b exponent pseudo-floating point reciprocal square-root approximation on even elements
- BBE_FPRSQRTNX16_1 - 16-bit mantissa + 7b exponent pseudo-floating point reciprocal square-root approximation on odd elements

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

4.8.3 Advanced Vector Reciprocal & Reciprocal Square Root

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

The advanced precision - 40-bit mantissa, 7-bit exponent - reciprocal (RECIP) and reciprocal square root (RSQRT) options provide operations that compute lookup table based terms to support Taylor's series expansion. The actual expansion may be performed using the core MAC operations. These options are recommended when more than 16-bit precision is required, particularly for inversions of ill-conditioned matrices. The throughput of this option set is about three 32-bit results per cycle. This estimate also includes the relevant MAC and normalization operations needed in the sequence.

The input to these operations needs to be normalized first. And, the operations compute either even or odd elements within Nx16 or Nx40 vectors. To hide the three cycle latency, they may be issued in pairs and software pipelined. Overflow or saturation conditions resulting from zero valued input elements are designated with the most negative representable number at the 40-bit wide output.

The Taylor's series approximation of `y(x)` is given as:

```
y = f(x0) + (x-x0)*f'(x0) + (x-x0)^2*f"(x0)/2 = A + (x-x0)*(B+(x-x0)*C)
```

We use lookup tables to approximate `A B, and C` with `x0` being the low end of segment range. The RECIP and RSQRT functions compute N-way 32-bit outputs in multiple steps.

1. Start by finding normalization amounts and then normalize wide precision inputs. Use `BBE_NSANX40` for RECIP, and `BBE_NSAENX40` for RSQRT, and apply `BBE_SLLNX40` to both.
2. Next, use `BBE_RECIPLUNX40_{0,1}` / `BBE_RSQRTLUNX40_{0,1}` even-odd pairs appropriately. These generate a wide output approximation for term `A` along with scaled `(x-x0)` inputs and an adjusted slope term `B=(x-x0)*C`. Use the signed MAC operation `BBE_MULUSANX16` for RECIP and the unsigned MAC operation `BBE_MULUUSNX16` for RSQRT to complete the rest of the Tayalor's series expansion as shown above.
3. Now, to obtain a floating-point output format, use `BBE_PACKVNX40` to pack the Nx40 values into Nx16, and use the normalization amounts to find the appropriate output exponent.

4. Or, to obtain fixed-point output format with desired denormalization into a wide vector register, use `BBE_SRSNX40` with an appropriate shift amount.

In the default use case, the input is assumed to be Q39 - in the range between 0.5 and 1.0 - and the output after MAC sequence without any shifts will be Q1.38 - greater than equal to 1.0 but under 2.0. Here's a code sequence illustrating this use case of ConnX BBE64EP RSQRT operations along with the said data formats of its inputs and outputs:

```
xb_vecNx40 inpw, inpnormw;    // inputs
xb_vecNx40 a ;                // to hold the a's from lookups
xb_vecNx16 inp_scaled;        // to hold scaled input
xb_vecNx16 b;                 // to hold b's from lookup
vsaN norm, renorm;            // normalization amounts

// inpw in Q39 format
norm = BBE_NSAENX40(inpw);
inpnormw = BBE_SLLNX40(inpw,norm);
BBE_RSQRTLUNX40_0(a, inp_scaled, b, inpnormw);
BBE_RSQRTLUNX40_1(a, inp_scaled, b, inpnormw);
BBE_MULUUSNX16(a, inp_scaled, b);

// a contaits mantissa of rsqrt output in Q1.38 format
// Q1.38 format => there is one sign bit, one integer bit, and 38 fractional bits
renorm = BBE_SUBSR1SAVSN(0,norm);
// Output exp = -norm/2
// BBE_SUBSR1SAVSN(0,norm) calculates 0 - (norm/2)
```

If the input data is not in Q39 format, it first needs to be converted to Q39 since the operation requires it. The same shift amount may be used to compute the output exponent.

For ConnX BBE64EP RECIP, the input format is assumed to be normalized Q39. The sequence of operations involved in implementing the Taylor's series approximation will output a result in Q2.37 format. Here's a similar example illustrating the code sequence:

```
xb_vecNx40 inpw, inpnormw; // inputs
xb_vecNx40 a ;                // to hold the a's from lookups
xb_vecNx16    inp_scaled; // to hold scaled input
xb_vecNx16 b;                 // to hold b's from lookup
vsaN norm, renorm;            // normalization amounts

norm = BBE_NSANX40(inpw);
inpnormw = BBE_SLLNX40(inpw,norm);
BBE_RECIPLUNX40_0(a, inp_scaled, b, inpnormw);
BBE_RECIPLUNX40_1(a, inp_scaled, b, inpnormw);
BBE_MULUSANX16(a, b, inp_scaled);
renorm = BBE_SUBSAVSN(0,norm); // output mantissa in a in Q2.37, output exponent in renorm
```

Refer to the ISA HTML of the following operations for more details on implementation and setup.

- Advanced precision vector reciprocal (RECIP)

  - `BBE_RECIPLUNX40_0` - Compute normalization and table lookup factors for advanced reciprocal approximation - even elements (...,4,2,0)

- BBE_RECIPLUNX40_1 - Compute normalization and table lookup factors for advanced reciprocal approximation - odd elements (...,5,3,1)
- Advanced precision vector reciprocal square root (RSQRT)
    - BBE_RSQRTLUNX40_0 - Compute normalization and table lookup factors for advanced reciprocal square root approximation - even elements (...,4,2,0)
    - BBE_RSQRTLUNX40_1 - Compute normalization and table lookup factors for advanced reciprocal square root approximation - odd elements (...,5,3,1)

4.9 Advanced Precision Multiply/Add

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

The ConnX BBE64EP advanced precision multiply/add is a configurable option that enables two new classes of data representation and related computations:

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

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

23-bit Floating Point

The mantissas are held in Nx16-bit vector registers (vec) and represent normalized signed Q15 values. For complex numbers, at least one element of the real-imaginary pair is normalized. The exponents are held in Nx7-bit vsa registers (vsa) and denote the amount of right shift needed to convert the mantissa back to fixed Q15 value. The legitimate range of values is -64<exponent<63. The special values 63 and -64 represents zero and BIG NaN respectively. BBE_FLUSH_TO_ZERO state is set when an resulting exponent is 63 (designating a zero). By convention, complex numbers have a common exponent.

32-bit Fixed Point

All the 32-bit fixed point data is held in wide vector registers (wvec), or in some cases in a pair of narrow vector registers (vec) as low and high 16-bit parts. The related operations use 16-bit multiply hardware and helper operations to emulate 32-bit multiplication with appropriate shifting and component gathering/accumulation. The intermediate multiply results are stored into new 2-entry Nx32-bit register file (mvec), then shifted appropriately and accumulated into regular Nx40-bit wide vector registers (wvec).

Decoupling of advanced precision multiply/add operations allow multiply-shift-accumulate sequences to be folded into parallel slots with a pipeline depth of 2-3 stages.

Table 15: Advanced Precision Multiply/Add Operations Overview on page 87 presents an overview of the various classes of ConnX BBE64EP operations added by the advanced precision multiply/add option.

Table 15: Advanced Precision Multiply/Add Operations Overview

Operation Type	Operations	Description
Data organization	`BBE_MOVMVNX32,` `BBE_MOVSVWXL, BBE_MOVSVWXH,` `BBE_MOVSWVX;` `BBE_UNPKNVNX16,` `BBE_UNPKVENX16,` `BBE_PACKNVNX40,` `BBE_FPPACKNVNX40;` `BBE_FPCVTCRN`	Move between vec/wvec/mvec register files; packing/unpacking from/to wvec; convert real to complex formats
Multiply, add, subtract, pack	`BBE_MULUUMNX16,` `BBE_MULUSMNX16,` `BBE_MULMNX16,` `BBE_MULUSMNX16C,` `BBE_MULMNX16C,` `BBE_MULUSMNX16J,` `BBE_MULSUMNX16J,` `BBE_MULMNX16J;` `BBE_FPPACKNX40,` `BBE_FPPACKNX40C,` `BBE_FPNORMNX40,` `BBE_FPNORMNX40C;` `BBE_FPADDNX16,` `BBE_FPSUBNX16`	Signed/unsigned real/complex multiplies into mvec; pack/normalize denormalized values after multiplies; add/subtract vec floating point into wvec
Shift, accumulate, normalize	`BBE_SRAMNX40,` `BBE_SRAMADDNX40,` `BBE_SRAMSUBNX40,` `BBE_SRAIMNX40,` `BBE_SRAIMRNX40,` `BBE_RNDIMNX40,` `BBE_SRAIMADDNX40,` `BBE_SRAIMADDRNX40,` `BBE_SRAIMSUBNX40,` `BBE_SRAIMSUBRNX40,` `BBE_SRAIWADDMNX40,` `BBE_SRAIWSUBMNX40,` `BBE_SLLIMNX40,` `BBE_SLLIMADDNX40,`	Shift and accumulate mvec to wvec with possible rounding; add/subtract mvec or vec floating point; find vec/wvec normalization amount

Operation Type	Operations	Description
	BBE_SLLIMSUBNX40; BBE_FPADDMNX40, BBE_FPSUBMNX40, BBE_FPADDMNX40C, BBE_FPSUBMNX40C, BBE_FPADDVNX40C, BBE_FPSUBVNX40C, BBE_FPSUBRVNX40C; BBE_NSAZNX16, BBE_NSAZNX16C, BBE_NSANZNX40, BBE_NSANZNX40C, BBE_NSANRZNX40, BBE_NSANRZNX40C	
Exponent	BBE_ADDSAVSN, BBE_SUBSAVSN, BBE_SUBSR1SAVSN, BBE_ADDSVSN, BBE_ADDSTVSN, BBE_SUBSVSN; BBE_MAXVSN, BBE_MINVSN, BBE_SLS1VSN, BBE_SRA1VSN, BBE_SHFLVSNI, BBE_SELVSNI, BBE_SUBSRA1SVSN, BBE_ABSSUBSVSN, BBE_DIFFOFFSVSN, BBE_UNCPRSVSNC, BBE_CPRSVSNC_0, BBE_CPRSVSNC_1, BBE_UNCPRSVSN, BBE_CPRSVSN	Add, subtract, shift, shuffle, compare, compress, uncompress exponent values in vsa register

The following brief examples show the usage of some of the advanced precision multiply/add operations.

Real floating point addition

Two floating point vectors added. Operation BBE_FPADDNX16 appropriately aligns mantissas and adjust exponents for proper sum. The result is in 40-bit wvec then packed to output 16-bit mantissa and new exponent using operation BBE_FPPACKNX40.

```
xb_vecNx16 x0_mant, y0_mant, z_mant;
vsaN x0_exp, y0_exp, w_exp, z_exp;
xb_vecNx40 w_mant;
BBE_FPADDNX16(w_mant, w_exp, x0_mant, x0_exp, y0_mant, y0_exp); // add floating point
```

```
    z_exp = w_exp;
    BBE_FPPACKNX40(z_mant, z_exp, w_mant, 0); // pack/normalize, update exponents
```

Complex floating point multiply-accumulate

Two sets of products of floating point vectors accumulated. Intermediate products into 32-bit mvec, accumulation into 40-bit wvec using operation BBE_FPADDMNX40C, then pack with operation BBE_FPPACKNX40.

```
    xb_vecNx16 x0_mant, y0_mant, x1_mant, y1_mant, z_mant;
    vsaN x0_exp, y0_exp, w_exp, x1_exp, y1_exp, z_exp, acc_exp;
    xb_vecNx40 acc_mant = BBE_MULNX16C(x0_mant, y0_mant);  // 1st mantissa set
    acc_exp = BBE_ADDSTVSN(x0_exp, y0_exp); // 1st exponent set
    xb_mvecNx32 m1_mant = BBE_MULMNX16C(x1_mant, y1_mant); // 2nd mantissa
    vsaN m1_exp = BBE_ADDSTVSN(x1_exp, y1_exp); // 2nd exponent
    BBE_FPADDMNX40C(acc_mant, acc_exp, m1_mant, m1_exp); //  accumulate products
    z_exp=acc_exp;
    BBE_FPPACKNX40(z_mant, z_exp, acc_mant, 0); // normalize pack and update exponents
```

Complex high precision fixed point multiply-accumulate(keep top 32 bits of full precision 64-bit result)

Move each 32-bit number into low/high 16-bit parts. Use appropriate singed/unsigned multiply to compute low/high, high/low, high/high products and accumulate with operations BBE_SRAIMNX40 with appropriate shifts.

```
    xb_vecNx40 x0, y0, x1, y1, wvec;
    xb_vecNx16 xl,xh,yl,yh;

    // 1st set of complex multiplies
    xl=BBE_MOVSVWXL(x0);
    xh=BBE_MOVSVWXH(x0); // move saturated 32-bit into low/high 16-bit
    yl=BBE_MOVSVWXL(y0);
    yh=BBE_MOVSVWXH(y0); // move saturated 32-bit into low/high 16-bit
    xb_mvecNx32 mlh = BBE_MULUSMNX16C(yl, xh); // complex low/high multiply
    wvec = BBE_SRAIMNX40(mlh, 15); // accumulate with proper shift adjustment
    xb_mvecNx32 mhl = BBE_MULUSMNX16C(xl,yh); // high/low
    BBE_SRAIMADDNX40(wvec, mhl, 15);
    xb_mvecNx32 mh = BBE_MULMNX16C(yh, xh); // high/high product
    BBE_SRAIMADDNX40(wvec, mh, 0);

    // 2nd set of complex multiplies to accumulate
    xl=BBE_MOVSVWXL(x1);
    xh=BBE_MOVSVWXH(x1);
    yl=BBE_MOVSVWXL(y1);
    yh=BBE_MOVSVWXH(y1);
    mlh = BBE_MULUSMNX16C(yl, xh); //low/low
    BBE_SRAIMADDNX40(wvec, mlh, 15);
    mhl = BBE_MULUSMNX16C(xl, yh); // high/low
    BBE_SRAIMADDNX40(wvec, mhl, 15);
    mh = BBE_MULMNX16C(yh,xh); // high/high
    BBE_SRAIMADDNX40(wvec, mh, 0);
```

4.10 Inverse Log-likelihood Ratio (LLR)

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

Turbo Equalization: Turbo equalizer is based on the Turbo principle and approaches the performance of MAP (Maximum aposteriori) receiver via iterative message passing between a soft-in soft-out (SISO) equalizer and a SISO decoder. This technique is used in MIMO-OFDM and CDMA systems including 3GPP LTE, WCDMA/HSPA, DVB etc. The decoder used is the SISO Turbo decoder, whereas the equalizer algorithm can be LMMSE equalizer using a-priori information based on feedback from Turbo Decoder. This technique can also be used in conjunction with SIC (Sequential interference cancelation) & PIC(Parallel interference cancelation).

The inverse LLR calculation converts the extrinsic LLR values output by the Turbo decoder to mean and variance values which can be used by the next iteration of the MMSE algorithm as shown below. The mean and variance calculation needs to be performed per symbol.
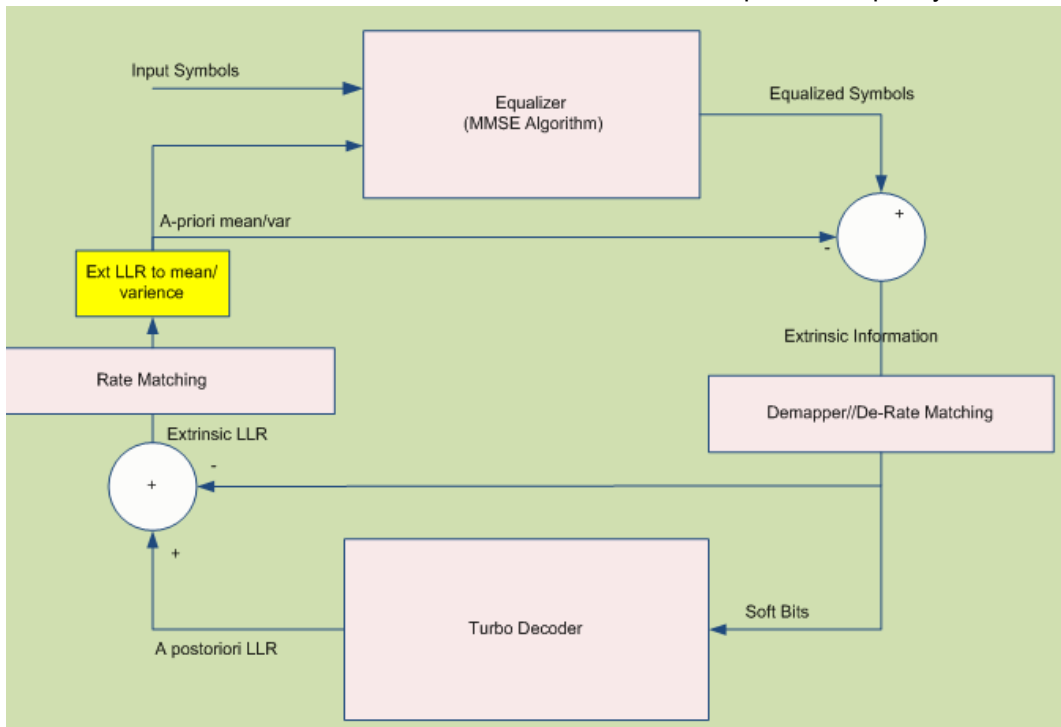

Figure 6: Inverse LLR Calculation

ConnX BBE64EP includes optional operations to facilitate efficient estimation of soft-complex QAM symbols from log-likelihood ratios (LLR). This process uses a lookup table for the

computation of non-linear hyperbolic tangent (tanh). Also, different operations are needed for each size of supported constellations (4/16/64/256-QAM).

The inputs, to the `BBE_INVLLRNX16C` operation, represent LLR values for a vector of N/2 complex, QAM symbols. For example, 6 such LLRs are needed for each 64-QAM complex symbol - 3 for real part and 3 for imaginary part.

- Immediate-value inputs select whether the sequence of LLR values, per complex point, are to be considered interleaved (3GGP) or not (IEEE), and whether or not they should be negated before processing further
- Input LLR values are packed in bytes, however, only the 6 LSBs are used in a Q2.3 format

Output of the operation is a vector of N-bit probabilities, in Q5.10 format, computed after a 11-bit lookup table approximation of *tanh* nonlinearity. The bit-probabilities for each complex element are interleaved as real-imaginary pairs. A sequence of operations is needed to obtain all bit-probabilities for any given QAM case, for example 256-QAM requires four issues to obtain the four pairs of bit probabilities. An immediate allows obtaining the value (2-bit probability) for a certain bit to accelerate the estimation process.

Operation `BBE_INVLLRNX16C` computes bit probabilities as the *tanh* of input LLRs for different QAM sizes. Once the bit probabilities are available, the mean and variance of the complex constellation point can be estimated using regular ConnX BBE64EP MAC operations.

1. Denote `pi` the bit probabilities at the output. Then the complex mean is given by
   - **256-QAM mean** - `real = p0*(8-p1*(4-p2*(2-p3))) & imag = p4*(8-p5*(4-p6*(2-p7)))`
   - **64-QAM mean** - `real = p0*(4-p1*(2-p2)) & imag = p3*(4-p4*(2-p5))`
   - **16-QAM mean** - `real = p0*(2-p1) & imag = p2*(2-p3)`
   - **4-QAM mean** - `real = p0 and imag = p1`
2. The real-imaginary parts for each bit probability pi are interleaved at the output so that helps in computing SIMD-way the quantities above for N/2 complex element vectors
3. Start with (`2-pi`) quantity in high QAM cases and successively real multiply-add the next bit probability N/2-way to compute the complex mean as equations dictate
   - For (`2-pi`) use special immediate setting to enable the factor 2, otherwise disable to produce only the `pi` value

Some iterative equalization methods require both the mean and variance of the complex constellation point computed (i.e. Turbo equalization). Variance is estimated applying a similar approach and the use of real MAC operations, mean, and bit probabilities already computed above. For example, for 64-QAM, with mean denoted as `mean64`,

```
64-QAM variance = 26+(2-p2)*(4-8p1) + (2-p5)*(4-8p4) - (mean64)^2
```

4.11 Single and Dual Peak Search

Single and dual peak (value and its index) search of 16/32-bit, signed/unsigned, real/complex inputs. In baseband, this option accelerates peak search of sequences resulting from real and complex correlation processes, for example to detect frame alignment during acquisition, or extracting frequency/phase correction.

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

Several state registers are added upon configuring a ConnX BBE64EP with the peak search option in order to not increase register usage.

- `BBE_MAX` and `BBE_MAX2` are 512-bit [SIMD] states added to hold first and second peak values respectively.
- `BBE_MAXIDX` and `BBE_MAXIDX2` are 192-bit [SIMD] states added to hold first and second peak indices respectively. The indices here are stored with relative SIMD lane offsets and are converted into actual index values at the final stage.
- `BBE_IDX` is a 11-bit state added to hold count of the number of peak search passes. This is internally used inside `BBE_MAXIDX/BBE_MAXIDX2` to compute relative offsets to indices of peaks.

Use model of these peak search operations is a multi-step process involving a setup (initialization), followed by operations that compare, aggregate and extract peaks and values from intermediate vectors. Let's go through the flow:

1. Initialize all index-states with `zero` and value-states with `-infinity` using `BBE_SETDUALMAX()`.
2. Use one of the following operations to process the loaded vector input, and appropriately save any single/dual peaks and their respective indices into the aforementioned states.
    - For 16-bit signed/unsigned single peak search, use `BBE_DMAX[U]NX16()`
    - For 16-bit signed/unsigned dual peak search, use `BBE_DUALMAX[U]NX16()`
    - For 32-bit signed/unsigned single/dual peak search, use `BBE_DUALMAX[U]WNX32()`
3. Next, for 16-bit single peak search use `BBE_GTMAXNX16()` and `BBE_MOVDUALMAXT()`, and for all other cases use only `BBE_MOVDUALMAXT()` to aggregate peaks and indices held in two states into a single vector.
4. Finally, `BBE_RBDUALMAXR()` and `BBE_SELMAXIDX()` are used to extract the single/dual peaks and their indices from the single vector obtained in the previous step.

Typically, in implementing complex correlation functions, `BBE_MAGINX16C()` is used to compute magnitudes of complex values. This output data of this operation is interleaved. To

account for this data reordering, a 1-bit immediate passed to `BBE_SELMAXIDX` controls the index value computation of the absolute maximum value; see ISA HTML for more details. Optionally, a right shift may be performed when using `BBE_DUALMAX{U}WNx32()` to avoid any potential input overflow. An example code of 32-bit complex dual peak search is provided below:

```
void bbe_dualpeak_mag_32b(int16_t * data_in, uint32_t * __restrict peaks, int16_t *__restrict
indices, int32_t size) {

xb_vecNx16 * __restrict data_ptr = (xb_vecNx16 *)data_in;
int32_t j;
int32_t size_N = (size/XCHAL_BBEN_SIMD_WIDTH) ;
uint32_t maxpeak;
uint32_t secondpeak;
vboolN selector, selector2;
BBE_SETDUALMAX(0); // initialize all dual peak state registers

// Find power of complex data and search for dual peaks/indexes
for(j=0;j<size_N;j=j+1) {
 xb_vecNx16 vec0,vec1;
 xb_vecNx40 wvec0;
 /* process two complex vectors */
 vec0 = data_ptr[2*j];
 vec1 = data_ptr[2*j+1];
 wvec0 = BBE_MAGINX16C(vec1,vec0); // this interleaves data, maxindex has to deinterleave below
 BBE_DUALMAXUWNX32(wvec0, 0);
}

// get the maximum and lane flag for max
BBE_RBDUALMAXUR(maxpeak,selector);
// get corresponding index, note deinterleaving index mode
int32_t maxindex = BBE_SELMAXIDX(selector,1);
// replace max with max2 for max lane
BBE_MOVDUALMAXT(selector);
// get the maximum and lane flag for max2
BBE_RBDUALMAXUR(secondpeak,selector2);
// get corresponding index for max2, deinterleaving mode
int32_t secondindex = BBE_SELMAXIDX(selector2,1);

peaks[0]   = maxpeak;
peaks[1]   = secondpeak ; // output dual peaks
indices[0] = maxindex;
indices[1] = secondindex ; // output dual indexes

}
```

Lastly, the table below provides a stage-wise overview of the different use cases supported and the set of operations involved in each of them.

Table 16: Use cases of peak search operations

Single peak search	SIMD vector search	Merge values/ indices vectors	Peak/index extraction
16b signed/unsigned	`DMAX{U}NX16`	`GTMAXNX16` and `MOVDUALMAXT`	`RBDUALMAXR` and `SELMAXIDX`

32b signed/unsigned	`DUALMAX{U}WNX32`	N/A	`RBDUALMAXR` and `SELMAXIDX`	
Dual peak search	**SIMD vector search**	**First peak/index extraction**	**Merge values/ indices vectors**	**Second peak/index extraction**
16b signed/unsigned	`DUALMAX{U}NX16`	`RBDUALMAXR` and `SELMAXIDX`	`MOVDUALMAXT`	`RBDUALMAXR` and `SELMAXIDX`
32b signed/unsigned	`DUALMAX{U}WNX32`	`RBDUALMAXR` and `SELMAXIDX`	`MOVDUALMAXT`	`RBDUALMAXR` and `SELMAXIDX`

5. Special Operations

Topics:

- *Polynomial Evaluation*
- *Matrix Computation*
- *Pairwise Real Multiply Operation*
- *Descramble Operations*
- *Vector Compression and Expansion*
- *Predicated Vector Operations*

The ConnX BBE64EP has operations to support fast Taylor's series expansion. These operations are useful to compute approximations of non-linear functions f(x), where $f(x) = f(x0) + f'(x0)(x-x0) + \ldots$ The entire range of f(x) is divided into multiple subdivisions, the center of each being x0. Thus, these operations can be used to evaluate transcendental functions (sines, cosines and such) that can be expressed in the form of a series.

5.1 Polynomial Evaluation

The polynomial evaluation operations are designed to work together to facilitate quick SIMD evaluation of iterative polynomial functions of the form yi = xi * yi-1 + ci, where xi is a coefficient for the ith step, and ci is a constant for the ith step. Vector polynomial operationsaccelerate N-way real vector multiply-add kernels of the following form:

```
for (i=0; i<terms; i++)
{
  Poly _out[i] = Lookup[x0] + Poly _in[i] * (x[i] - x0);
}
```

The vector polynomial operations compute N results in parallel and evaluates the function expansion by decomposition:

```
v = a + b*(x-x0) + c*(x-x0)2 + d*(x-x0)3
v = a + (x-x0)*(b + (x-x0)*(c + (x-x0)*(d + (x-x0)*0)))
```

The polynomial acceleration process needs an initial setup, once for every narrow vector (Nx16-bit). The prototype of this operation looks like:

```
BBE_POLYNX16_OFF(xb_vecNx16 x, immediate sl, immediate sa)
```

where,

- Input 16-bit vector x to compute vector (x-x0) factors output
- Choose sl (range 12-15) to extract bit-range from input and index/mid-point of its subdivision
- Choose sa for output shift (range 0..3, allows scaling by 1, 2, 4, 8)

```
Offv = InputData[(sl-2):0] - (1024 << (sl-12));
Offset = (Offv << sa) & 0xFFFF;
```

After completing the initial setup, execute iteratively until a desired precision is achieved:

- Use `(sl-1)/(sl-0)` as the shift amount in BBE_MOVVSV operation to extract the sub-range index for BBE_SHFL16X16/SEL16X16 operation respectively. These are indexes into the lookup table vector values
- Perform vector multiplication of the (x-x0) factors with the previous computed polynomial result and/or the lookup values

It is important to note that the table entries for lookup depend on the SIMD width of the machine. For ConnX BBE64EP, 32 subdivisions are allowed for the range of *f*(x) by using SHFL and 64 subdivisions by using SELECT. One can use SELECT or SHFL depending on the operation needs.

5.2 Matrix Computation

ConnX BBE64EP instruction set provides operations optimized for several types of matrix computation tasks. These operations support matrices stored in both packed and streaming order. Let's say we have three 2x2 matrices - A0, A1 and A2; the following figures illustrate this difference of matrix storage in memory.
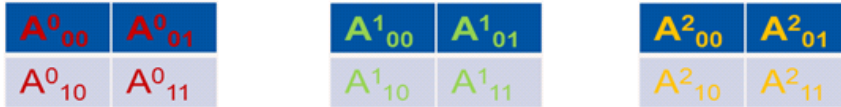
Now, in packed or natural order, the elements of each matrix are stored contiguously in memory

$$A^0_{00} \; A^0_{01} \; A^0_{10} \; A^0_{11} \; A^1_{00} \; A^1_{01} \; A^1_{10} \; A^1_{11} \; A^2_{00} \; A^2_{01} \; A^2_{10} \; A^2_{11}$$

However, in streaming order, corresponding elements of each matrix form individual streams as they are stored in memory.

$$A^0_{00} \; A^1_{00} \; A^2_{00} \; A^0_{01} \; A^1_{01} \; A^2_{01} \; A^0_{10} \; A^1_{10} \; A^2_{10} \; A^0_{11} \; A^1_{11} \; A^2_{11}$$

ConnX BBE64EP supports efficient computation of small, complex packed matrix multiplications – 2x2, 4x4, 8x8 and variants. For streaming order matrices, regular multiply operations from the ISA can be used to get efficient code schedules. For packed order matrices, special select/shuffle patterns are required that enhance the regular shuffle operation `BBE_SHFLNX16I` and select operation `BBE_SELNX16I`; see *Packed Complex Matrix Multiply* on page 71. These complex packed matrix multiplications are accomplished by decomposing computation into smaller tasks:

- Load matrix data into the ConnX BBE64EP vector registers
- In a multi-step execution routine, use special select/shuffle patterns to reorganize the order of data depending on matrix size

  - For square matrices, using regular multiplies after shuffle.
  - For rectangular matrices, using special multiplies with replication.
- Perform multi-step MAC operations and storing the result

When the matrices are not square or multiplying a matrix by a vector, the `BBE_MULNX16PC_{0/1}()` pair of protos can be used. The inputs to these protos are reordered outputs of the special shuffle/select operations and the outputs are full-precision results of the shuffled element multiplications. `BBE_MULNX16PC_{0/1}()` pair of protos perform N/2 complex multiplications and add adjacent pairs to form N/2 full-precision complex results in one wide `wvec` register. For multiply-accumulate type operations, a similar pair of protos is also provided – `BBE_MULANX16PC_{0/1}()`.

The following figure illustrates packed matrix multiplication of two 2x2 complex matrices:
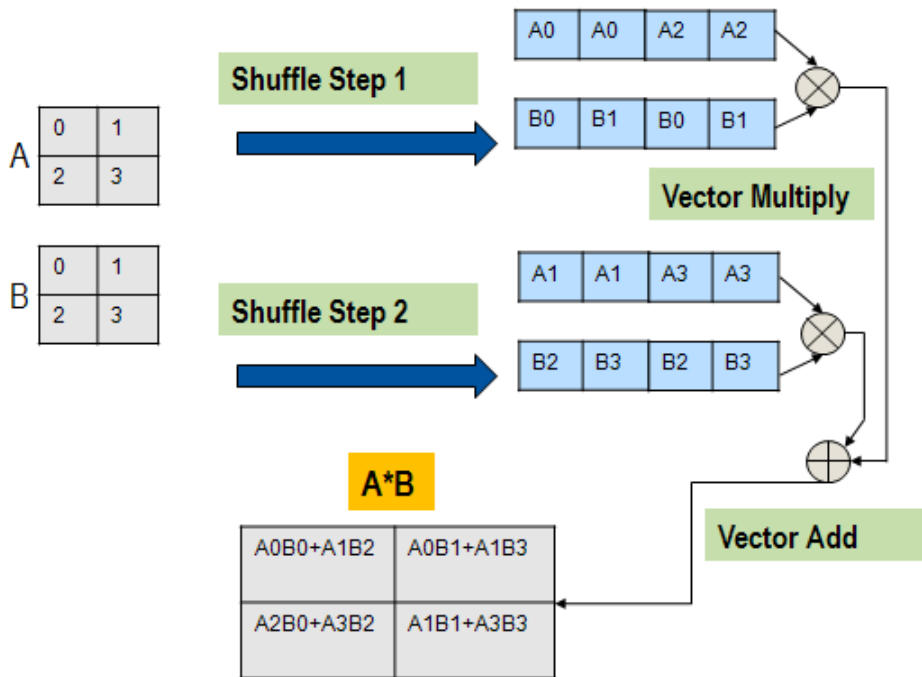
5.3 Pairwise Real Multiply Operation

The pairwise real multiply operations in the ConnX BBE64EP support 64 MACs per cycle on real data. The `BBE_MUL[A]NX16PR` operation finds natural use in accelerating symmetric FIR operations with real coefficients as described in the previous section. The general form of a pairwise real multiply operation is:

```
out_wvec[i] = in_vecA[i] * p + in_vecB[i] * q
```

where, in_vecA and in_vecB are N-way 16-bit narrow vectors; p & q are 16-bit scalars; and out_wvec is a N-way 40-bit wide vector. The 32-bit signed product per vector element is pairwise added between the two input narrow vectors. The pairwise sum is then sign-extended to 40-bits per element and written to the output wide vector. When implementing a symmetric FIR operation, p & q represent real coefficients of the filter.

In addition, the `BBE_MUL[A]NX16PR` operation finds use in accelerating real matrix-matrix multiplies, in particular product of 2x2 matrices, real matrix-vector multiplies and in real dot

product. Refer to the ISA HTML of `BBE_MUL[A]NX16PR` for more information on how the operation is set up.

5.4 Descramble Operations

Use of 1D single-code 16-bit complex despread function for combined descrambling and de-spreading operation on the downlink control channel DPCCH/FDPCH.

In 3G, scrambling by itself only involves complex multiplication, and since we usually need to work with a single (primary) scrambling code (as most channels are scrambled using a single scrambling code), 1D single-code type function is required for the descrambling operation.

ConnX BBE64EP provides ISA support for descrambling. A descrambling operation multiplies an input data sample with a code. This type of an operation is suited for vector processing through element-wise signed multiplication of a vector of input data samples by a code vector. The input samples may be real or complex, and accordingly define the nature of code used in the descrambling operation.

An N-way (N = SIMD size of the ConnX BBE64EP) descrambling operation on real data can be described as follows:

```
FOR i in {0..N}
re_res[i] = real_ip[i] * real_code[i]);
```

The `BBE_DESCRNX16{}` proto performs an N-way descrambling of 16-bit real data with a real code. A real code is always {1, -1} and is encoded by a 1-bit code as a vector input to the proto - '1' encoded by '0' and '-1' encoded by '1'. Thus, a set of 32-bits of code is required for a full vector of 32-way real inputs. Thus, a 512-bit narrow vec register can hold thirty-two code sets, 16-bits each. The result of `BBE_DESCRNX16{}` is a vector of thirty-two 16- bit real elements every cycle. The resulting output values saturate when the 16-bit range is exceeded.

For complex data samples, however, the descrambling operation in the ConnX BBE64EP is supported by two different protos. The use of these protos depends on the nature of input data samples.

- `BBE_DESCRNX8C{}` - 32-way descramble of 8-bit complex samples
- `BBE_DESCRN_2X16C{}` - 16-way descramble of 16-bit complex samples.

Similar to despreading, complex codes for descrambling can also be of two types - {+/-1, +/-j} or {+/-1+/-j}. A one bit immediate to the complex descramble proto selects the type of code to be used. For each sample of complex input data, two bits are required to encode the four possible code values regardless of the code type. *Table 12: Decoding Complex Codes for Despreading* on page 76 can be used again to show how complex codes are decoded within a complex descrambling operation. A vector-form description of a complex descrambling operation can be shown as:

```
FOR i in {0..N}
re_res[i] = re_ip[i] * re_code[i] + im_ip[i] * im_code[i];
im_res[i] = re_ip [i] * im_code[i] + im_ip[i] * re_code[i];
```

The outputs of the complex descramble protos are complex vectors whose values saturate when their range is exceeded. The real and imaginary parts of the complex result(s) in the output narrow vec register are interleaved. Refer to the ISA HTML for data-type information of inputs and outputs, and the manner in which each operation is set up. A summary of all the protos available for descrambling in the ConnX BBE64EP is shown in *Table 17: Protos for Descrambling* on page 100.

Table 17: Protos for Descrambling

Proto Name	SIMD	Input Type (`vec`)	Code Values	Code Sets (`vec`)	Output Type (`vec`)
NX16	32	16b complex	{1, -1}	16 x (32x1b)	16b real
NX8C	32	8b complex	{+/-1, +/-j} | {+/-1+/-j}	8 x (32x2b)	8b complex
N_2X16C	16	16b complex	{+/-1, +/-j} | {+/-1+/-j}	16 x (16x2b)	16b complex

5.5 Vector Compression and Expansion

ConnX BBE64EP has special operations to help with vector compression and expansion. Vector compression involves extracting elements from a full vector source, indexed by their positions, to form a new vector consisting of the extracted elements. Vector compression can be achieved by combining a squeeze and a shuffle operation. As an example, to compress a source vector `vec_in = {v_{n-1}, v_{n-2}, …, v1, v0}` to only its 0th, 1st and 4th elements, follow the steps below:

1. Define a Boolean vector mask in the `vbool` register `vbr`. The bit positions in the mask correspond to the position of the desired elements in the source vector. In this case, `vbr = {0,0,...,0,1,0,0,1,1}`.
2. Use the proto `BBE_ SQZN()` to generate a shuffle pattern in a `vsa` register `st` and a count of the number of bytes squeezed in an `ar` register `art`. Now, `st = {0, 0,..., 0, 4, 1, 0}` (n=32, SIMD width of the core) and `art = 6` (3 elements x 2 bytes/element).
3. Now, use the shuffle pattern produced in `st` to shuffle the source vector `vec_in` by choosing a `BBE_SHFL` proto appropriate to the data-type of `vec_in`. The output `vec_shfl = {v0, v0,…, v0, v4, v1, v0}`. Notice that after extracting the specified elements on the lower end, the rest of the output vector is filled with the lowermost element of the source vector (`v0`).

4. Additionally, the corresponding `BBE _SAVNX*.XP()` proto can be used on `vec_shfl` along with the size computed in `art` to store only the compressed vector elements.

Vector expansion follows a similar principle by using a combination of a un-squeeze and a shuffle instruction. Assuming the source vector `vec_in = {v_{n-1}, v_{n-2},…, v1, v0}`,

1. Define a Boolean vector mask in a `vbool` register `vbr` and generate a shuffle pattern in a `vsa` register st using the `BBE_ UNSQZN()` proto. `BBE_ UNSQZN()` also tracks the count of logical ones in the mask and computes the number of bytes to be un-squeezed as output in an ar register `art`.

2. The lowest order element (element zero) in `st` is always set to zero. Each higher order element (from one up to n-1) will contain a count of the number of ones in the input Boolean vector `vbr`, starting at the next lower element and going down to element zero. Now, if `vbr = 0b1001001` to represent the expansion of 0th, 3rd and 6th elements, then the un-squeeze instruction sets `st = 0d32221110` and `art = 6`.

3. The index values in `st` correspond to the positions of logical ones in the Boolean mask. Use the generated vector of index values in `st` within a `BBE_ SHFLNX*()` proto to un-squeeze the vector elements and produce an expanded output which in this case is `vec_shfl = {v3, ..., v3, v2, v2, v2, v1, v1, v1, v0}`.

5.6 Predicated Vector Operations

ConnX BBE64EP offers an enhanced ISA support for predicated vector operations natural to the core SIMD size. Without ISA support for predicated vector operations inherently parallel code cannot be parallelized by a vectorizing compiler. Consider a simple example:

```
for (i=0; i<32; i++) {
  if (a[i] != 0)
    b[i]++;
}
```

However, using predicated vector operations – in this case BBE_ADDNX16T; the above code can be easily vectorized. With a higher number of predicated operations supported for vectors than other Tensilica® Baseband architectures, the ConnX BBE64EP provides an improved performance in the presence of control flow both inside and outside loops. *Table 18: Predicated Vector Operations* on page 101 lists all the predicated operations supported by the ConnX BBE64EP for parallelizing control flow in vector code.

Table 18: Predicated Vector Operations

Type	Predicated TRUE Operation	Predicated FALSE Operation
Add	BBE_ADDNX16T	BBE_ADDNX16F
Signed add	BBE_ADDSNX16T	BBE_ADDSNX16F

Type	Predicated TRUE Operation	Predicated FALSE Operation
Complex, complex conjugate	BBE_CONJNX16CT	BBE_CONJNX16CF
Complex, complex conjugate - saturating	BBE_CONJSNX16CT	BBE_CONJSNX16CF
Signed maximum	BBE_MAXNX16T	BBE_MAXNX16F
Unsigned maximum	BBE_MAXUNX16T	BBE_MAXUNX16F
Signed minimum	BBE_MINNX16T	BBE_MINNX16F
Unsigned minimum	BBE_MINUNX16T	BBE_MINUNX16F
Narrow signed move	BBE_MOVNX16T	Use TRUE type
Wide signed move	BBE_MOVNX40T	Use TRUE type
Complex signed multiply-accumulate	BBE_MULANX16CT	Use TRUE type
Complex conjugate signed multiply-accumulate	BBE_MULANX16JT	Use TRUE type
Multiply-accumulate	BBE_MULANX16T	Use TRUE type
Real unsigned*signed Multiply-accumulate	BBE_MULUSANX16T	Use TRUE type
Real unsigned*unsigned Multiply-accumulate	BBE_MULUUANX16T	Use TRUE type
Signed negative	BBE_NEGNX16T	BBE_NEGNX16F
Signed saturating negative	BBE_NEGSNX16T	BBE_NEGSNX16F
Complex signed reduction sum	BBE_RADDNX16CT	BBE_RADDNX16CF
Signed reduction sum	BBE_RADDNX16T	BBE_RADDNX16F
Complex signed saturating reduction sum	BBE_RADDSNX16CT	BBE_RADDSNX16CF
Signed saturating reduction sum	BBE_RADDSNX16T	BBE_RADDSNX16F
Signed reduction maximum	BBE_RMAXNX16T	BBE_RMAXNX16F
Unsigned reduction maximum	BBE_RMAXUNX16T	BBE_RMAXUNX16F
Signed reduction minimum	BBE_RMINNX16T	BBE_RMINNX16F

Type	Predicated TRUE Operation	Predicated FALSE Operation
Unsigned reduction minimum	BBE_RMINUNX16T	BBE_RMINUNX16F
Signed difference	BBE_SUBNX16T	BBE_SUBNX16F
Signed saturating difference	BBE_SUBSNX16T	BBE_SUBSNX16F

6. Load & Store Operations

Topics:

The ConnX BBE64EP has an extremely wide variety of load and store operations for vectors, scalars, pairs of scalars (which can represent complex numbers as an (imaginary, real) pair) and a large number of addressing modes. There are over 185 load and store operations which support 16/32-bit scalars, 32x16-bit vectors in memory and 40-bit elements in vector registers. Unaligned vectors are supported by aligning loads and stores. The five addressing modes include a variety of indexed, immediate and updating addressing. There are 512/1280-bit spills and restore operations primarily for compiler use. The 1280-bit spills are implemented by extending them to 2048-bits and moving into four 512-bit registers for four subsequent stores/restores.

6.1 ConnX BBE64EP Addressing Modes

Table 19: ConnX BBE64EP Addressing Modes on page 106 provides a description of the ConnX BBE64EP addressing modes .

Table 19: ConnX BBE64EP Addressing Modes

Name	Notation	Description
Immediate	`_I`	Offset is an immediate operand
Immediate with Post-increment	`_IP`	Offset is an immediate operand. Update base address post-increment.
Indexed	`_X`	Offset from a register
Indexed with Post-increment	`_XP`	Offset from a register. Update base address post-increment.
Narrow Immediate	`_I_N`	Offset is a narrow immediate operand. Compiler alias of `_I` version.

☞ **Note:** There are specialized versions of the immediate loads (suffixed with ".N", which you may see in the disassembly. You do not need to use these directly; the compiler automatically chooses between the `.N` and the normal version depending on the scheduling.

Apart from the above five basic addressing modes, a special addressing modes supported by ConnX BBE64EP is the `_IC` notation used with operations that support circular addressing.

6.2 Aligning Loads and Stores

In ConnX BBE64EP, the support from aligning operations enables vector load/store of unaligned data. The aligning vector load and store operations move 512-bit vectors between the ConnX BBE64EP registers and memory addresses that may or may not be aligned to 64-byte boundaries. If the address is aligned, these operations perform the same operation as aligned load and store operations. For unaligned addresses, these operations use the ConnX BBE64EP alignment register file to provide a throughput of one aligning load or store operation per operation. The aligning vector load and store operations rely on two mechanisms to do this. One mechanism is the rotation of load and store data based on the least significant address bits of the virtual address. The other mechanism is the appropriate merging of the vector data with the contents of the alignment register.

Aligning Loads

A special priming operation is used to begin the process of loading an array of unaligned data. This operation conditionally loads the alignment register if the target address is unaligned. If the memory address is not aligned to a 64-byte boundary, this load initializes the contents of the alignment register. The subsequent aligning load operation merges data loaded from the target location with the appropriate data bytes already residing in the alignment register to form the completed vector, which is then written to the vector register. The base address is incremented after the completion of the aligning load operation i.e. post-incrementation. Data from this load then overwrites the alignment register, priming it for the next load. Subsequent load operations provide a throughput of one aligning load per operation.

The design of the priming load and aligning load operations is such that they can be used in situations where the alignment of the address is unknown. If the address is aligned to a 64 boundary, the priming load operation does nothing. Subsequent aligning load operations will not use the alignment register and will directly load the memory data into the vector register. Thus, the load sequence works whether the starting address is aligned or not.

Any C code for the ConnX BBE64EP compiled using the Xtensa C Compiler (XCC) always aligns arrays to 2N-bytes, where N is the SIMD size. If the C code development is not in a XCC environment, users may need to explicitly force alignment appropriate to the environment used.

Aligning Stores

Aligning stores operate in a slightly different manner. Each aligning store operation is sensitive to the value of the flag bit in the alignment register. If the flag bit is 1, appropriate bytes of the alignment register are combined with appropriate bytes of the vector register to form the 512-bit store data written to memory. On the other hand, if the flag bit is 0, then the store is a partial store and only the relevant bytes of the vector register are written to memory. Data from the alignment register is not used. No data will be written to one or more bytes starting at the 64 byte aligned address. This store will only write data starting from the byte corresponding to the memory address of the store.

Each aligning store operation (independent of the value of the flag bit) will also update the appropriate bytes in the alignment register, priming it for the next store operation. Every aligning store operation also sets the alignment register's flag bit to 1. When the last aligning store operation executes, some data may be left in the alignment register, which must be flushed to memory. A special flush operation copies this data from the alignment register to memory if needed.

Start with the BBE_ZALIGN operation to store an array of vectors beginning at an aligning memory address. This operation initializes the alignment register's contents and clears the flag bit. A series of aligning stores following the BBE_ZALIGN operation will store one vector to memory per operation. Note that the first store operation of this series will perform a partial store because the flag bit was cleared by the BBE_ZALIGN operation. Each subsequent store will perform a full 512-bit store because the first (and subsequent) store operations set

the flag bit. Finally, a flush operation flushes out the last remaining bytes in the alignment register.

Once again, the design of the aligning store and flush operations allows them to work even if the memory address is aligned to a 64 8 byte boundary. Specifically, if the address is aligned, the store operations store data only from the vector register. The alignment register is not used if the addresses are aligned. Similarly, if the addresses are aligned, the flush operation does nothing.

Note that these operations move data between memory and the ConnX BBE64EP registers if the memory addresses are not aligned to a vector boundary. However, these operations do assume that the addresses are aligned to 16-bit scalar boundaries. If these operations are used with addresses that are not aligned to 16-bit boundaries, the result is undefined. The priming and flush operations support only the immediate addressing mode, while the other operations support all four addressing modes.

6.3 Circular Addressing

The ConnX BBE64EP supports circular addressing using two special register states - CBEGIN and CEND. There are certain requirements to initialize these states before a circular buffer addressing operation is set up, as follows:

- The circular buffers need to be 64-byte aligned. This requires both CBEGIN and CEND to be 64-byte aligned addresses. Otherwise, circular buffer loads and stores may behave incorrectly.
- CBEGIN should be set to the start address of the circular buffer pointing at the first vector.
- CEND should be set to 64 bytes after the last aligned vector in the buffer.
- (CEND-CBEGIN) must be a multiple of 64-bytes.

☞ **Note:** Comparison for wrap-around is done after the post-increment .

The ConnX BBE64EP only supports a stride of 64 bytes in circular addressing. Other than these considerations, the load and store operations perform as expected, and can be either aligned or aligning.

There are four basic operations to access the circular buffer states as shown in the following table.

Table 20: Circular Buffer State Registers

Operation	Description
RUR.CBEGIN()	Reads from user register CBEGIN
RUR.CEND()	Reads from user register CEND

Operation	Description
WUR.CBEGIN(v)	Writes to user register CBEGIN
WUR.CEND(v)	Writes to user register CEND

The load/store operations with the `_IC` suffix or the corresponding `_IC` C intrinsic functions refer to the circular buffer addressing. Also, the `BBE_LAPOS_PC` priming load operation and the corresponding `BBE_LAPOS_PC` intrinsic function use the circular buffer registers.

6.4 Variable Element Vector Aligning Loads and Stores

ConnX BBE64EP contains a set of store operations that compact vectors at the same time as they do the store, according to patterns defined by the user. These operations are BBE_LAVNX16.XP, BBE_SAVNX16.XP and BBE_SAPOS.FP. There are also protos that define variations for various data types including fractional, complex, fractional complex, etc. as defined in the earlier tables. Further details about the operations, their protos and usage may be found in the ConnX BBE64EP ISA/HTML.

6.5 Update Post-increment in Loads and Stores

The ConnX BBE64EP has many loads and stores, as described in earlier sections, which perform an address update as part of their operation. The updating loads and stores do a post-increment of the address after doing the load or store.

☞ **Note:** Some other architectures, like Vectra, ConnX BBE16, SSP16, etc., perform pre-incrementing loads and stores as a part of their address update strategy.

7. Nature DSP Signal Library

The ConnX BBE64EP comes packaged together with a generic DSP library. The library comes with two Xplorer projects - `bbe64ep_integrit_re_lib_v<version_num>.xws` and `bbe64ep_integrit_re_demo_v<version_num>.xws`. The former is the library. The latter is a test application that exercises all the functions in the library. Both are delivered in source form so that the user can use the library as is or use it as a starting point for their own development. The library contains functions for basic math functions, matrix operations, communication operations and others.

From the library project, under doc, is a reference manual, *NatureDSP Signal Library Reference for Cadence® ConnX BBE64EP DSP*, which describes the library and the test program in depth.

8. Implementation Methodology

Topics:

The ConnX BBE64EP is a part of the *ConnX Baseband Engine* family of DSPs specially optimized for infrastructure and user-equipment applications. This family of DSPs is based on Xtensa ISA version LX6.0. The ConnX BBE64EP is included as a check box option in the Xplorer Processor Generator (XPG) interface in Xtensa Xplorer. This option is featured in the RF-2014.1 release of Xtensa Xplorer, XPG, and Xtensa tools. The following section provides guidelines for using XPG to configure a custom ConnX BBE64EP. The last section in this chapter discusses synthesis and place-and-route aspects of the ConnX BBE64EP.

8.1 Configuring a ConnX BBE64EP

Creating your own ConnX BBE64EP DSP configuration consists of the following steps using XPG in the Xtensa Xplorer IDE:

1. Open the System Overview tab by navigating to the Show View menu in the Window dropdown item from the menu bar. Right click the Configuration directory and select the New Configuration option.
2. Select the Create new configuration with a new core ISA option and make sure to test your XPG access by entering a valid customer\username and password. Click next after XPG access test is passed.
3. Enter a name for your configuration, an optional description and select the processor ISA version LX6.0 or greater. Click Finish.
4. Open the created configuration by finding it within the Uninstalled Configs sub-directory. In the Configuration Overview pane, click the Edit button in the Workspace Config area under Configuration section.
5. Open the Processors tab in the Configuration Editor and select the ConnX BBE64EP DSP from the ConnX BBE-EP DSP family.

   A red cross-mark may appear if there are any required modules to be included in the configuration. Hover over the cross-mark to read any errors and resolve them by selecting appropriate options to include all the necessary modules. Some errors may need to be resolved by selecting correct options from the Interfaces tab.
6. To extend your ConnX BBE64EP configuration with one or more options, select the relevant check box options under the ConnX BBE-EP DSP Options section:

   • FFT
   • Symmetrical FIR
   • Packed complex matrix multiply
   • LFSR & convolutional encoding
   • Linear block decoder
   • 1D Despreader
   • Soft-bit demapping
   • Vector divide
   • Fast reciprocal & reciprocal square root
   • Advanced precision reciprocal & reciprocal square root
   • Advanced precision multiply/add
   • Inverse log-likelihood ratio (LLR)
   • Single and dual peak search

   Note: Including any configurable option adds additional operations and associated hardware to the ConnX BBE64EP configuration.

Figure 7: ConnX BBE64EP Configuration Options in Xtensa Xplorer

XRC_B64EP_MIN, Optionally, there are sample configuration templates provided for the ConnX BBE64EP, called XRC_B64EP_MAX and XRC_B64EP_MAX2 which are described in *Sample Configuration Templates for the ConnX BBE64EP* on page 122. They provide useful starting points for users who wish to either use the configuration as described by the template, or create their own variation. It is useful, but not essential, to start with a ConnX BBE64EP template. Following are the steps to create your configuration in the Xtensa Xplorer IDE using a template:

1. Follow steps 1-4 from earlier procedure to create your own ConnX BBE64EP configuration.
2. Click Load a Configuration Template from the Software pane if you wish to use one of the templates as a starting point for your ConnX BBE64EP configuration
3. Click Select from standard templates and pick either XRC_B64EP_Base_DSP, XRC_B64EP_eNB_RX, XRC_B64EP_eNB_TX, XRC_B64EP_WiFi_4x4, or XRC_B64EP_MIN to apply and click OK.

As you are customizing your ConnX BBE64EP, keep in mind the following restrictions. The ConnX BBE64EP has 48/96-bit instruction formats. The following may need to be set during the configuration process:

- The Maximum Instruction Width in bytes to 12 in the ISA Configuration Options section under the Instructions pane;
- Pipeline Width to 7 in the ISA Configuration Options section under the Instructions pane;
- Width of Instruction Fetch Interface to 128 [bits] in the PIF/Memory Interface Widths section under the Interfaces pane; and
- Width of Data Memory/Cache Interface to 512 [bits] in the PIF/Memory Interface Widths section under the Interfaces pane.

Once your ConnX BBE64EP has been configured, save your configuration and open the Configuration Overview pane. You can now click the Upload/Build button to submit your ConnX BBE64EP configuration to XPG. Once your configuration is built, you may download and install it in the Xtensa Xplorer IDE.

8.2 XPG Estimation for Size, Performance and Power

In the RF-2014.1 release of the XPG, the estimation of size (area), performance, and power for the ConnX BBE64EP is available.

These estimations are adjusted when users modify their ConnX BBE64EP configuration. Sometimes hardware is shared between one or more configurable options. This makes the combined cost of those options less than the linear sum of their individual costs. In ConnX BBE64EP, for example,the FFT and symmetric FIR options share a significant amount of hardware - including one of them makes it relatively inexpensive to include the other.

8.3 Basic ConnX BBE64EP Characteristics

Some of the relevant configuration characteristics of the ConnX BBE64EP include:

* ConnX BBE64EP instruction set
* Vector Boolean registers
* Little-endian byte order
* Two load/store units
* 512-bit local data memory interface
* MUL16 and MUL32 (Pipelined + UH/SH) arithmetic operation options
* 128-bit PIF interface. A 32-bit PIF interface is theoretically possible, but is not advised for ConnX BBE64EP, as it is intended for high performance applications; a 32-bit PIF is likely to be inadequate for optimal ConnX BBE64EP performance.

8.4 Extending a ConnX BBE64EP with User TIE

The ConnX BBE64EP can be extended with user TIE defining new operations. These instructions can be assigned to the 24-bit regular instruction format, or you may wish to use the 48/96-bit FLIX instruction format that is available for user instruction additions. Due to encoding restrictions, there are actually fewer bits available in the user-defined instruction format than the length of the instruction.

We illustrate the process of defining the user FLIX instruction format and adding instructions to it using two simple TIE examples, as follows. The first TIE example defines a new FLIX format, consisting of five FLIX slots. The following TIE template can be used to define a new 96-bit wide FLIX format:

```
// Define a new 96-bit wide FLIX format for user instructions in ConnX BBE64EP
// Use this format declaration:

format user96 96 { user_wide_slot0, user_wide_slot1, user_wide_slot2,
user_wide_slot3, user_wide_slot4 }
```

```
// For now, only assign NOPs to the user slots
slot_opcodes user_wide_slot0 {NOP}
slot_opcodes user_wide_slot1 {NOP}
slot_opcodes user_wide_slot2 {NOP}
slot_opcodes user_wide_slot3 {NOP}
slot_opcodes user_wide_slot4 {NOP}
```

Based on the scheduling requirements in the target application, users may now add one or more operations - either from the ConnX BBE64EP ISA or user defined TIE operations - to the slot_opcodes defined. Based on the amount of encoding space available, the TIE Compiler will attempt to encode the format and opcodes assigned to its one or more slots.

Similarly, the following TIE template can be used to define a new 48-bit narrow FLIX format:

```
// Define a new 48-bit (narrow) FLIX format for user instructions in ConnX BBE64EP
// Use this format declaration:

format user48 48 { user_narrow_slot0, user_narrow_slot1 }

// For now, only assign NOPs to the user slots
slot_opcodes user_narrow_slot0 {NOP}
slot_opcodes user_narrow_slot1 {NOP}
```

☞ **Restriction:**

- TIE Compiler Error
- Error: (TIE_FORMAT_MAX_SLOT_COUNT), A total slot count of [n] has been reached which exceeds the maximum slot limit of 64. Each format may contain a maximum of 30 slots but the total slot count across all formats may not exceed 64
- For a ConnX BBE64EP configuration with the "*Advanced Precision Multiply/Add*" option turned ON, a restriction in the current release provides only four user FLIX slots. This means no more than four slots may be defined and used regardless of the number of user TIE formats being added.

Users can also write their own operation to extend an Xtensa base machine like a ConnX BBE64EP. The *Tensilica® Instruction Extension (TIE) Language User's Guide* is a good reference to learn the fundamentals of TIE and write TIE code to extend a core's ISA.

Compiling User TIE

Following are the instructions to compile an example user TIE for the ConnX BBE64EP:

1. Start with a ConnX BBE64EP configuration in Xtensa Xplorer. The configuration should not include the desired user TIE.
2. Clone the configuration and attach TIE files. Select **ConnX BBE64EP_user _format.tie** and **ConnX BBE64EP_user _slots.tie**. Choose a name for the TDB.
3. Choose a name for the new configuration.
4. Compile the TDK for the new configuration with user TIE. Note: Generating cstub libraries requires more time than a standard TDK compilation.

5. After compiling the TDK, use the new configuration to compile and run code.

Name Space Restrictions for User TIE

There are restrictions in the name space that can be used in a user TIE file when using the ConnX BBE64EP. The following TIE elements are reserved and cannot be used in TIE added to a configuration that includes the ConnX BBE64EP. In general, the prefix BBE_, bbe_, and xb_ are reserved for the coprocessor. More specifically, the names of the different TIE features used by the coprocessor are discussed below. The following table shows the state and register files used by the ConnX BBE64EP:

Table 21: State and Register File Names

Name	Type	Description
vec	regfile	
wvec	regfile	
valign	regfile	
vsa	regfile	
vbool	regfile	
mvec	regfile	Vector divide option
BBE_BMUL _STATE	state	LFSR & convolutional encoding option
BBE_BMUL _ACC	state	LFSR & convolutional encoding option
BBE_STATEA	state	FFT / symmetric FIR option
BBE_STATEB	state	FFT / symmetric FIR option
BBE_STATEC	state	FFT / symmetric FIR option
BBE_STATED	state	FFT / symmetric FIR option
CBEGIN	state	
CEND	state	
FFTCTRL	state	FFT option
RANGE	state	FFT option
FLUSH_TO_ZERO	state	Advanced precision multiply/add option

Name	Type	Description
BBE_PQUO0	state	Vector divide option
BBE_PQUO1	state	Vector divide option
BBE_PREM0	state	Vector divide option
BBE_PREM1	state	Vector divide option

Table 22: User Register Entries

User Register Name	User Register Number
BBE_STATEA	0-15
BBE_STATEB	16-31
BBE_STATEC	32-47
BBE_STATED	48-63
RANGE	96
FLUSH_TO_ZERO	97
BBE_BMUL_STATE	192-223
BBE_BMUL_ACC	224
BBE_PQUO0	64-71
BBE_PQUO1	72-79
BBE_PREM0	80-87
BBE_PREM1	88-95
FFTCTRL	241
CBEGIN, CEND	246, 247

☞ **Note:** User TIE should not use User Registers above 223)

- **Coprocessor Number: 1**
- **ctype Names:** All ConnX BBE64EP `ctype` names are reserved names. They are all prefixed with "xb_". In addition, the `ctype` starting with "`vbool`", "`vsa`", and "`valign`" are also reserved.
- **Operation Names:** All ConnX BBE64EP operation names are reserved names, which are either: Prefixed with BBE_, Named RUR.<User register name as defined above> or Named WUR.<User register name as defined above>.

- **Proto Names:** All ConnX BBE64EP intrinsic (proto) names are reserved intrinsic names. These all have the prefix "BBE_", or the prefix that begins with the `ctype` names "xb _", "`vsa`", "`vbool`", "`valign`" (for data type conversions protos).
- **TIE Function Names**: The ConnX BBE64EP uses a number of TIE functions, which you should not use for your own TIE functions. All functions have the prefix "bbe_", which should not be used by user TIE.
- **Semantic Names**: The ConnX BBE64EP uses a number of semantic names, which you should not use within your own TIE semantics. All semantics have the prefix "bbe_", which should not be used by user TIE.

8.5 XPG Configuration Options and Capabilities

The following tables (**Table 8- 4** through **Table 8- 9**) list the configuration options and any constraints on the ConnX BBE64EP .

Table 23: Simulation Modeling Capabilities

Capability	Allowed?
NGO flow	Yes
Pin-level XTSC	Yes
XTMP	Yes
XTSC	Yes

Table 24: Instruction Extension

Option	Allowed?
16AR	Yes
CLAMPS	Mandatory
MAC16	Yes
Extended L32R	No
MUL32 Implementation Selection	Pipelined with UH/SH
MUL16	Yes
DIV32	Yes
NSA (Normalized Shift Amount)	Mandatory
MINMAX (min and Max values)	Mandatory
SEXT (Sign Extended)	Mandatory
BOOLEANS	Mandatory

Option	Allowed?
Density instructions supported (16-bit)	Yes

Table 25: Allowed Architecture Definition

Option	Constraint
Load/Stores	Two
Action when handling an unaligned load/store	Take exception
Supported instruction widths	16, 24, 48, & 96 required

Table 26: Instruction Width

Option	Allowed?	Constraint
Pipeline length	Yes	Only seven-stage allowed.
Minimum number of interrupts needed		0
Minimum number of timers needed		0
Inbound PIF needed?	Not needed	
PIF widths supported		32, 64 and 128.
Byte Enables	Mandatory	
* (instrwidthbytes = 8)		

Table 27: Coprocessor Configuration Options

Option	Allowed?
Number of coprocessors	At least two
Single-precision floating point	Yes
Double-precision acceleration HW	Yes
Vectra LX	No
HiFi2	No
ConnX D2	No

Table 28: Local Memories

Option	Allowed?
D-Cache supported	No
I-Cache supported	Yes

Option	Allowed?
CAMMU	Yes
CAXLT	Yes
Full MMU (4KB page)	No
XLMI	No

Table 29: TIE Option Packages

Option	Allowed?
FLIX3	No
QIF32	Yes
GPIO32	Yes
User-defined TIE ports	Yes
User-defined TIE queues	Yes
User-defined TIE lookups (with writes)	Yes

8.6 Sample Configuration Templates for the ConnX BBE64EP

Sample configuration templates are provided in XPG for ConnX BBE64EP: XRC_B64EP_Base_DSP, XRC_B64EP_eNB_RX, XRC_B64EP_eNB_TX, XRC_B64EP_WiFi_4x4 & XRC_B64EP_MIN.These templates contain a valid core configuration and different sets of optional ISA packages available for ConnX BBE64EP. Effectively, these reference cores bracket a wide range of configuration possibilities for the ConnX BBE64EP. Users can start with any of these templates and modify configuration options, within the constraints defined by XPG, until a required configuration is built.

XRC_B64EP_MIN is a minimal configuration DSP. It represents a configuration that might be well suited for limited scope problems such as matrix operations. XRC_B64EP_Base_DSP is a DSP that includes support for filtering, FFT and fully featured matrix operations but without Advanced Precision and many of the other communication specific options. XRC_B64EP_eNB_RX is a fully loaded configuration suited for use in an eNodeB base station transceiver. It includes features such as Advanced Precision to support 4x4 MIMO operations. XRC_B64EP_eNB_TX is a reduced configuration for an eNodeB base station transmitter. It can result in a smaller area/power design when paired with and eNB_RX. XRC_B32EP_WiFi_4x4 is a heavy lifting DSP suitable for use in a 4x4 WiFi implementation.

For complete information on the base core that constitutes the XRC_B64EP_Base_DSP, XRC_B64EP_eNB_RX, XRC_B64EP_eNB_TX, XRC_B64EP_WiFi_4x4 & XRC_B64EP_MINonfiguration templates, see next section. Users are free to use any of these templates as a starting point, or not use a template at all, and then modify the

configuration options for their ConnX BBE64EP configuration within the constraints and assumptions defined in XPG and the build process.

Base Configuration

specifies the base core configuration in the XRC_B64EP_Base_DSP, XRC_B64EP_eNB_RX, XRC_B64EP_eNB_TX, XRC_B64EP_WiFi_4x4 & XRC_B64EP_MIN templates, which uses Xtensa ISA version LX6.0 and Xtensa Exception Architecture 2 (XEA2).

Table 30: ConnX BBE64EP Base Configuration

Option	Available
Xtensa ISA version	LX6.0
Instruction options	
16-bit MAC with 40 bit Accumulator	No
MUL16	Yes
MUL32	Pipelined + UH/SH
32 bit integer divider	Yes
Single Precision FP (coprocessor id 0)	No
Double Precision FP Accelerator	No
CLAMPS	Yes
NSA/NSAU	Yes
MIN/MAX and MINU/MAXU	Yes
SEXT	Yes
Boolean Registers	Yes
Number of Coprocessors (NCP)	3
Enable Density Instructions	Yes
Enable Processor ID	Yes
Zero-overhead loop instructions	Yes
Synchronize instruction	Yes
Conditional store synchronize instruction	Yes
TIE arbitrary byte enables	Yes

Option	Available
Count of Load/Store units	2
Max instruction width (bytes)	12
L32R hardware support option	Normal L32R
Pipeline length	7
Thread Pointer	No
GPIO32: 32-bit GPIO interface	No
QIF32: 32-bit Queue Interface	No
Interrupts enabled?	Yes
Interrupt count	17
Int 0 type / priority level	ExtLevel / 1
Int 1 type / priority level	ExtLevel / 1
Int 2 type / priority level	ExtLevel / 1
Int 3 type / priority level	ExtLevel / 1
Int 4 type / priority level	ExtLevel / 1
Int 5 type / priority level	ExtLevel / 1
Int 6 type / priority level	Timer / 1
Int 7 type / priority level	Software / 1
Int 8 type / priority level	ExtLevel / 2
Int 9 type / priority level	ExtLevel / 2
Int 10 type / priority level	Timer / 2
Int 11 type / priority level	Software / 2
Int 12 type / priority level	ExtEdge / 1
Int 13 type / priority level	ExtEdge / 1
Int 14 type / priority level	ExtEdge / 2
Int 15 type / priority level	ExtEdge / 2
Int 16 type / priority level	NMI / 4
High Priority Interrupts	Yes

Option	Available
Interrupt Level count	3
Medium Level Interrupts	Yes
Highest Medium Interrupt Level	2
Timer count	Yes
Timer count	2
Timer 0	6
Timer 1	10
Byte ordering (endianness)	Little Endian
Address registers available for call windows	32
Miscellaneous Special Register count	2
On unaligned load/store address	Exception
Enable Processor Interface (PIF)	Yes
PIF version 3.2	No
Asynchronous PIF interface	No
Write buffer entries	2
Enable PIF Write Responses	Yes
Prioritize Load Before Store	No
Widths of Cache and Memory Interfaces	
Width of Instruction Fetch interface	128
Width of Data Memory/Cache interface	512
Width of PIF interface	128
Instruction Cache	Not Selected
Data Cache	Not Selected
Debug	Yes
Data address breakpoint registers	2
Instruction address breakpoint registers	2
Debug interrupt level	3

Option	Available
On Chip Debug(OCD)	Yes
Use array of 4 Debug Instruction Registers (DIRs)	Yes
External Debug Interrupt	Yes
Trace port (address trace and pipeline status)	Yes
Add data trace	No
Xtensa Exception Architecture	XEA2
Memory Protection/MMU	Region Protection
System RAM start address / size	0x60000000 - 0x63ffffff / 64M
System ROM start address / size	0x50000000 - 0x50ffffff / 16M
Inbound PIF request buffer depth	2
Local Memory	
Instruction RAM start address / size	0x40000000 - 0x4001ffff / 128K
Instruction ROM start address / size	Not selected
Data RAM [0] start address / size	0x3ffc0000 - 0x3ffdffff / 128K
Data RAM [1] start address / size	0x3ffe0000 - 0x3fffffff / 128K
Data ROM start address / size	Not selected
XLMI start address / size	Not selected
Vector configuration	
Reset Vector start address / size	0x50000000 / 0x2e0
Kernel (Stacked) Exception Vector start address / size	0x400001dc / 0x1c
User (Program) Exception Vector start address / size	0x400001fc / 0x1c
Double Exception Vector start address / size	0x4000021c / 0x1c
Window Register Overflow Vector start address / size	0x40000000 / 0x178
Level 2 Interrupt Vector start address / size	0x4000017c / 0x1c
Level 3 Interrupt Vector start address / size	0x4000019c / 0x1c
Level 4 Interrupt Vector (NMI vector) start address / size	0x400001bc / 0x1c
Relocatable Vectors	Yes

Option	Available
Selected Static Vector set	Primary
Primary Static Vector Group Base Address	0x50000000
Alternate Static Vector Group Base Address	0x40000240
Alternate Reset Vector Address	0x40000240
Default Dynamic Vector Group Base Address (VECBASE)	0x40000000
Target & CAD options [Nominal]	
RTL description	Verilog
Synthesis / P&R flow	Physical Synthesis, Route
Geometry / Process	45gs / Worst
Core Speed	976 MHz
User Defined Estimator Library	Default
User Area Scaling factor	1.0
User Speed Scaling factor	1.0
User Dynamic Power Scaling factor	1.0
User Leak Power Scaling factor	1.0
User Area To Gate Scaling factor	1.0
Functional Unit Clock Gating	Yes
Global Clock Gating	Yes
Register file implementation block (Latches are deprecated)	Flip-flops
Asynchronous Reset	No
Full scan	Yes
Software Target Options	
Xtensa Tools should use Extended L32R	No
Software ABI	Windowed
C Libraries	Newlib
Compatibility Checking	

Option	Available
Generic RTOS compatibility	No
Target Linux compatibility	No

Differences in Reference Configurations

The following table highlights all the differences between the reference configuration templates.

Table 31: ConnX BBE64EP Configuration Options

Option	MIN	Base_DSP	eNB_RX	eNB_TX	WiFi_4x4
Integer and fractional vector divide	No	No	No	Yes	No
Fast vector reciprocal & reciprocal square root	Yes	Yes	Yes	Yes	Yes
Advanced vector reciprocal & reciprocal square root	No	Yes	Yes	No	Yes
Bit-multiplication support for LFSR applications and convolutional encoding	No	No	Yes	Yes	No
1D despread	No	No	Yes	Yes	Yes
Packed complex matrix multiply	No	No	Yes	No	Yes
Symmetric FIR	No	Yes	Yes	Yes	Yes
Linear block decoder	No	No	Yes	No	No
FFT	No	Yes	Yes	Yes	Yes
3GPP soft-bit demap	No	No	No	No	Yes
Advanced precision multiply/add	No	No	Yes	No	No
Inverse LLR	No	No	No	No	No

Option	MIN	Base_DSP	eNB_RX	eNB_TX	WiFi_4x4
Single and dual peak search	No	No	Yes	No	Yes

8.7 Synthesis and Place-and-Route

When the ConnX BBE64EP coprocessor is included in an Xtensa processor configuration, the synthesis and place-and-route scripts that are included with the software build can be used with the usual methodology, which is outlined in the *Xtensa LX Hardware User's Guide*.

For timing closure between synthesis and place-route, Tensilica® recommends using a layout aware synthesis tool. Because of the data path-intensive nature of the ConnX BBE64EP netlist, the tool may need to be further controlled with some of its parameters depending upon the process technology, the foundry, and the library vendor.

8.8 ConnX BBE64EP Memory Floor-planning Suggestions

Tensilica® provides an automated, push-button, standard cells layout flow for ConnX BBE64EP. Before proceeding with floor-planning ConnX BBE64EP with memory, it is imperative that out-of-the-box push-button flow is exercised for ConnX BBE64EP. The advantage is that it will provide accurate measure of flop-to-flop timing inside ConnX BBE64EP without extraneous factors such as specific memory placement or floor-plan dimensions that may cause layout tool to severely degrade the performance of ConnX BBE64EP. Secondly, it helps identify potential timing issues on memory paths based on only on memory timing budgets. Finally, the flow provides an estimate of ConnX BBE64EP size to determine the dimensions of floor-plan for ConnX BBE64EP with its associated memories.

The ConnX BBE64EP interfaces with instruction and data memories. The instruction memories interface with the Program Counter and Instruction Fetch (PCIF) unit in ConnX BBE64EP. The data memories interface with the Load/Store (LS) unit in ConnX BBE64EP. The layout tool attempts to place standard cells for PCIF and LS units close to instruction and data memories respectively. When all memory instances are placed on one side of the floor-plan, the tendency of layout tools to place standard cells of both PCIF and LS units close to the memory will likely cause congestion. When instruction memories are placed on one side and data memories on other, the congestion at memory interfaces is alleviated. As a general guideline:

• The data memory interfaces must be contiguous,
• Instruction memory interfaces must be contiguous,
• Data and instruction memory interfaces should not be intermixed.

The number of pipeline stages in ConnX BBE64EP is 7. With large memories, the ConnX BBE64EP is configured with 7-pipeline stages so that a full cycle is available to register the

memory data. Specifying accurate memory timing budgets in synthesis ensures that netlist generated by synthesis tool right sizes the gates and inserts buffers to address the timing on memory interface paths. This minimizes the risk of congestion causing area increase on memory interface paths during layout.

ConnX BBE64EP muxes data from each instruction cache way, instruction ram 0, and instruction ram 1 on the instruction memory interface. Depending on the instruction memory interface width, each of instruction cache way and instruction rams may be implemented as a bank of memory instances. For example, a 128-bit instruction memory interface may be implemented using four banks, each one with 32-bit interface. As data is muxed inside ConnX BBE64EP, it is useful to interleave the memory banks. For example, consider 128-bit interface for instruction memory configuration of Instruction Ram 0 (IRam0) and Instruction Ram 1 (IRam1), each of which is 64 KB. Each instruction ram is implemented as 4 banks of 16 KB with 32-bit interface (IRam<N>_0, IRam<N> _1, IRam<N> _2, and IRam<N> _3). In this situation, the floor-plan should interleave IRam0 and IRam1 instances by placing corresponding bank instances next to each other (e.g. IRam0 _0 next to IRam1 _0). The same guideline is applicable for data memory interface. The data from ConnX BBE64EP to memory (memory write data) and memory address signals need to be routed to each memory bank. For this 7-stage machine, the address setup path may be timing critical. In this situation, interleaving the memory banks may lead to accentuating the congestion and timing challenge at memory address and write data pins. In summary, interleaving instances for memory banks is one of the alternatives that should be carefully considered for creating ConnX BBE64EP floor-plan.

In addition to considering memory interfaces, it is important to allocate sufficient floor-plan space for ConnX BBE64EP. Our experience suggests that a starting utilization of 60% is a reasonable preliminary estimate for floor-planning experiments. The starting utilization refers to the floor-plan area allocated to ConnX BBE64EP standard cells divided by the post-synthesis cell area. The starting utilization ratio need to be adjusted down if floor-plan dimensions and memory instance dimensions leads to notches and narrow channels. The ratio may be adjusted up if the design easily meets the timing.

As with any floor-plan creation, the design of the ConnX BBE64EP floor-plan is partly art and partly science. The first level consideration must take into account the following:

- Size and aspect ratio of floor-plan space allocated to ConnX BBE64EP and associated memories
- Data and instruction memory configuration
- Relative size of ConnX BBE64EP and its associated memories

At this stage it is important to estimate the memory sizes and floor-plan real estate available to place ConnX BBE64EP standard cells. It is important that sufficient notch-free and channel-free space is available for placement of the ConnX BBE64EP standard cells.

The next level of detail must evaluate banking structure to implement the ConnX BBE64EP memory configuration. The banking may be along "data bits" or "address bits" (for example, top 2 bits to select data from one of the four banks) or both. The memory banking enables use of smaller memory instances instead of a large monolithic instance or may be the only

choice when it is not possible to generate a monolithic instance. In addition, the column mux ratio may be employed as a parameter to determine the memory instance dimensions in the context of the floor-plan dimensions.

The logical connectivity of instruction and data memories to ConnX BBE64EP units must be exploited to place the memory instances. It is important to differentiate type of memories from the ConnX BBE64EP perspective and place memory of a specific type contiguously in the floor-plan. The interleaving of memory bank instances must be considered to improve the congestion characteristics of the floor-plan.

For optimal congestion-free ConnX BBE64EP floor-plan that meets the timing, the importance of early, floor-plan exploration experiments cannot be overemphasized.

8.9 Mapping the ConnX BBE64EP to FPGA

Experiments have been carried out mapping the ConnX BBE64EP to Xilinx Kintex/Virtex 7 FPGAs. The characterization runs were carried out on ConnX BBE64EP configurations from a previous release (RE-2014.0). These results, as shown in the tables below, should be similar to a ConnX BBE64EP configuration in the current RF-2014.1 release.

ConnX BBE64EP Synthesis - Xilinx

Table 32: Xilinx Synthesis Results (RE-2014.0)

Config	Device	Utilization	Target Frequency	Achieved Frequency
XRC_B64EP_MIN	7vx690t	92	33.0MHz	33.86MHz

☞ **Note:** The XRC_B64EP_MAX configurations used for the FPGA mapping experiment do not include the following three configurable options:

- Soft-bit demapping
- Advanced precision multiply/add
- Inverse log-likelihood ratio (LLR)

9. On-Line ISA, Protos and Configuration Information

Xtensa Xplorer offers a number of on-line Instruction Set Architecture (ISA) documentation references in HTML format for Xtensa configurations, including the ConnX BBE64EP. These references are accessed from the Xplorer's Configuration Overview window by clicking the View Details button from the Installed Builds column. Note: Be sure to click the View Details button from the Installed Builds column, and not from the Workspace Config column, which displays configuration information only.

When you click View Details, an HTML page is displayed within Xplorer (on Windows) and in your web browser (on Linux). From this page you can access a number of HTML pages describing the ISA, protos, and other documentation information. These are listed under the Instruction Set Architecture area and the Hardware Documentation area. The information offered includes:

- Instruction Formats, including a complete operation slot assignment table called the "Operation Slot Compatibility Matrix"
- Instruction Descriptions, sometimes informally called "ISA HTML". This is an HTML page describing each instruction in some detail
- Instruction Opcodes
- C-Types, Operators, and Instruction Prototypes (the latter also known as "Protos")

    - List of Prototypes (Categorized by Functional Class)
    - Description of Prototypes (C Function Protoype, TIE Definitions)
- Instruction Pipelining
- State List for ISA internal state
- A hardware Port List for the configuration, under the Hardware Documentation area

The Instruction Descriptions contain live links to the Instruction Prototypes within which these instructions are used. In addition, the Instruction Prototypes contain live

links to the particular instruction descriptions that they use.