cadence®

ConnX BBE16 DSP Engine

User's Guide

For Xtensa Processor Cores

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

Issue Date:11/2014

RF-2014.1

PD-14-3222-10-01

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

Contents

1. ConnX BBE16 Overview 1
1.1 Purpose of this User Guide 1
1.2 Installation Overview 2
1.3 ConnX BBE16 Architecture Overview 2
1.4 ConnX BBE16 Instruction Set Overview 4
1.5 ConnX BBE16 Programming Models and XCC Vectorization 6
1.6 Changes in Instruction Set Architecture (ISA) between LX3 (RC Release) and LX4 (RD Release) 6
2. ConnX BBE16 Features 9
2.1 ConnX BBE16 Architecture Behavior 9
2.2 ConnX BBE16 ISA Concepts 10
2.3 Instruction Naming Conventions 10
2.4 Fixed Point Values and Fixed Point Arithmetic 15
2.4.1 Representation of Fixed Point Values 16
2.4.2 Arithmetic with Fixed Point Values 17
2.5 Data Types Mapped to the Vector Register File 17
2.5.1 Scalar Data Types Mapped to the Vector Register File 17
2.5.2 Vector Data Types Mapped to the Vector Register File 19
2.6 Data Typing 20
2.7 Vector Groups 21
2.7.1 Load and Store Groups 23
2.7.2 Multiply Groups 24
2.7.3 Other Groups 27
2.8 ConnX BBE16 Multiplies: Real and Complex Arithmetic 30
2.9 Select Instructions 34
2.10 Vector Initialization and Some Additional Select Patterns 35
2.11 BBE_REP8X20 and BBE_ESHFT8X20 Special Instructions 37
2.12 Block Floating Point 38
2.13 Butterfly Instructions 39
2.14 4-Way Complex Conjugate Instructions 39
2.15 Vector Polynomial Evaluation Instructions: BBE_POLY8X20_SU and BBE_POLY8X20_STEP 39
2.15.1 BBE_POLY8X20_SU: Setup Polynomial Evaluation 40
2.15.2 BBE_POLY8X20_STEP: Evaluation of one Step of the Polynomial Evaluation Process 40
2.15.3 Example with Sine 42
Example Code for Sine Computation 43
2.15.4 Cosine Computation 45
2.15.5 Derivation of Coefficients 46
2.16 Dual Peak Max Index and Max Absolute Index and Values 47
2.16.1 Example using Dual Peak Max index instructions 49
2.17 Despread Operation (Optional) 50
2.17.1 Hadamard Transform Operations (part of Optional Despread Operation) 53
2.18 Divide and Reciprocal Square Root Instructions (Optional) 54

2.19 FLIX Slots and Formats 55
2.20 Instruction List – Showing Slot Assignments 55
3. Programming ConnX BBE16 57
3.1 ConnX BBE16 Data Types 57
3.1.1 Automatic Type Conversion 60
Converting int16 and int32 (C short and C int) to all Types 60
Converting Between Unguarded Types to Guarded Types 60
Converting Between Guarded Types to Unguarded Types 61
Other Type Conversions, Including Manual Conversions 62
3.2 Xtensa Xplorer Display Format Support 63
3.3 Operator Overloading and Vectorization 64
3.4 Prerequisite Reading 64
3.5 Programming Styles 64
3.6 Auto-Vectorization 65
3.7 Operator Overloading and Inferencing 66
3.8 Vectorization and Inferencing Examples 67
3.9 Manual Vectorization 71
3.10 Intrinsic-Based Programming 72
3.11 Using the Two Local Data RAMs and Two Load/Store Units 73
3.12 Other Compiler Switches 76
3.13 TI C6x Intrinsic Porting Assistance Library 76
3.13.1 Porting TI C6X Code Examples 77
Example 1: _mpy 77
Example 2: _add2 79
Example 3: _add2 and _sub2 80
Manual Vectorization 80
4. FFT Instructions 81
4.1 FFT Acceleration 81
4.2 FFT Control Register 81
4.3 General FFT Structure 82
4.4 Summary of ConnX BBE16 FFT Operations and Instructions 84
4.5 BFL Operation - Radix-4 DIF Butterfly 84
4.6 FFT Normalization 85
4.7 FFT Dynamic Range 86
4.8 Inner Loop C Code for FFT 86
4.9 BFLR4 Operation 87
4.10 BFLR2R4 Operation 88
4.11 FFT_ADD3MUL Operation 89
4.12 ConnX BBE16 FFT Instruction Usage 92
4.13 Radix-4 DIF FFT Result Order 93
4.14 Bit-Reversed Addressing 94
4.15 BBE_SR8X16S.XU: FFT Indexed Store with Update 96
5. General Multi-Mode, Multi-Mode Multiply and Multiply-Add Instructions with Extended Precision 97
5.1 Details of the Instructions 99
5.1.1 32-Bit Select Operand (sel) 100
5.2 Using the Multi-Way, Multi-Type, Multi-Mode Multiply and Multiply-Add Instructions 105
5.2.1 Real 2x2 and 4x4 Matrix Multiply Modes 107

5.2.2 Complex 2x2 and 4x4 Matrix Multiply Multi Modes 109
5.2.3 FIR Operations 110
5.3 Examples 115
5.3.1 2x2 Complex Matrix Multiply 115
5.3.2 2x2 Real Matrix Multiply 117
5.3.3 16-Tap Complex-Complex FIR Example 118
6. Load/Store Instructions 121
6.1 Load and Store Intrinsic Names 121
6.2 ConnX BBE16 Addressing Modes 122
6.3 Circular Addressing 122
6.4 Aligning Loads and Stores 123
6.5 Vector Compacting Aligning Stores 125
6.6 Pre-Decrementing in Updating Loads and Stores 126
7. Implementation Methodology 127
7.1 Configuring a ConnX BBE16 127
7.2 XPG Estimation for ConnX BBE16 Size, Performance, and Power 130
7.3 Basic ConnX BBE16 Characteristics 130
7.4 Extending a ConnX BBE16 with User TIE 130
7.4.1 Compiling User TIE 133
7.4.2 User Warning to Ignore 133
7.4.3 Name Space Restrictions for User TIE 133
7.5 XPG Configuration Options and Capabilities 136
7.6 Optional Configuration Templates for ConnX BBE16 138
7.6.1 XRC_B16LP Configuration 139
7.6.2 XRC_B16PM Configuration 145
7.7 Synthesis and Place-and-Route 151
7.7.1 CadSetup.file Variable Selection for Optimal Performance 151
7.7.2 DRC Fanout Constraint Suggestions 151
7.8 ConnX BBE16 and Memory Floorplanning Suggestions 152
7.9 Mapping ConnX BBE16 to FPGA 154
A. On-Line ISA, Protos, and Other ConnX BBE16 Information 157
A.1 Protos 157
A.1.1 Extract Protos 158
Extract Protos Example 160
A.1.2 Combine Protos 160
Combine Protos Example 164
A.2 Operator Protos 165
B. Vectra LX Compatibility 167
B.1 Vectra LX Specific Complex and Real Vector Multiplies and Multiply-Adds 171

List of Figures

Figure 1–1. Architecture of ConnX BBE16 3
Figure 2–2. Basic Multiplier Details 31
Figure 2–3. Low-Precision Real Multiply Example 33
Figure 2–4. High-Precision Real Multiply Example 33
Figure 2–5. High-Precision Complex Multiply Example 34
Figure 2–6. BBE_ESHFT8X20 Flow Example 37
Figure 2–7. BBE_DSPR4XC20 Codeword Format 51
Figure 2–8. BBE_DSPR4XC20 Operation Detail for Complex Outputs 52
Figure 2–9. BBE_DSPRA4XC20 Operation Detail for Complex Accumulators 52
Figure 4–10. Mixed Radix (4/2) Parallel 32 Point FFT 83
Figure 4–11. Radix-4 FFT butterfly 84
Figure 4–12. FFT Normalization 86
Figure 4–13. Radix-4 Bit-Reversed Order Result 93
Figure 5–14. Complex FIR and Matrix Multiplier 98
Figure 5–15. Structure of Quad-Multiplier Element 99
Figure 5–16. 16-Tap Real Symmetric FIR Example 115
Figure 5–17. 2x2 Complex Matrix Multiply 116
Figure 5–18. 2X2 Real Matrix Multiply Example 118
Figure 7–19. ConnX BBE16 Configuration Options in Xplorer 128
Figure B–20. MUL18 Even Data Path (MUL18.0) 172
Figure B–21. MUL18 Odd Data Path (MUL18.1) 172
Figure B–22. RMUL18 Data Path 172
Figure B–23. IMUL18 Data Path 173
Figure B–24. RMUL18 Details 173
Figure B–25. IMUL18 Details 174

List of Tables

Table 1–1. ConnX BBE16 64-Bit Instruction Formats 4
Table 2–2. ConnX BBE16 Instruction Name Categories 10
Table 2–3. Naming Conventions for ConnX BBE16 Load/Store Instructions and Intrinsics 14
Table 2–4. Vector Data Types Mapped to Vector Register Files 19
Table 2–5. Scalar Memory Data Types 20
Table 2–6. Scalar Register Data Types 20
Table 2–7. Vector Memory Data Types 20
Table 2–8. Vector Register Data Types 21
Table 2–9. Major Vector Load Group 21
Table 2–10. Major Vector Store Group 22
Table 2–11. Vector Pack and Store and Pack Group 23
Table 2–12. Load/Store Special and Support Group 23
Table 2–13. Multiply to 40-bit Result Group 24
Table 2–14. Multiply to 20-bit Result Group 25
Table 2–15. Special Multiply Group 25
Table 2–16. Shift Group 26
Table 2–17. Vector Arithmetic and Logical Group 26
Table 2–18. Reduction Instructions Group 27
Table 2–19. Moves Group 27
Table 2–20. Boolean Set Instructions Group 29
Table 2–21. Complex Conjugate Group 30
Table 2–22. Dual Peak Max Index Group 30
Table 2–23. Multiply Instructions 32
Table 2–24. BBE_8X20SELI Type Definitions 35
Table 2–25. BBE_MOVVI8X20 Instruction Arguments 35
Table 2–26. BBE_SEL4V8X20 Instruction Arguments 36
Table 2–27. POLY_SU Arguments 42
Table 2–28. Codeword Values 51
Table 2–29. Codeword Values 53
Table 3–30. Scalar Memory Data Types 57
Table 3–31. Scalar Register Data Types 57
Table 3–32. Vector Memory Data Types 58
Table 3–33. Vector Register Data Types 58
Table 3–34. Unguarded Types to Guarded Types Conversion 61
Table 3–35. Guarded Types Converted to Unguarded Types 61
Table 4–36. MODE[2:0] Register 82
Table 4–37. FFTOP[1:0] Register 82
Table 4–38. ConnX BBE16 FFT Operations 84
Table 4–39. Radix-4 Butterfly Outputs 85
Table 4–40. Sel Outputs 85
Table 4–41. FFT Instructions 92
Table 4–42. FFT Bit-Reversed Order 93
Table 4–43. FFT Control State Registers 95
Table 4–44. B_pos FFT Sizes 95

Table 5–45. 32-Bit Select Operand Fields 101
Table 5–46. Mode and Resulting Offsets 105
Table 5–47. Valid Combinations of the Select Operand Fields 105
Table 6–48. Data Types for ConnX BBE16 Load/Stores 121
Table 6–49. ConnX BBE16 Addressing Modes 122
Table 6–50. Register Instructions 123
Table 7–51. Simulation Modeling Capabilities 136
Table 7–52. Instruction Extensions Configuration Options and Constraints 136
Table 7–54. Instruction Width* Configuration Options and Constraints 137
Table 7–55. Coprocessor Configuration Options 137
Table 7–56. Local Memories Configuration Options 137
Table 7–53. Allowed Architecture Definition Configuration Options and Constraints 137
Table 7–57. TIE Option Packages 138
Table 7–58. XRC_B16LP Instruction Details 139
Table 7–59. Configuration Options 140
Table 7–60. Interrupt Number Details 141
Table 7–61. PIF Option Details 142
Table 7–62. Cache and Memory Interface Widths 142
Table 7–63. Debug Details 142
Table 7–64. Memory Protection/MMU (Region Protection) 142
Table 7–65. Local Memory Details 143
Table 7–66. Vector Configuration Details 143
Table 7–67. Relocatable Vector Details 143
Table 7–68. Target and CAD Options 143
Table 7–69. User-Defined Estimator Library Scaling Factors 144
Table 7–70. Software Target Options 144
Table 7–71. Compatibility Checking Features 144
Table 7–72. Instruction Details 145
Table 7–73. Configuration Options 146
Table 7–74. Interrupt Details 147
Table 7–75. PIF Option Details 148
Table 7–76. Cache and Memory Interface Widths 148
Table 7–77. Debug Details 148
Table 7–78. Memory Protection/MMU (Region Protection) 149
Table 7–79. Local Memory Details 149
Table 7–80. Vector Configuration Details 149
Table 7–81. Relocatable Vector Details 149
Table 7–82. Target and CAD Options 150
Table 7–83. User Defined Estimator Library Scaling Factors 150
Table 7–84. Software Target Options 150
Table 7–85. Compatibility Checking Features 150
Table B-86. Vectra LX Instructions Mapped to ConnX BBE16 167

Preface

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- $ represents your shell prompt, in user-session examples.
- **`literal_input`** indicates literal command-line input.
- *`variable`* indicates a user parameter.
- `literal_keyword` (in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- *`... output ...`* indicates unspecified program output.
- *`[optional-variable]`* indicates an optional parameter.
- **`[`***`variable`***`]`** indicates a parameter within literal square-braces.
- **`{`***`variable`***`}`** indicates a parameter within literal curly-braces.
- **`(`***`variable`***`)`** indicates a parameter within literal parentheses.
- | means *OR*.
- *`(var1 | var2)`* indicates a required choice between one of multiple parameters.
- *`[var1 | var2]`* indicates an optional choice between one of multiple parameters.
- *`var1 [, varn]*`* indicates a list of 1 or more parameters (0 or more repetitions).
- `4'b0010` is a 4-bit value specified in binary.
- `12'o7016` is a 12-bit value specified in octal.
- `10'd4839` is a 10-bit value specified in decimal.
- `32'hff2a` or `32'HFF2A` is a 32-bit value specified in hexadecimal.

Terms

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

Related Xtensa Documents

Customizable DPU Data Books

- *Xtensa® LX6 Microprocessor Data Book*
- *Xtensa® 11 Microprocessor Data Book*
- *Xtensa® Instruction Set Architecture (ISA) Reference Manual*

DPU User Guides and Reference Manuals

ConnX Communications DPUs

- *ConnX D2 DSP Engine User's Guide*
- *ConnX Vectra™ LX DSP Engine Guide*
- *ConnX BBE16 DSP Engine User's Guide*
- *ConnX BBE32EP DSP User's Guide*
- *ConnX BBE64EP DSP User's Guide*
- *ConnX BSP3: Bit Stream Processor 3 Guide*
- *ConnX SSP16: Soft Stream Processor User's Guide*

HiFi Audio DSPs

- *HiFi Audio Engine Codecs Programmer's Guides*
- *HiFi 2/EP Audio Engine User's Guide*
- *HiFi 3 Audio Engine User's Guide*
- *HiFi 4 Audio Engine User's Guide*
- *HiFi Mini Audio Engine User's Guide*

Hardware Design Guides

SOC Implementation

- *Xtensa LX Hardware Implementation and Verification Guide*
- *Xtensa® System Designer's Guide*
- *Xtensa Bus Designer's Toolkit Guide*
- *Xtensa Bus Interface User's Guide*
- *Xtensa® Processor Interface Protocol Reference Manual*
- *Xtensa® Prototyping User's Guide for the Xilinx ML605 (XT-ML605) Board*
- *Xtensa® Prototyping User's Guide for the Xilinx KC705 (XT-KC705) Board*

TIE Creation and Processor Optimization

- *Tensilica Instruction Extension (TIE) Language Reference Manual*
- *Tensilica Instruction Extension (TIE) Language User's Guide*

- *Xtensa® Upgrade Guide*

Software Development and Simulation Tools Guides

- *Xtensa® Software Development Toolkit User's Guide*
- *Xtensa® Development Tools Installation Guide*
- *Xtensa C Application Programmer's Guide*
- *Xtensa® C and C++ Compiler User's Guide*
- *Xtensa® Linker Support Packages (LSPs) Reference Manual*
- *Xtensa® OSKit™ Guide*
- *Xtensa® Microprocessor Programmer's Guide*
- *Xtensa® System Software Reference Manual*

System Simulation

- *Xtensa® Instruction Set Simulator (ISS) User's Guide*
- *Xtensa® Modeling Protocol (XTMP) User's Guide*
- *Xtensa® SystemC® (XTSC) Reference Manual*
- *Xtensa® SystemC® (XTSC) User's Guide*
- *Xtensa® SystemC® (XTSC) for Carbon SoC Designer User's Guide*

Utilities and Libraries

- *GNU Assembler User's Guide*
- *GNU Binary Utilities User's Guide*
- *GNU Debugger User's Guide*
- *GNU Linker User's Guide*
- *GNU Profiler User's Guide*
- *Red Hat newlib C Library Reference Manual*
- *Red Hat newlib C Math Library Reference Manual*

Debug and Trace Product Guides

Xtensa Debug User's Guide

Changes from the Previous Version

The following changes were made to this document for the Cadence Tensilica RF-2014.1 release of ConnX BBE16:

Additional details added:

- ConnX BBE16 customization requirements in Section 7.1.
- XPG estimation calculations in Section 7.2.
- Example FLIX format in Section 7.4.
- Semantic name usage in Section 7.4.3.

1. ConnX BBE16 Overview

The ConnX BBE16 (16-MAC Baseband Engine) is a high performance DSP designed for use in next generation communication baseband processors, such as those found in LTE and 4G cellular radios and multi-standard broadcast receivers. The high computation requirements in such applications require new and innovative architectures with a high degree of parallelism and efficient I/Os. The ConnX BBE16 DSP Engine meets these needs by combining a 8-way SIMD, 3-slot VLIW processing pipeline with a rich and extensible set of interfaces.

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

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

1.1 Purpose of this User Guide

This guide provides an overview of the ConnX BBE16 architecture and its instruction set. It will help programmers using ConnX BBE16 by identifying some of the techniques that are commonly used to vectorize algorithms. It gives guidelines to achieve improved performance by using ConnX BBE16 instructions, intrinsics, protos, and primitives. This guide also serves as a C/C++ usage reference for the appropriate way to use ConnX BBE16 features in C/C++ software development. This guide will also assist Xtensa BBE16 users who wish to add additional instructions to the BBE16 architecture.

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

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

1.2 Installation Overview

To install a ConnX BBE16 configuration, follow the same procedures described in the *Xtensa Development Tools Installation Guide.* The ConnX BBE16 library of example source code files is provided as a separate file of examples in an Xtensa Xplorer workspace called `bbe16_examples.xws`.

The ConnX BBE16 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_bbe16.h`

There are also some specialized include files for Vectra LX compatibility, which are discussed in Appendix B. Another additional helpful include file is xt_bbe16_fft.h, which assists in writing FFT routines. Finally, there is an include header file for TI C6x source code compatibility, which is discussed in Chapter 3. This include file maps TI C6x intrinsics into standard C code and is meant as a porting assist only.

1.3 ConnX BBE16 Architecture Overview

ConnX BBE16, a SIMD (single-instruction/multiple-data) processor, has the ability to work in parallel on several data items at the same time. ConnX BBE16 executes a single operation simultaneously across different data elements. For example, it allows for eight 20-bit additions in parallel or four 40-bit additions in parallel. It inherits many of the features of our ConnX Vectra LX DSP Engine, in a new form, but adds a wide range of new operations and programmer state, to achieve very high performance on DSP operations typically found in 3G and 4G wireless communications and other high-data rate PHY layer processing. These operations include optimized instructions for FFT, complex multiplication and multiply-accumulation, vector division (optional), vector reciprocal square root (optional), despreading (optional) and other high performance operations.

ConnX BBE16 is also a 3-way VLIW architecture, in which three operations are dispatched in parallel every cycle. This allows the processor to sustain 16 multiply-accumulate operations in parallel with a load of eight operands and a store of eight results in every cycle.

To supply the memory bandwidth required for this high computation rate, the ConnX BBE16 contains two independent load/store units, which can communicate with the two local data memories. Each data memory interface is 128-bits wide. At 400MHz, this provides almost 13GB per second of data memory bandwidth.

For efficiency, ConnX BBE16 fetches instructions out of a 64-bit wide local instruction memory. The processor can also read and write system memory and devices attached to standard system buses. Other processors or DMA engines can transfer data into and out of the local memories, in parallel with the processor pipeline. The processor can also have an instruction cache.

ConnX BBE16 can also be implemented with any number of wide, high-speed I/O interfaces (TIE ports and queues) to directly control devices or hardware blocks, and to move data directly into and out of the processor register files.

The ConnX BBE16 architecture (as illustrated in Figure 1–1) uses variable length instructions, with encodings of 16- and 24-bits for its baseline RISC instructions, and 64-bits for the three VLIW operations that may be issued in parallel. The Xtensa compiler optimally schedules different operations into the three VLIW slots as required by the program. The first VLIW instruction slot is used to issue load/store operations or Xtensa core instructions, and some specialized BBE16 instructions. The second instruction slot allows for real and complex multiply, FFT butterfly, FIR, or vector select operations. The third instruction slot is used for load/store operations using the second load/store unit as well as arithmetic and logical operations.

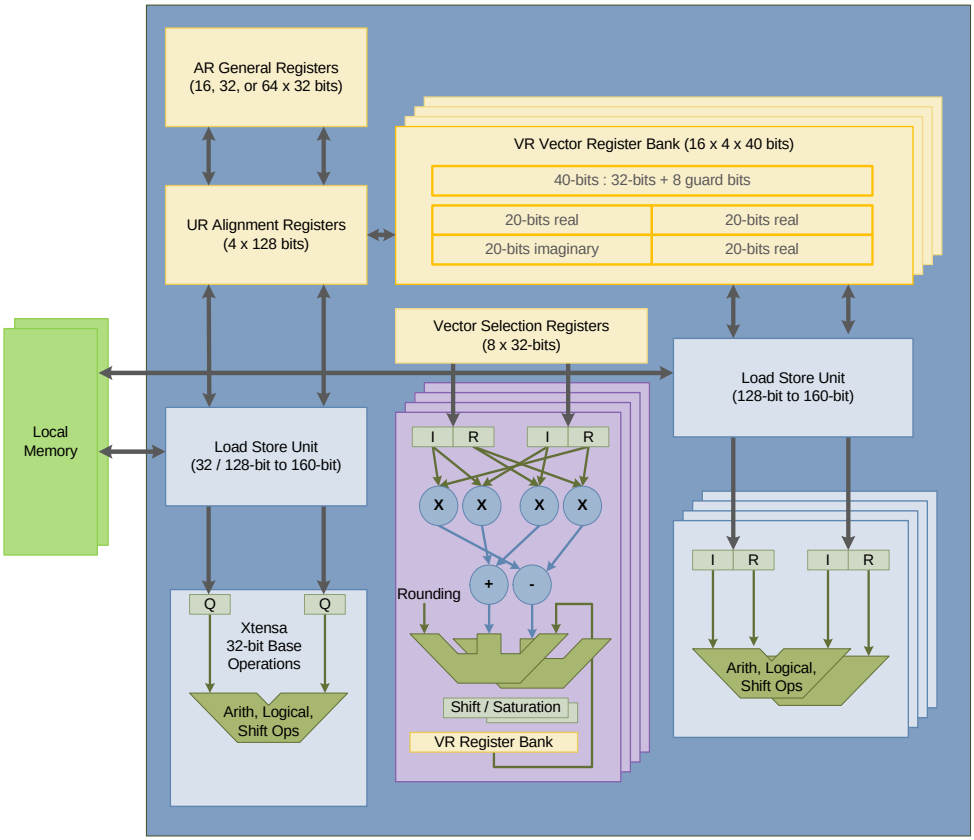


Figure 1–1. Architecture of ConnX BBE16

Table 1–1 illustrates the basic instruction formats supported by ConnX BBE16. Note that the third 64-bit format is available for user-defined instructions.

Table 1–1. ConnX BBE16 64-Bit Instruction Formats

BBE16 Format 0 (x 64)		
Slot 0 (vsLDST)	Slot 1 (vsMAC)	Slot 2 (vsALU)
• Load/Store Unit 0 • Base Instructions • Scalar Arithmetic • Move Instructions • Scalar Logical Operations	• Real and Complex MAC subset • Element Select	• Load/Store Unit 1 subset • Vector ALU and Logic • Vector Pack • Shift
BBE16 Format 1- bbe_fftfir		
Slot 0 (fftfir0)	Slot 1 (fftfir1)	Slot 2 (fftfir2)
• Load Unit 0 subset • Twiddle Loads	• FFT Butterfly • Real and Complex MAC • Multi-mode Multiply • Element Select • Optional Vector Divide • Optional Reciprocal Sqrt • Optional Despread	• Load/Store Unit 1 • Bit-reverse stores • Vector Pack • Shift • Vector ALU • Vector Select • Moves • Polynomial
Slot(s)		
• Available for User TIE		

1.4 ConnX BBE16 Instruction Set Overview

The ConnX BBE16 DSP Engine is built on the baseline Xtensa RISC architecture, which implements a rich set of generic instructions optimized for efficient embedded processing. The power of ConnX BBE16 comes from a comprehensive DSP and baseband instruction set with over 500 instructions. A wide variety of load/store operations supports six different addressing modes with support for 16b/32b scalar and vector data types. Unaligned load/stores with masking deliver full bandwidth loads and stores for unaligned data. Vector data management is supported with data packing and shifting.

Multiply operations include complex and real 18bx18b multiply, multiply-round, multiply-add and multiply-subtract functions. Complex-number functions include support for conjugate arithmetic and magnitude operations as well as full precision arithmetic and saturated/rounded outputs. ConnX BBE16 is capable of performing up to 16 multiplies per operation. ConnX BBE16 includes extended precision with guard bits on all register data, full support of double precision data, and 40-bit accumulation on all MAC operations

without performance penalty. A wide variety of rich arithmetic, logical, and shift operations are supported for up to eight data words per cycle. There is full support for matrix multiplication with acceleration for OFDM matrix operations.

ConnX BBE16 directly supports single cycle radix-4 and combined radix-4/radix-2 butterfly operations enabling efficient high-speed FFT implementations. Support for a single cycle 4-tap FIR filter and single cycle 16-tap FIR filter with real taps allows efficient filtering operations. Special instructions supporting radix 3/5 FFT are also provided. Symmetric filters on real and complex data at double rate are also supported.

Complex FFT									
Size (complex points)		128	256	512	1024	2048	4096	8192	32768
FFT – natural order	cycles	226	455	804	1,784	3,456	7,841	15561	70,098

Complex FIR							
Size (complex taps, complex data)		512	1024	2048	4096	8192	16384
FIR –16 tap	MAC/cycle	15.72	15.86	15.93	15.96	15.98	15.99

Real FIR							
Size (real taps, real data)		512	1024	2048	4096	8192	16384
FIR –16 tap	MAC/cycle	14.98	15.47	15.73	15.86	15.93	15.97

Real Tap Symmetric FIR							
Size (real taps, complex data)		512	1024	2048	4096	8192	16384
FIR –16 tap	Effective MAC/cycle	25.05	25.97	26.45	26.69	26.90	26.88

Real Symmetric FIR							
Size (real taps, real data)		512	1024	2048	4096	8192	16384
FIR –16 tap	Effective MAC/cycle	27.96	29.84	30.88	31.43	31.71	31.86

For further application acceleration, optional instruction packages are available for 8-way SIMD integer and fractional divide, 4-way SIMD reciprocal square root and despreading functions (16 complex MACs/cycle). The ConnX BBE16 instruction set is described in more detail in Chapter 2.

1.5 ConnX BBE16 Programming Models and XCC Vectorization

The ConnX BBE16 DSP Engine supports a number of programming models -- including standard C/C++, ConnX BBE16-specific integer and fixed-point data types with operator overloads and a level of automated vectorization, scalar intrinsics and vector intrinsics.

ConnX BBE16 contains a number of integer and fixed-point data types that can be used explicitly by a programmer writing new code. These data types can be used with built-in operators in C and C++ or with intrinsics as described in Chapter 3.

Vectorization, which can be manual or automatic, analyzes an application program for vector parallelism and restructures it to run efficiently on your ConnX BBE16 configuration. Manual vectorization using ConnX BBE16 data types and intrinsics is discussed in Chapter 3. The Xtensa C and C++ compiler (XCC) contains a feature to perform automatic vectorization on many ConnX BBE16 supported data types. The compiler analyzes and vectorizes the program with little or no user intervention. It generates code for loops by using ConnX BBE16 instructions. 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*.

1.6 Changes in Instruction Set Architecture (ISA) between LX3 (RC Release) and LX4 (RD Release)

Some instructions were added to both the base ConnX BBE16 and to the Despread option between the RC family of releases (Xtensa LX3-based) and the RD family of releases (Xtensa LX4-based). If you have an RC (LX3) configuration of ConnX BBE16, and do a software upgrade in Xtensa Xplorer using an RD upgrade, the ISA of the upgraded configuration will reflect the RC (LX3) ISA and will NOT have the additional instructions.

These changes in ISA include:

- Dual Max Peak Index instructions: BBE_MAXIDX8X20, BBE_MAXABSIDX8X20, BBE_MAXIDX4X40, and BBE_MAXABSIDX4X40 discussed in Section 2.16.
- Instructions added to the Despread option to support Hadamard Transforms: BBE_ADDSUB8X20P, BBE_NEG8X20T detailed in Section Section 2.17.1.
- Special vector compacting stores: BBE_SAC8X16S.U, BBE_SAC8X16U.U, BBE_-SAC4X32S.U, BBE_SAC4X32U.U, and BBE_SAC128.F discussed in Section 6.5.
- Saturating left shift of vector using shift amount from AR register: BBE_SLSR8X20 and BBE_SLSR4X40 as listed in Table 2–16.
- New complex vector magnitude instructions: BBE_MAG8XC18 and BBE_MAG8XC18PACKS as listed in Table 2–14 and Table 2–23.
- Complex conjugate instructions: BBE_CONJ8X20C and BBE_CONJ4X40C as listed in Table 2–21.

- New unpack instruction: BBE_UNPKQ8X20 listed in Table 2–19.

2. ConnX BBE16 Features

The ConnX BBE16 DSP Engine contains a set of sixteen 160-bit general purpose vector registers (vec) that hold the operands and results of its SIMD operations. Each register can hold either eight 20-bit values or four 40-bit values, depending on how the register is used by the software.

ConnX BBE16 includes four 128-bit Alignment registers, and eight 32-bit specialized Select registers, for use with the select function that can manipulate the contents of vector registers. It also includes the following eight special state registers:

- Three 160-bit FFT twiddle value registers, A, B, and C, each hold four complex pairs, each 20-bit real and imaginary values. These registers can also be used for holding contents of the vector registers as temporary storage without going to and from memory.
- A 32-bit control register for the FFT instruction BBE_FFT8X18CPACKQ (FFTCTRL). This is divided into five fields that act as controls or monitors for the FFT instruction BBE_FFT8X18CPACKQ.
- 5-bit unsigned Variable Shift Amount (VSAR)
- 40-bit rounding register (ROUND)
- 32-bit circular buffer support begin address (CBEGIN)
- 32-bit circular buffer support end address (CEND).

ConnX BBE16 supports 128-bit loads and stores between registers and memory.

2.1 ConnX BBE16 Architecture Behavior

The ConnX BBE16 architecture includes guard bits in the data path and register file to avoid overflow on ALU and MAC operations. This means the typical data flow on this machine is:

- Load data at lower precision
- Compute at higher precision, using guard bits.
- Store results at lower precision.

For example, if we load 16-bit data into 20-bit register elements, we can accumulate 16-bit data, using up to two guard bits before sending to 18-bit multipliers. Each 18x18 bit multiply produces 36-bit data, in 40-bit register elements. This allows up to four guard bits for accumulation in 40-bit register elements.

Unlike some other architectures, ConnX BBE16 does not set flags or take exceptions if operations overflow their 20-bit or 40-bit range.

Some ConnX BBE16 operations also saturate results. For instance, Pack operations extract 20-bit data from 40-bit register elements, and saturate results to the range [-2^{19} .. $2^{19}-1$]. Most Store or Move operations that save 16-bit or 32-bit results also saturate.

2.2 ConnX BBE16 ISA Concepts

ConnX BBE16 has several types of instructions. The base instruction set comes from the basic Xtensa LX ISA, consisting of approximately 200 instructions. In addition, BBE16 has a number of instructions derived from the Xtensa Vectra LX DSP coprocessor, renamed to fit in with the BBE16 instruction naming convention. Finally, it has a number of new instructions. The instructions are supported as a set of basic operations and additional variations (intrinsics, which are sometimes called "protos" or prototypes) for different data types. These are discussed in much more detail in Appendix A. There are over 3000 BBE16 operations and intrinsics, although many of these are provided to assist the compiler in automated inference and vectorization as described in Chapter 3.

ConnX BBE16 utilizes two of the three or more possible 64-bit FLIX instruction formats. This allows you to define an additional instruction format, or more than one if you want to encode two different 64-bit formats. Due to reserved bits in the 64-bit encoding there are actually 59 bits available to define your own instructions in TIE.

2.3 Instruction Naming Conventions

ConnX BBE16 uses some standard naming conventions to provide for greater consistency and predictability in determining the names of instructions and operations. The following tables describe the conventions.

Table 2–2. ConnX BBE16 Instruction Name Categories

Category	Mnemonic	Type	Description
FFT	BBE_FFT		FFT type operations
LOAD	BBE_L	V	Load vector
		A	Load and align vector
		E	
		S	Load scalar
		P	Load pair (complex scalar)
		TA	Load twiddle A
		TB	Load twiddle B

Table 2–2. ConnX BBE16 Instruction Name Categories (continued)

Category	Mnemonic	Type	Description
		TC	Load twiddle C
MOVE	BBE_MOV	T	Move on true
		F	Move on false
		AB	Move Boolean to Address reg
		AFM	Move Special FFT reg to AR reg
		ATO	Move AR reg to Special FFT reg
		AV	Move to AR reg
		B	Move to Boolean reg
		LS	Move from a vec reg into another vec reg
		SS	Move from a vec reg into another vec reg
		TA	Move register to twiddle A
		TB	Move register to twiddle B
		TC	Move register to twiddle C
		VA	Copy complex element from AR reg into VR reg
		VI	Move value from imm arg to VR reg
		VTA	Move twiddle A to register
		VTB	Move twiddle B to register
		VTC	Move twiddle C to register
	BBE_MALIGN		Move alignment register
	BBE_MSEL		Move selection control register
	BBE_UNPK	S,Q	Move from one vector to two vectors
MULTIPLY	BBE_MUL		Multiply operation
		A	Multiply/accumulate operation
		E	Multi-mode multiply, extended precision
		C	Multiply complex, high precision
		J	Multiply complex conjugate, high precision
		JCPACKQ	Complex and Complex Conjugate Fractional Multiply. Low precision
		PACKQ	Real fractional multiply, low precision
		PACKS	Real integer multiply, low precision
		R	Multiply Round register

Table 2–2. ConnX BBE16 Instruction Name Categories (continued)

Category	Mnemonic	Type	Description
		S	Multiply/subtract
		SGN	Multiply/Sign
PACK	BBE_PACK	L	Converts from 40-bit vectors to 20-bit vectors
		S	Packs low precision integer vector
		Q	Packs low precision fraction vector
		V	Packs arbitrary fixed point vector to low precision
SELECT	BBE_SEL		Select elements of vector
STORE	BBE_S	V	Store vector
		A	Store and align vector
		S	Store scalar (complex scalar)
		SEL	Store selection control register
		P	Store pair (complex scalar)
		R	Store selected elements
		M	Stored masked vector
	BBE_SAC		Store Compressed Vector Aligned
		F	Store Compressed Vector Aligned Flush
ABS	BBE_ABS		Absolute value
CONJUGATE	BBE_CONJ		Complex Conjugate
DIVISION	BBE_DIV	Q	Fractional divide
		S	Fractional signed divide
		U	Fractional unsigned divide
DESPREAD	BBE_DSPR		Despreading operation
		A	Despread and Add operation
EQUALITY	BBE_EQ		Equality vector check
EXTRACT	BBE_EXTRACT		Extract elements of boolean
HADAMARD	BBE_ADDSUB	P	Add or subtract with predicate control
	BBE_NEG	T	Conditional Negate
INTERLEAVE	BBE_ITLV		Interleave operations
JOIN	BBE_JOIN		Join booleans
MAGNITUDE	BBE_MAG		Magnitude of complex vectors high precision

Table 2–2. ConnX BBE16 Instruction Name Categories (continued)

Category	Mnemonic	Type	Description
		PACKQ	Magnitude of complex fractional vectors, low precision
		PACKS	Magnitude of complex integer vectors, low precision
MAX	BBE_MAX		Max of vector
MAX DUAL PEAK	BBE_MAXIDX		Dual peak max and max index of vector
	BBE_MAXABSIDX		Dual peak absolute max and max index of vector
MIN	BBE_MIN		Min of vector
NAND	BBE_NAND		NAND of 160 bit vector
NSA	BBE_NSA		Normalize shift amount
OR	BBE_OR		OR of 160 bit vector
POLYNOMIAL	BBE_POLY		Polynomial evaluation
RECIPROCAL SQRT	BBE_RSQRT		Reciprocal Square Root
REPLICATE	BBE_REP		Replicate elements
SHIFT	BBE_SLL	(I|R|V)	Shift left
	BBE_SLS		Shift left with saturate
		R	Shift left with saturate with shift value from AR register
	BBE_SRA	(I|R|V)	Shift right
	BBE_ESHFT		Shift vector
	BBE_RNSA	(A|U)	Normalize shift amount
SATURATE	BBE_SAT		Saturate signed vector
ARITHMETIC	BBE_<*NAME*>		Various arithmetic operations, e.g., ADD, AND, RADD, SUB, etc.

Within each class, further conventions are used to describe types of operations, data type layouts, signed/unsigned, data formatting, addressing modes, etc., depending on the type of operation. Table 2–3 is an abbreviated list of this information; for detailed information about the instructions, 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 BBE16 configuration in the System Overview to open the Configuration Summary window.
2. Click **View Details** for the installed build to open a Configuration Overview window.
3. Select **All Instructions** to open a complete list of instruction descriptions.

Table 2–3. Naming Conventions for ConnX BBE16 Load/Store Instructions and Intrinsics

Type	Prefix	Data Type	Pack Type	Addressing
Vector Load	BBE_LV	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Aligning Vector Load	BBE_LA	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Scalar Load	BBE_LS	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Scalar Pair Load	BBE_LP	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU)
Twiddle Load	BBE_LT(A|B|C)	(8X16S|4XC16|8XQ15|4XCQ15)		(I|IU)
Vector Store	BBE_SV	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Masked Store	BBE_SM	(8X16|8X16S|8X16U|4XC16|8XQ15|4XCQ15)		IU
Aligning Vector Store	BBE_SA	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Scalar Store	BBE_SS	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Scalar Pair Store	BBE_SP	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU)
Vector Int Pack Store	BBE_SV	(8X16 | 4XC16)	PACKS	IU
Vector Fract Pack Store	BBE_SV	(8XQ15 | 4XCQ15)	PACKQ	IU
Scalar Int Pack Store	BBE_SS	(8X16 | 4XC16)		IU
Scalar Fract Pack Store	BBE_SS	(8XQ15 | 4XCQ15)	PACKQ	IU
Vector FFT Store	BBE_SR	(4XC16 | 4XCQ15)		[XU|BR|BRU
Aligning, Compacting Vector Store	BBE_SAC	(8X16S | 8X16U | 4XC16 | 8XQ15 | 4XCQ15 | 4X32S | 4X32U | 2XC32 | 4XQ1_30 | 2XCQ1_30)		U
Aligning, Compacting Vector Store Flush	BBE_SAC	(128 | 8X16S | 8X16U | 4XC16 | 8XQ15 | 4XCQ15 | 4X32S | 4X32U | 2XC32 | 4XQ1_30 | 2XCQ1_30)	F	

Following are a few examples to illustrate the naming conventions. Note that where the assembly instruction would use periods, for example, .I, .IU, etc., the "C" prototype replaces the period (".") with an underscore ("_").

- BBE_LV8X16S_I — Vector load of an 8-way vector of 16-bit signed integer elements (signed shorts, in C terms), using an immediate for an address.
- BBE_LTA8XQ15_IU — Vector load of an 8-way vector of Q15 fixed point elements into Twiddle register A. It uses an immediate for an address and the update addressing mode.
- BBE_LV4XC16_IU — Vector load of a 4-way vector of complex 16-bit integer elements (pairs of shorts). Since a complex is an (imaginary, real) pair of 16-bit elements, this is actually an 8x16 vector of elements, but protos for the operations allow convenient use of alternative types in names of instructions. This simplifies programming through the use of appropriate types. The addressing mode uses immediates and updates.
- BBE_LV4XCQ15_IU — Vector load of complex elements, in this case the base data type is Q15.
- BBE_MOVVTB8X20 — Move of an 8x20 vector from Twiddle register B to a vector register. Recall that vector registers (as opposed to vectors in memory) are 160 bits, as are the twiddle registers.
- BBE_MUL8X18PACKS — This is a multiply instruction. ConnX BBE16 vector multiplies are 8-way for real data (16-way, or 4-way complex, for complex data) and use 18 of the 20 bits in the vector register - hence the 8X18 designator. There are a variety of multiply instructions. The PACK variants convert the results, which would otherwise end up being 36-bit results stored in two 4X40 vector registers, back into 16 bit results stored in one 8X20 bit vector register (16 bits plus sign extension) using different schemes. PACKS and PACKL do a right shift to grab the low order bits of the results — thus a presumed integer. PACKQ grabs the high order bits of the result, which are thus treated as fixed point fractions. PACKV uses the VSAR register to determine what shift is being used. PACKL truncates the high order bits of the result and only uses the low order bits, without using the VSAR register.

2.4 Fixed Point Values and Fixed Point Arithmetic

The ConnX BBE16 DSP Engine contains instructions for implementing fixed point arithmetic. This section describes the representation and interpretation of fixed point values as well as some operations on fixed point values.

2.4.1 Representation of Fixed Point Values

A fixed point data type Q*m*.*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 2^n.

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

39 Sign (1 bit)	38 Integer (9 bits) 30	29 Fraction (30 bits) 0
0	0 0000 0001	10 0000 0000 0000 0000 0000 0000 0000
0x0	0x1	0x2000 0000

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

15 Sign (1 bit)	14 Fraction (15 bits) 0
1	100 0000 0000 0000
0x1	0x4000

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

ConnX BBE16 instructions use operations on Q15, Q1.30 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`).
- Q1.30 32-bit fixed point data type with 1 sign bit, 1 integer bit, and 30 bits to the right of the binary point.
- Q9.30 40-bit fixed point data type with 1 sign bit, 9 bits to the left of the binary point and 30 bits of fraction to the right of the binary point.

2.4.2 Arithmetic with Fixed Point Values

When multiplying fixed point numbers $Qm_0.n_0 * Qm_1.n_1$, with a standard signed integer multiplier, the natural result of the multiple will be a $Qm.n$ data type where $n = n_0+n_1$ and $m = m_0+m_1+1$. So multiplying a Q15 by a Q15 generates a Q1.30. Since ConnX BBE16 has 18-bit x 18-bit multipliers, it multiplies two Q2.15 values to produce Q5.30 results in a 40-bit register element.

To convert a Q1.30 to a Q8.31 requires shifting left by 1 bit to generate a Q1.31 and sign extending the result to obtain the Q8.31 that fills a 40-bit register. To convert a Q1.30 to a Q31 also requires shifting left shift by 1 bit. However, some values that fit in a Q1.31 do not fit in a Q31. For multiply operations that generate a Q31 result, signed saturation is used to clamp the value to the maximum or minimum value that the data type can represent. Thus, the Q31 result generated by multiplying the Q15s representing -1.0 * -1.0 = 1.0-2 $^{-31}$.

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.

2.5.1 Scalar Data Types Mapped to the Vector Register File

The real integer values are:

- xb_int16 A 16-bit signed integer stored in the least significant 16 bits of a 20-bit vector register element. The rest of the bits are sign extended from bit 15.
- xb_int20 A 20-bit signed integer stored in a 20-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 values are:

- xb_c16 A signed complex integer value with 16-bit imaginary and 16-bit real parts. The real and imaginary pair are stored in two 20-bit elements of a vector register file, with the real part in the less significant element. The values are sign extended from bit 15.
- xb_c20 A signed complex integer value with 20-bit imaginary and 20-bit real parts. The real and imaginary pair occupy two 20-bit elements of a vector register file, with the real part in the less 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 occupy two 40-bit elements of a vector register file, with the real part in the less significant element.

In addition to the integer data types, ConnX BBE16 also supports a programming model with explicit fixed-point 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 real data type values are:

- xb_q15 A signed Q15 data type stored in the least significant 16 bits of a 20-bit vector register element. The rest of the bits are sign extended from bit 15.
- xb_q4_15 A signed Q4.15 data type that occupies a 20-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_q9_30 A signed Q9.30 data type that uses all 40 bits of a vector register element.

The complex fixed point values are:

- xb_cq15 A signed complex fixed point value with Q15 imaginary and Q15 real parts. The real and imaginary pair are stored in two 20-bit elements of a vector register file, with the real part in the less significant element. The values are sign extended from bit 15.
- xb_cq4_15 A signed complex fixed point value with Q4.15 imaginary and Q4.15 real parts. The real and imaginary pair occupy two 20-bit elements of a vector register file, with the real part in the less 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 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_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 less significant element.

2.5.2 Vector Data Types Mapped to the Vector Register File

Following is a list of the vector register files and their corresponding vector data types.

Table 2–4. Vector Data Types Mapped to Vector Register Files

Register Vector	Vector 20	Vector 40	Double Vector 40
Integer	xb_vec8x20	xb_vec4x40	xb_vec8x40
Fixed-point	xb_vec8xq4_15	xb_vec4xq9_30	xb_vec8xq9_30
Complex int	xb_vec4xc20	xb_vec2xc40	xb_vec4xc40
Complex fixed-point	xb_vec4xcq4_15	xb_vec2xcq9_30	xb_vec4xcq9_30

Memory Vector	Vector 16	Vector 32	Double Vector 32
Integer	xb_vec8x16	xb_vec4x32	xb_vec8x32
Unsigned int	xb_vec8x16U	xb_vec4x32U	
Fixed-point	xb_vec8xq15	xb_vec4xq1_30	xb_vec8xq1_30
Complex int	xb_vec4xc16	xb_vec2xc32	xb_vec4xc32
Complex fixed-point	xb_vec4xcq15	xb_vec2xcq1_30	xb_vec4xcq1_30

2.6 Data Typing

The following tables lay out the complete instruction and intrinsic naming convention for both instructions and protos. The basic data types are used to understand the naming convention. All instructions can be accessed using intrinsics with the same name as the instruction using one of the base data types (xb_vec8x20 and xb_vec4x32). Alternatively, intrinsics are provided to give the same functionality of each instruction mapped appropriately to the set of data types.

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

Table 2–5. Scalar Memory Data Types

Scalar	Scalar 16	Scalar 32
Int	xb_int16	xb_int32
Fixed pt	xb_q15	xb_q1_30
Complex int	xb_c16	xb_c32
Complex fixed pt	xb_cq15	xb_cq1_30

Table 2–6. Scalar Register Data Types

Scalar	Scalar 20	Scalar 40
Int	xb_int20	xb_int40
Fixed pt	xb_q4_15	xb_q9_30
Complex int	xb_c20	xb_c40
Complex fixed pt	xb_cq4_15	xb_cq9_30

Table 2–7. Vector Memory Data Types

Mem Vector	Vector 16	Vector 32	Double Vector 32
Int	xb_vec8x16	xb_vec4x32	xb_vec8x32
Unsigned int	xb_vec8x16U	xb_vec4x32U	
Fixed pt	xb_vec8xq15	xb_vec4xq1_30	xb_vec8xq1_30
Complex int	xb_vec4xc16	xb_vec2xc32	xb_vec4xc32
Complex fixed pt	xb_vec4xcq15	xb_vec2xcq1_30	xb_vec4xcq1_30

Table 2–8. Vector Register Data Types

Temp Vector	Vector 20	Vector 40	Doub Vec 40
Int	xb_vec8x20	xb_vec4x40	xb_vec8x40
Fixed pt	xb_vec8xq4_15	xb_vec4xq9_30	xb_vec8xq9_30
Complex int	xb_vec4xc20	xb_vec2xc40	xb_vec4xc40
Complex fixed pt	xb_vec4xcq4_15	xb_vec2xcq9_30	xb_vec4xcq9_30

2.7 Vector Groups

Table 2–9. Major Vector Load Group

Type	Prefix	Data Type	Addressing	Load Type
Inst	BBE_LV	(8X16) (S|U)	(I|IU|X|XU|CU)	Vector
Proto	BBE_LV	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)	(I|IU|X|XU|CU)	Vector
Inst	BBE_LV	(4X32) (S|U)	(I|IU|X|XU|CU)	Vector
Proto	BBE_LV	(4X32S|4X32U|2XC32|4XQ1_30|2XCQ1_30)	(I|IU|X|XU|CU)	Vector
Inst	BBE_LA	(8X16) (S|U)	(I|IU|X|XU|CU)	Align
Proto	BBE_LA	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)	(I|IU|X|XU|CU|P|CP)	Align
Inst:	BBE_LA	(4X32) (S|U)	(I|IU|X|XU|CU)	Align
Proto	BBE_LA	(4X32S|4X32U|2XC32|4XQ1_30|2XCQ1_30)	(I|IU|X|XU|CU|P|CP)	Align
Inst:	BBE_LA	128	(P|CP)	Align Prime
Inst:	BBE_LS	(8X16) (S|U)	(I|IU|X|XU)	Scalar
Proto	BBE_LS	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)	(I|IU|X|XU|CU)	Scalar
Inst	BBE_LS	(4X32) (S|U)	(I|IU|X|XU)	Scalar
Proto	BBE_LS	(4X32S|4X32U|2XC32|4XQ1_30|2XCQ1_30)	(I|IU|X|XU|CU)	Scalar
Inst	BBE_LP	(8X16) (S|U)	(I|IU|X|XU)	Scalar Pair
Proto	BBE_LP	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)	(I|IU|X|XU)	Scalar Pair
Inst	BBE_LP	(4X32) (S|U)	(I|IU|X|XU)	Scalar Pair
Proto	BBE_LP	(4X32S|4X32U|2XC32|4XQ1_30|2XCQ1_30)	(I|IU|X|XU)	Scalar Pair
Inst	BBE_LT	(A|B|C) 8X16 S	(I|IU)	Twiddle
Proto	BBE_LT	(A|B|C) (8X16S|4XC16|8XQ15|4XCQ15)	(I|IU)	Twiddle

Table 2–10. Major Vector Store Group

Type	Prefix	Data Type	Addressing	Store Type
Inst	BBE_SV	(8X16) (S|U)	(I|IU|X|XU|CU)	Vector
Proto	BBE_SV	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)	(I|IU|X|XU|CU)	Vector
Inst	BBE_SV	(4X32) (S|U)	(I|IU|X|XU|CU)	Vector
Proto	BBE_SV	(4X32S|4X32U|2XC32|4XQ1_30|2XCQ1_30)	(I|IU|X|XU|CU)	Vector
Inst	BBE_SA	(8X16) (S|U)	(I|IU|X|XU|CU)	Align
Proto	BBE_SA	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)	(I|IU|X|XU|CU|P|CP)	Align
Inst:	BBE_SA	(4X32) (S|U)	(I|IU|X|XU|CU)	Align
Proto	BBE_SA	(4X32S|4X32U|2XC32|4XQ1_30|2XCQ1_30)	(I|IU|X|XU|CU|P|CP)	Align
Inst:	BBE_SA	128	F	Align Flush
Inst	BBE_SS	(8X16) (S|U)	(I|IU|X|XU)	Scalar
Proto	BBE_SS	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)	(I|IU|X|XU|CU)	Scalar
Inst	BBE_SS	(4X32) (S|U)	(I|IU|X|XU)	Scalar
Proto	BBE_SS	(4X32S|4X32U|2XC32|4XQ1_30|2XCQ1_30)	(I|IU|X|XU|CU)	Scalar
Inst	BBE_SP	(8X16) (S|U)	(I|IU|X|XU)	Scalar Pair
Proto	BBE_SP	(8X16S|8X16U||8XQ15|4XCQ15)	(I|IU|X|XU)	Scalar Pair
Inst	BBE_SP	(4X32) (S|U)	(I|IU|X|XU)	Scalar Pair
Proto	BBE_SP	(4X32S|4X32U|4XQ1_30|2XCQ1_30)	(I|IU|X|XU)	Scalar Pair
Inst	BBE_SR	8X16 S	(XU|BR|BRU)	Bit Reversed (BBE_S2VS16.*)
Proto	BBE_SR	(4XC16|4XCQ15)	(XU|BR|BRU)	Bit Reversed
Inst	BBE_SM	8X16 (S|U)	IU	Masked
Proto	BBE_SM	(8X16|8X16S|8X16U|4XC16|8XQ15|4XCQ15)	IU	Masked
Inst	BBE_SAC	(8X16S | 8X16U | 4X32S | 4X32U)	U	Aligning, compacted vector

Table 2–11. Vector Pack and Store and Pack Group

Type	Prefix	Data Type	Addressing	Pack/Store Type
Inst	BBE_SV	8X16 PACK (S|Q)	IU	Pack (BBE_SVPACK*)
Proto	BBE_SV	(8X16|4XC16)	IU	PACKS In: 8X40, 4XC40
Proto	BBE_SV	(8XQ15|4XCQ15)	IU	PACKQ In: 8XQ9_30, 4XCQ9_30
Inst	BBE_SS	8X16 PACK (S|Q)	(I|IU|X|XU)	Pack (BBE_SSPACK*)
Proto	BBE_SS	(8X16|4XC16)	IU	PACKS
Proto	BBE_SS	(8XQ15|4XCQ15)	IU	PACKQ Additional to use on smaller sizes
Proto	BBE_SS	(4X16|2XC16)	IU	PACKS
Proto	BBE_SS	(4XQ15)	IU	PACKQ
Inst	BBE_PACK	(V|S|Q|L) (8X40|4XC40V)		
Proto	BBE_PACKV	(8X40|8X40V|8XQ9_30|4XCQ9_30)		
Proto	BBE_PACKS	(8X40|4XC40)		
Proto	BBE_PACKQ	(8XQ9_30|4XCQ9_30)		
Proto	BBE_PACKL	(8X40)		
Proto	BBE_PACK	(8X40L|4XC40L)		
Inst	BBE_PACK	(8X40) .01 (Vectra compatibility)		
Proto	BBE_PACK	(8X40) .01		

2.7.1 Load and Store Groups

Table 2–12. Load/Store Special and Support Group

Type	Prefix	Data Type	Addressing	Load/Store Type
Inst	BBE_LV	(2X40H|4X20L)	I	Loads the high/low part of a reg
Inst	BBE_SV	(2X40H|4X20L)	I	Stores the high/low part of a reg
Inst	BBE_SVL	8X16	I	Stores vector low element bits
Inst	BBE_LSEL	8X4	(I|IU)	Loads selection control reg
Inst	BBE_SSEL	8X4	I	Stores selection control reg

Table 2–12. Load/Store Special and Support Group (continued)

Type	Prefix	Data Type	Addressing	Load/Store Type
Inst	BBE_LALIGN	128	I	Loads alignment reg
Inst	BBE_SALIGN	128	I	Stores alignment reg
Inst	BBE_ZALIGN	128		Clears alignment reg contents

2.7.2 Multiply Groups

Table 2–13. Multiply to 40-bit Result Group

Type	Prefix	Data Type	Attributes*
Inst	BBE_MUL	SGN4x40	
Inst	BBE_MUL	(|A) 8X18	(|C|J|E)
Inst	BBE_MUL	(R|S) 18	
Proto	BBE_MUL	(|A) (8X20|8XQ4_15)	(|E)
Proto	BBE_MUL	(|A) (4XC20|4XCQ4_15)	(|J|E)
Inst	BBE_MUL	(A) 18 (Vectra compatibility	(0|1|I|R)

* Notes:
- (blank) is High precision Real (8 multiplies)
- C is High precision Complex (16 multiplies)
- J is High precision Complex with a conjugate input (16 multiplies)
- E is High precision with Chosen Element inputs (16 multiplies)

Table 2–14. Multiply to 20-bit Result Group

Type	Prefix	Data Type	Attributes	Multiply Type*
Inst	BBE_MUL	SGN8x20		
Inst	BBE_MUL	(|A) 8X18	(|C|J|E)	PACKQ
Proto	BBE_MUL	(|A) 8XQ4_15)		PACKQ
Proto	BBE_MUL	(|A) 4XCQ4_15	(|J)	PACKQ
Inst	BBE_MUL	8X18	JC	PACKQ
Inst	BBE_MUL	8X18		PACKS
Inst	BBE_MAG	8XC18		
Inst	BBE_MAG	8XC18		PACKQ
Inst	BBE_MAG	8XC18		PACKS
Proto	BBE_MAG	2X4XC20		
Proto	BBE_MAG	2X4XCQ4_15		
Proto	BBE_MAG	2X4XC20		PACKS
Proto	BBE_MAG	2X4XCQ4_15		PACKQ
Proto	BBE_MUL	4XCQ4_15	JC	PACKQ
Proto	BBE_MUL	8X20		PACKS

* Multiply Type Details:

- PACKQ is Low precision Real Frac (8 multiplies)
- CPACKQ is Low precision Complex Frac (16 multiplies)
- JPACKQ is Low precision Complex Frac one Conjugate Input (16 multiplies)
- JCPACKQ is Low precision Complex true/conj Output (16 multiplies)
- PACKS is Low precision Real Integ (8 multiplies)
- MAG*PACKQ is Low precision Complex Frac Magnitude Output (16 multiplies)

MAG*PACKS is Low precision Complex Integer Magnitude Output (16 multiplies)

MAG is High precision Complex Magnitude Output (16 multiplies)

Table 2–15. Special Multiply Group

Type	Prefix	Data Type	Addressing	Multiply Type
Inst	BBE_FFT	8X18		CPACKQ
Inst	BBE_POLY	8X20		STEP

Table 2–15. Special Multiply Group (continued)

Type	Prefix	Data Type	Addressing	Multiply Type
Inst	BBE_POLY	8X20		SU
Proto	BBE_POLY	8XQ4_15		STEP
Proto	BBE_POLY	8XQ4_15		SU

Table 2–16. Shift Group

Type	Prefix	Data Type	Shift Amount Source *
Inst	BBE_SLL	(8X20|4X40)	(|I|R|V)
Inst	BBE_SLS	(8X20|4X40)	(R)
Inst	BBE_SRA	(8X20|4X40)	(|I|R|V)
Inst	BBE_LFSR. STEP		
Inst	BBE_RNSA		(A|UAV)
Proto	BBE_SLL	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)	(|I|R|V)
Proto	BBE_SLS	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)	(R)
Proto	BBE_SRA	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)	(|I|R|V)

*I = immediate. R= amount in variable, V = vector

Table 2–17. Vector Arithmetic and Logical Group

Type	Prefix	Data Type
Inst	BBE_ (ABS|ADD|NEG|SUB)	(8X20|4X40)
Inst	BBE_ MULSGN	(8X20|4X40)
Inst	BBE_ (AND|NAND|OR|XOR)	160
Inst	BBE_ (MAX|MIN)	(8X20|4X40)
Inst	BBE_ (NSAUV|NSAV)	(8X20|4X40)
Inst	BBE_ (NSAUV|NSAV)	4X40

*NOT is NAND self, ZERO is XOR self

Table 2–17. Vector Arithmetic and Logical Group (continued)

Type	Prefix	Data Type
Inst	BBE_SAT	(4X32S|8X16S)
Proto	BBE_(ABS|ADD|NEG|SUB)	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)
Proto	BBE_MULSGN	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)
Proto	BBE_(AND|NAND|OR|XOR)	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30) |160)
Proto	BBE_(NOT|ZERO)	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30) |160)*
Proto	BBE_(MAX|MIN)	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)
Proto	BBE_(NSA|NSAU)	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)
Proto	BBE_(NSAUV|NSAV)	(4X40|8X40|2XC40|4XC40|4XQ9_30|8XQ9_30|2XCQ9_30|4XCQ9_30)

*NOT is NAND self, ZERO is XOR self

2.7.3 Other Groups

Table 2–18. Reduction Instructions Group

Type	Prefix	Data Type	Attributes
Inst	BBE_RADD	(8X20|4X40)	(|C)
Proto	BBE_RADD	(8X20|4X40|8X40|8XQ4_15|4XQ9_30|8XQ9_30)	
Proto	BBE_RADD	(4XC20|2XC40|4XC40|4XCQ4_15|2XCQ9_30|4XCQ9_30)	

Table 2–19. Moves Group

Type	Prefix	Data Type	Attributes	Move Type
Inst	BBE_MOV160			
Inst	BBE_MOVTV	8X20	(A|B|C|BC)	Move Vector register to Twiddle size
Inst	BBE_MOVVT	8X20	(A|B|C)	Move Twiddle register to Vector register
Inst	BBE_MOV	(8X20|4X40|2X80)	(T|F)	Conditional Moves

Table 2–19. Moves Group

Type	Prefix	Data Type	Attributes	Move Type
Proto	BBE_MOV	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)		
Proto	BBE_MOVTV	(8X20|4XC20|8XQ4_15|4XCQ4_15)	(A|B|C|BC)	Move Vector register to Twiddle size
Proto	BBE_MOVVT	(8X20|4XC20|8XQ4_15|4XCQ4_15)	(A|B|C|BC)	Move Twiddle register to Vector register
Proto	BBE_MOV	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)	(T|F)	Conditional Moves
Inst	BBE_SEL	(8X20|4V8X20)	(|I)	Select elements of vector
Inst	BBE_ (ESHFT|REP)	(8X20|4X40)		Shift Vector/ Replicate elements
Proto	BBE_SEL	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)	(|I)	
Proto	BBE_(ESHFT|REP)	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)		
Inst	BBE_MOVAV	(16|32|16C|8X4)		Move vector reg into AR reg
Inst	BBE_MOVVA	(20|32|16C|32C)		Move AR reg into vector reg
Inst	BBE_MOVAB	(4|8)		Move Boolean to AR (reversed)
Inst	BBE_MOVBA	(4|8)		Move AR to Boolean
Inst	BBE_MOVVI	8X20		Move imm arg to vec reg
Proto	MOVAV	(8X20|4X40|8X40|4XC20|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15)		
Proto	MOVVA	(8X20|4X40|8X40|4XC20|2XC40|4XC40|8XQ4_15|4XQ9_30|8XQ9_30|4XCQ4_15|2XCQ9_30|4XCQ9_30)		

Table 2–19. Moves Group

Type	Prefix	Data Type	Attributes	Move Type
Inst	BBE_MALIGN	128		Move alignment reg
Inst	BBE_MSEL	8X4		Move selection control reg
Inst	BBE_UNPK	8X20	S,Q	Move vec from one reg into two 4X40-bit vec with either integer style sign extension or fractional type extension

Table 2–20. Boolean Set Instructions Group

Type	Prefix	Data Type	Boolean Type
Inst	BBE_(MAXB|MINB)	(8X20|4X40)	Calculates max/min value for each element of vec reg. Results written to vec reg.
Inst	BBE_(EQ|LE|LT)	(8X20|4X40)	Compares vec reg vs and vr. Results written to Boolean reg.
Inst	BBE_EQ	2X80	Two parallel 80-bit equality comparison
Inst	BBE_EXTRACT	(B2B4|B28B)	Extracts low/high part of Boolean
Proto	BBE_(MAXB|MINB)	(8X20|4X40|8X40|8XQ4_15|4XQ9_30|8XQ9_30)	
Proto	BBE_(EQ|LE|LT)	(8X20|4X40|8X40|8XQ4_15|4XQ9_30|8XQ9_30)	
Proto	BBE_EQ	(4XC20|2XC40|4XC40|4XCQ4_15|2XCQ9_30|4XCQ9_30)	
Inst	BBE_JOIN	B8B4|B4B2	
Inst	BBE_SEL4V	8X20	
Inst	BBE_NAND	160	Calculates vs and vs NAND, writes to vt

Table 2–21. Complex Conjugate Group

Type	Prefix	Data Type	Attributes
Inst	BBE_CONJ	(8X20 | 4X40)	C
Proto	BBE_CONJ	(8X20 | 4X40)	C
Proto	BBE_CONJ	(4XC20|2XC40|4XC40|4XCQ4_15|2XCQ9_30|4XCQ9_30)	

Table 2–22. Dual Peak Max Index Group

Type	Prefix	Data Type	Attributes /Meaning
Inst	BBE_MAXIDX	(8X20 | 4X40)	Dual peak max index
Proto	BBE_MAXIDX	(8X20 | 8XQ4_15 | 4X40 | 4XQ9_30)	(V, VAL, VAL2, IDX, IDX2, OFFSET, INIT)
Inst	BBE_MAXABSIDX	(8X20 | 4X40)	Dual peak max absolute index
Proto	BBE_MAXABSIDX	(8X20 | 8XQ4_15 | 4X40 | 4XQ9_30)	(V, VAL, VAL2, IDX, IDX2, SGNSINFLG, OFFSET, INIT)

2.8 ConnX BBE16 Multiplies: Real and Complex Arithmetic

The ConnX BBE16 DSP Engine can perform multiply, multiply/accumulate, multiply/subtract and multiply/pack operations on real and complex data. These use a set of 16 18x18 bit SIMD multipliers and associated adders provided as resources for ConnX BBE16 computation.

The basic element operation is an 18x18 multiplier, which results in a 36-bit precision result, expanded to a 40-bit precision element. Because the source elements are 20-bits precision, the top two bits are ignored in the multiplication. Also, because the operation takes two 8x20 vectors, the complete result is an 8x40 result stored in two registers. Some operations, for example, BBE_MULA8X18JPACKQ, BBE_MULA8X18PACKQ, and BBE_MULA8X18PACKS retain just 20 bits of the results creating 8x20 vectors.

The data layout for the main multiply instructions is a vector of real, or a vector of complex elements, where the complex (imaginary, real) parts are interleaved. That is, if we consider vectors of eight 20-bit elements, used as inputs into the multiply instructions, and as outputs from these instructions, then the vector contains real elements [7:0] from

nominal left to right. If the vector contains complex elements, then these are ordered, nominally left to right, as (imaginary[3], real[3], imaginary[2], real[2], imaginary[1], real[1], imaginary[0], real[0]).

There are actually several types of multiply instructions, depending on input data type and precision.

- The high-precision multiplies produce 36-bit products, and return the results in two 4x40-bit vectors.
- Low precision multiply instructions retain only 20-bits of the products. There are two types of low precision multiply operations, for fixed-point data (fractional) or for integer data.
- Low precision multiplies of two fractional inputs retain the most significant bits of the product.
- Low precision multiplies of integer data retain the least significant 20 bits of the results.

Almost all multiply operations exist in multiply and multiply-accumulate variations. The basic types of multiply include real, complex, and complex conjugate. Basic data types include both integer (short 16-bit in 20 bit vectors) and fixed point fractional (Q1_15 stored as Q4_15 in vectors). The multipliers are actually 18 x 18 bits, which allows use of some of the guard bits in vector registers. Figure 2 illustrates the basic multiplier details.

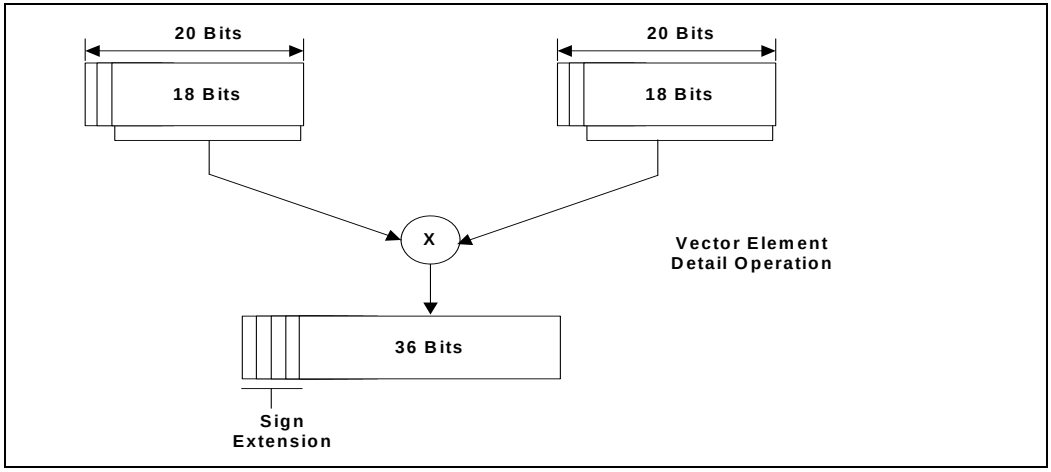


Figure 2–2. Basic Multiplier Details

There are some multiply instructions that operate in a somewhat different way because they are based on the ConnX Vectra LX instruction set. Because these are less efficient than the ConnX BBE16 native multiplies, use of them is discouraged. They are supported for Vectra LX compatibility; the use of these instructions are described in Appendix B.

The following table summarizes the supported multiply instructions:

Table 2–23. Multiply Instructions

Operation	Definition
Real Vector Multiply and Multiply-Add	
BBE_MUL8X18PACKQ	Real Multiply: 20-bit fractional results
BBE_MULA8X18PACKQ	Real Multiply-Add: 20-bit fractional results
BBE_MUL8X18PACKS	Real Multiply: 20-bit integer results
BBE_MULA8X18PACKS	Real Multiply-Add: 20-bit integer results
BBE_MUL8X18	Real Multiply: 40-bit results
BBE_MULA8X18	Real Multiply-Add: 40-bit results
Complex Vector Multiply and Multiply-Add	
BBE_MUL8X18CPACKQ	Complex Multiply: 20-bit fractional results
BBE_MULA8X18CPACKQ	Complex Multiply-Add: 20-bit fractional results
BBE_MUL8X18C	Complex Multiply: 40-bit results
BBE_MULA8X18C	Complex Multiply-Add: 40-bit results
Complex Conjugate Vector Multiply and Multiply-Add, and Magnitude	
BBE_MUL8X18JPACKQ	Complex Conjugate Multiply: 20-bit fractional results
BBE_MULA8X18JPACKQ	Complex Conjugate Multiply-Add: 20-bit fractional results
BBE_MUL8X18JCPACKQ	Complex and Complex Conjugate Multiply: 20-bit fractional results
BBE_MUL8X18J	Complex Conjugate Multiply: 40-bit results
BBE_MULA8X18J	Complex Conjugate Multiply-Add: 40-bit results
BBE_MAG8XC18PACKQ	Sum of square of real and imaginary parts
Special Multi-Mode Multiplies	
BBE_MUL8X18E	General multi-way, multi-type, multi-mode Multiply instruction by selected element with Extended Precision
BBE_MULA8X18E	General multi-way, multi-type, multi-mode Multiply-Add instruction by selected element with Extended Precision

Figure 2–3 shows details of a low precision real integer multiply. The inputs are two 8X20 bit vectors and the output is one 8X20 bit vector. The operation does an 8-way SIMD multiply (18-bit x 18-bit) and computes eight 36-bit vector products. This operation extracts 20 bits of each element, with a result of xb_vec8X20 in VT.

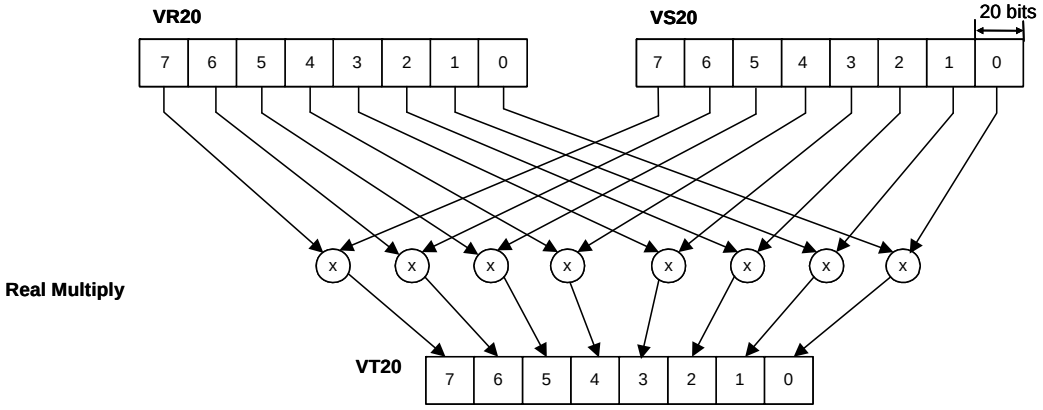


Figure 2–3. Low-Precision Real Multiply Example

Figure 2–4 shows the details of a vector high-precision real multiply. The inputs are two 8X20-bit vectors. This operation does an 8-way SIMD multiply (18-bit x 18-bit), computes eight 36-bit vector products and sign-extends the results into 40-bit fields. The result of this operation is two vectors, xb_vec4X40.

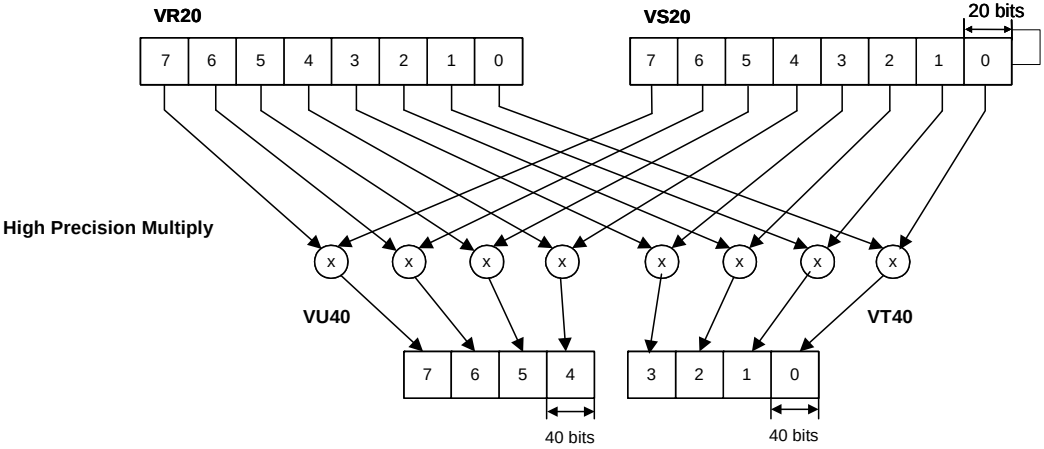


Figure 2–4. High-Precision Real Multiply Example

Complex vectors have interleaved real and imaginary elements. A ConnX BBE16 register vec8x20 will accommodate four complex elements.

The "high precision" operations, such as BBE_MUL8X18C, use a pair of registers, from the vec 16x160-bit register file, to hold a high precision result. These are split into the low order results (1 and 0) and the high order results (3 and 2).

Figure 2–5 shows a high-precision complex integer multiply. The inputs to the operation are two complex pairs of two 8x20 vectors. The operation extracts 18 bits from each element, computes four 36-bit products, adds the appropriate parts to form two 37-bit sums, and sign-extends the results into two 40-bit fields.

Two "extended precision" multi-mode, multi-way, multi-data type, multiply instructions, BBE_MUL8X18E and BBE_MULA8X18E, are discussed in Chapter 5.

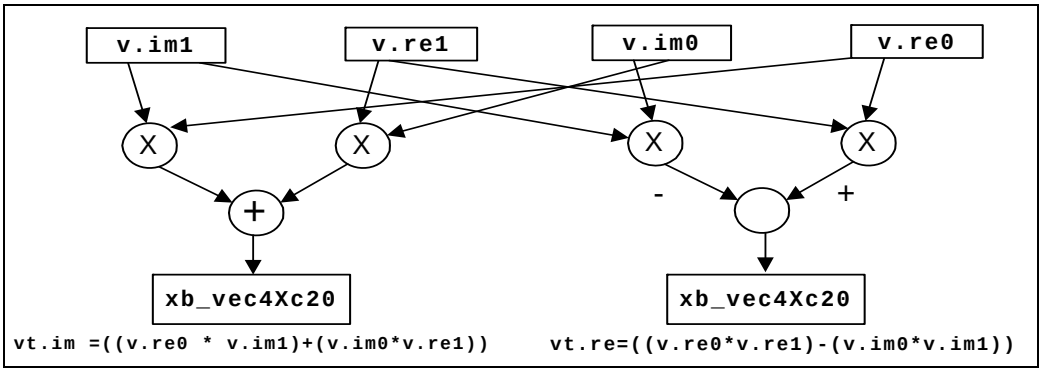


Figure 2–5. High-Precision Complex Multiply Example

There are some additional multiplies in ConnX BBE16 that are derived from the ConnX Vectra LX DSP Engine. In general, the use of these are discouraged in preference to the more efficient BBE16 multiply and multiply-accumulate instructions. The Vectra LX versions are discussed in Appendix B.

2.9 Select Instructions

The select instructions (BBE_SEL8X20) allows elements from two vectors to be copied to a third vector, and at the same time the elements for the target vector can be individually selected from the source vectors. With this general definition of element transferring, it is easy to implement replication, rotation, shift, and interleaving with the same basic instruction.

To define the transfer matrix, BBE_SEL8X20 takes two vec8x20 source vectors and one vec8x20 target vector, as well as a `sel` register. The sel register contains your defined pattern for the transfer.

BBE_SEL8X20I allows the most common selections without having to set a selection register as described in Table 2–24. The values in the `sel` Matrix column represent the values necessary for the `sel` register to perform the equivalent operation.

Table 2–24. BBE_8X20SELI Type Definitions

SELI Type	sel Matrix	Description
0	0x87654321	Shifts right by one element
1	0x98765432	Shifts right by two elements
2	0xedcba987	Shifts left by one element
3	0xdcba9876	Shifts left by two elements
4	0x6f4d2b09	Even/Odd elements interleaved
5	0xba987654	Selects middle eight elements
6	0xba983210	Lower four elements of each vector
7	0xfedc7654	Upper four elements of each vector

2.10 Vector Initialization and Some Additional Select Patterns

ConnX BBE16 also contains a special move instruction, BBE_MOVVI8X20, which is used for vector initialization based on an immediate value. This can also be used in a two instruction combination to create additional select patterns for the BBE_SEL8X20 instruction. The following tables illustrate what is possible.

Table 2–25. BBE_MOVVI8X20 Instruction Arguments

Immediate Argument	Format	Value
-1	Integer	-1
0	Zero	0
1	Integer	1
2	Q4.15	+1.0 or int16 MININT
3	Q4.15	-1.0
4	int20	MININT20
5	int20	MAXINT20
6	int40	MININT40
7	int40	MAXINT40
8	C20	+1 or int40 1
9	C20	(0)+i
10	C20	-1 or even select
11	C20	0-i or odd select
12	CQ4.15	+1
13	CQ4.15	0+i
14	CQ4.15	-1

Table 2–25. BBE_MOVVI8X20 Instruction Arguments

Immediate Argument	Format	Value
15	CQ4.15	0-i
16	Lower Char	
17	Not Lower Char	
18	Upper Char	
19	Not Upper Char	
20	Q9.30	1
21	Q9.30	-1
22	int16	MAXINT16
23	int32	MININT32
24	int32	MAXINT32
48	Special Bit pattern	(see * below)
49	Special Bit pattern	(see * below)
50	Special Bit pattern	(see * below)
56	8x20 sequence	7,6,5,4,3,2,1,0
57	4x40 sequence	3,2,1,0

*****Immediate arguments of 48, 49, and 50 will produce a bit pattern that can be transformed to the following values using a BBE_SEL4V8X20 instruction and the appropriate value for its immediate as shown in Table 2–26.

Table 2–26. BBE_SEL4V8X20 Instruction Arguments

Immediate Argument for BBE_MOVVI8X20	Immediate Argument for BBE_SEL4V8X20	Resulting Select Register Pattern	Selection Effect
48	0	0xf7e6d5c4	Interleave x20 Upper
48	4	0xb3a29180	Interleave x20 Lower
48	8	0xeca86420	Extract Even x20
48	12	0xfdb97531	Extract Odd x20
49	0	0xfe76dc54	Interleave x40 Upper
49	4	0xba329810	Interleave x40 Lower
49	8	0xdc985410	Extract Even x40
49	12	0xfeba7632	Extract Odd x40
50	0	0xf6d4b290	Real/Imag x20 Merge

Table 2–26. BBE_SEL4V8X20 Instruction Arguments

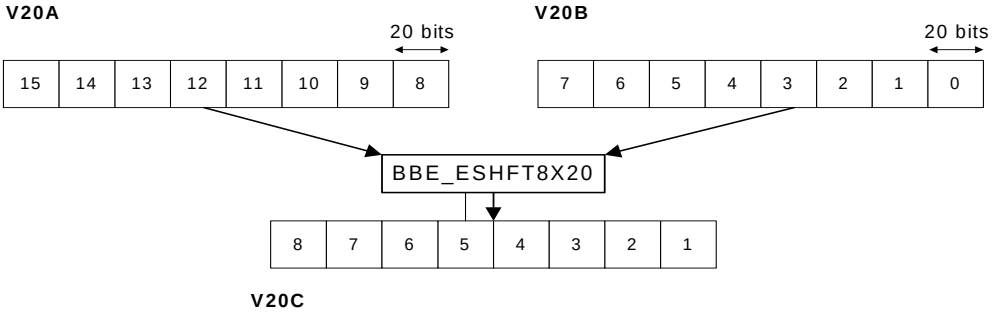
Immediate Argument for BBE_MOVVI8X20	Immediate Argument for BBE_SEL4V8X20	Resulting Select Register Pattern	Selection Effect
50	4	0xfe54ba10	Real/Imag x40 Merge
50	8	0xe6c4a280	Interleave Real x20
50	12	0xf7d5b391	Interleave Imag x20

2.11 BBE_REP8X20 and BBE_ESHFT8X20 Special Instructions

To increase the performance of some algorithms, ConnX BBE16 uses special purpose operations, such as BBE_REP8X20 and BBE_ESHFT8X20.

BBE_REP8X20 replicates a scalar that already exists in a vector register into the other elements of the vector. The developer specifies which of the eight elements to replicate. BBE_REP8X20 is a special case of BBE_SEL8X20, except BBE_REP8X20 is an operation for slot 0, while BBE_SEL is a slot 1 operation.

BBE_ESHFT8X20 performs a right element rotation of a vector register pair. Assuming a little-endian numbering, the shift is a right shift. This causes elements 1 to 7 to move to Elements 0 to 6 and Element 8 moves to Element 7. BBE_ESHFT8X20 is a special case of BBE_SEL8X20. Because BBE_ESHFT8X20 is modeled after BBE_SEL8X20, the source vectors are not modified.




```
ESHFT20  V20C, V20A, V20B
```

Figure 2–6. BBE_ESHFT8X20 Flow Example

2.12 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 instructions BBE_NSA8X20, BBE_NSA4X40, BBE_NSAU8X20, and BBE_NSAU4X40 facilitate block floating point. These instructions calculate the left shift amount required to normalize each element to a 20-bit or 40-bit value. The result is returned in another register, and can be used with the instruction BBE_SLLV8X20 and BBE_SLLV4X40 to normalize the elements of a single vector. We illustrate the use on the 8X20 variation. For each vector element, BBE_NSA8X20 returns 19 if the input is 0 or -1, or returns the number of sign bits minus 1. Using BBE_SLLV8X20 to shift the vector left by the BBE_NSA8X20 result produces eight vector elements in which bit 19 and bit 18 differ, or the result is 0.

For each vector element, BBE_NSAU8X20 returns 20 if the input is 0, or returns the number of leading zeros. Using BBE_SLLV8X20 to shift the vector left by the BBE_NSAU8X20 result produces eight vector elements in which bit 19 is set, unless the input is 0.

The corresponding instructions for xb_vec4x40 data are BBE_NSA4X40 and BBE_NSAU4X40.

To implement block floating point, BBE_NSA8X20 or BBE_NSAU8X20 is applied to the entire data set, and the minimum normalization shift is calculated using BBE_MIN8X20. Then the same shift is applied to the entire data set, typically using BBE_SLL8X20. This shift amount can be used at a later stage of the computation, with BBE_SRA8X20 to return the data to non-normalized form.

BBE_RNSAAV4X40, an 8-way minimum Normalize Shift Amount, calculates for two input 4-way 40-bit vectors, the minimum number of shifts required to normalize to a 40-bit size. It operates across all eight elements of the vectors. It returns the minimum number of sign bits minus 1. If all elements are zero or minus 1 the instruction returns 39.

BBE_RNSAUAV4X40, an 8-way minimum unsigned Normalize Shift Amount, calculates for two 4-way unsigned 40-bit vectors, the minimum number of shifts required to normalize to a 40-bit size. It operates across all eight elements of the vectors. It returns the minimum number of leading zeros in register at. If all elements are zero, the instruction returns 40.

2.13 Butterfly Instructions

The use of the butterfly instructions in FFT acceleration is described as part of the examples in Chapter 4.

2.14 4-Way Complex Conjugate Instructions

A complex conjugate multiply of two complex numbers a and b is defined as:

r = (a.real + j a.imag) * (b.real - j b.imag)

There are five complex conjugate multiply or multiply-add instructions:

- BBE_MUL8X18JPACKQ
- BBE_MULA8X18JPACKQ
- BBE_MUL8X18J
- BBE_MULA8X18J
- BBE_MUL8X18JCPACKQ.

BBE_MUL8X18JPACKQ and BBE_MULA8X18JPACKQ are low-precision fractional (Q15) complex conjugate multiply and multiply-add, respectively. BBE_MUL8X18J and BBE_MULA8X18J are high precision complex conjugate multiply and multiply-add respectively.

For example, BBE_MUL8X18J takes as input two 8x20 bit vectors, in which the real and imaginary portions of four complex numbers are interleaved. It produces four results in two 4X40 registers where the high order (3,2) results are in one register and the low order (1,0) results are in the other register, both as interleaved (imaginary, real) pairs.

BBE_MUL8X18JCPACKQ is a composite low precision fractional (Q15) operation that does both complex and complex conjugate multiply.

2.15 Vector Polynomial Evaluation Instructions: BBE_POLY8X20_SU and BBE_POLY8X20_STEP

ConnX BBE16 has two instructions that are designed to work together to allow quick SIMD evaluation of iterative polynomial functions of the form $y_i = x_i * y_{i-1} + c_i$, where x_i is a coefficient for the i'th step, and c_i is a constant for the i'th step. Thus, these instructions can be used to evaluate series functions, and thus evaluate transcendental functions (sines, cosines and the such) that can be expressed in the form of series.

The two instructions are a setup instruction, BBE_POLY8X20_SU, and a step-wise evaluation instruction, BBE_POLY8X20_STEP. In this section, these instructions are first described individually, and then there is a discussion of their usage for a real example (vector sine), including an example piece of code. The process is intended for the Q15 (or Q4.15 in registers) fractional data types. There is also a discussion about how to derive coefficients for the sine example.

2.15.1 BBE_POLY8X20_SU: Setup Polynomial Evaluation

BBE_POLY8X20_SU sets up a vector polynomial evaluation process for use by the BBE_POLY8X20_STEP instruction. The basic prototype for this instruction in C is as follows:

```
extern xb_vec8x20 BBE_POLY8X20_SU(xb_vec8x20 vr, immediate poly_sl,
immediate poly_sa);
```

It reads an input vector of type xb_vec8xq4_15, containing eight Q4.15 values, from vr, and transforms the input data into a xb_vec8xq4_15 output vector in vt.

Each 20-bit element of the output vector encodes two values:

- Output bits [19:16] contain an index into a segmented polynomial and
- Output bits [15:0] contain an offset within the segment for each argument.

The 4-bit index is used to divide the range of possible input data values into 16 equal segments. Each valid input data value falls into one of those 16 segments. The 16-bit offset represents the position of that input value within that segment, as a signed offset from the mid-point of that segment.

BBE_POLY8X20_SU has some flexibility in how this is done. One immediate to the instruction says which bits of the operand become the index in bits[19:16]. The index can come from bits[18:15], bits[17:14], bits[16:13], or bits[15:12] of the input data. For an operand which is a Q15 number sign extended into the register, bits[15:12] are the right choice. The other choices exist to deal with cases where more accuracy is needed and the operand is the result of other computations.

The second immediate to the instruction gives an amount to shift input data left to produce the offset. Thus bits[15:0] are the offset within the segment multiplied by 1, 2, 4, or 8. This multiplication is done to allow coefficients larger than one to be used more conveniently.

2.15.2 BBE_POLY8X20_STEP: Evaluation of one Step of the Polynomial Evaluation Process

After the call to BBE_POLY8X20_SU, the second instruction, BBE_POLY8X20_STEP, can be called several times to evaluate the polynomial until sufficiently precise results are obtained. The default prototype (intrinsic) for the instruction is:

```
extern void BBE_POLY8X20_STEP(xb_vec8x20 vw /*inout*/, xb_vec8x20 vr,
xb_vec8x20 vx, xb_vec8x20 vs);
```

An alternative prototype expressed in terms of the intended data types is:

```
proto BBE_POLY8XQ4_15_STEP {inout xb_vec8xq4_15 Poly, in xb_vec8xq4_15
lu_h, in xb_vec8xq4_15 lu_l, in xb_vec8xq4_15 x}
```

The first argument is an inout vector of eight Q4.15 values. The input is the partial polynomial evaluated so far and the output is the new partial polynomial after this step is executed.

The second and third arguments are the high and low segments of 16 coefficients of which one will be chosen for each of the eight polynomials being created. Each coefficient is a Q4.15 type.

The fourth argument is a modified version of the polynomial argument. This modified version is set up by the BBE_POLY8X20_SU instruction discussed earlier. The four guard bits of this value contain a polynomial segment index. The low 16 bits of this value contain a position within the segment for this argument.

The BBE_POLY8X20_STEP instruction implements the following equation:

```
for (i=0;i<8;i++) {
    Poly-out[i] = Poly-in[i] * Offset(x[i])  + Lookup( x[i]);
}
```

The values of i from 0 to 7 represent the positions in the vector. Poly-out and Poly-in represent the output and input values of the first argument, Poly.x[i] represents the fourth argument.

The multiplication is done by sign extending x[15:0] by two bits and multiplying by Poly-in[17:0], which gives a 36-bit product. A round value of 0x000004000 is added to the product and the result is shifted right by 15 bits giving a 21-bit value. Because of the 2-bit sign extension of one of the operands, this can be converted to a 20-bit result without saturation.

In parallel with the multiplication process, the Lookup value is created separately for each vector element by using x[19:16] as an index into eight elements in lu_l and lu_h. Index values 0-7 read values in little endian order from lu_l while index values 8-15 read values in little endian order from lu_h.

The Lookup value selected in this way is added to the rounded and shifted product to create the output value. The addition wraps at the left end and does not saturate.

2.15.3 Example with Sine

Using the sine function as an example, assume that the input to the function is a Q15 number in the range [-1.0,+1.0) and that what we want is the sine of pi times the input.

That means the input represents the entire unit circle starting from the negative real axis, going through the negative imaginary axis, through the positive real axis and the positive imaginary axis to just before the negative real axis again. The output sine function will go through the values 0.0, -1.0, 0.0, +1.0, 0.0.

The range of the argument in hex in the Q4.15 format goes from 0xF8000 to 0x07FFF. For the following ranges, the index and offset are given from Table 2–27. Note that the offset is always in range [0xFF800..0x007FF].

Table 2–27. POLY_SU Arguments

POLY_SU Argument in Range	Index	Center
[0xF8000..0xF8FFF]	8	0xF8800
[0xF9000..0xF9FFF]	9	0xF9800
[0xFA000..0xFAFFF]	A	0xFA800
[0xFB000..0xFBFFF]	B	0xFB800
[0xFC000..0xFCFFF]	C	0xFC800
[0xFD000..0xFDFFF]	D	0xFD800
[0xFE000..0xFEFFF]	E	0xFE800
[0xFF000..0xFFFFF]	F	0xFF800
[0x00000..0x00FFF]	0	0x00800
[0x01000..0x01FFF]	1	0x01800
[0x02000..0x02FFF]	2	0x02800
[0x03000..0x03FFF]	3	0x03800
[0x04000..0x04FFF]	4	0x04800
[0x05000..0x05FFF]	5	0x05800
[0x06000..0x06FFF]	6	0x06800
[0x07000..0x07FFF]	7	0x07800

An argument in any position in the vector which falls into the range in the first row will have an index of 8 and will use the eighth element of the lookup in every stage of the polynomial. The polynomial will approximate a Taylor series centered around 0xF8800 and an offset will be used in the range [0xFF800..0x007FF]. Effectively then, the vector registers holding the coefficients are holding 16 separate series of coefficient for 16 polynomial segments.

Following is code for the vector sine routine, which is drawn from the `VectorSine_proj` example contained in `bbe16_examples.xws`. Following the code is a discussion about how it is implemented. Note that the data types used here are the standard default xb_vec8x20 and xb_vec8x16 rather than the fractional 20- and 16-bit Q4_15 and Q1_15 data types.

Example Code for Sine Computation

```
void compute_sin_vec_bbe(short *out, const short *in,
                unsigned n_16,
                const short *tables) {
  xb_vec8x16 *__restrict in_p = (xb_vec8x16*)in;
  xb_vec8x16 *__restrict table_p = (xb_vec8x16*)tables;
  xb_vec8x16 *__restrict out_p = (xb_vec8x16*)out;
  int i;

  xb_vec8x20 table3_hi = table_p[2*3+1];
  xb_vec8x20 table3_lo = table_p[2*3+0];
  xb_vec8x20 table2_hi = table_p[2*2+1];
  xb_vec8x20 table2_lo = table_p[2*2+0];
  xb_vec8x20 table1_hi = table_p[2*1+1];
  xb_vec8x20 table1_lo = table_p[2*1+0];
  xb_vec8x20 table0_hi = table_p[2*0+1];
  xb_vec8x20 table0_lo = table_p[2*0+0];

  for (i=0; i<n_16; i++) {
     xb_vec8x20 in0 = in_p[i*2];
     xb_vec8x20 in1 = in_p[i*2+1];
    xb_vec8x20 x0 = BBE_POLY8X20_SU(in0, 0+12, 2);
    xb_vec8x20 x1 = BBE_POLY8X20_SU(in1, 0+12, 2);
    xb_vec8x20 par0 = (short)0;
    xb_vec8x20 par1 = (short)0;
      /*xb_vec8x20 x1 = In the above code, the immediate value of 12
      means it will extract bits [15:12] of input data. The Immediate
      value of 2 means the offset will be multiplied by four.*/

    BBE_POLY8X20_STEP(par0, table3_hi, table3_lo, x0);
    BBE_POLY8X20_STEP(par1, table3_hi, table3_lo, x1);
    BBE_POLY8X20_STEP(par0, table2_hi, table2_lo, x0);
    BBE_POLY8X20_STEP(par1, table2_hi, table2_lo, x1);
    BBE_POLY8X20_STEP(par0, table1_hi, table1_lo, x0);
    BBE_POLY8X20_STEP(par1, table1_hi, table1_lo, x1);
    BBE_POLY8X20_STEP(par0, table0_hi, table0_lo, x0);
    BBE_POLY8X20_STEP(par1, table0_hi, table0_lo, x1);
    out_p[i*2]   = par0;
    out_p[i*2+1] = par1;
  }
```

```
}
```

The tables used are hex equivalents of the following four tables:

```
xb_vec8xq15 Table0[16] = // sin(pi*(2n+1)/16), n=0..15
         { 0.19509032,  0.55557023,  0.83146961,  0.98078528,
           0.98078528,  0.83146961,  0.55557023,  0.19509032.
          -0.19509032, -0.55557023, -0.83146961, -0.98078528,
          -0.98078528, -0.83146961, -0.55557023, -0.19509032 };

xb_vec8xq15 Table1[16] = //   (1/4)*pi^1*cos(pi*(2n+1)/16), n=0..15
         { 0.77030696,  0.65303471,  0.43634384,  0.15322358,
          -0.15322358, -0.43634384, -0.65303471, -0.77030696,
          -0.77030696, -0.65303471, -0.43634384, -0.15322358,
           0.15322358,  0.43634384,  0.65303471,  0.77030696 };

xb_vec8xq15 Table2[16] = // -(1/32)*pi^2*sin(pi*(2n+1)/16), n=0..15
         {-0.06017076, -0.17135183, -0.25644613, -0.30249883,
          -0.30249883, -0.25644613, -0.17135183, -0.06017076,
           0.06017076,  0.17135183,  0.25644613,  0.30249883,
           0.30249883,  0.25644613,  0.17135183,  0.06017076 };

xb_vec8xq15 Table3[16] = //-(1/384)*pi^3*cos(pi*(2n+1)/16), n=0..15
         {-0.07919401, -0.06713744, -0.04485980, -0.01575267,
           0.01575267,  0.04485980,  0.06713744,  0.07919401,
           0.07919401,  0.06713744,  0.04485980,  0.01575267,
          -0.01575267, -0.04485980, -0.06713744, -0.07919401 };
```

The coefficients given in the table are simple truncated Taylor Series-derived numbers and the resulting function is accurate to around 3-4 ULP (Unit of Least Precision, or Unit in the Last Place). Adjustment for the truncation of the Taylor Series and care with arguments near zero should bring that error down to around 1-2 ULP.

Each of the instructions works on eight elements at a time and two sets of instructions are interleaved because of the 2-cycle recurrence in the polynomial step instruction so that the loop works on 16 elements at a time. The compiler should be able to turn the loop into an 8-cycle loop bringing about a result of 0.5 cycles per Sine result.

Using sin(pi/6) as an example, that will be represented as an input of 0x01555. The BBE_POLY8X20_SU instruction would, with both immediates 0, create an index of 1 and an offset of 0xFFD55. Since the second immediate is 2 (as shown in “Example Code for Sine Computation” on page 43), the offset created will be 4x larger, or 0xFF554.

If z is the offset, and n is the index, then the result created by four BBE_POLY8X20_-STEP instructions will be:

```
Poly_result = Table0[1] + z*(Table1[1] + z*(Table2[1] + z*Table3[1]))
```

or, substituting for the particular case of sin(pi/6) with z as 0xFF554 or -0.0833:

```
Poly_result= 0.555 - 0.0833*(0.653 - 0.0833*(-0.171 - 0.0833*(-
0.067)))
```

which comes out to 0.4994.

2.15.4 Cosine Computation

Cosine can be very similar to sine. In radians cos(x) = sin(x+pi/2). In these units (radians/pi) for the argument, cos(x) = sin(x+0.5). There are two ways to do this. The first is simply to add 0.5 to the input and run the sine algorithm on it. This should not take additional time because the add can be done at the beginning in slot-2. The other method is to rotate the coefficients.

The first method has minor code changes to do the BBE_ADD8X20.

The second method has tables which would need to be converted to short equivalents as was done for the sine example above:

```
xb_vec8xq15 Table0[16] =
         { 0.98078528,  0.83146961,  0.55557023,  0.19509032.
          -0.19509032, -0.55557023, -0.83146961, -0.98078528,
          -0.98078528, -0.83146961, -0.55557023, -0.19509032,
           0.19509032,  0.55557023,  0.83146961,  0.98078528 };

xb_vec8xq15 Table1[16] =
         {-0.15322358, -0.43634384, -0.65303471, -0.77030696,
          -0.77030696, -0.65303471, -0.43634384, -0.15322358,
           0.15322358,  0.43634384,  0.65303471,  0.77030696,
           0.77030696,  0.65303471,  0.43634384,  0.15322358 };

xb_vec8xq15 Table2[16] =
         {-0.30249883, -0.25644613, -0.17135183, -0.06017076,
           0.06017076,  0.17135183,  0.25644613,  0.30249883,
           0.30249883,  0.25644613,  0.17135183,  0.06017076,
          -0.06017076, -0.17135183, -0.25644613, -0.30249883 };

xb_vec8xq15 Table3[16] =
         { 0.01575267,  0.04485980,  0.06713744,  0.07919401,
           0.07919401,  0.06713744,  0.04485980,  0.01575267,
          -0.01575267, -0.04485980, -0.06713744, -0.07919401,
          -0.07919401, -0.06713744, -0.04485980, -0.01575267 };
```

The second method should be a little more straightforward if a vector cosine is desired. The first method should be easier if both sine and cosine are desired to be computed. For example, you could add 0.5 to the input arguments and set up a second value, and then do the cosine and sine evaluations alternately.

2.15.5 Derivation of Coefficients

The details of the coefficients are determined (for sine) by the following reasoning: We want sin(x). But we're given y=x/pi with |y| =< 1.0. This is a Taylor series expansion of sine around `x0`.

```
sin(x) = sin(x0)
```

$$+\frac{1}{1!}\times\frac{d(\sin(x_0))}{dx}\times(x-x_0)$$

$$+\frac{1}{2!}\times\frac{d^2(\sin(x_0))}{dx^2}\times(x-x_0)^2$$

$$+\frac{1}{3!}\times\frac{d^3(\sin(x_0))}{dx^3}\times(x-x_0)^3$$

where x0 is one of pi*(2*n-15)/16 for n=0..15 with an error near $\frac{1}{4!}\times\frac{d^4(\sin(x_0))}{dx^4}\times(x-x_0)$

Since |x-x0| is less than pi/16, this error approximation gives about 6e-5, which is about two units in the least significant place of a Q15. We assume that error can be reduced sufficiently by careful selection of coefficients which are not precisely Taylor series. If that's not so, then one more term can be added and two cycles added to the inner loop we illustrated.

Substituting derivatives...

```
sin(x) =       sin(x0)
           +       cos(x0)*(x-x0)^1
           - (1/2)*sin(x0)*(x-x0)^2
           - (1/6)*cos(x0)*(x-x0)^3
```

where x0 is one of pi*(2*n-15)/16 for n=0..15

Noting that x = pi*y...

```
sin(pi*y) =       sin(pi*y0)
          +       cos(pi*y0)*(pi*(y-y0))^1
          - (1/2)*sin(pi*y0)*(pi*(y-y0))^2
          - (1/6)*cos(pi*y0)*(pi*(y-y0))^3
```

Note that x0 is one of pi*(2*n-15)/16 for n=0..15. Since x = pi*y, y0 = x0/pi thus, y0 is one of (2*n-15)/16 for n=0..15, where y0 is one of (2*n-15)/16 for n=0..15
or...

```
sin(pi*y) = a * b*(y-y0) + c*(y-y0)^2 + d*(y-y0)^3
        a =             sin(pi*y0)
        b =        pi^1*cos(pi*y0)
        c = -(1/2)*pi^2*sin(pi*y0)
        d = -(1/6)*pi^3*cos(pi*y0)
```

or...

```
sin(pi*y) = a + (y-y0)*[b + (y-y0)*(c + (y-y0)*[d + (y-y0)*0])]
        a =             sin(pi*y0)   // |a| < 1
        b =        pi^1*cos(pi*y0)   // |b| < pi     ~= 3.2
        c = -(1/2)*pi^2*sin(pi*y0)   // |c| < pi^2/2 ~= 5.1
        d = -(1/6)*pi^3*cos(pi*y0)   // |d| < pi^3/6 ~= 5.3
```

Let us substitute z=4*(y-y0)...

```
sin(pi*y) = a + (z/4)*[b + (z/4)*(c + (z/4)*[d + (z/4)*0])]
        a =             sin(pi*y0)   // |a| < 1
        b =        pi^1*cos(pi*y0)   // |b| < pi     ~= 3.2
        c = -(1/2)*pi^2*sin(pi*y0)   // |c| < pi^2/2 ~= 5.1
        d = -(1/6)*pi^3*cos(pi*y0)   // |d| < pi^3/6 ~= 5.3
```

where z = 4*(y-y0) // |z| < 0.25

And push the 4s through....

```
sin(pi*y) = a + z*[b/4 + z*(c/16 + z*[d/64 + z*(0/256)])]
        a =             sin(pi*y0)   // |a| < 1
        b =        pi^1*cos(pi*y0)   // |b| < pi     ~= 3.2
        c = -(1/2)*pi^2*sin(pi*y0)   // |c| < pi^2/2 ~= 5.1
        d = -(1/6)*pi^3*cos(pi*y0)   // |d| < pi^3/6 ~= 5.3
```

and z = 4*(y-y0) // |z| < 0.25

Adjusting the values of a, b, c, and d...

```
sin(pi*y) = a + z*[b + z*(c + z*[d + z*0])]
        a =               sin(pi*y0)   // |a| < 1
        b =    (1/4)*pi^1*cos(pi*y0)   // |b| < ~0.8
        c =  -(1/32)*pi^2*sin(pi*y0)   // |c| < ~0.4
        d = -(1/384)*pi^3*cos(pi*y0)   // |d| < ~0.1
```

and z = 4*(y-y0) // |z| < 0.25

The above can be put into the BBE_POLY8X20_STEP form. BBE_POLY8X20_SU produces 4*(y-y0) and the four calls to BBE_POLY8X20_STEP compute the polynomial.

2.16 Dual Peak Max Index and Max Absolute Index and Values

ConnX BBE16 contains instructions to speed up the determination of the dual peaks (maximum, and next to maximum, values) in arrays of 20-bit and 40-bit values. These instructions will also produce the indices - the locations in the array of the maximum and next-to-maximum values. There are also instructions for computing the dual peak maximum absolute values and indices as an alternative, since in an array containing positive and negative values, those whose absolute value deviate the largest amount from zero (and second-largest amount) may be of interest.

The four basic instructions are as follows:

```
BBE_MAXIDX8X20(xb_vec8x20 a /*inout*/,xb_vec8x20 b)

BBE_MAXIDX4X40(xb_vec4x40 a /*inout*/,xb_vec4x40 b)

BBE_MAXABSIDX8X20(xb_vec8x20 a /*inout*/,xb_vec8x20 b)

BBE_MAXABSIDX4X40(xb_vec4x40 a /*inout*/,xb_vec4x40 b)
```

The idea behind these instructions is that the inout vector register a contains the running maximum, and next-to-maximum values and indices determined by successive application of the MAXIDX or MAXABSIDX instruction on successive vectors of the array. This variable also contains other housekeeping items such as the current array index value, the signs of the maximum and next-to-maximum absolute value (in the case of the MAXABSIDX instructions) and an initialization flag. Details on the structure of inout vector register a and the arguments for the operations can be found in the ISA HTML descriptions.

Protos are defined so that these instructions can be applied to fractional real types. Note that the instructions are only for real integers and fixed-point types (not complex). In addition, protos are defined to initialize the inout running maximum vector a. For example:

```
BBE_MAXIDX8X20_INIT(xb_vec8x20 a /*out*/)
```

which sets all fields in ‘a’ to zero. Variations of these protos exist for all supported types.

There is also a proto that allows the inout vector register a to be treated as a separate input and output register, for ease of programming. This is illustrated in the example that follows.

```
BBE_MAXIDX8X20_V { out xb_vec8x20 a, in xb_vec8x20 b, in xb_vec8x20 c }
```

In addition, protos exist to allow the extraction of the fields in the running maximum vector after the complete set of MAXIDX operations has run. These are:

```
BBE_MAXIDX8X20_VAL(xb_int20 a/*out*/, xb_vec8x20 b)

BBE_MAXIDX8X20_VAL2(xb_int20 a/*out*/, xb_vec8x20 b)

BBE_MAXIDX8X20_IDX(xb_int20 a/*out*/, xb_vec8x20 b)

BBE_MAXIDX8X20_IDX2(xb_int20 a/*out*/, xb_vec8x20 b)
```

The VAL and VAL2 protos replicate the maximum and next-to-maximum values determined (or absolute values, in the case of MAXABSIDX) into output vector ‘a’. The IDX and IDX2 replicate the index in the array of the maximum and next-to-maximum values.

Finally, for the MAXABSIDX operations, there is one final proto that returns the values of the sign bits for the maximum and next-to-maximum absolute values (i.e., the original signed values in the array). This also contains the initialization flag, which after the first operation of the MAXIDX operation, is set to zero. The proto is:

```
BBE_MAXABSIDX8X20_SGNSINFLG(xb_int20 a/*out*/, xb_vec8x20 b)
```

The returned value a contains in each field the sign bits and initialization flag. Bit 2 is the sign bit for the original next-to-maximum value (0 = positive, 1 = negative). Bit 1 is the sign bit for the original maximum value. Bit 0 is the initialization flag and can be ignored.

Thus if the value returned in each field of a is 0x7 or 0x6, then the maximum and next-to-maximum original values were both negative. If it is 0x5 or 0x4, then the maximum was positive and the next-to-maximum was negative. If it is 0x3 or 0x2, then the maximum was negative and the next-to-maximum was positive. If it is 0x1 or 0x0, then both maximum and next-to-maximum were positive.

2.16.1 Example using Dual Peak Max index instructions

The following code illustrates how one might use these instructions to extract the maximum, and next-to-maximum values and indices from an array.

```
void dualpeak_i16_bbe(int *max_idx_p, int *max_idx2_p, int *max_val_p,
           int *max_val_p2, const short *a, int n_8)
{
  int i;
  xb_vec8x20 maxdata;
maxdata=BBE_MAXIDX8X20_INIT();
  const xb_vec8x16 * __restrict a_v = (xb_vec8x16 *)a;
  for (i = 0; i < n_8; ++i) {
    xb_vec8x20 val= a_v[i];
    maxdata = BBE_MAXIDX8X20_V(maxdata, val);
  }
  *max_idx_p = BBE_MAXIDX8X20_IDX(maxdata);
  *max_idx2_p = BBE_MAXIDX8X20_IDX2(maxdata);
*max_val_p = BBE_MAXIDX8X20_VAL(maxdata);
  *max_val2_p = BBE_MAXIDX8X20_VAL2(maxdata);
}
```

The vector register variable `maxdata` is used to store the running maximum and next-to-maximum values and indices. Note the use of the _INIT proto to initialize it. Next the vector is walked through in the loop and the _V proto form of the operation is used to update `maxdata`. Finally at the end, the _IDX, _IDX2, _VAL and _VAL2 protos are used to extract the maximum and next-to-maximum indexes and values from the array.

2.17 Despread Operation (Optional)

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

```
BBE_DSPR4XC20(xb_vec8x20 vt /*out*/, xb_vec8x20 vs, xb_vec4x40
vr, imm2x1)
```

```
BBE_DSPRA4XC20(xb_vec8x20 vt /*inout*/, xb_vec8x20 vs, xb_vec4x40
vr, imm2x1);
```

The vector register vs contains four complex 20-bit pairs (eight 20-bit elements) with the real and imaginary parts interleaved:

{v.imag3,v.real3, v.imag2,v.real2, v.imag1,v.real1, v.imag0,v.real0}

The vector register vr contains 64 2-bit unsigned values, divided into 16 8-bit codewords. Four codewords are stored in each of the least significant 32 bits of each 40-bit vector element in the vector register. The eight guard bits in each vector element of the vector register vr are unused.

There are also protos for the optional despread operations that take in xb_4xc20 as inputs and produce xb_4xc20 outputs. These are called BBE_DSPREAD4XC20 and BBE_DSPREADA4XC20:

```
BBE_DSPREAD4XC20 { xb_vec4xc20 a /*out*/, xb_vec4xc20 b,
xb_vec4x40 c, imm}
```

```
BBE_DSPREADA4XC20 { xb_vec4xc20 a /*inout*/, xb_vec4xc20 b,
xb_vec4x40 c, imm}
```

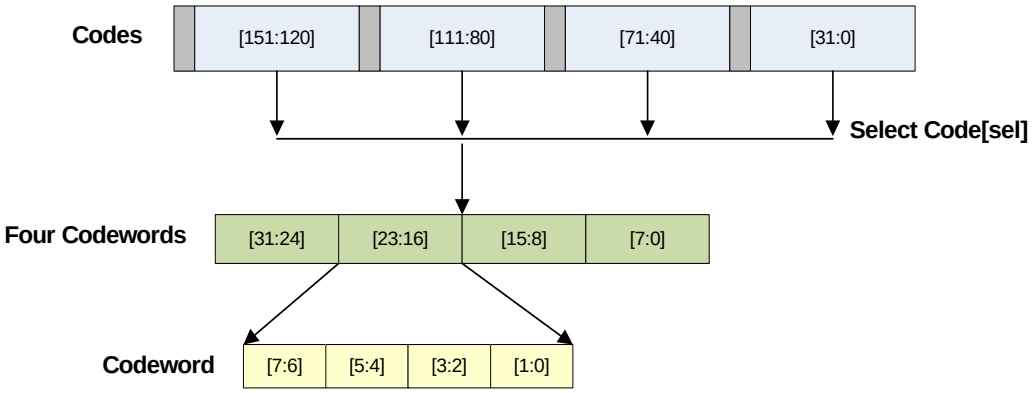


Figure 2–7. BBE_DSPR4XC20 Codeword Format

In each codeword, each unsigned 2-bit value c represents the code value in complex number $e^{(-i * \Pi * c/2)}$, which will be multiplied by complex input data in vs. Table 2–28 lists the code values and details.

Table 2–28. Codeword Values

Code Value (c)	Multiplier	Meaning
0	1	$e^{-i \Pi 0/2}$
1	-i	$e^{-i \Pi 1/2}$
2	-1	$e^{-i \Pi 2/2}$
3	i	$e^{-i \Pi 3/2}$

Following are illustrations of the BBE_DSPRA4XC20 operation details.

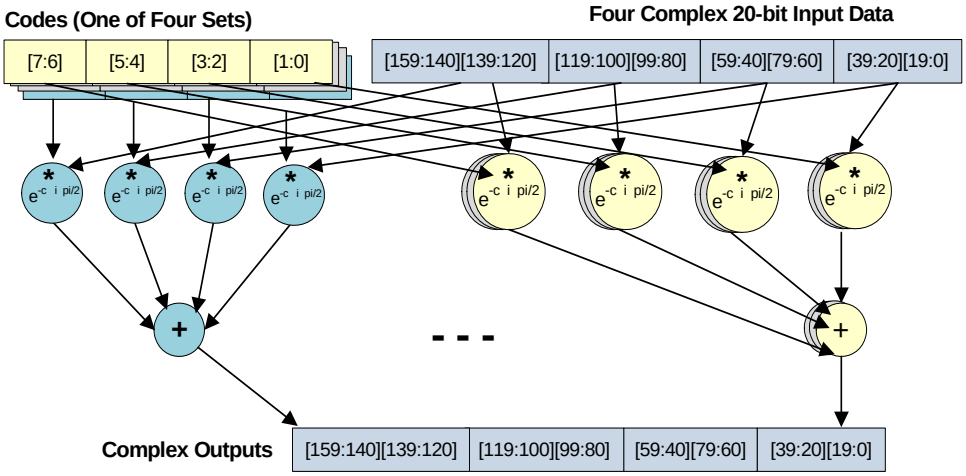


Figure 2–8. BBE_DSPR4XC20 Operation Detail for Complex Outputs

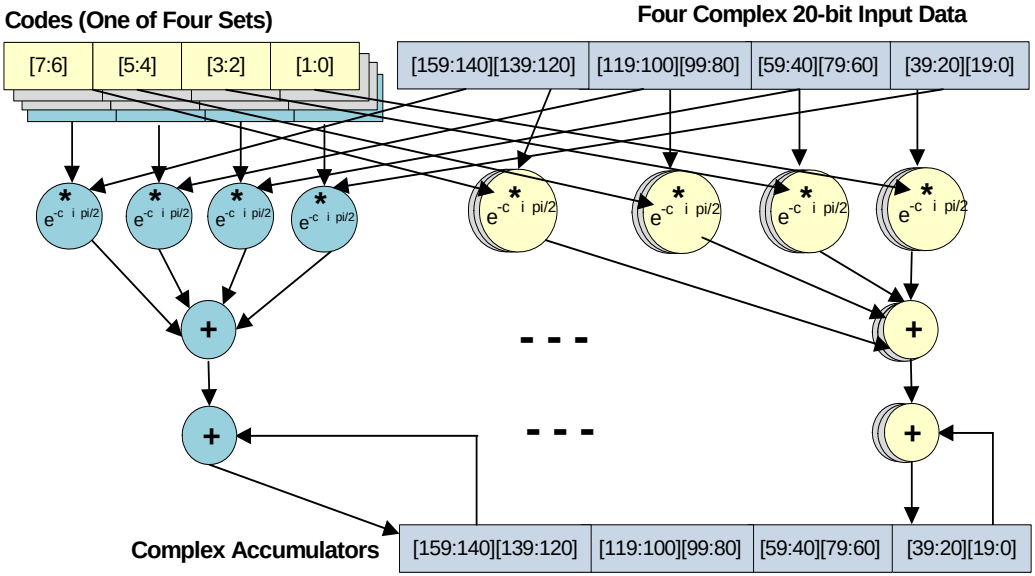


Figure 2–9. BBE_DSPRA4XC20 Operation Detail for Complex Accumulators

The 2-bit values are organized into 16 codewords with each codeword containing four contiguous 2-bit values. The imm2x1 immediate operand selects four codewords (16 2-bit values) from the vector register vr. A select value of 0 selects the four codewords values in vr[31:0] and select of 3 chooses most significant four codewords.

Table 2–29. Codeword Values

Select imm2x1	4x4 Codewords	Four Codewords
0	vr[31: 0]	(vr[31: 24] vr[23: 16] vr[15: 8] vr[7: 0])
1	vr[63: 32]	(vr[63: 56] vr[55: 48] vr[47: 40] vr[39: 32])
2	vr[95: 64]	(vr[95: 88] vr[87: 80] vr[79: 72] vr[71: 64])
3	vr[127: 96]	(vr[127:120] vr[119:112] vr[111:104] vr[103: 96])

Each of the 8-bit codewords contains four 2-bit codes. The complex numbers represented by each of the four codes in the codeword are multiplied by the corresponding complex element in vector register vs. Those four complex products are added to produce a single 20-bit complex sum. That sum complex products from a codeword is written to one of the complex elements of the vector register vt.

Computing that sum of products for each of four codewords produces four complex output results in vector register vt. The result contains four complex pairs (eight 20-bit elements) with the real and imaginary parts interleaved:

{v.imag3,v.real3, v.imag2,v.real2, v.imag1,v.real1, v.imag0,v.real0}

(Where 0 is the least significant element of the register).

The hardware efficient implementation of this instruction uses adders instead of complex multiplication hardware.

2.17.1 Hadamard Transform Operations (part of Optional Despread Operation)

The optional despread operation also contains two instructions used for the computation of Hadamard transforms, BBE_ADDSUB8X20P and BBE_NEG8X20T.

BBE_ADDSUB8X20P is an 8-way vector add or subtract under predicate control. Its operation is:

```
BBE_ADDSUB8X20P { out xb_vec8x20 vouthi, inout xb_vec8x20 lo, in
xb_vec8x20 vinhi, in vsel sel }
```

This operation takes the two inputs `lo` and `vinhi` and produces results in the outputs `lo` and `vouthi` (`lo` is an inout register) under the control of the selection register `sel`. Each pair of inputs, one from `lo` and one from `vinhi`, produces two outputs. Bit values in `sel` control whether the inputs are multiplied by +1 or -1 before they are added together.

More details on the operation of this instruction are in the instruction HTML description.

BBE_NEG8X20T is a conditional negate. Its operation is:

```
BBE_NEG8X20T { out xb_vec8x20 a, in xb_vec8x20 b }
```

This instruction will either copy an element from b or the negative of the element from b into the corresponding element of output a, depending on a bit pattern obtained from twiddle register TWIDDLEA. The twiddle register can hold up to 16 8-bit patterns, embedded in eight 20-bit elements; these are rotated 8 bits to the right (after extraction from the embedded format) on each application of the BBE_NEG8X20T instruction. Thus one load of TWIDDLEA with the right set of patterns can serve for 16 applications of the BBE_NEG8X20T instruction, and if the patterns recur in groups of 2, 4, 8 or 16, the same twiddle register contents can be reused multiple times.

2.18 Divide and Reciprocal Square Root Instructions (Optional)

These optional instructions perform the common, but complicated arithmetic functions of fixed point divide and square root. Eight divides are performed per instruction, with a 16-bit dividend divided by a 16-bit divisor to create a 16-bit quotient. Inputs are in two 8x20 bit vectors and the output is an 8x20 bit vector.

The divides come in four forms:

- BBE_DIV8X16S(xb_vec8x20 vr, xb_vec8x20 vs) is a signed 16-bit divide, which interprets the inputs as signed integers. The inputs are eight signed, 16-bit dividends in vr and eight signed, 16-divisors in vs; the higher-order bits are ignored. This instruction computes an 8-way SIMD signed divide in a 6-cycle operation in which the vector divides are not pipelined. It then returns eight, 16-bit signed integer results, in range [-2^{15} .. 2^{15} -1]; if there is a division by zero, it returns 0x7FFFF or 0x80000.
- BBE_DIV8X16U (xb_vec8x20 vr, xb_vec8x20 vs) is an unsigned 16-bit divide, which interprets the inputs as unsigned integers. The inputs are eight unsigned, 16-bit dividends in vr and eight unsigned, 16-bit divisors in vs; the higher-order bits are ignored. This instruction computes an 8-way SIMD unsigned divide in a 6-cycle operation. It then returns eight,16-bit unsigned integer results, in the range [0 .. 2^{16}-1]; if there is a division by zero, it returns a result of 0x7FFFF.
- BBE_DIV8X16Q(xb_vec8x20 vr, xb_vec8x20 vs) is an unsigned fractional divide which shifts its 16-bit dividend input left by 16 bits before performing unsigned divide. This allows computation of a fractional Q15 quotient. The Inputs are eight unsigned, 16-bit dividends in vr and eight unsigned, 16-bit divisors in vs; the higher-order bits are ignored and each dividend must be less than its divisor. This instruction computes an 8-way SIMD unsigned divide in a 6-cycle operation, which results in eight 16-bit positive fractional results; valid results are in the range [0 .. 1) or [0 .. 0x07FFF]. If there is a division by zero, it returns 0x7FFFF; If the dividend is greater than or equal to the divisor, it returns 0x7FFFF.

- BBE_DIV8X32S(xb_vec4x40 vu, xb_vec4x40 vr, xb_vec8x20 vs) is a signed 32-bit by 16-bit divide, which means it divides eight 32-bit numbers in two vector registers by 16-bit numbers in one vector register. The 8-way divide creates exact integer quotients. The inputs are eight signed, 32-bit dividends in vu and vr eight signed, 16-bit divisors in vs; the higher-order bits are ignored. This instruction computes an 8-way SIMD signed divide in a 6-cycle operation in which the vector divides are not pipelined. It then returns eight, 16-bit signed integer results, in range [-2^{15} .. 2^{15} -1]. If there is a division by zero, it returns 0x7FFFF or 0x80000. In case of overflow, it returns 0x7FFFF or 0x80000.

For the operation BBE_RSQRT4X32 {out xb_vec4x40 y, in xb_vec4x40 x}, one of the protos is BBE_RSQRT4X40 {out xb_vec8xq4_15 y, in xb_vec4x40 x}. The reciprocal square root operation is important for algorithms that involve accurate normalization. The input is a 4x40 bit vector and the output is the same type. The 4-way SIMD instruction assumes its inputs are 32-bit integers and computes 16-bit fractional outputs. Alternatively, the input can be interpreted as a 32-bit fractional value and the output as 16-bit integers. The reciprocal square root is accurate to approximately one bit in the least significant position. This operation is undefined on negative or zero inputs, returning 0x7FFFF. This is a pipelined 4-cycle operation.

Note that BBE_RSQRT4X32 is a bit unusual in that it returns four positive 16-bit results in a 4x40 register.

2.19 FLIX Slots and Formats

ConnX BBE16 can issue three operations in a single instruction bundle using Xtensa LX FLIX (VLIW) technology. It contains scalar and vector SIMD (single-instruction/multiple-data) operations.

BBE16 is implemented with three 64-bit formats in addition to the standard 24 and optional 16-bit instruction formats in the Xtensa LX architecture. Each of the 64-bit formats can bundle three operations into three separate FLIX slots.

2.20 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 BBE16. Consult Appendix A for information about this on-line information. Since this list is automatically generated from the machine description, it is comprehensive and up to date.

3. Programming ConnX BBE16

ConnX BBE16 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 ConnX BBE16 and explores the capabilities of automated instruction inference and vectorization, and cases where the use of intrinsic-based programming is appropriate.

To use ConnX BBE16 data types and intrinsics in C, please include the appropriate header file via the following preprocessor directive:

```
#include <xtensa/tie/xt_bbe16.h>
```

3.1 ConnX BBE16 Data Types

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

Table 3–30. Scalar Memory Data Types

Scalar	Scalar 16	Scalar 32
Int	xb_int16	xb_int32
Fixed pt	xb_q15	xb_q1_30
Complex int	xb_c16	xb_c32
Complex fixed pt	xb_cq15	xb_cq1_30

Table 3–31. Scalar Register Data Types

Scalar	Scalar 20	Scalar 40
Int	xb_int20	xb_int40
Fixed pt	xb_q4_15	xb_q9_30
Complex int	xb_c20	xb_c40
Complex fixed pt	xb_cq4_15	xb_cq9_30

Table 3–32. Vector Memory Data Types

Mem Vector	Vector 16	Vector 32	Double Vec 32	Double Vec 40
Int	xb_vec8x16	xb_vec4x32	xb_vec8x32	xb_vec8x40
Unsigned int	xb_vec8x16U	xb_vec4x32U		
Fixed pt	xb_vec8xq15	xb_vec4xq1_30	xb_vec8xq1_30	xb_vec8xq9_30
Complex int	xb_vec4xc16	xb_vec2xc32	xb_vec4xc32	xb_vec4xc40
Complex fixed pt	xb_vec4xcq15	xb_vec2xcq1_30	xb_vec4xcq1_30	xb_vec4xcq9_30

Table 3–33. Vector Register Data Types

Temp Vector	Vector 20	Vector 40	Double Vec 40
Int	xb_vec8x20	xb_vec4x40	xb_vec8x40
Fixed pt	xb_vec8xq4_15	xb_vec4xq9_30	xb_vec8xq9_30
Complex int	xb_vec4xc20	xb_vec2xc40	xb_vec4xc40
Complex fixed pt	xb_vec4xcq4_15	xb_vec2xcq9_30	xb_vec4xcq9_30

The ConnX BBE16 DSP Engine processes both fixed-point and integer data. The basic data element, xb_vec8x16 for signed and xb_vec8x16U for unsigned, is 16-bits wide, and a vector consists of eight such 16-bit elements. Thus, an input vector in memory is 128-bits wide. ConnX BBE16 also supports, xb_vec4x32 and xb_vec4x32U, a wider 32-bit data element with four such elements in a 128-bit vector. This double-width data type is typically generated as the result of multiply or multiply/accumulate operations on 16-bit data elements.

When a 128-bit vector is loaded from memory into a ConnX BBE16 register, the ConnX BBE16 automatically expands the vector to 160 bits by adding guard bits, creating a new data type, xb_vec8x20 or xb_vec4x40. The programmer should use the 128-bit data types for variables that will typically be stored in memory such as arrays and the 160-bit data types for temporary variables. The extra guard bits provide higher precision for intermediate computations. During a load operation, ConnX BBE16 expands 16-bit scalar values to 20 bits and it expands 32-bit scalar values to 40 bits.

ConnX BBE16 supports both signed and unsigned data types in memory and provides a symmetrical set of load and store instructions for both these data types. For signed data, a load results in sign extension from 16 to 20 (or 32 to 40) bits. For unsigned data, loads result in zero extension.

During memory stores, ConnX BBE16 automatically compresses the expanded data elements back to their original size. ConnX BBE16 store operations results in saturation of data if the data-element value held in the ConnX BBE16 register exceeds the capacity of the data element's width as stored in memory (16 or 32 bits).

Because of the presence of guard bits and the semantics of the load and store instructions, the data-processing instructions (such as the ALU and MAC instructions) come in only one variety —they treat the data as signed. This convention allows the ConnX BBE16 to correctly process unsigned data as long as there is no over-flow into the most significant bit of the elements in a vector register. Software must ensure that the input data is scaled appropriately to prevent such an overflow when processing unsigned numbers. Note that this problem is no different than the general problem of preventing overflows when running code on any fixed-point processor.

Automatic type conversion is also supported between vector types and the associated scalar types, for example between xb_vec8x20 and `short` and between xb_vec4x40 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. See Section 3.1.1 on page 60 for details about automatic type conversion.

The C compiler also supports the xb_vec8x40 type, a vector of eight, 40-bit elements. While not directly supported by hardware, as no register can hold 320 bits, this data type is useful because logically the multiplication of two variables of type xb_vec8x20 returns a variable of type xb_vec8x40. The xb_vec8x40 type is implemented using a set of two registers.

ConnX BBE16 data is intended to be stored in memory without guard bits, for example, as xb_vec8x16. Therefore, data types must be converted when loaded from and stored to memory. Consider the following code fragment:

```
xb_vec8x16 *a, *b, *c;
*c++ = BBE_ADD8X20(*a++, *b++);
```

a, b, and c all point to `xb_vec8x16` data, but the `ADD20` instruction expects `xb_vec8x20` data for arguments and produces `xb_vec8x20` data as output. The compiler converts the incoming `xb_vec8x16` data into the `xb_vec8x20` type by sign extending each 16-bit element to 20 bits and converting the outgoing `xb_vec8x20` type into a `xb_vec8x16` type using saturation on each data element of the vector. Because ConnX BBE16 load and store operations themselves perform sign-extension and saturation respectively, there is no cost for the conversions.

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 Chapter 2, there are alternative prototypes (protos) for the BBE16 instructions 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 complex scalar, q1_30 for fixed point with 1 bit for integer and 30 bits for fractional, sign bit implied, 64 bits total
- `xb_vec4xcq_9_30`: Register vector of four complex fixed point elements, each 9-bit integer and 30-bit fractional, sign bit implied, total 320 bits
- `xb_vec4x32U`**:** Memory real vector, four elements by 32 bits each unsigned, total 128 bits
- `xb_vec4xcq15`: Memory complex vector, four elements of Q15 fraction, total 128 bits

3.1.1 Automatic Type Conversion

Variables of one data type may be assigned to variables of another data type using the normal C "=" operator. The compiler will appropriately convert the values as discussed in this section.

Converting int16 and int32 (C short and C int) to all Types

Converting int16 and int32 to all type conversions are provided as follows:

- For vector types the value is replicated in all elements of the vector.
- For complex types the value is put in the real part, the imaginary gets assigned 0.
- For the integer types the conversion has the normal C meaning.
- For the fixed point types, the integer number is converted to the integer part of the fixed point number, with saturation if it exceeds the allowed values. This implies that the range of allowed values is very limited: for the q15 and q1.30 types only 0 and -1 are allowed.
- For q4.15, the range of -16 to 15 is allowed; for q9.30, the range of -512 to 511 is allowed.

Converting Between Unguarded Types to Guarded Types

All unguarded types are converted to the corresponding guarded types with a simple move, the value does not change.

Table 3–34 is a list of conversions from unguarded to guarded types.

Table 3–34. Unguarded Types to Guarded Types Conversion

Unguarded Type	To	Guarded Type
xb_c16	- >	xb_c20
xb_c32	- >	xb_c40
xb_cq15	- >	xb_cq4_15
xb_cq1_30	- >	xb_cq9_30
xb_int16	- >	xb_int20
xb_int32	- >	xb_int40
xb_q15	- >	xb_q4_15
xb_q1_30	- >	xb_q9_30
xb_vec2xc32	- >	xb_vec2xc40
xb_vec2xcq1_30	- >	xb_vec2xcq9_30
xb_vec4x32	- >	xb_vec4x40
xb_vec4xc16	- >	xb_vec4xc20
xb_vec4xc32	- >	xb_vec4xc40
xb_vec4xcq15	- >	xb_vec4xcq4_15
xb_vec4xcq1_30	- >	xb_vec4xcq9_30
xb_vec4xq1_30	- >	xb_vec4xq9_30
xb_vec8x16	- >	xb_vec8x20
xb_vec8x32	- >	xb_vec8x40
xb_vec8xq15	- >	xb_vec8xq4_15
xb_vec8xq1_30	- >	xb_vec8xq9_30

Converting Between Guarded Types to Unguarded Types

Guarded types are converted to the corresponding unguarded types by saturating the guarded value.

Table 3–35 is a list converting guarded types to unguarded types:

Table 3–35. Guarded Types Converted to Unguarded Types

Guarded Type	To	Unguarded Type
xb_c20	- >	xb_c16
xb_c40	- >	xb_c32
xb_cq4_15	- >	xb_cq15
xb_cq9_30	- >	xb_cq1_30
xb_int20	- >	xb_int16

Table 3–35. Guarded Types Converted to Unguarded Types (continued)

Guarded Type	To	Unguarded Type
xb_int40	- >	xb_int32
xb_q4_15	- >	xb_q15
xb_q9_30	- >	xb_q1_30
xb_vec2xc40	- >	xb_vec2xc32
xb_vec2xcq9_30	- >	xb_vec2xcq1_30
xb_vec4x40	- >	xb_vec4x32
xb_vec4xc20	- >	xb_vec4xc16
xb_vec4xc40	- >	xb_vec4xc32
xb_vec4xcq4_15	- >	xb_vec4xcq15
xb_vec4xcq9_30	- >	xb_vec4xcq1_30
xb_vec4xq9_30	- >	xb_vec4xq1_30
xb_vec8x20	- >	xb_vec8x16
xb_vec8x40	- >	xb_vec8x32
xb_vec8xq4_15	- >	xb_vec8xq15
xb_vec8xq9_30	- >	xb_vec8xq1_30

Other Type Conversions, Including Manual Conversions

- **Special typecasts between two types of the same element width:**

  Vector 160-bit fixed-point and complex data types can be converted (cast) to and from standard xb_vec8x20 or xb_vec4x40 types using explicit intrinsics. These conversions do not modify the underlying data, only reinterpret it as a different type.

  For example, the intrinsic BBE_MOV8X20_FROM4XC20() casts a complex vector to a real vector of 20-bit elements.

- **Conversion between real and complex vector types:**

  Assigning a real vector or scalar to a complex variable of the same element size sets the real part of the complex variable from the real variable, and zeroes the imaginary part. BBE_COMBINE intrinsics can be used for more general assignments of complex variables from real ones.

  See Appendix A for a discussion and example of these intrinsics.

- **Conversion between complex and real vector types:**

  Assigning a complex vector or scalar to a real variable of the same element size sets the real variable to the real part extracted from the complex variable. BBE_EXTRACT intrinsics can be used for more general assignments of real variables from complex ones.

  See Appendix A for a discussion and example of these intrinsics.

- **Conversion between scalar and vector types:**

  Assigning a standard scalar variable (e.g. short) to a vector of the compatible element type automatically sets each element of the vector to that scalar value. For example:

  ```
  xb_vec8x20 va = 42;
  ```

  Assigning BBE scalar variables (e.g. xb_int20) to vector variables will also logically replicate the scalar into every element of the vector. However, since BBE scalars are loaded into vector registers as replicated values, converting to a vector type typically does not require any additional operations. For example:

  ```
  xb_int20 sb= s_arr[5];
  xb_vec8x20 vb = sb + sb;
  ```

- **Conversion between vector and scalar types:**

  Assigning a vector variable to a compatible scalar variable automatically assigns element 0 of the vector to the scalar variable. For example:

  ```
  xb_vec8x20 vc = vec_arr[2];
  xb_int20 sc = vc;
  ```

- **Conversion between narrow types and wider types:**

  Assigning a vector of narrow (20-bit) elements to a vector of wide (40-bit) elements unpacks and sign-extends the narrow elements into wide elements. For example:

  ```
  xb_vec8x20 vn = vec_arr[3];
  xb_vec8x40 wb = vn;
  ```

- **Conversion between wider types and narrow types:**

  Assigning a vector of wide (40-bit) elements to a vector of narrow (20-bit) elements packs and saturates the wide elements into narrow elements. For example:

  ```
  xb_vec8x40 wm = vbarr[2];
  xb_vec8x20 vd = wm + wm;
  ```

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_vec8x20 types, but you can change the format to any other format.

Some examples of these display formats for a 160-bit variable are:

- xb_vec8x20m displays hex and decimal for each real element of vector
- (14, 40, 56, 59, 32, 30, 27, 23)=
  (0x0000e, 0x00028, 0x00038, 0x0003b, 0x00020, 0x0001e, 0x0001b, 0x00017)

- xb_vec4xc20m displays above as four complex numbers instead
- (14i+40, 56i+59, 32i+30, 27i+23)=
  (0x0000ei+0x00028, 0x00038i+0x0003b, 0x00020i+0x0001e, 0x0001bi+0x00017)

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

3.3 Operator Overloading and Vectorization

Common ConnX BBE16 operations can be accessed in C or C++ by applying standard C operators to the ConnX BBE16 data types. There are many operator overloads defined for the various types, and the compiler will infer the correct ConnX BBE16 vector operation in many cases.

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 ConnX BBE16 scalar types may turn into a loop of ConnX BBE16 vector operations that is as tightly packed and efficient as manual code using BBE16 intrinsics.

For operations that do not map to standard operators, intrinsics can be used.

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 ConnX BBE16 can be programmed, including the example results.

3.4 Prerequisite Reading

Cadence recommends the two following important Xtensa manuals that you should read and be familiar with before attempting to obtain optimal results by programming ConnX BBE16:

- *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.

3.5 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.

There are three basic programming styles that can be used, in increasing order of manual effort. These 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

One strategy is to start with legacy C code or to write the algorithm in a natural style using scalar types (possibly using ConnX BBE16 special scalars, e.g., for Q15 or complex or complex Q15 data) and seeing the limits of what automatic vectorization and operator overloading or inference can achieve. By profiling the code, computational-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 are discussed in the next sections.

3.6 Auto-Vectorization

Auto-vectorization of scalar C code using ConnX BBE16 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. Cadence 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 16-byte boundaries. The XCC compiler will naturally align arrays to start on 16-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.

3.7 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 BBE16 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 following examples illustrate how they work in conjunction.

3.8 Vectorization and Inferencing Examples

The examples provided with ConnX BBE16 include a project, bbe_vectorize_types, which illustrates some of the extent and the limits of automatic compiler capabilities. This includes three very simple algorithms: vector add, vector dot product and matrix multiply, and several scalar data types: int, short, xb_q15, xb_c16, xb_cq15, xb_q1_30, xb_c32, xb_cq1_30, xb_q4_15, xb_c20, xb_cq4_15.

Almost all these examples vectorize. There is no room to show all of them here, but let's look at a few. 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];
    }
```

When we compile the following code is produced:

```
{ bbe_lv4x32s.i v4, a3,16; bbe_add4x40 v3, v1, v0; bbe_lv4x32s.iu v1, a3, 32 }
{ bbe_sv4x32s.i v5, a4,16; nop; bbe_lv4x32s.iu v0, a2, 32 }
{ bbe_lv4x32s.i v2, a2,16; bbe_add4x40 v5, v4, v2; bbe_sv4x32s.iu v3, a4, 32 }
```

Note the use of vectorized loads, stores and vector 4x40 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_lv8x16s.i.n v0, a2, 16; bbe_add8x20 v2, v2, v0; bbe_lv8x16s.i.n v1, a3, 16 }
{ bbe_lv8x16s.iu v0, a2, 32; bbe_add8x20 v2, v1, v0; bbe_sv8x16s.iu v2, a4, 32 }
{ bbe_sv8x16s.i v2, a4, 16; nop; bbe_lv8x16s.iu v2, a3, 32 }
```

Note the mapping of complex q15 types into 8x16 loads and stores and 8x20 adds.

A vector dot product short:

```
short vec_dot_short() {
    int i;
    int sum = 0;
    for (i = 0; i < VEC_SIZE; i++) {
        sum += as[i] * bs[i];
    }

return sum;
```

produces code

```
loopgtz a4, 5ffc06c8 <vec_dot_short+0x24>
{ bbe_lv8x16s.iu v0, a2, 16; bbe_mula8x18packs v2, v1, v0; bbe_lv8x16s.iu v1, a3, 16 }
vec_dot_short+0x24:
{ nop; bbe_mula8x18packs v2, v1, v0; nop }
{ nop; nop; bbe_radd8x20 v0, v2 }
{ bbe_movav16 a2, v0; nop; nop }
retw.n
```

Note the automatic selection of the BBE_MULA8X18PACKS 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. Note that the compiler vectorizer has effectively promoted the short data type from 16-bits into 20-bits. If in the original example, the sum were to overflow 16-bits, the vectorized version would not overflow and would therefore behave differently than the original code.

Matrix multiplies are a little more challenging. The following subroutine vectorizes. The compiler can stride through array bm15 for small sized arrays. The code accumulates sum of products in a 40-bit sum, then saves to memory as 16-bit results:

```
// Matrix product fixed point 16 bits to 32 bits
xb_q15 am15[ARRAY_SIZE][ARRAY_SIZE], bm15[ARRAY_SIZE][ARRAY_SIZE];
xb_q15 cm15[ARRAY_SIZE][ARRAY_SIZE];

void mm_q15() {
    int i, j, k;
    for (i = 0; i < ARRAY_SIZE ; i += 1) {
        for (j = 0; j < ARRAY_SIZE ; j += 1) {
            xb_q9_30 sum=0;
            for (k = 0; k < ARRAY_SIZE; k += 1) {
                sum += am15[i][k] * bm15[k][j];
            }
            cm15[i][j] = sum;
        }
    }
```

The disassembly code is:

```
        loopgtza4,.LBB28_mm_q15
    {
        bbe_ls8x16s.iuv1,a3,2
        bbe_mula8x18v8,v9,v1,v0
        bbe_lv8x16s.iuv0,a2,32
    }
    {
        bbe_ls8x16s.iv1,a3,32
        bbe_mula8x18v2,v3,v1,v0
        nop
    }
    {
        bbe_ls8x16s.iv1,a3,64
        bbe_mula8x18v4,v5,v1,v0
        nop
    }
    {
        bbe_ls8x16s.iv1,a3,96
        bbe_mula8x18v6,v7,v1,v0
        nop
    }
.LBB28_mm_q15:
```

Here we see a PACKQ form of the multiply-accumulate, and the use of loads of scalars to replicate values appropriately.

In all these cases that used VEC_SIZE and ARRAY_SIZE, the VEC_SIZE used was 1024 and ARRAY_SIZE was 8.

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 ConnX BBE16 offers 16-bit matrix multiplication (via its 16 18x18 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 (not in bbe16_vectorize_types):

```
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()
{
    i;
    xb_cq9_30 red_0 = test_global_red_0;
    (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:

```
     512       5ffc12b0     { bbe_lv8x16s.iu v0, a2, 16; bbe_mula8x18j
v2, v3, v1, v0; bbe_lv8x16s.iu v1, a3, 16 }
```

Note the automatic inference of a complex conjugate multiply BBE_MULA8X18J from the construct in C X[i] * ~X[i], where X is defined as a complex Q15 type.

Another even more intriguing example puts both a complex multiply and a complex conjugate multiply into one loop (also not in the bbe16_examples.xws):

```
xb_cq15 test_arr_0[VEC_SIZE], test_arr_1[VEC_SIZE]
       ,  test_arr_2[VEC_SIZE],  test_arr_3[VEC_SIZE];

void test_MUL_MULJ_xb_cq15()
{
  int i;
  for (i = 0; i < VEC_SIZE; i++) {
     test_arr_0[i] = (test_arr_1[i] * ~test_arr_2[i]);
     test_arr_3[i] = (test_arr_1[i] * test_arr_2[i]);
  }
}
```

What results from this is the disassembly for the inner loop:

```
     128       5ffc1318     { bbe_lv8x16s.i.n v0, a2, 16;
bbe_mul8x18jcpackq v4, v3, v3, v0; bbe_lv8x16s.i.n v1, a3, 16 }
```

```
     253       5ffc1320     { bbe_lv8x16s.iu v0, a2, 32;
bbe_mul8x18jcpackq v2, v1, v1, v0; bbe_sv8x16s.i v2, a5, 16 }
```

Note the automated inference of a double-barrelled bbe_mul18x18jcpackq instruction, which does a complex and complex conjugate multiply for Q15 data at the same time.

3.9 Manual Vectorization

Of course, even the best compiler cannot automatically vectorize all code and all loop nests even if all the guidelines have been followed. In this case, the next step is to move from scalar BBE16 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, shorts by xb_vec8x16 (memory variables) or xb_vec8X20 (vector register variables). The compiler deals with ConnX BBE16 vector data types as it does with any other type, except that non-C-standard types can not be passed as arguments to C functions.

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_vec8x16 *va = (xb_vec8x16*)a;
     const xb_vec8x16 *vb = (xb_vec8x16*)b;
  // Assume no pointer aliasing
     xb_vec8x16 * __restrict vc = (xb_vec8x16*)c;
  // Change loop count to work on 8 elements at a time
     for (i = 0; i < len/8; i += 1) {
          vc[i] = va[i] + vb[i];
     }
}
```

Here we see the loop count divided by eight and the use of xb_vec8x16 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 eight. If it is not, then the programmer needs to write extra code for the more general situation. However, if the data is arranged to al-

ways 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 eight in the case above) is also often easy to do.

3.10 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_vec8x16 *pa = ((xb_vec8x16 *) a);
    // Declare sum as 8 element vector and initialize to zero
    xb_vec8x20 sum = 0;
    xb_vec8x20 avec;
    for (i = 0; i < n; i += 8) {
     sum += (pa++);
   }
   // Add vector of 8 sums and return short result
  return BBE_RADD8X20(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_RADD8X20.
- 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 ConnX BBE16 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 Appendix A.
- Some functions such as the FFT operations discussed in Chapter 4 and the multi mode multiply discussed in Chapter 5 are too complex to ever be inferred. Programming with these complex instructions requires intrinsic-based programming. Refer to the examples in the bbe16_examples.xws file for several examples of doing this, and consult Chapter 4 and Chapter 5.
- 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.

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

The ConnX BBE16 DSP Engine 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 BBE16 configuration in order to access the two local data RAMs. For example, the two default BBE16 templates described in Chapter 7, Implementation Methodology, the XRC_B16LP and XRC_B16PM configuration templates both have the CBox option enabled.

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 modelled 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 modelling when studying the code performance.

If you are using the standard set of LSPs (Linker Support Packages) provided with your BBE16 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, Cadence 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 16-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 modelling; 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(16),section(".dram1.data")));

complex b[ARRAY_SIZE][ARRAY_SIZE][NSAMPLES] __attribute__
((aligned(16),section(".dram0.data")));

xb_vec4xc16 c_auto_opt[ARRAY_SIZE][ARRAY_SIZE][NSAMPLES/4]
__attribute__ ((aligned(16),section(".dram0.data")));

void mm_auto_opt_4x4_stream_complex (xb_vec4xc16  (*__restrict
a)[ARRAY_SIZE][NSAMPLES/4],
          xb_vec4xc16  (*__restrict b)[ARRAY_SIZE][NSAMPLES/4],
        xb_vec4xc16  (* __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_vec4xc16 (*) [ARRAY_SIZE][NSAMPLES/4])a,
               (xb_vec4xc16 (*) [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 modelling enabled in the ISS. The combination of the ymemory pragma and the mcbox compiler directive produces better results than if only one was used.

3.12 Other Compiler Switches

The following two other compiler switches are important:

- -mcoproc, as 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.
- Optimization level 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.

3.13 TI C6x Intrinsic Porting Assistance Library

Cadence provides the following include header file:

```
<install_path>/XtDevTools/install/tools/RC-2010.2-
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 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 ConnX BBE16; to optimize the code further, you need to manually modify it using either BBE16 data types that the compiler can vectorize and infer from, and/or BBE16 intrinsics. 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 BBE16. As it handles most intrinsics, the code should, with little manual effort, compile and execute successfully on BBE16.
- 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 BBE16 data types and compiler automatic vectorization, or BBE16 intrinsics, to maximize application performance. If you have access to pre-optimized BBE16 libraries, these could be substituted 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.

3.13.1 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

```
#include <xtensa/tie/xt_bbe16.h>
#include "c6x-compat.h"
#define VEC_SIZE 128
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.

When compiled at –O2 and run on a BBE16_LP configuration without memory modelling, test_mpy takes 572 cycles when profiled.

Partial disassembly is:

Cycle Count	Instruction Address	Instruction
17	600001d9	l16si a3, a2, 0
16	600001dc	l16si a5, a4, 0
16	600001df	l16si a7, a2, 4
16	600001e2	l16si a8, a4, 4
16	600001e5	mul16s a3, a5, a3
16	600001e8	l16si a5, a2, 8
16	600001eb	l16si a9, a4, 8
16	600001ee	mul16s a7, a8, a7
16	600001f1	s32i.n a7, a6, 4

However, for a few TI instructions, the compiler can do better with BBE16 vectorization and automatic inference of intrinsics. When we compile test_mpy at O3 SIMD (-g -LNO:simd -O3) the profile now takes 79 cycles.

Partial disassembly is:

```
1    600001fd     loopgtz a5, 60000220 <test_mpy+0x50>
                          test_mpy+0x30
16   60000200     { bbe_lv4x32s.iu v0, a2, 32; bbe_mul8x18 v4, v5, v4,
v0; bbe_lv4x32s.iu v1, a3, 32 }
                          test_mpy+0x38
     15   60000208     { bbe_lv4x32s.i.n v2, a2, 16; nop;
bbe_lv4x32s.i.n v3, a3, 16 }
     15   60000210     { bbe_sv4x32s.iu v5, a4, 32; nop; bbe_packs8x40
v0, v2, v0 }
                          test_mpy+0x48
     15   60000218     { bbe_sv4x32s.i v4, a4, 16; nop; bbe_packs8x40
v4, v3, v1 }
                          test_mpy+0x50
     1    60000220     { nop; bbe_mul8x18 v2, v3, v4, v0; nop }
```

Note the compiler automatically uses 8-way BBE16 multiplies, 4-way loads (for ints which are cast to shorts), packs (to convert 40-bit results to shorts, 4-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]);
     }
}
```

At O3, the SIMD profile takes 2,317 cycles. One approach is to do the unpacking and packing in separate loops and then transform add_2 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. At O3 SIMD, profiling gives calc 162 cycles and merge 700 cycles.

The calc disassembly vectorizes:

```
 16 60000468  { bbe_lv4x32s.i v2, a2, 48; bbe_add8x20 v5, v4, v2;
                    bbe_lv4x32s.iu v0, a2, 32 }
                              test_add2_transform_calc+0x74
 15 60000470  { bbe_lv4x32s.iu v3, a3, 32; nop; bbe_srai4x40 v1, v0, 16 }
```

```
15 60000478 { bbe_sv8x16s.iu v4, a7, 16; nop; bbe_packs8x40 v0, v2, v0 }
15 60000480 { bbe_sv8x16s.iu v5, a9, 16; nop; bbe_srai4x40 v2, v2, 16 }
15 60000488 { bbe_lv4x32s.i v5, a3, 16; nop; bbe_srai4x40 v4, v3, 16 }
15 60000490 { bbe_sv8x16s.iu v0, a4, 16; nop; bbe_packs8x40 v3, v5, v3 }
```

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. When we tried this by creating an _add2_sub2_calc routine, followed by a merge routine, which does the pack back into the TI merged form, the calc routine took 211 cycles, and the _add2_sub2_merge routine took 700 cycles.

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. ConnX BBE16 does not saturate on most operations, nor does it have an overflow register. Instead, it uses 40/20-bit vector registers with guard bits; saturates occur on storing from 40- or 20-bit vector registers to 32 or 16 bits 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. FFT Instructions

4.1 FFT Acceleration

The ConnX BBE16 DSP Engine includes an optimized instruction for FFT operations. This instruction contains four different sub-operations that can be set in order to perform four different FFT operations. These operations support decimation in frequency (DIF) Radix-4 butterfly operations, including support for combining Radix-2 and Radix-4 operations.

The four different FFT operations are selected based on the setting of a control switch called FFTOP. In addition, there are several other control and feedback states contained in a special register. The FFT instruction is called BBE_FFT8X18CPACKQ.

The four different operations that the FFT instruction supports are a standard Radix-4 operation BFL (also known as BFLT); a special Radix-4 operation for the last pass, BFLR4 (also known as BFLTR4); a combined Radix-2 and Radix-4 operation BFLR2R4 (also known as BFLTR2R4); and an ADD3MUL operation that is used for odd-radix (e.g. Radix-3 and Radix-5) support.

The FFT instruction also supports optional normalization and dynamic range computation. In addition there are special bit-reversing store instructions to return data in natural order, which means there is no penalty for using natural order.

To support programming with the FFT instructions and the various operational modes and settings, an include file “xt_bbe16_fft.h” is provided with the ConnX BBE16 include files contained in the directory:

```
<xtensa_root>/xtensa-elf/arch/include/xtensa/tie
```

4.2 FFT Control Register

The FFT instruction BBE_FFT8X18CPACKQ is controlled by a 32-bit control register FFTCTRL. This is divided into five fields that act as controls or monitors for the FFT instruction BBE_FFT8X18CPACKQ. These fields are separated in the control register by one or two zero bits. In addition, instructions exist that allow these to be accessed as if they were completely separate state registers: BBE_MOVAFM<field> will move the contents of the field to an AR register; BBE_MOVATO<field> will move relevant bits of the contents of an AR register into the field.

- A 3-bit control field MODE[2:0] that controls normalization and shifting for the FFT instruction. Values 0 to 7 are defined, as follows:

Table 4–36. MODE[2:0] Register

MODE	Function
0	adds 0x04000 and shifts right by 15 bits
1	adds 0x08000 and shifts right by 16 bits
2	adds 0x10000 and shifts right by 17 bits
3	adds 0x20000 and shifts right by 18 bits
4	adds 0x40000 and shifts right by 19 bits
5,6,7	same as MODE 4 above

- A 3-bit monitor field RANGE[2:0] that contains the maximum number of non-sign bits in the range [19:12] across all FFT results. This can be used for adjusting FFT results.
- A 2-bit control field FFTOP[1:0] which controls the particular FFT operation being executed by the instruction BBE_FFT8X18CPACKQ. There are four different possible operations, as follows:

Table 4–37. FFTOP[1:0] Register

FFTOP	Name	Function
0	BFL	Standard Radix-4 DIF FFT butterfly. Also known as BFLT.
1	BFLR4	Simple Radix-4 butterfly for last pass. Also known as BFLTR4.
2	BFLR2R4	Combined Radix-2, then simple Radix-4 butterfly. Also known as BFLTR2R4.
3	ADD3MUL	Add-multiply ((A + B + C) * T) for odd-radix FFT

- A 4-bit control field BITREV_POS[3:0] that is used in FFT bit-reversed addressing to increment the offset. The bit reverse offset is incremented by 0x1000 right-shifted by the value in BITREV_POS. Thus, if BITREV_POS is 0, the increment is 0x1000; if it is 1, 0x0800, and so on.
- A 13-bit control field BITREV_OFF[12:0] that is used in FFT bit-reversed addressing.

4.3 General FFT Structure

A general algorithm for implementing FFT operations using the ConnX BBE16 butterfly instructions is as follows:

- For size $N=4^P$ FFT, use $P=\log_4(N)$ phases of Radix-4 butterflies.

- For size $N=2^R$ FFT, use floor($\log_4(N)$)-1 Radix-4 phases, followed by one final combined Radix-2/Radix-4 phase.
  (Where "floor" means the largest integer less than or equal to $\log_4(N)$.)
- All FFT phases load from source array and write to destination array.
- Alternate between two data buffers.
- After computation, use special stores to reorder output data from bit-reversed indexing back to natural order.

Figure 4–10 shows a 32 point FFT (N = 32). Thus we must use one Radix-4 phase, (lower($\log_4(32)$) -1 = lower(2.5)-1 = 1), followed by one mixed Radix-2/Radix-4 phase.

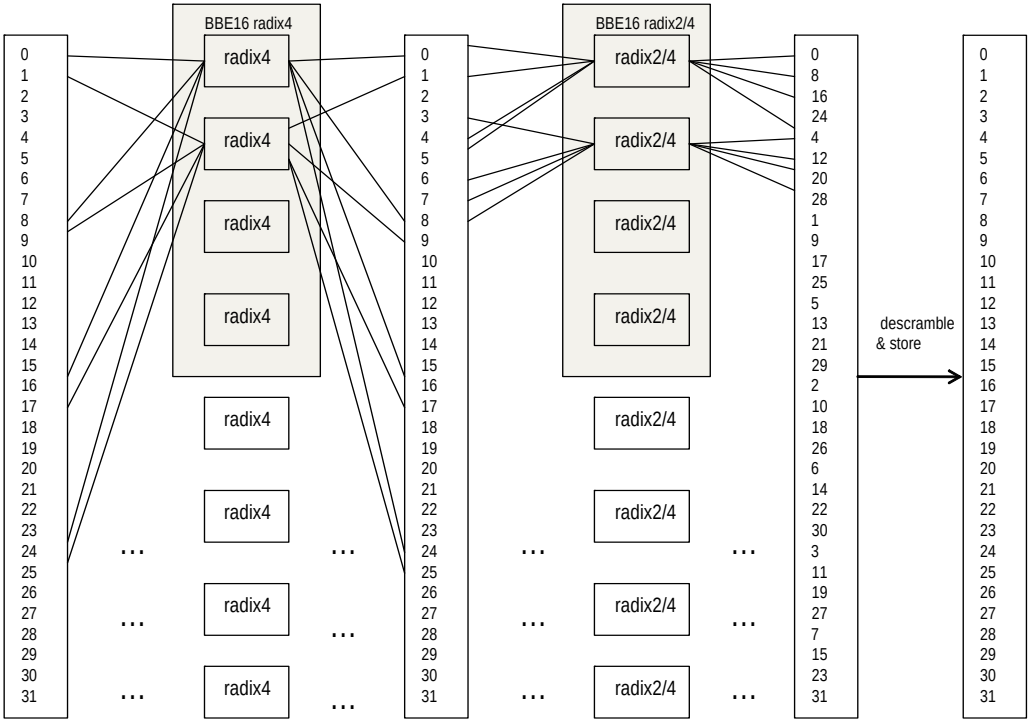


Figure 4–10. Mixed Radix (4/2) Parallel 32 Point FFT

4.4 Summary of ConnX BBE16 FFT Operations and Instructions

Table 4–38. ConnX BBE16 FFT Operations

Instruction/Operation	Definition
BFL	Standard Radix-4 DIF FFT butterfly. Also known as BFLT
BFLR4	Simple Radix-4 butterfly for last pass. Also known as BFLTR4.
BFLR2R4	Combined Radix-2 then simple Radix-4 butterfly. Also known as BFLTR2R4.
ADD3MUL	Add-multiply ((A + B + C) * T) for odd-radix FFT
BBE_SR8X16S.BR	Store selected elements of vectors with bit-reversed address
BBE_SR8X16S.BRU	Store selected elements of vectors with bit-reversed address and update
BBE_SR8X16S.XU	Store selected elements of vectors (indexed with update of address register)

4.5 BFL Operation - Radix-4 DIF Butterfly

BFL is a Radix-4 DIF (Decimation In Frequency) butterfly instruction. The concept is illustrated in Figure 4–11.

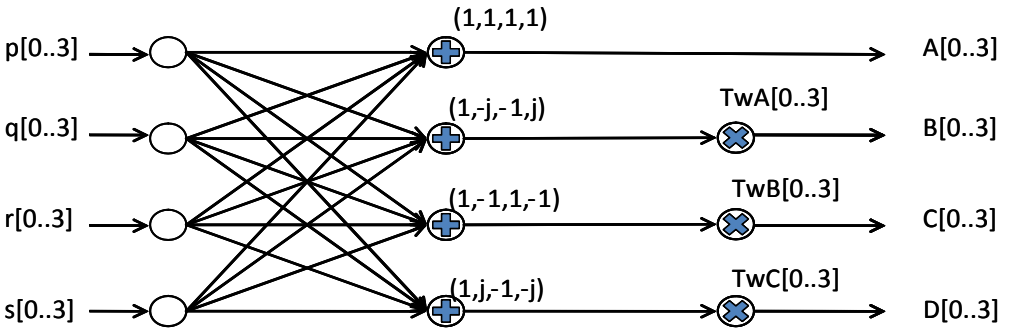


Figure 4–11. Radix-4 FFT butterfly

In this figure {Twa, Twb, Twc} are twiddle factors.(Complex roots of unity).

BFL is the main FFT butterfly operation. It is invoked by setting the FFTOP and then invoking the main FFT operation whose prototype is:

```
BBE_FFT4XCQ4_15CPACKQ {xb_vec4xcq4_15 vp, xb_vec4xcq4_15 vq,
xb_vec4xcq4_15 vr, xb_vec4xcq4_15 vs, imm2 sel}
```

BFL computes four complex SIMD results per instruction, using four complex data inputs per vector register, four complex coefficients per twiddle register, and producing 4x20-bit complex output values.

The immediate input sel chooses one of four Radix-4 butterfly outputs as shown in Table 4–39.

Table 4–39. Radix-4 Butterfly Outputs

SEL	Function
0	(p + q + r + s)
1	(p - j * q - r + j * s) * BBE_TWIDDLEA
2	(p - q + r - s) * BBE_TWIDDLEB
3	(p + j * q - r - j * s) * BBE_TWIDDLEC

It takes four operations to compute 16 results from 16 inputs as each operation produces four outputs. The sel immediate chooses which output is produced as follows:

Table 4–40. Sel Outputs

Result	SEL	Intrinsic
A[0..3]	0	BBE_FFT4XCQ4_15CPACKQ(p,q,r,s,0)
B[0..3]	1	BBE_FFT4XCQ4_15CPACKQ(p,q,r,s,1)
C[0..3]	2	BBE_FFT4XCQ4_15CPACKQ(p,q,r,s,2)
D[0..3]	3	BBE_FFT4XCQ4_15CPACKQ(p,q,r,s,3)

To efficiently use the operation, load all 16 complex data values before doing the butterfly. Each input vector register holds four complex inputs (e.g., p[0..3]). Inputs are complex fixed-point Q4.15 pairs. The multiplies are 18x18, and each operation computes four complex pairs of Q4.15 results. We round and right-shift each 36-bit multiplication result. Additionally, there is an optional further right shift for normalization controlled by the MODE special register field described earlier. An xb_vec4xcq4_15 complex vector type is returned with four complex Q4.15 results. This can then be stored saturating to memory as four complex Q15 pairs.

4.6 FFT Normalization

FFT normalization of results is controlled by setting the MODE special register.

- If set to 0, round and shift by 15 bits to extract a Q4.15 result.
- If set to 1, round and shift by 16 bits to divide by 2.
- If set to 2, round and shift by 17 bits to divide by 4.
- If set to 3, round and shift by 18 bits to divide by 8
- If set to 4, round and shift by 19 bits to divide by 16
- If set to 5,6, or 7, perform the same round and shift as MODE 4

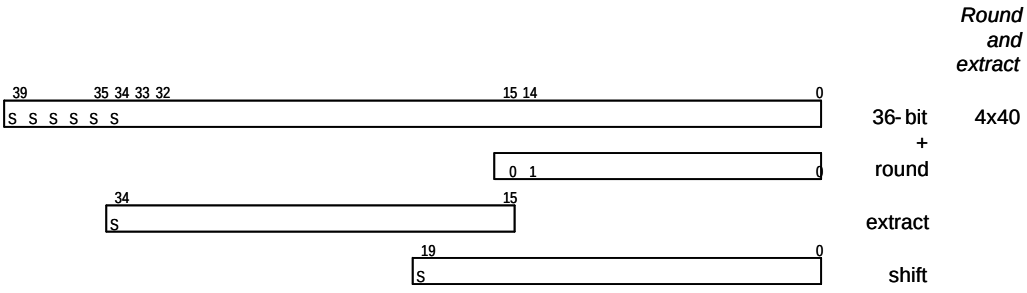


Figure 4–12. FFT Normalization

4.7 FFT Dynamic Range

To analyze and use the dynamic range capabilities of the FFT operations, set a constant MODE shift for every FFT stage. In addition, before the operations for each phase, set the RANGE special register to 0. After the operations for the phase, the RANGE special register will contain the dynamic range of the results. This can then be used before the next phase to set the MODE shift for that next phase.

RANGE represents the maximum significant bits in any result. A range of 0 means the results are in the range [- 2^{12} .. 2^{12} -1]. A range of 6 is the range [- 2^{18} .. 2^{18} -1] and a range of 7 is the range [- 2^{19} .. 2^{19} -1]. You can use the instruction BBE_MOVAFM.RANGE to move the value of RANGE to an AR register or normal variable.

4.8 Inner Loop C Code for FFT

Below is typical inner loop C code for the FFT operation, where db0 is the base address of the buffer holding input data for the pass, and db1 is the base address of the buffer to hold output data for the pass:

```
/* radix-4 processing */
      BBE_MOVATO_MODE(normalize); // Optional normalize shift on each
pass
      BBE_MOVATO_RANGE(0); // Reset dynamic range monitor
      BBE_MOVATO_FFTOP(BBE_FFT_BFL);        // radix-4 butterfly mode

//n_vec = 1024, quarter_n_vec = 256, repcnt = 1
      loop_incr = -4*(3*quarter_n_vec-4);
      p = db0 - loop_incr/sizeof(xb_vec4xcq15);  // pre-decrement to
compensate for pre-increment addressing
```

```
      q = db1 - loop_incr/sizeof(xb_vec4xcq15); // pre-decrement to
compensate for pre-increment addressing

      for (li = 0; li < quarter_n_vec/4; li++) {

        BBE_LTA4XCQ15_IU(tw,sizeof(xb_vec4xcq15));
        BBE_LTB4XCQ15_IU(tw,sizeof(xb_vec4xcq15));
        BBE_LTC4XCQ15_IU(tw,sizeof(xb_vec4xcq15));

        BBE_LV4XCQ15_XU(d0,p,loop_incr);
        BBE_LV4XCQ15_XU(d1,p,step);
        BBE_LV4XCQ15_XU(d2,p,step);
        BBE_LV4XCQ15_XU(d3,p,step);

        r0 = BBE_FFT4XCQ4_15CPACKQ(d0,d1,d2,d3,0);
        r1 = BBE_FFT4XCQ4_15CPACKQ(d0,d1,d2,d3,1);
        r2 = BBE_FFT4XCQ4_15CPACKQ(d0,d1,d2,d3,2);
        r3 = BBE_FFT4XCQ4_15CPACKQ(d0,d1,d2,d3,3);

        *(q+=loop_incr/sizeof(xb_vec4xcq15)) = r0;
        *(q+=step/sizeof(xb_vec4xcq15)) = r2;
        *(q+=step/sizeof(xb_vec4xcq15)) = r1;
        *(q+=step/sizeof(xb_vec4xcq15)) = r3;
      }
      repcnt <<= 2;
      quarter_n_vec >>= 2;
      step >>=2;
```

4.9 BFLR4 Operation

The operation BFLR4 performs a Radix-4 butterfly for the last pass of the DIF FFT, when the four inputs are adjacent in memory and represented in a single vector of four complex pairs. This applies to a final phase of a size 4^P FFT. It does not use the twiddle values. The operation is set using the FFTOP special register field.

The input vector is an 8x20 bit vector with four complex numbers, real and imaginary stored adjacent. The computation is:

```
g0 = (p0 +     p1 + p2 +     p3)
g1 = (p0 - j * p1 - p2 + j * p3)
g2 = (p0 -     p1 + p2 -     p3)
g3 = (p0 + j * p1 - p2 - j * p3)

h0 = (q0 +     q1 + q2 +     q3)
h1 = (q0 - j * q1 - q2 + j * q3)
h2 = (q0 -     q1 + q2 -     q3)
h3 = (q0 + j * q1 - q2 - j * q3)
```

The results are rounded and normalized using the optional MODE shift. The immediate sel is used to pick pairs of complex results:

- SEL= 0: R = {h1,h0,g1,g0}
- SEL= 1: R = {h3,h2,g3,g2}

4.10 BFLR2R4 Operation

This operation is used for the last pass of a mixed Radix-2/Radix-4 FFT where the size is not a power of 4. First it does a Radix-2 Butterfly, and then a simple Radix-4. It retains high-precision intermediate values and rounds and normalizes with the optional MODE shift.

It operates on complex inputs selected from four vector registers: p, q, r and s. The twiddle state registers are not used.

The result is a vector rx of 4 complex pairs (eight 20-bit elements)

The computation takes place in three stages.

1. The four corresponding complex pairs of p and q, and of r and s are added (SEL[0] = 0) or subtracted (SEL[0] = 1) as part of the Radix-2 calculation:

```
SEL[0]= 0:
 k = p + q;
 l = r + s;

SEL[0] = 1:
 k = p - q;
 l = r - s;
```

2. These sums and differences are multiplied by built-in twiddle values, with the following complex values T = {T.imag,T.real}:

```
SEL[0] = 0:
 t0 = {0x00000, 0x08000} = (0i + 1)
 t1 = {0x00000, 0x08000} = (0i + 1)
 t2 = {0x00000, 0x08000} = (0i + 1)
 t3 = {0x00000, 0x08000} = (0i + 1)

SEL[0] = 1:
 t0 = {0x00000, 0x08000} = (0i + 1)
 t1 = {0xfa57e, 0x05a82} = (sin(-pi/4)i + sin(pi/4))
 t2 = {0xf8000, 0x00000} = (-i + 0)
 t3 = {0xfa57e, 0xfa57e} = (sin(-pi/4)i + sin(-pi/4))
```

so the following complex multiplies are performed:

```
m0 = k0 * t0
 m1 = k1 * t1
```

```
 m2 = k2 * t2
 m3 = k3 * t3

 and
n0 = l0 * t0
 n1 = l1 * t1
 n2 = l2 * t2
 n3 = l3 * t3
```

Only the least significant 18 bits of each element are multiplied. Any higher-order bits are ignored.

3. The final Radix-4 operation is combined with the add and subtraction needed to complete the complex multiplications above, but only four the eight possible complex pairs are computed, depending on SEL[1]:

```
SEL[1] = 0:
 w0 = m0 + m1 + m2 + m3;
 x0 = m0 - j*m1 - m2 + j*m3;
 w1 = n0 + n1 + n2 + n3;
 x1 = n0 - j*n1 - n2 + j*n3;

SEL[1] = 1:
 w0 = m0 - m1 + m2 - m3;
 x0 = m0 + j*m1 - m2 - j*m3;
 w1 = n0 - n1 + n2 - n3;
 x1 = n0 + j*n1 - n2 - j*n3;
```

Up through this point, all adds are performed to full precision, so that the result components have approximately 38-bits of range for each component. The instruction converts these values to 20-bit fixed-point Q4.15 elements, rounding and normalizing with the optional MODE shift.

There is no saturation of results. A vector of four complex pairs, each consisting of two 20-bit elements is returned as the result:

```
rx = {x1.imag,x1.real, w1.imag,w1.real, x0.imag,x0.real, w0.imag,w0.real}
```

4.11 FFT_ADD3MUL Operation

FFT_ADD3MUL (FFTOP = 3) is a 3-input add and multiply operation for other butterfly computations such as odd-radix.

This operation provides a general method to add the four complex value of three input vectors, and multiply each complex value by the corresponding complex values of a fourth input vector. It is commonly used in odd-radix FFT butterfly kernels. It operates on complex inputs of four vector registers: p, q, r and s, where each vector contains four complex pairs, with each real and imaginary element (eight 20-bit elements, real and

imaginary interleaves, real in less significant position). The result is a vector of four complex pairs of the same format written to vector register rx. The twiddle registers and select value are ignored.

The corresponding elements of p, q, and r are added together:

```
y0 = p0 + q0 + r0
 y1 = p1 + q1 + r1
 y2 = p2 + q2 + r2
 y3 = p3 + q3 + r3
```

A complex multiply is performance for each complex sum by the corresponding complex value in the vectors:

```
z0 = y0 * s0
 z1 = y1 * s1
 z2 = y2 * s2
 z3 = y3 * s3
```

Following are Radix-3 and Radix-5 examples in pseudo-code form.

Note: The following is a suggested method for implementing an FFT using BBE16 instructions; it does not necessarily match the implementation approach used in BBE16 libraries and examples.

For Radix-3 implementation we need to perform:

```
load vector [4 complex pairs] A
load vector [4 complex pairs] B
load vector [4 complex pairs] C
B1 = B * t1
B2 = B *(conjugate(t1))
C1 = C * t1
C2 = C* (conjugate(t1))
V = ADD3MUL(A,B,C,tv)
W = ADD3MUL(A,B1,C2,tw)
X = ADD3MUL(A,B2,C1,tx)
store vector [4 complex pairs] V
store vector [4 complex pairs] W
store vector [4 complex pairs] X
```

where t1 is special twiddle exp(-j*2*pi/3) and the tv, tw, tx are appropriate twiddle factors based on the size of FFT.

For Radix-5 implementation we need:

```
load vector [4 complex pairs] A
load vector [4 complex pairs] B
load vector [4 complex pairs] C
load vector [4 complex pairs] D
load vector [4 complex pairs] E

B1=B * t1
B4= B *(conjugate(t1))
B2=B * t2
B3= B *(conjugate(t2))
C1=C * t1
C4= C* (conjugate(t1))
C2=C * t2
C3= C*(conjugate(t2))
D1=D * t1
D4= D*(conjugate(t1))
D2=D * t2
D3= D*(conjugate(t2))
E1=E * t1
E4= e* (conjugate(t1))
E2=E * t2
E3= E*(conjugate(t2))

V = ADD3MUL(A,ADD(B,C),ADD(D,E),tv)
W = ADD3MUL(A,ADD(B1,C2),ADD(D3,E4),tw)
X = ADD3MUL(A,ADD(B2,C4),ADD(D1,E3),tx)
Y = ADD3MUL(A,ADD(B3,C1),ADD(D4,E2),ty)
Z = ADD3MUL(A,ADD(B4,C3),ADD(D2,E1),tz)

store vector [4 complex pairs] V
store vector [4 complex pairs] W
store vector [4 complex pairs] X
store vector [4 complex pairs] Y
store vector [4 complex pairs] Z
```

where t1=exp(-j*2*pi/5), t2= exp(-j*4*pi/5), and the tv, tw, tx, ty, tz are appropriate twiddle factors based on the size of FFT.

Up through this point, all adds are performed to full precision, so that the result components have approximately 37-bits of range for each component. The instruction converts these values to 20-bit fixed-point Q4.15 elements. These component results are rounded to the nearest by adding a 1 at the shift position and shifting right according to the shift amount indicated in bits [2:0] of the MODE register.

MODE[2:0] Normalization

- 0: Add rounding value 0x04000 and shift right 15-bits
- 1: Add rounding value 0x08000 and shift right 16-bits
- 2: Add rounding value 0x10000 and shift right 17-bits
- 3: Add rounding value 0x20000 and shift right 18-bits
- 4: Add rounding value 0x40000 and shift right 19 bits
- 5-7: same round and shift as MODE 4

The purpose of the variable shift is to allow FFT software to reduce the occurrence of overflow for intermediate FFT results. Depending on the shift amount, the resulting values can have up to 22 bits of significance, representing a numerical range (for the 15 bit shift) of [-48.0,48.0) - sign bit, six bits of integer, and 15 bits of fraction.

All shift amounts greater than 15 in all passes must be compensated for in the final result to provide results in the expected Q15 range. Final results can be shifted left by the sum of the number of shift amounts greater than 15 across all passes to accomplish this.

There is no saturation of results. A vector of four complex pairs, each consisting of two 20-bit elements is returned as the result:

```
rx = {z3.imag,z3.real, z2.imag,z2.real, z1.imag,z1.real, z0.imag,z0.real}
```

4.12 ConnX BBE16 FFT Instruction Usage

In line with the discussion in the previous section, the basic algorithm, using the appropriate instructions based on the size of FFT, is:

- Do upper($\log_4$(N)) phases in total, where "upper" is the lowest integer greater than or equal to $\log_4$(N).
- Use BFL (full Radix-4 Butterfly) for the first (upper($\log_4$(N)) - 1) phases.
- Use BFR4 (simple Radix-4) for the final phase of a size 4^P FFT - e.g. 1024, 4096.
- Use BFADDR4 and BFR2R4 in combination for Radix-4/Radix-2 butterflies for the final phase of a size 2^R FFT - e.g., 512, 2048, 8192.

Table 4–41 illustrates the instructions used for different sizes of FFT.

Table 4–41. FFT Instructions

FFT Size	1	2	3	4	5	6
fft512	BFLT	BFLT	BFLT	BFLTR2R4		
fft1024	BFLT	BFLT	BFLT	BFLT	BFLTR4	
fft2048	BFLT	BFLT	BFLT	BFLT	BFLTR2R4	
fft4096	BFLT	BFLT	BFLT	BFLT	BFLT	BFLTR4

For example, for an 2048 sized FFT:

```
// Phase 1, 2, 3, 4
BBE_MOVATO_FFTOP(BBE_FFT_BFL); // radix-4 butterfly mode

// Phase 5
BBE_MOVATO_FFTOP(BBE_FFT_BFLR2R4);        // trivial r4 butterfly mode
```

4.13 Radix-4 DIF FFT Result Order

For a power of four FFT, we use $\log_4(N)$ phases for an FFT of size $N=4^P$. The results are in bit-reversed order as illustrated in Figure 4–13.

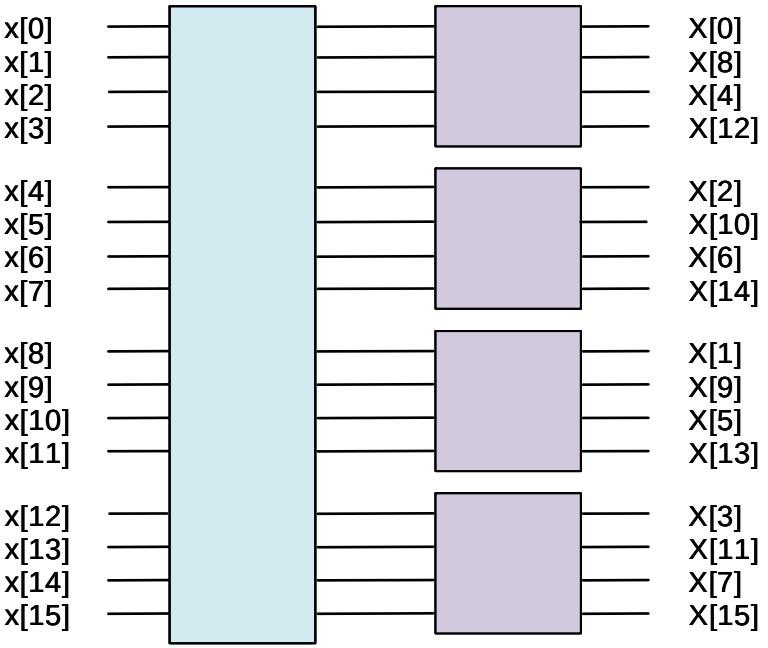


Figure 4–13. Radix-4 Bit-Reversed Order Result

Table 4–42. FFT Bit-Reversed Order

n	Bin(n)	Bitrev(n)	N
0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1110	12
4	0100	0010	2

Table 4–42. FFT Bit-Reversed Order (continued)

n	Bin(n)	Bitrev(n)	N
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	11
15	1111	1111	15

These can be converted back to normal ordering using two special bit-reversed store instructions as discussed next.

4.14 Bit-Reversed Addressing

Two special store instructions, BBE_SR8X16S.BR and BBE_SR8X16S.BRU, implement bit-reversed addressing for use in properly ordering results of the FFT. Like the store instruction BBE_SR8X16S.XU, these extract four complex 20-bit elements from two vector registers depending on a sel immediate.

The prototype for BBE_SR8X16S.BR is:

```
BBE_SR8X16S_BR {xb_vec4xcq4_15 vr, xb_vec4xcq4_15 vu, xb_vec4xcq15*
ar, int32 ir, imm2 sel}
sel==0:
 {vr.im2,vr.re2, vu.im2,vu.re2, vr.im0,vr.re0, vu.im0,vu.re0}
sel==1:
 {vr.im3,vr.re3, vu.im3,vu.re3, vr.im1,vr.re1, vu.im1,vu.re1}
```

It saturates to 16-bit elements and stores 128 bits to a bit-reversed address. This is calculated as

```
addr[31:0] <=  ar[31:0] + ir[31:0] + (Boffset << 4)
```

Where `Boffset` is a vector (16-byte) offset. This instruction does not update the base register `ar`. `Boffset` comes from the FFT control state register as shown in Table 4–43.

Table 4–43. FFT Control State Registers

Field	Alias	Use
BITREV_OFF[12:0]	Boffset	Vector (16-byte) offset used for bit-reverse address generation.
BITREV_POS[3:0]	Bpos	Bit position for bit-reversed increment.

BBE_SR8X16S.BRU is very similar, but it increments the vector offset using Bpos. Its prototype is

```
BBE_SR8X16S_BRU {xb_vec4xcq4_15 vr, xb_vec4xcq4_15 vu, xb_vec4xcq15* ar,
  int32 ir, imm2 sel}
```

BBE_SR8X16S_BRU operates identically to BBE_SR8X16S.BR, except it increments the vector offset:

```
Boffset= bit_rev(  bit_rev(Boffset) + (0x1000 >> Bpos) )
```

The following table shows the choices for Bpos:

Table 4–44. B_pos FFT Sizes

B_pos	Stride	FFT Size
0	1	32
1	2	64
2	4	128
3	8	256
4	16	512
5	32	1024
6	64	2048
7	128	4096
8	256	8192
9	512	16384
10	1024	32768
11	2048	65536
12	4096	131072

BBE16 can reorder FFT data back to natural order using these bit-reversing store instructions.

This is used in the final stage of FFT processing for power of two sized FFT.

To use these instructions, before starting the last stage of FFT processing, initialize BITREV_OFF state to zero and set the BITREV_POS state to a constant depending on the size of the FFT, as listed in table 45.

(In addition, header file <xtensa/tie/xt_bbe16_fft.h> defines the BITREV_POS values for different sized FFTs).

For example, before the last stage of processing a 1024 FFT, set:

```
BBE_MOVATO_BITREV_OFF(0);

BBE_MOVATO_BITREV_POS(BBE_BR_POS_1K)
```

Thereafter, you do not need to update the two states anymore, just use the BBE_SR8X16S.BR and BBE_SR8X16S.BRU store operations to reorder vectors.

In order to leave FFT data in bit-reversed order, you do not need to use the bit-reversing store instructions or states. In that case you skip the two settings and simply use BBE_SR8X16S.XU for stores, as described in Section 4.15.

4.15 BBE_SR8X16S.XU: FFT Indexed Store with Update

BBE_SR8X16S.XU is an FFT indexed store with update. The most useful prototype for this special indexed store is

```
BBE_SR4XCQ15_XU {xb_vec4xcq4_15 vr, xb_vec4xcq4_15 vu, xb_vec4xcq15*
ar, int32 ir, imm2 sel}
```

The operation extracts four complex 20-bit elements from two vector registers, depending on the value of **sel**:

- sel==0:

  {vr.im1,vr.re1, vu.im1,vu.re1, vr.im0,vr.re0, vu.im0,vu.re0}

- sel==1:

  {vr.im3,vr.re3, vu.im3,vu.re3, vr.im2,vr.re2, vu.im2,vu.re2}

It clamps (saturates) the output data to 16 bits, and stores the 8x16 bit vector to an address formed by a base+index computation (ar + ir). Finally, it updates the ar address register with the computed address.

5. General Multi-Mode, Multi-Mode Multiply and Multiply-Add Instructions with Extended Precision

ConnX BBE16 contains two instructions that support a very general multiply and multiply-accumulate capability for vectors. These produce extended precision results and support both real and complex elements. They also support a variety of selection mechanisms directly to allow FIR, symmetric FIR and small matrix multiplies to be efficiently computed. These mechanisms also allow mixtures of real and complex data to be used - for example, for FIR with real coefficients or taps and complex data 2x2 and 4x4 matrix multiplies are supported, along with matrix-vector and vector-matrix multiplies of these dimensions.

These two instructions are called BBE_MUL8X18E and BBE_MULA8X18E. They use the 16 multiply resources to perform 16 18x18 multiplies or multiply-accumulates. A 32-bit select operand controls the wide variety of modes, data types, and element selection supported by these instructions.

Multi-way refers to the 16-way parallel multiplies possible. Multi-type refers to support of both real and complex data. Multi-mode refers to the various combinations of FIR, vector, and matrix multiply operations that are possible using these instructions. A select operand is used to control what kind of operational mode is used.

The basic idea behind the instructions is to divide the multipliers into four groups of four, each with flexible selection of the operands. Each group of multipliers is preceded by adders, which allow symmetric filtering operations. All of this is controlled by the 32-bit select operand.

To avoid unnecessary special registers, these operations reuse the twiddle registers. The basic structure of the operations is shown in Figure 12. Note that this figure contains five input registers, each composed of four sets of operand pairs (each pair 40 bits) and two output/accumulator registers, each composed of four sets of 40-bit pairs.

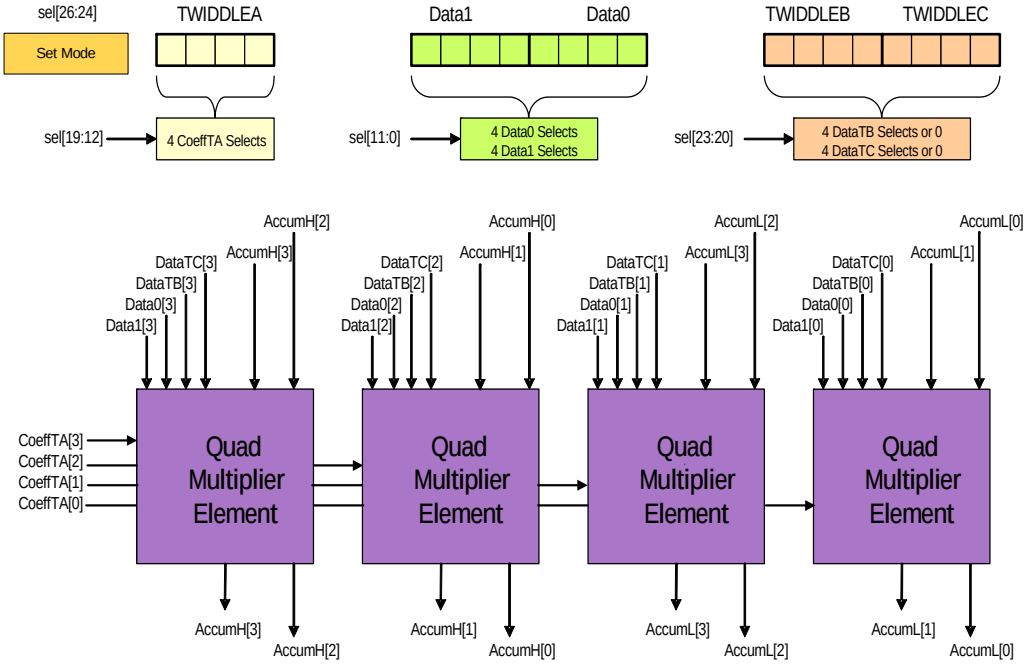


Figure 5–14. Complex FIR and Matrix Multiplier

As this figure shows, Twiddle register A holds coefficients for FIR operations. Twiddle registers B and C hold additional data to be selected. Two input registers hold data to be selected.

The structure of the groups of multipliers (the quad multiplier element) is shown in Figure 5–15.

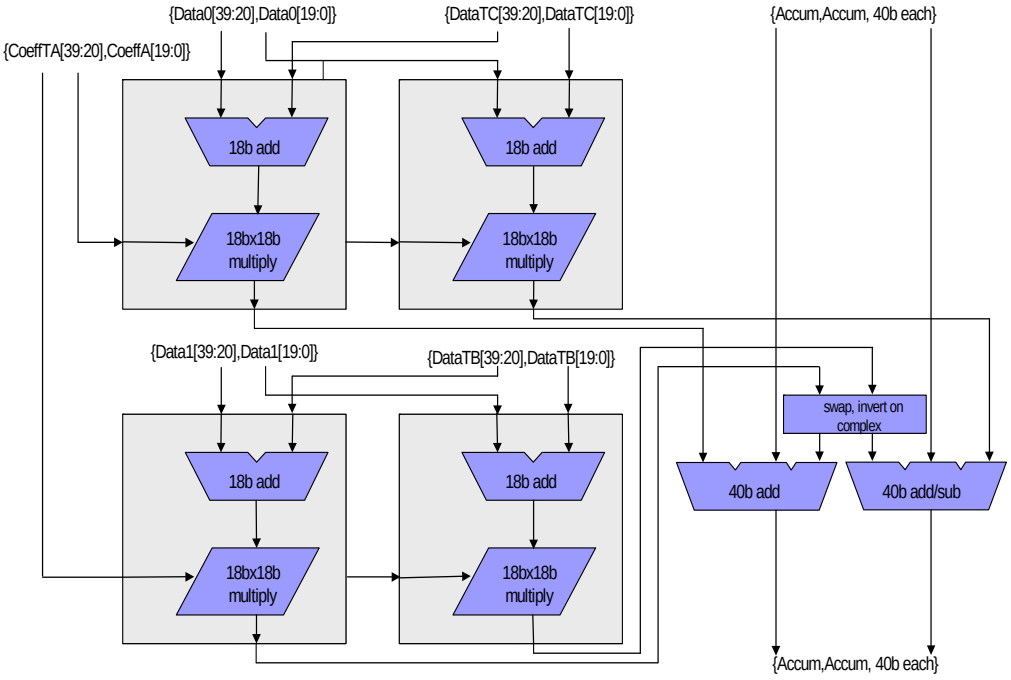


Figure 5–15. Structure of Quad-Multiplier Element

5.1 Details of the Instructions

The BBE_MUL8X18E prototype is as follows:

```
extern void BBE_MUL8X18E(xb_vec4x40 acc_h /*out*/, xb_vec4x40 acc_l
/*out*/, xb_vec8x20 p, xb_vec8x20 q, vsel sel);
```

The BBE_MULA8X18E is almost identical, but the two xb_vec4x40 registers are input-output because they are used to accumulate the results of this multiply with previous values held in the vector registers.

This instruction operates by arranging the selected elements of input FIR and matrix or vector components to be multiplied, creating partial sums that will be accumulated to create final outputs. The instructions operate on input vectors and create output vectors.

The sel operand is a 32-bit select register that contains control information that specifies the specific input and output configurations. There are four basic patterns:

1. One 8x20 vector of coefficients from TWIDDLEA special register
2. One 16x20 vector of data from two input vector registers

3. One 16x20 vector of data for symmetric FIR cases from TWIDDLEB/TWIDDLEC special registers
4. The output is one 8x40 accumulation inout register (mapped to two 4x40 output registers, one for high order results and one for low order results) for BBE_MULA8X18E. These are outputs only for BBE_MUL8X18E and overwrite previously stored results.

The 18-bit components to be multiplied are sign extended from the least significant 18 bits of the 20-bit elements in the operands of inputs p (data_h), q (data_l), and the special registers TWIDDLEA, TWIDDLEB, and TWIDDLEC. Since this is a multi-type instruction, the 16 multiplies can be set up to do 8-way dual real-multiplies, or 4-way complex multiplies. In both cases, the outputs are sign-extended and then written into two 4x40-bit vector registers that can be used as result registers, or re-used as accumulators. These are identified by the operands acc_h and acc_l.

Because these instructions support various modes and use cases, they offer control over input operand selection, output operand destinations, and the special use of stored operands. For example, for symmetric FIR cases, they make use of the TWIDDLEB and TWIDDLEC special registers for additional inputs as well as TWIDDLEA for coefficients, whereas for regular FIR cases and matrix multiplies they only use the TWIDDLEA special register.

5.1.1 32-Bit Select Operand (sel)

The operation of the instructions is controlled by a 32-bit select operand sel. There are 10 different fields in the 32-bit operand that must be set in order to control the operation successfully. Note that when some of the operand fields are set incorrectly (e.g. using reserved values), the operation of the instructions is undefined and the results are undefined. The fields of the select operand are described below. The bit range for each field of the sel operand is designated by [high:low]. These fields are described in logical order of instruction operation, not bit range order.

The first set of four fields select the elements from the data inputs that are to be multiplied with the data in the TWIDDLEA register. These fields also determine the output group position for the results. The selection is not completely arbitrary, since there are 16 input elements in the vector registers p and q. The fields are 3 bits wide, giving 8 different selection possibilities. Pairs of elements are selected at a time. Thus for complex data, each pair will be a single complex element in (imaginary, real) order. For real data, each pair will be two adjacent real data elements.

Since the multiplication operation produces extended precision outputs that are 40 bits wide, 16 data inputs will produce 8 data outputs. There are four output groups, each containing two 40 bit data items. These are organized into the two output registers as follows: acc_l will receive group 0 and 1 outputs, and acc_h will receive groups 2 and 3.

Each of the bit fields in the description is given a symbolic name, as shown in the Selector column in Table 5–45.

Table 5–45. 32-Bit Select Operand Fields

Selector	Output Data Group	Input Data	Twiddle Coeff	Twiddle B Data	Twiddle C Data	Input Data Offset	Twiddle B/C Data Offset for 1st Pair Selected
sel[11:9]	3	0-7 pair position					
sel[8:6]	2	0-7 pair position					
sel[5:3]	1	0-7 pair position					
sel[2:0]	0	0-7 pair position					
sel[19:18]	3		0-3 pair element				
sel[17:16]	2		0-3 pair element				
sel[15:14]	1		0-3 pair position				
sel[13:12]	0		0-3 pair position				
sel[23:20]				7-0 pair position (if 15, ignore) 8-14 reserved	7-0 pair position (if 15, ignore) 8-14 reserved		
sel[26:24]						0,1,2,4,8 element	1,2,4,8 element

Following are the descriptions for each selector:

- sel[11:9]=selDataIn_out3 — Directs output data to the group 3 location (*i.e.*, the highest pair position in acc_h). Input data is selected from eight position pairs of registers vr and vs indexed by the 3 bits. For real data multiplies, the first data pair is selected from these bits, whereas an additional second pair is implicitly selected by adding the value of this field to the offset stored in bits sel[26:24]: sel_mode. For complex data the index simply selects one of eight complex numbers (with real and imaginary parts).
- sel[8:6]=selDataIn_out2 — Directs output data to the group 2 location (next highest pair position in acc_h). Input data is selected from eight position pairs of registers vr and vs indexed by the 3 bits. For real data multiplies, the first data pair is selected from these bits, whereas an additional second pair is implicitly selected by adding the value of this field to the offset stored in bits sel[26:24]: sel_mode. For complex data the index simply selects one of eight complex numbers (with real and imaginary parts).
- sel[5:3]=selDataIn_out1 — Directs output data to the group 1 location (highest pair position in acc_l). Input data is selected from eight position pairs of registers vr and vs indexed by the 3 bits. For real data multiplies, the first data pair is selected from these bits, whereas an additional second pair is implicitly selected by adding the value of this field to the offset stored in bits sel[26:24]: sel_mode. For complex data the index simply selects one of eight complex numbers (with real and imaginary parts).
- sel[2:0]=selDataIn_out0 — Directs output data to the group 0 location (next highest pair position in acc_l). Input data is selected from eight position pairs of registers vr and vs indexed by the 3 bits. For real data multiplies, the first data pair is selected from these bits, whereas an additional second pair is implicitly selected by adding the value of this field to the offset stored in bits sel[26:24]: sel_mode. For complex data the index simply selects one of eight complex numbers (with real and imaginary parts).

The next set of four fields choose the coefficients used with the first set of multipliers taken from the TWIDDLEA special register. The field always picks a pair of values, thus for complex cases these are real and imaginary parts and for real data these are just two real values. These pairs of data items are then split up into high and low multiplier inputs. When used with matrix multiplies, these are data input items - when used for FIR, these are taps or coefficients. The bit fields are described below.

- sel[19:18]=selCoeff3 — Selects one of four pairs of elements for real or one of four complex numbers from register TwiddleA into position group 3 to be multiplied with input elements of same group chosen in selDataIn_out3.

- sel[17:16] =selCoeff2 — Selects one of four pairs of elements for real or one of four complex numbers from TwiddleA into position group 2 to be multiplied with input elements of same group chosen in selDataIn_out2.
- sel[15:14] =selCoeff1 — Selects one of four pairs of elements for real or one of four complex numbers from register TwiddleA into position group 1 to be multiplied with input elements of same group chosen in selDataIn_out1.
- sel[13:12] =selCoeff0 — Selects one of four pairs of elements for real or one of four complex numbers from register TwiddleA into position group 0 to be multiplied with input elements of same group chosen in selDataIn_out0.

The next field is used for type selection, and also controls the amount of element offset used.in the case of symmetric FIR.

- sel[23:20]=sel_symFirOffset — If this field is 15 in decimal value, (0xF), data input from TWIDDLEB and TWIDDLEC registers is ignored. In that case, the TWIDDLEA special register is the only one used. For example this is the case with matrix multiplies and normal (non-symmetric) FIR. For symmetric FIR cases, however, this field controls the positions of the data inputs from the TWIDDLEB and TWIDDLEC registers as explained below.

Note that decimal values between 14 and 8 are reserved. Decimal values between 7 and 0 select the offset position of data pair elements to be selected from TWIDDLEB and TWIDDLEC registers when performing symmetric FIR computations.

The bits correspond to the amount of offset in 20-bit element units from the start of the TWIDDLE registers 0th position. In symmetric FIR operations, input data comes from the two input registers and TWIDDLEB and TWIDDLEC registers (thus there are four distinct data pairs). The two sets of pairs are added and then multiplied by data in the TWIDDLEA register. Input data of length 320 bits (a 16x20 vector) comes from input registers vr and vs and an additional 320 bits of input data from the TWIDDLEB and TWIDDLEC registers. To select which pairs to pick from the input registers and where the output goes, we use the index fields selDataIn_out fields reviewed before. To select the pairs for the TWIDDLEB and TWIDDLEC registers in complex data cases, we use the bits of sel_symFirOffset field directly. In the real data cases, the first item pair to be picked derives its index by adding the offset of sel_mode reviewed next, and the offset of sel_symFirOffset, whereas the second item only uses sel_symFirOffset.

The next field specifies the mode of operation and the offset used in real multiplies as discussed below.

- sel[26:24]: sel_mode A 3-bit field with some valid settings and others reserved.

This field determines the amount of spacing or offset applied to the pair of data elements being processed.

For real data, the pairs are two real values, but for complex data the pair is a set of real and imaginary parts or one complex number.

As explained before, when real data is loaded into the input registers vr and vs, the data is stored in 20-bit elements that form pairs. Registers vr and vs can each hold 4 such pairs (so two vectors of 8x20 bits each) for a total of 320 bits. The 16 real multiplies that are possible in these instructions are split into 4 sets of 4 multiplies. Thus each set can operate on two pairs of real values, and the offset specifies the spacing (in single 20 bit element units) between these pairs to be picked from the 16x20 elements stored.

For example, 2x2 real matrix multiplies use an offset of 2. This specifies a spacing of two positions of real elements between the data to be used in the first multiplier pair and the second pair (that has the offset relative to the index of first pair). Similarly, for 4x4 real matrix multiplication, the offset should be set to 4.

Note that this setting is most typically used for real data. For complex data in matrix multiplies, the setting should be 7 decimal which does not create offsets and treats everything as complex. However, for FIR cases that have a mix of complex data and real coefficients, this setting can still be used to offset the complex data by certain amounts.

Table 5–46 lists modes of operation and the resulting offsets.

Table 5–46. Mode and Resulting Offsets

Mode	Offset
0	Offset in 20-bit unit elements = 8
1	Offset in 20-bit unit elements = 1
2	Offset in 20-bit unit elements = 2
3	Reserved
4	Offset in 20-bit unit elements = 4
5	Reserved
6	Reserved
7	Complex, no offset

Notes:

- The last field in the sel operand is a reserved field.
- sel[31:27]: must be set to 5'b00000, other values are reserved.
- For Symmetric FIR uses, because the instructions carry out an add of the symmetric taps or coefficients before multiplication, overflow of the results may occur with lower input data values than in the non-symmetric FIR uses.

Table 5–47. Valid Combinations of the Select Operand Fields

Reserved	Mode	symFirOFF	Coeff3..0	DataIn03	DataIn02	DataIn01	DataIn00	Description
[31:27]	[26:24]	[23:20]	[19:12]	[11:9]	[8:6]	[5:3]	[2:0]	
0	0	F	any	0..3	0..3	0..3	0..3	Matrix Multiply
0	1	0..7,F	any	0..6	0..6	0..6	0..6	Real x Real
0	2	0,2,4,6,F	any	0..6	0..6	0..6	0..6	2x2 and Real x Complex
0	4	F	any	0..5	0..5	0..5	0..5	4x4
0	7	0,2,4,6,F	any	0..7	0..7	0..7	0..7	Complex

5.2 Using the Multi-Way, Multi-Type, Multi-Mode Multiply and Multiply-Add Instructions

It is easier to understand some of the variety of operations supported by these instructions through small code examples. In this case we define three C code macros (#defines), SEL_DESC (for real FIR and real matrix multiplies), CSEL_DESC (for complex FIR and complex matrix multiplies), and FSEL_DESC (for symmetric FIR operation). We first give the three macros, which take nine or eight inputs, and fill out the control select

operation into a variable of type vsel that will be stored in the sel register. The macros effectively shift the various bit fields into the right positions of the 32-selection operand sel to control the 10 fields described above.

```
#define SEL_DESC(_e0, _e1, _e2, _e3, _e4, _e5, _e6, _e7, offset) ( \
     ((_e0/2) << (3*0)) | \
     ((_e1/2) << (3*1)) | \
     ((_e2/2) << (3*2)) | \
     ((_e3/2) << (3*3)) | \
     ((_e4/2) << (12+2*0)) | \
     ((_e5/2) << (12+2*1)) | \
     ((_e6/2) << (12+2*2)) | \
     ((_e7/2) << (12+2*3)) | \
     ((offset) << (20)))


#define CSEL_DESC(_e0, _e1, _e2, _e3, _e4, _e5, _e6, _e7) ( \
        ((_e0) << (3*0)) | \
        ((_e1) << (3*1)) | \
        ((_e2) << (3*2)) | \
        ((_e3) << (3*3)) | \
        ((_e4) << (12+2*0)) | \
        ((_e5) << (12+2*1)) | \
        ((_e6) << (12+2*2)) | \
        ((_e7) << (12+2*3)) | \
        ((0x7F) << (20)))


#define FSEL_DESC(_e0, _e1, _e2, _e3, _e4, _e5, _e6, _e7, offset ) ( \
```

```
((_e0) << (3*0)) | \
((_e1) << (3*1)) | \
((_e2) << (3*2)) | \
((_e3) << (3*3)) | \
((_e4) << (12+2*0)) | \
((_e5) << (12+2*1)) | \
((_e6) << (12+2*2)) | \
((_e7) << (12+2*3)) | \
((offset) << (20)))
```

Following are illustrations of the use of the control operands for various cases.

5.2.1 Real 2x2 and 4x4 Matrix Multiply Modes

In a real 2x2 matrix multiply mode, we use the following vsel register pattern that will support two simultaneous 2x2 real matrix multiplies:

```
vsel sel = SEL_DESC(0, 0, 4, 4, 0, 2, 4, 6, 0x2F);
```

Referring to the C macro for SEL_DESC, this uses a value of 2 for bits of sel_mode, which means real data with an offset of 2 is loaded into input register vr. As the register can hold 8x20 values, we can load two complete 2x2 matrices, call them A and B, 4x20 bit vectors each, split into rows that have offset of two elements between them. It also sets sel_symmFirOffset to 15 (0xF), which means that only data from TWIDDLEA special register (in conjunction with fist input vr) is used. The TWIDDLEA register will hold another two 2x2 real matrices, call them T0 and T1, also 4x20 bit vectors, to be used as the multiplier.The first four values of 0,0,4,4 are input indexes selDataIn_outi, i=0…3, and converted by the macro to 0,0,2,2. They are used to select the input pairs for the four matrices to be multiplied. Their position also shows where the output will be directed. For example, the first value of selDataIn_out0=0 directs the output to the lowest pair of acc_l vector output. Thus we want to perform T0*A and T1*B, which will be supported by this single-instruction 16 MAC capability. The two rows of the A matrix are selected by the first 0 value of selDataIn_out0, which picks the first pair (first row) and implicitly the second pair offset by two elements, which is essentially the second row. Similarly for

the second 0 of selDataIn_out1. The third 2 of selDataIn_out2 will select the first row of the second matrix B and the fourth 2 of selDataIn_out3 its offset by 2, or the second row of B matrix. In other words, both 2x2 matrices are selected.

The next four values are the selCoeffi, i=0…3 variables, where 0,2,4,6 are converted to 0, 1, 2, 3 by the macro. The selCoeff0=0 variable selects the first row of T0 and will multiply the A matrix. The result of the multiplication, 2x40 bit pair, goes into the low elements of acc_l output based on the position of the selDataIn_out0 index (first position). The second row of T0 is selected by value selCoeff1=1 and multiplied again by index selDataIn_out1=0 of input, which is again the whole matrix A (two rows). The result goes into the high element pairs of acc_l based on the second position of the index selDataIn_out1. Similarly the T1 rows are selected by 2, 3 indexes selCoeff2 and selCoeff3, and the input matrix B is selected by input index 2, 2 of selDataIn_out2 and selDataIn_out3. In each case two pairs are selected, the third pair along with a pair offset by two elements from the third pair, which is really both rows of second matrix B. The multiplication of T1*B gives the result into the 4x40 vector stored in the acc_h register, with each pair of outputs coming from the respective position of the input register index based on the group positions selected by the value order.

In a real 4x4 matrix multiply, we store rows of matrices in the TWIDDLEA and input registers. Call input matrix A with elements a00, a01..a44 (row major notation here refers to element on ith row and jth column) and TWIDDLE matrix T with elements t00, t01…t44. We will perform R=T*A for 4x4 case, where R is a 4x4 resulting matrix with elements r00, r01...r44. We use two vsel register patterns to perform the first two row multiplies and then repeat for the final two rows of the result. The first vsel is used for an initial multiply and the second for an multiply-accumulate.

```
vsel s0 = SEL_DESC(0,2,0,2,0,0,4,4,0x4F);

vsel s1 = SEL_DESC(0,2,0,2,2,2,6,6,0x4F);
```

This uses a value of 4 for sel_mode[, which means real data with an offset of 4 (since the load of 4x4 real matrices into the vector register vr will load two rows of a complete matrix A, and each row is offset from the next by four). It also sets sel_symmFirOffset to 15 (0xF), which means that only data from TWIDDLEA special register is used along with register vr.

In the first select register value, the first four values of selDataIn_outi, i=0…3, are 0,2,0,2, which are converted by the macro to 0,1,0,1. The four distinct values for the TWIDDLE matrix selections selCoeffi, i=0…3, are converted from 0, 4, 2, 6 to 0, 2, 1, 3, Each pair of TWIDDLE and input data indexes perform four multiplies. Input data index selDataIn_out0 =0 will select elements a00, a01 pairs, and also a11, a12 pairs from second row (since offset is 4). The TWIDDLE first value is selCoeff0=0 and selects the first pair of the first row of T matrix, t00 and t01. Once we multiply the three pairs, we will get the first partial sums of result matrix R, r00 and r01, stored in acc_l's low pair position. Similarly, for the next set of four multiplies, a02, a03, and a12, a13 are selected by the

input index selDataIn_out1, which are multiplied by TWIDDLEA elements t00, t01 since the value of TWIDDLEA selCoeff1 is still 0. The result will be partial sum for r02 and r03, which will go to acc_l high pair position. Continuing in this way, values 0,1 for selDataIn_out2 and selDataIn_out3 for inputs and 2, 2, for TWIDDLEA selCoeff2 and selCoeff3 will complete the partial sums for r10, r11, r12 and r13 in result matrix R. These will go into acc_h output pair positions accordingly. Thus the first two row partial sums in the R matrix are ready. The next vsel should operate on the next two rows of the A matrix and multiply with pairs 1, 3 of the T matrix. The sums are to be accumulated with the previous results, completing the first two rows full sum. The process has to be repeated for the next two rows of the T matrix to create the last two rows of the result R matrix.

5.2.2 Complex 2x2 and 4x4 Matrix Multiply Multi Modes

In a complex 2x2 matrix multiply, we use two vsel register patterns. The first is used for the initial Multiply and the second for the Multiply-accumulate.

```
vsel sel0     =  CSEL_DESC(0, 0, 2, 2, 0, 1, 0, 1);

vsel sel1     =  CSEL_DESC(1, 1, 3, 3, 2, 3, 2, 3);
```

These select register values use the CSEL_DESC define, which sets the sel_mode to 7 and sel_symmFirOffset to 0xF automatically since they are always the same for complex. The first select register is used for a multiply and the second for a multiply-accumulate. One 2x2 complex matrix output (consisting of four complex (imag, real) pairs) is produced using the two select registers and the multiply and multiply-accumulate. Denote the two complex matrices to be operated A with elements [a00 a01;a10 a11] and the second B with elements [b00 b01; b10 b11]. The first A matrix is saved in input register vr and B into TWIDDLEA. The first multiply will compute four output values: a00 * b00, a00 * b01, a10 * b00, a11 * b01. The second multiply-accumulate will compute four values: a01*b10, a01*b11, a11*b10, a11*b11 and add them to the four values computed in the first pass. Thus the two operations will compute the whole 2x2 matrix.

In a complex 4x4 matrix multiply, we use four vsel register patterns. The first is used for the initial Multiply and the remaining three for three Multiply-accumulates. After this round of four operations, the first row of the output complex matrix consisting of four complex pairs (i,r) is ready to be output. This is then repeated three more times.

```
vsel s0 = CSEL_DESC(0,1,2,3,0,0,0,0);

vsel s1 = CSEL_DESC(0,1,2,3,1,1,1,1);

vsel s2 = CSEL_DESC(0,1,2,3,2,2,2,2);

vsel s3 = CSEL_DESC(0,1,2,3,3,3,3,3);
```

Using the same notation as before for matrices A and B but augmenting them to a 4x4 complex case, the computation here uses these patterns to first load a row of the A matrix into TWIDDLEA, and the first rows of the B matrix into input register vr. Each select register value will effectively replicate a value of the A matrix row four times, and multiply them against the four values in the row of the B matrix in input register vr. Thus the first computation will produce a00 * b00, a00 * b01, a00 * b02, a00 * b03. The second computation which is a multiply-accumulate will use the second row of the B matrix. Thus it will add to the first outputs the values a01 * b10, a01 * b11, a02 * b12, a03 * b13 and so on.

5.2.3 FIR Operations

For FIR operations, we want to use a sliding window that moves across the data items, and apply the same taps (coefficients) to the sliding window while accumulating the results across the number of taps. We use the two input vector registers to load in data, and the TWIDDLEA special register to hold taps or coefficients. We use the TWIDDLEB AND TWIDDLEC special registers to hold additional of data for symmetric FIR cases.

For example, a complex 16 tap FIR (complex taps or coefficients, complex data) will use TWIDDLEA to hold four complex taps (four pairs of (imaginary, real)), and the two vector input registers vr and vs will contain eight data items.

```
vsel sel0 =  CSEL_DESC(0, 1, 2, 3, 0, 0, 0, 0);

vsel sel1 =  CSEL_DESC(1, 2, 3, 4, 1, 1, 1, 1);

vsel sel2 =  CSEL_DESC(2, 3, 4, 5, 2, 2, 2, 2);

vsel sel3 =  CSEL_DESC(3, 4, 5, 6, 3, 3, 3, 3);
```

The four select registers will select first data items 0-3 by setting selDataIn_outi, i=0…3, and apply tap 0 to them with selection of selCoeff0=0; then items 1-4 and apply tap 1; then 2-5 and apply tap 2; finally 3-6 and apply tap 4. We then load in four more data items and slide the window over, and load in four more taps. At the end of a sequence of one multiply and fifteen multiply-accumulates, we have four outputs for a 16-tap filter.

To understand the setup, consider the four consecutive outputs of a FIR operation. Outputs at times t0, t1, t2, t3 are denoted o0, o1, o2, o3. The first output will be equal to

```
o0=c0*d0+c1*d1+…cN*dN, for coefficients ci and data di, i=0 to N.
```

The next three outputs will be the same coefficients with shifted data

```
o1=c0*d1+c1*d2+…cN*dN+1
o2=c0*d2+c1*d3+…cN*dN+2
o3=c0*d3+c1*d4+…cN*dN+3
```

Thus we can accumulate partially each output with each multiplication of data *coefficient and then shift the next data item in to proceed with the next partial sum until each output accumulates the whole sum.

For a 16-tap real-complex FIR (real taps, complex data), we use a different set of patterns because the complex data is effectively treated as two real value for multiplying by the real taps. Similar to above, we need to create four outputs in time as follows:

```
o0=c0*d0+c1*d1+…cN*dN
o1=c0*d1+c1*d2+…cN*dN+1
o2=c0*d2+c1*d3+…cN*dN+2
o3=c0*d3+c1*d4+…cN*dN+3
```

We use an offset of 2 because we can actually compute four multiplies at each pair of indexes passed in the vsel, that is, each two real coefficients multiplied by two complex data points. The TWIDDLEA special register will contain eight real taps.

```
vsel sel0 =  FSEL_DESC(0, 1, 2, 3, 0, 0, 0, 0, 0x2f);

vsel sel1 =  FSEL_DESC(2, 3, 4, 5, 1, 1, 1, 1, 0x2f);

(load new data here)

vsel sel2 =  FSEL_DESC(0, 1, 2, 3, 2, 2, 2, 2, 0x2f);

vsel sel3 =  FSEL_DESC(2, 3, 4, 5, 3, 3, 3, 3, 0x2f);
```

Thus in the above first vsel line, with selDataIn_out0=0 for input indexes selects two complex numbers, d0 and a second offset by 2 (real) elements, d1. The selCoeff0=0 for TWIDDLEA selects two real taps, in this case c0 and c1. So we will compute c0*d0 (2 multiplies) and c1*d1 (2 multiplies). As you see this will start computing the first partial sum of output o0. The rest of the values will process the four multiplies and first sum for o1, o2, and o3 as well. To continue with the rest of the partial sums, we have to start now at data point d2 since the first two have been processed already. That is shown in vsel line 2 above. Finally, we need to load new coefficients and new data to continue the next eight points and repeat with the same values as shown above. Essentially, the load of new coefficients into TWIDDLEA completes the 16 real taps needed. The first two selects are applied, followed by a load of new data and the second two selects are applied; at this point, the four complex outputs are ready to be written out.

For 16-tap real-real FIR, the offset is 1 since every element is a new data point rather than being part of a complex (imaginary, real) pair. This pattern is like the set of patterns for complex-complex FIR, but only half the number of data and coefficient loads are required since every load brings in eight real items. A sequence of eight multiply/multiply accumulates is required for eight real outputs. We would like to compute the following:

```
o0=c0*d0+c1*d1+…cN*dN
o1=c0*d1+c1*d2+…cN*dN+1
o2=c0*d2+c1*d3+…cN*dN+2
o3=c0*d3+c1*d4+…cN*dN+3
o4=c0*d4+c1*d5+…cN*dN+4
o5=c0*d5+c1*d6+…cN*dN+5
o6=c0*d6+c1*d7+…cN*dN+6
o7=c0*d7+c1*d8+…cN*dN+7

vsel sel0 =  FSEL_DESC(0, 1, 2, 3, 0, 0, 0, 0, 0x1f);

vsel sel1 =  FSEL_DESC(1, 2, 3, 4, 1, 1, 1, 1, 0x1f);

vsel sel2 =  FSEL_DESC(2, 3, 4, 5, 2, 2, 2, 2, 0x1f);

vsel sel3 =  FSEL_DESC(3, 4, 5, 6, 3, 3, 3, 3, 0x1f);
```

Each pair of indexes will do the usual four multiplies. For the first line we pick real data d0 and d1 pair with selection selDataIn_out0=0, and the offset pair of sel_mode=1 element, which is d1 and d2. Those pairs are multiplied by the coefficient pair 0, which is c0 and c1. We produce c0*d0+c1*d1, and c0*d1+ c1*d2, which fill the first partial sum of the first two outputs o0, o1. The remaining lines produce the rest of the output partial sums and then new data is loaded to continue.

For symmetric FIR, we use TWIDDLEB and TWIDDLEC special registers as additional storage of data, so we have 32x20 bit space. These hold data items which will be multiplied and added to the output of normal tap-data multiplication The taps are stored in TWIDDLEA as usual. Because we use the extra set of multiplications, the select patterns do not set sel_symmFirOffset to 0xF; rather, they are used to define offsets for the second data inputs in TWIDDLEB and TWIDDLEC. This has the effect of multiplying each tap against two different data items. Note that the complex-complex case does not use complex conjugates for the symmetric part of the taps but does typical complex multiplication. It is your responsibility to load the right data into TWIDDLEB and TWIDDLEC so the right two data numbers are added and multiplied by the correct coefficients.

For complex-complex 16-tap symmetric FIR, we use patterns similar to normal FIR, except we also do the second set of multiply-adds, and only use a sequence of eight multiply/multiply-accumulates with a new coefficient load after the first four. The assumption is that complex inputs d0 through d7 are stored in input registers vr and vs and complex inputs d12 to d19 are stored in TWIDDLEB and TWIDDLEC (the low portion goes to TWIDDLEC).

```
vsel sel0 =  FSEL_DESC(0, 1, 2, 3, 0, 0, 0, 0, 0x76);

vsel sel1 =  FSEL_DESC(1, 2, 3, 4, 1, 1, 1, 1, 0x74);

vsel sel2 =  FSEL_DESC(2, 3, 4, 5, 2, 2, 2, 2, 0x72);
```

```
vsel sel3 =  FSEL_DESC(3, 4, 5, 6, 3, 3, 3, 3, 0x70);
```

Referring to the above first vsel, the first selDataIn_out0=0 in input indexes selects the first complex number d0 stored in the input registers vr and vs, which will be added to a complex number stored in the TWIDDLEB and TWIDDLEC registers; the latter is chosen by offset sel_symmFirOffset=6 element pairs (or three complex numbers), so effectively the fourth complex number is chosen in TWIDDLEC, which is d15 (counting for the delays of all four registers). After the addition (d0+d15), the result is multiplied by the first complex number in the TWIDDLEA index, c0, selected by the selCoeff0=0 value. The next pair will be d1 and d16, again multiplied by c0, and so on. Note that each advancement of pair indices in the twiddles we increment the offset by 2, so it would be 6 at first, then 8, 10, and 12. Those will index twiddle data numbers d15, d16, d17, d18, respectively. We continue the same operation with data shifted by one in both input and TWIDDLEB and TWIDDLEC registers and multiply by the next coefficient of TWIDDLEA register. Once all four coefficients are completed, we load the next four and start again. The operations are as follows for coefficients ci, data di, and outputs oi:

```
o0=c0*(d0+d15)+c1*(d1+d14)+…
o1=c0*(d1+d16)+c1*(d2+d15)+…
o2=c0*(d2+d17)+c1*(d3+d16)+…
o3=c0*(d3+d18)+c1*(d4+d17)+…
```

For 16-tap real-complex (real taps, complex data) symmetric FIR we use an offset of sel_mode=2. Each index selCoeffi, i=0…3, into TWIDDLEA selects a pair of real values for taps and each index of input data selDataIn_outi, i=0…3, selects two complex numbers from registers vr and vs. An additional two numbers from TWIDDLEB and TWIDDLEC registers are selected with offset sel_symmFirOffset. Following is an example:

```
vsel sel0 =  FSEL_DESC(0, 1, 2, 3, 0, 0, 0, 0, 0x24);

vsel sel1 =  FSEL_DESC(2, 3, 4, 5, 1, 1, 1, 1, 0x20);

vsel sel2 =  FSEL_DESC(0, 1, 2, 3, 2, 2, 2, 2, 0x24);

vsel sel3 =  FSEL_DESC(2, 3, 4, 5, 3, 3, 3, 3, 0x20);
```

Refer to the first vsel line. We first select selCoeff0=0 for the TWIDDLEA index, which selects coefficient pair c0 and c1, both real. Then we select selDataIn_out0=0 value for the first index in input registers, which selects the complex number 0 and one offset by sel_mode=2 elements, shown in the first digit of the select operand. So we essentially pick d0 and d1 complex numbers. We also select two complex numbers from data in TWIDDLEB and TWIDDLEC. For this selection, there are two offsets to be applied, from sel_mode and sel_symmFirOffset fields. The left offset is sel_mode=2 and the right off-

set is sel_symmFirOffset=4. The left offset shows the offset between the input register 20-bit elements as before, and the right offset shows the offset from the start of 8x20 vector register data in TWIDDLEB and TWIDDLEC. Thus, the first complex number will be indexed by the offset of (sel_mode+ sel_symmFirOffset)=2+4=6. The second complex number is only offset by the right offset sel_symmFirOffset, so it would be 4. Based on the delays chosen, the complex numbers d15 and d14 will be chosen. So we perform c0*(d0+d15)+c1*(d1+d14) for the first four multiplies to go to output o0. Similarly, the rest of the selections will complete partial sums for outputs o1, o2 and o3. More data is loaded and the process is repeated to accumulate all the output results. The summary is:

```
o0=c0*(d0+d15)+c1*(d1+d14)+…
o1=c0*(d1+d16)+c1*(d2+d15)+…
o2=c0*(d2+d17)+c1*(d1+d16)+…
o3=c0*(d3+d18)+c1*(d2+d17)+…
```

For 16-tap real-real symmetric FIR, we use patterns similar to the complex-complex case, except the offset is 1 and a sequence of four multiply/multiply-accumulates will produce eight outputs at a time since each multiply-accumulate will do effectively 32 multiplies (taking advantage of the symmetry).

```
vsel sel0 =  FSEL_DESC(0, 1, 2, 3, 0, 0, 0, 0, 0x16);

vsel sel1 =  FSEL_DESC(1, 2, 3, 4, 1, 1, 1, 1, 0x14);

vsel sel2 =  FSEL_DESC(2, 3, 4, 5, 2, 2, 2, 2, 0x12);

vsel sel3 =  FSEL_DESC(3, 4, 5, 6, 3, 3, 3, 3, 0x10);
```

The first select for example, chooses real data pair d0 and d1 (from selector selDataIn_out0=0) and pair d1 and d2 (from left offset sel_mode=1) coming from input registers vr, vs; we also select data points d16 and d15 pair (left offset sel_mode=1 plus right offset sel_symmFirOffset=6 for total 7) and pair d15 and d14 (right offset sel_symmFirOffset=6) from the TWIDDLEB and TWIDDLEC registers. Each index input/coefficient will do four multiplies, thus for example, the first four such multiplies will be c0*(d0+d15)+c1(d1+d14) for o0, and also c0*(d1+d16)+c1*(d2+d15) for o1.

Continuing for all selections and all loads of data, the operations here are 8 outputs at a time, with all real multiplications, as follows:

```
o0=c0*(d0+d15)+c1*(d1+d14)+…
o1=c0*(d1+d16)+c1*(d2+d15)+…
o2=c0*(d2+d17)+c1*(d3+d16)+…
o3=c0*(d3+d18)+c1*(d4+d17)+…
o4=c0*(d4+d19)+c1*(d5+d18)+…
o5=c0*(d5+d20)+c1*(d6+d19)+…
```

```
o6=c0*(d6+d21)+c1*(d7+d20)+…
o7=c0*(d7+d22)+c1*(d8+d21)+…
```

Figure 5–16 displays an example of how to pick a specific set of coefficient pairs from TWIDDLEA register and data input pairs from input registers and TWIDDLEB/C registers. Note the selector settings to set up correctly a symmetric FIR operation of the multi-mode instruction. Successive selections and instruction calls are needed to complete the FIR operation as shown in the following applicable C code.

Notes for Figure 5–16:

- sel_mode=1, real offset by one element for second pair of data input
- Elements are shown in real 20-bit values

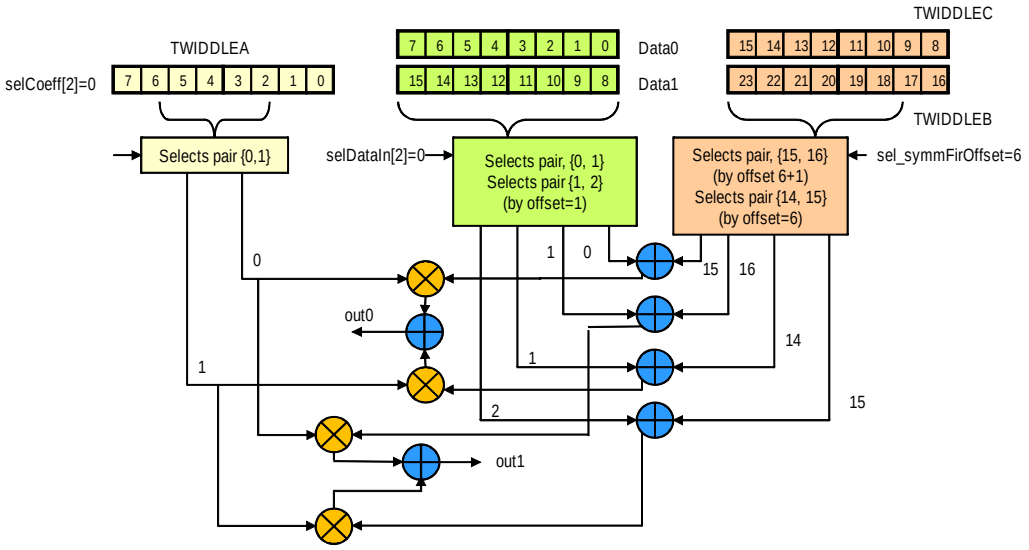


Figure 5–16. 16-Tap Real Symmetric FIR Example

5.3 Examples

5.3.1 2x2 Complex Matrix Multiply

Figure 5–17 shows what we want to do with the 2x2 complex matrix, which consists of four complex elements which occupy one 8x20 vector register. A second complex matrix we multiply again is another 8x20 vector register. We put the first matrix in the TwiddleA register and the second in an input register.

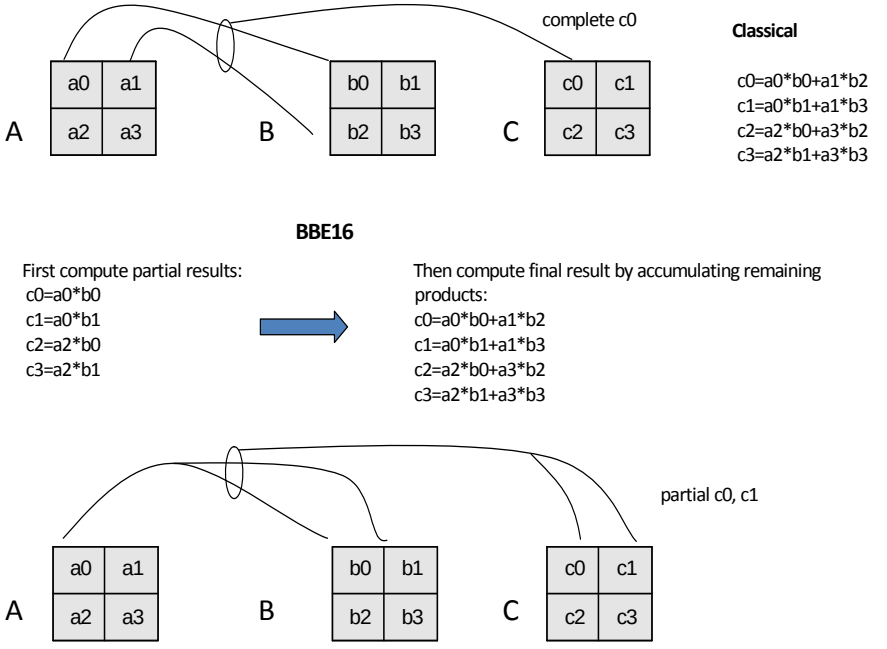


Figure 5–17. 2x2 Complex Matrix Multiply

The objective is to write the code in a way that would allow the compiler to efficiently load consecutive elements from memory into vectors, perform multiply operations, then store the results into consecutive locations again back in memory. Multiplying the matrices A and B in the classical way of an A row times a B column does not allow efficient processing. Although the rows can be loaded consecutively from matrix A, the columns of B matrix are loaded with non-unity strides from one element to the next and cannot be used in vector load instructions. Thus, we multiply a replicated element of matrix A times the consecutive elements of B matrix rows and add the accumulated results to generate the output matrix. So to produce the first row of the result matrix, multiply the first element of A's first row times the first B row, and accumulate it with a product of the second element of A's first row times the second B row, and so on for all elements of A first row and all rows of B. The code shows how the indexes are chosen.

As Figure 5–17 shows, there are eight complex multiplies required - four multiplies and four multiply-accumulates. We call a MUL and then a MULA to compute the first partial sums and then the second set. We use two select patterns set up with the CSEL macro, as per the following code:

```
// Using complex intrinsics

complex a[NSAMPLES][ARRAY_SIZE][ARRAY_SIZE] ; // A matrix allocation
complex b[NSAMPLES][ARRAY_SIZE][ARRAY_SIZE] ; // B matrix
```

```
complex c_opt[NSAMPLES][ARRAY_SIZE][ARRAY_SIZE] ; // result matrix C

void mm_opt_2x2_complexTypes_fir()
{
xb_vec4xc16  * __restrict vcp = (xb_vec4xc16 *)c_opt; // cast to
pointer of BBE16 vectors
xb_vec4xc16  * __restrict vap = (xb_vec4xc16 *)a;
xb_vec4xc16  * __restrict vbp = (xb_vec4xc16 *)b;
xb_vec4xc40 c; // result register
vsel sel0     =  CSEL_DESC(0, 0, 2, 2, 0, 1, 0, 1); // first 4 numbers
select inputs, last 4 numbers outputs
vsel sel1     =  CSEL_DESC(1, 1, 3, 3, 2, 3, 2, 3);
int h;

vbp--; // preincrement for load
BBE_LTA4XC16_IU(vbp, 16); // load twiddle A with matrix B
for (h=0; h < NSAMPLES; h++) {
     xb_vec4xc20 va = vap[h]; // load matrix A
     c = BBE_MUL4XC20E (va, va, sel0 ); // complex  multiply A*B,
partial sums
     BBE_MULA4XC20E (c, va, va, sel1); // complex multiply and
accumulate to complete samples
     vcp[h] = c; // result store
     BBE_LTA4XC16_IU(vbp, 16); // load next sample of B
     }
```

5.3.2 2x2 Real Matrix Multiply

In the 2x2 real matrix multiply example, note that each real 2x2 matrix is four elements, so a single load brings in two matrices. We load two in Twiddle A and two into a vector input register. The select register uses a mode of 2 per row for processing and offset of F as this is not symmetric FIR. Thus, the selector is 0x2F in the macro SEL_DESC. Remember each value in the SEL_DESC is divided by two, so the patterns are 0,0,2,2 to pick a pair of data elements for each SIMD lane and then 0,1,2,3 from the twiddle register A.

Figure 5–18 shows an example of how specific elements from input registers and Twiddle A register are selected, and how to set the mode selectors for this 2x2 real matrix multiply.

```
vsel sel = SEL_DESC(0, 0, 4, 4, 0, 2, 4, 6, 0x2F);
for (h=0; h < NSAMPLES/2; h++) {
     xb_vec8x20 vb = vbp[h];
     BBE_LTA8X16S_IU(vap, 16);
     BBE_MUL8X18E(chi, clow, vb, vb, sel );
     vcp[h] = BBE_PACKS8X40(chi, clow);
     }
```

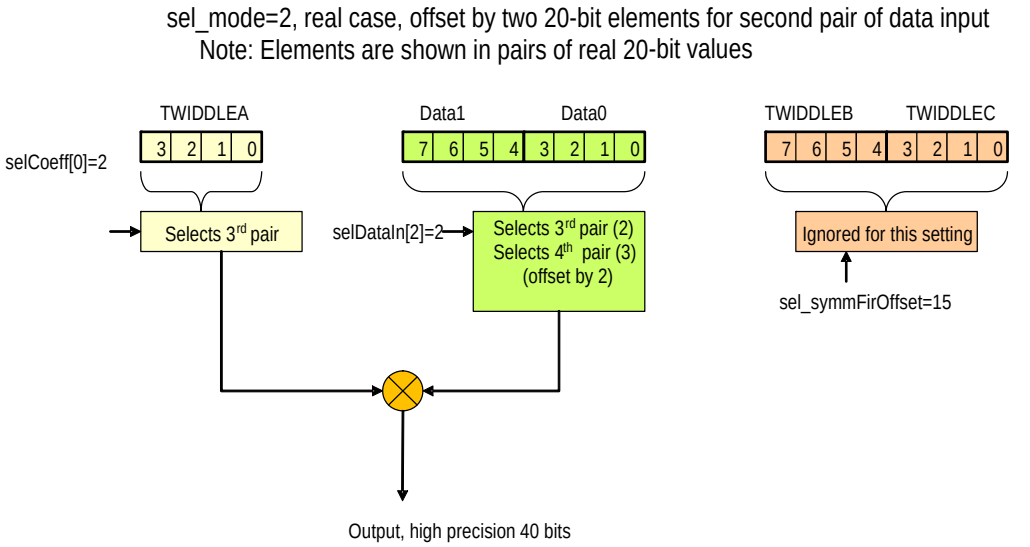


Figure 5–18. 2X2 Real Matrix Multiply Example

5.3.3 16-Tap Complex-Complex FIR Example

This example uses the BBE_MUL8X18E and BBE_MULA8X18E instructions for a 16-tap complex tap, complex data FIR. Note that first the ConnX BBE16 data types are used for 4-way complex vectors of Q4_15 elements: xb_vec4xcq4_15. Then protos for these data types are used to access the instructions— BBE_MUL4XCQ4_15 and BBE_MULA4XCQ4_15. There are four loads into Twiddle Register A to load the 16 complex taps or coefficients, and then four multiplies or multiply-accumulates after each twiddle load to apply the coefficients to the right set of data. There are four data loads to slide the coefficients across the data and four select patterns applied repeatedly to select the data items and coefficients.

```
     void fir_bbe16_cxcx(      complexT *data,       complexT *coeff,
complexT *rPtr,     size)
{
      j;
        xb_vec4xcq15 *            aP;
  xb_vec4xcq15 *            rP ;
  xb_vec4xcq15 * coeff_p = (xb_vec4xcq15 *) coeff ;
  xb_vec4xcq4_15 pV1, pV2, qV1, qV2;
  xb_vec4xcq4_15 pV3;

  vsel sel0 =  CSEL_DESC(0, 1, 2, 3, 0, 0, 0, 0);
  vsel sel1 =  CSEL_DESC(1, 2, 3, 4, 1, 1, 1, 1);
```

```
vsel sel2 =  CSEL_DESC(2, 3, 4, 5, 2, 2, 2, 2);
vsel sel3 =  CSEL_DESC(3, 4, 5, 6, 3, 3, 3, 3);

xb_vec4xcq9_30 c;


rP = (xb_vec4xcq15 *) &rPtr[0] ;

   (j=0; j<size/4; j++)
{
  aP = (xb_vec4xcq15 *) &data[j*4] ;


  pV1= *aP++;  //  aP[0]   DATA LOAD
  qV1= *aP++;  //  aP[1]   DATA LOAD
  BBE_LTA4XCQ15_I(coeff_p,0);


  c = BBE_MUL4XCQ4_15E(qV1, pV1, sel0);
  BBE_MULA4XCQ4_15E(c, qV1, pV1, sel1);
  BBE_MULA4XCQ4_15E(c, qV1, pV1, sel2);
  BBE_MULA4XCQ4_15E(c, qV1, pV1, sel3);

  pV2= *aP++;  //  aP[0]   DATA LOAD
  BBE_LTA4XCQ15_I(coeff_p+1,0);

  BBE_MULA4XCQ4_15E(c, pV2, qV1, sel0);
  BBE_MULA4XCQ4_15E(c, pV2, qV1, sel1);
  BBE_MULA4XCQ4_15E(c, pV2, qV1, sel2);
  BBE_MULA4XCQ4_15E(c, pV2, qV1, sel3);

  qV2= *aP++;  //  aP[1]   DATA LOAD
  BBE_LTA4XCQ15_I(coeff_p+2,0);

  BBE_MULA4XCQ4_15E(c, qV2, pV2, sel0);
  BBE_MULA4XCQ4_15E(c, qV2, pV2, sel1);
  BBE_MULA4XCQ4_15E(c, qV2, pV2, sel2);
  BBE_MULA4XCQ4_15E(c, qV2, pV2, sel3);

  pV3= *aP++;          //   DATA LOAD
  BBE_LTA4XCQ15_I(coeff_p+3,0);

  BBE_MULA4XCQ4_15E(c, pV3, qV2, sel0);
  BBE_MULA4XCQ4_15E(c, pV3, qV2, sel1);
  BBE_MULA4XCQ4_15E(c, pV3, qV2, sel2);
  BBE_MULA4XCQ4_15E(c, pV3, qV2, sel3);

  *rP++ = c;
```

6. Load/Store Instructions

ConnX BBE16 has an extremely wide variety of load and store instructions 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. These support both 16- and 32-bit scalars and 8x16 and 4x32 bit vectors in memory and 20- and 40-bit elements in vector registers. Unaligned vectors are supported by aligning loads and stores. There are also masking stores and combined pack and store operations (40 bits in registers to 16 bits in memory). The seven addressing modes include a variety of indexed, immediate, updating and circular buffer addressing. There are also 160-bit spill and restore instructions primarily for compiler use.

6.1 Load and Store Intrinsic Names

Table 6–48 lists the different type of load and store instructions, including the data and pack type, and addressing modes.

Table 6–48. Data Types for ConnX BBE16 Load/Stores

Type	Prefix	Data Type	Pack Type	Addressing
Vector Load	BBE_LV	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Aligning Vector Load	BBE_LA	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Scalar Load	BBE_LS	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Scalar Pair Load	BBE_LP	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU)
Twiddle Load	BBE_LT(A|B|C)	(8X16S|4XC16|8XQ15|4XCQ15)		(I|IU)
Masked Store	BBE_SM	(8X16|8X16S|8X16U|4XC16|8XQ15|4XCQ15)		IU
Vector Store	BBE_SV	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Aligning Vector Store	BBE_SA	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Scalar Store	BBE_SS	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU|CU)
Scalar Pair Store	BBE_SP	(8X16S|8X16U|4XC16|8XQ15|4XCQ15)		(I|IU|X|XU)
Vector Int Pack Store	BBE_SV	(8X16|4XC16)	PACKS	IU
Vector Fract Pack Store	BBE_SV	(8XQ15|4XCQ15)	PACKQ	IU
Scalar Int Pack Store	BBE_SS	(8X16|4XC16)		IU
Scalar Fract Pack Store	BBE_SS	(8XQ15|4XCQ15)	PACKQ	IU

Table 6–48. Data Types for ConnX BBE16 Load/Stores

Type	Prefix	Data Type	Pack Type	Addressing
Vector FFT Store	BBE_SR	(4XC16|4XCQ15)		[XU|BR|BRU
Aligning, Compacting Vector Store	BBE_SAC	(8X16S | 8X16U | 4XC16 | 8XQ15 | 4XCQ15 | 4X32S | 4X32U | 2XC32 | 4XQ1_30 | 2XCQ1_30)		U
Aligning, Compacting Vector Store Flush	BBE_SAC	(128 | 8X16S | 8X16U | 4XC16 | 8XQ15 | 4XCQ15 | 4X32S | 4X32U | 2XC32 | 4XQ1_30 | 2XCQ1_30)	F	

6.2 ConnX BBE16 Addressing Modes

Table 6–49 provides a description of the ConnX BBE16 addressing modes.

Table 6–49. ConnX BBE16 Addressing Modes

Name	Notation	Description
Immediate	.I	Offset is an immediate operand
Immediate with Update	.IU	Offset is an immediate operand. Update base address.
Index	.X	Offset from a register
Index with Update	.XU	Offset from a register. Update base address.
Circular with Update	.CU	Circular addressing with update. Increment by one vector.
Narrow Immediate	.I.N	Offset is a narrow immediate operand
Narrow Immediate with Update	.IU.N	Offset is a narrow immediate operand. Update base address
Bit-reverse store	.BR	Offset from bit-reverse register
Bit-reverse and update	.BRU	Update bit-reverse offset with increment

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

6.3 Circular Addressing

The ConnX BBE16 DSP Engine can handle circular addressing using two registers, `cbegin` and `cend`. These two registers have certain requirements when they are initialized as described below.

- Circular buffer addressing mode requires the circular buffer be 16-byte aligned. So both CBEGIN and CEND must be 16-byte aligned addresses. Otherwise, circular buffer loads and stores will not behave correctly.
- CBEGIN should be set to 16 bytes before the start address of the circular buffer.
- CEND should be set to 32 bytes before the end of the circular buffer, which is 16 bytes before the last aligned vector in the buffer.
- ConnX BBE16 only supports a stride of 16 bytes in circular addressing.

Other than these considerations, the load and store operations perform as expected, and can be either aligned or unaligned.

There are four basic instructions to access the Read, Write, and Exchange registers as shown in Table 6–50:

Table 6–50. Register Instructions

Instruction	Description
RUR.CBEGIN()	Reads user register CBEGIN
RUR.CEND()	Reads user register CEND
WUR.CBEGIN(v)	Writes user register CBEGIN
WUR.CEND(v)	Writes user register CEND

The load/store operations with a .CU suffix or the corresponding _CU C intrinsic functions refer to the circular buffer addressing. Also, the BBE_LVA.CP priming load operation and the corresponding BBE_LVA_CP intrinsic function use the circular buffer registers.

6.4 Aligning Loads and Stores

The aligning vector load and store instructions move 128-bit vectors between the ConnX BBE16 registers and memory addresses that may or may not be aligned to 128-bit boundaries. If the address is aligned, these instructions perform the same operation as aligned load and store instructions. For aligning addresses, these instructions use the ConnX BBE16 alignment register file to provide a throughput of one aligning load or store operation per instruction. The aligning vector load and store instructions 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.

A special priming instruction is used to begin the process of loading an array of unaligned data. This instruction conditionally loads the alignment register if the target address is unaligned. If the memory address is not aligned to a 128-bit boundary, this load initializes the contents of the alignment register. The subsequent aligning load instruc-

tion pre-increments the load target address and then 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. Data from this load then overwrites the alignment register, priming it for the next load. Subsequent load instructions provide a throughput of one aligning load per instruction.

The design of the priming load and aligning load instructions is such that they can be used in situations where the alignment of the address is unknown. If the address is aligned to a 128-bit boundary, the priming load instruction does nothing. Subsequent aligning load instructions 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.

Aligning stores operate in a slightly different manner. Each aligning store instruction 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 128-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 128-bit aligned address. This store will only write data starting from the byte corresponding to the memory address of the store.

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

Start with the BBE_ZALIGN instruction to store an array of vectors beginning at an aligning memory address. This instruction initializes the alignment register's contents and clears the flag bit. A series of aligning stores following the BBE_ZALIGN instruction will store one vector to memory per instruction. Note that the first store instruction of this series will perform a partial store because the flag bit was cleared by the BBE_ZALIGN instruction. Each subsequent store will perform a full 128-bit store because the first (and subsequent) store instructions set the flag bit. Finally, a flush instruction flushes out the last remaining bytes in the alignment register.

Once again, the design of the aligning store and flush instructions allows them to work even if the memory address is aligned to a 128-bit boundary. Specifically, if the address is aligned, the store instructions 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 instruction does nothing.

Note that these instructions move data between memory and the ConnX BBE16 registers if the memory addresses are not aligned to a vector boundary. However, these instructions do assume that the addresses are aligned to 16-bit scalar boundaries. If these instructions are used with addresses that are not aligned to 16-bit boundaries, the result is undefined. There are also 32-bit versions of these store and flush instructions that assume 32-bit scalar alignment.

The priming and flush instructions support only the immediate addressing mode, while the other instructions support all four addressing modes.

Following is code that zeros an aligning array, assuming the size is a multiple of 16 bytes.

```
void zero_array(...)( int n, short * a)
{
  int i;

  xb_vec8x16 *addr = (xb_vec8x16 *) &a[-8];
  valign A = BBE_ZALIGN128();

  for(i = 0; i < n; i = i + 8)
  {
    BBE_SA8X16S_IU(0, A, addr, 16);
  }
  BBE_SA8X16S_F(A, addr);

}
```

6.5 Vector Compacting Aligning Stores

BBE16 contains a set of store instructions that compact vectors at the same time as they do the store, according to patterns defined by the user. These instructions are BBE_SAC8X16S.U, BBE_SAC8X16U.U, BBE_SAC4X32S.U, BBE_SAC4X32U.U, and BBE_SAC128.F. There are also protos that define variations for various datatypes including fractional, complex, fractional complex, etc. as defined in the earlier tables.

The operation for BBE_SAC8X16S.U is:

```
BBE_SAC8X16S.U { in xb_vec8x20 a, inout valign b, inout xb_vec8x16
*addr, in int32 bv, in immediate idx }
```

Input vector a contains eight vector elements. Depending on an input mask that is contained in input variable `bv` (and selected by the input immediate `idx`), some of these input vector elements are stored to memory. Because we only want to write to memory when we have a full set of vector elements ready to be written, we use the alignment

register b as a temporary storage location for vector elements that have been selected to store to memory, but not written yet. We saturate any elements that go into the alignment register to 16-bit signed or unsigned format as appropriate.

If the operation is ready to write to memory, it uses the value in `addr` to determine where to write to. It also updates `addr` for the next BBE_SAC instruction. It is possible that vectors may only be written to the alignment register, or only be written to memory on any instantiation of the BBE_SAC instruction, depending on the data in the alignment register and the bit mask being used.

At the end of a sequence of BBE_SAC instructions, there may still be data left in the alignment buffer. The BBE_SAC128.F instruction is used to flush the alignment register. The alignment buffer can be initialized with the BBE_ZALIGN128 instruction.

6.6 Pre-Decrementing in Updating Loads and Stores

BBE16 has many loads and stores which do an address update as part of their operation, as described earlier in this chapter. The updating loads and stores do a pre-increment of the address before actually doing the load or store. Therefore, the initial address passed to the load or store must be pre-decremented before the first load and store by an appropriate amount — the size of the standard BBE16 vector load/store, which is 16 bytes. This can be done in several ways. One way is to cast an array reference to a pointer of the appropriate type and pre-decrement the pointer, which works if the pointer is to a 16-byte type. However, if the data type is a 32-byte type, a pointer decrement will decrement the address by 32 bytes, not 16.

The safe way to pre-decrement is to cast an address to a standard 8x16 vector type, then decrement by 1, for example:

```
xb_vec8x32 *pa = (xb_vec8x32*) (((xb_vec8x16*) &a[0]) - 1) ;
```

7. Implementation Methodology

The ConnX BBE16 DSP Engine is an optional coprocessor for the Xtensa LX (and later versions) core. ConnX BBE16 is provided as a set of four check box options in the Xplorer Processor Generator (XPG) interface in Xtensa Xplorer (XX). This option is included in the RC2010.2 release of Xtensa Xplorer, XPG, and Xtensa tools. This section includes guidelines for using XPG to configure a ConnX BBE16 coprocessor.

The last section in this chapter discusses synthesis and place-and-route.

7.1 Configuring a ConnX BBE16

Configuring a ConnX BBE16 in XPG is done by selecting one or more of the relevant check box options in the Xplorer Configuration editor, in the Instructions window under the category **ConnX BBE16 Coprocessor Family**:

- ConnX BBE16 DSP Baseband Engine
- 8-way vector divide
- 4-way reciprocal square root
- 16-way Despreader

The ConnX BBE16 DSP Engine option must be selected in order to select any of the other options, as shown in Figure 7–19. The other three options are complementary — you can select any combination from zero to all three of them. They add additional instructions and associated hardware to the ConnX BBE16 configuration.

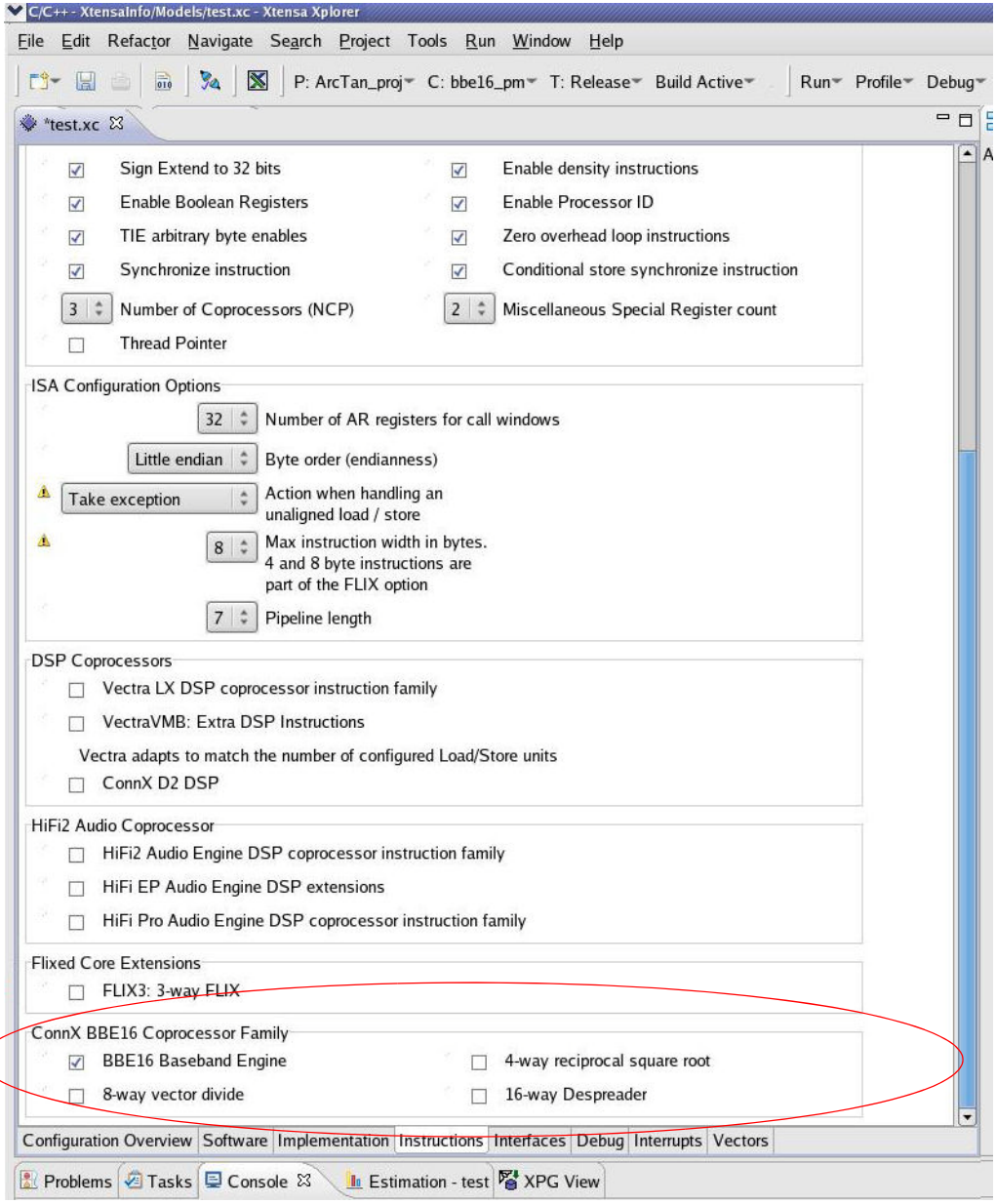


Figure 7–19. ConnX BBE16 Configuration Options in Xplorer

There are three optional configuration templates provided for ConnX BBE16, called XRC-B16LP, XRC-B16PM, and XRC_B16SA, which are described in Section 7.6. They provide useful starting points for a user who wishes 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 BBE16 template. Following are the steps to create your configuration in the Xtensa Xplorer IDE:

1. Click the Xplorer Quickstart Wizard button in the toolbar from any perspective. Then, select **Create a New Xtensa Configuration** and click **Next**.

   Alternatively, from the System Overview pane in the C/C++ perspective, right-click on **Configurations** and select **New Configuration**.
2. Select **Create new configuration with a new core ISA**. Enter the customer\username and password. Click **Test XPG Access**. When this succeeds, click **Next**.
3. Enter the configuration name and description. Select processor version LX4.0 or greater (although it is possible to select LX3.0 for the previous microarchitecture). Click **Finish**.
4. On the Configuration Summary pane, click **Edit** from the Configuration row of the Workspace Config column.
5. Click **Load a Configuration Template**, if you wish to use one of them as a starting point for your ConnX BBE16 configuration.
6. Click **Select** from standard templates. Then select the XRC-B16LP or XRC-B16PM template and click **OK**.

You can now customize the processor containing the ConnX BBE16 DSP Engine as described in the *Xtensa Development Tools Installation Guide*. As you are customizing the processor, keep in mind the following restrictions.

- The ConnX BBE16 option in the Instructions tab must be selected. In addition, you may select any of the other three ConnX BBE16 options.
- As ConnX BBE16 is always coprocessor number 1, the Number of Coprocessors must be at least two.
- ConnX BBE16 requires the selection of several options on the Instructions tab, as indicated below.
- The ConnX BBE16 option is incompatible with the some of the other DSP families as indicated blow.
- Because the ConnX BBE16 DSP Engine has some 64-bit instruction formats, the maximum instruction width must be eight bytes and a 64-bit instruction fetch is required. The data interfaces to local memories must be 128 bits.

Once a processor has been configured and downloaded it can be exercised in simulation.

7.2 XPG Estimation for ConnX BBE16 Size, Performance, and Power

The estimation of size (area), performance, and power for ConnX BBE16 is supported in the Xtensa Xplorer estimation tab. If you have a configuration definition without the basic ConnX BBE16 option or any of the other options selected, then the estimation will show values for size, performance and power. If you then select any of the ConnX BBE16 options or the base, the estimates will change to reflect an estimate of the extra hardware involved in BBE16: area, speed, and power. Keep in mind that this is an estimate only.

7.3 Basic ConnX BBE16 Characteristics

Some of the relevant configuration characteristics of the ConnX BBE16 coprocessor includes:

- ConnX BBE16 instruction set
- Boolean registers
- Little-endian byte order
- Two load/store units
- 128 bit local data memory interface
- 5 or 7-stage pipeline. However, note that this choice has several implications. A 5-stage pipeline will result in a smaller configuration, but the maximum speed that it is possible to synthesize and layout will be less than is possible with a 7-stage pipeline. In addition, large local memories (e.g., 128 KB or larger) may operate better with a 7-stage pipeline configuration that has extra memory access stages. Thus, depending on the application, consider these trade-offs. These are discussed in more detail in Section 7.8.
- 64-bit or 128-bit PIF interface. A 32-bit PIF interface is theoretically possible, but is not advised for ConnX BBE16, as it is intended for high performance applications; a 32-bit PIF is likely to be inadequate for optimal ConnX BBE16 performance.

7.4 Extending a ConnX BBE16 with User TIE

ConnX BBE16 can be extended with user-TIE defining new instructions. These can be assigned to the 24-bit regular instruction format, or you may wish to use the 64-bit FLIX instruction format that is available for user instruction additions. Due to encoding restrictions, there are actually 59 bits available in the user 64-bit instruction format.

We illustrate the process of defining the user FLIX instruction format and adding instructions to it using two simple TIE examples, as follows. These are also in an example `example_user_tie` file contained in the `bbe16_examples.xws` available with ConnX BBE16.

The first TIE example, bbe16_user_format.tie, defines a new 59-bit wide FLIX format, consisting of three FLIX slots. The following TIE template can be used to define a new FLIX format:

```
// Define new FLIX format for user instructions
// Use this format declaration:
//
length user_length64 64 {InstBuf[4:0] == 5'h1f}
format bbe_user_format user_length64 { user_slot0, user_slot1,
user_slot2 }
// For now, only assign NOP instruction to the three user slots

slot_opcodes user_slot0 {
    NOP
}

slot_opcodes user_slot1 {
    NOP
}

slot_opcodes user_slot2 {
    NOP
}
```

The second TIE example, bbe16_user_slots.tie, assigns some existing BBE16 instructions to those new FLIX slots. This of course is a pure example and does not define any new TIE instructions. Users are of course free to define new instructions and add them to the user FLIX slots.

```
/*************************************************************************
**
 *
 *   Example instruction assignment to user defined FLIX slots
 *
*************************************************************************
*/

/*************************************************************************
**
    SLOT 0
*************************************************************************
*/

slot_opcodes user_slot0 {
    ADD,
```

```
    ADDI,
    ADDMI,
    ADDX2,
    ADDX4,
    ADDX8,
    SUB,
    SUBX2,
    SUBX4,
    SUBX8,
    BBE_LV8X16S.I,
    BBE_LV8X16S.IU,
    BBE_LV8X16S.X,
    BBE_LV8X16S.XU,
    BBE_SV8X16S.I,
    BBE_SV8X16S.IU,
    BBE_SV8X16S.X,
    BBE_SV8X16S.XU,
    NOP
}

/*************************************************************************
SLOT 1
*************************************************************************
*/
slot_opcodes user_slot1 {
    BBE_MOV160,
    BBE_SEL8X20,
    BBE_SEL8X20I,
    BBE_ADD8X20,
    BBE_ADD4X40,
    BBE_SUB8X20,
    BBE_SUB4X40,
    NOP
}

/*************************************************************************
SLOT 2
*************************************************************************
*/
slot_opcodes user_slot2 {
    BBE_MOV160,
    BBE_LV8X16S.IU,
    BBE_LV4X32S.IU,
    BBE_SV8X16S.IU,
    BBE_SV4X32S.IU,
    NOP
}
```

7.4.1 Compiling User TIE

Following are instructions to compile the example user TIE:

1. Start with a ConnX BBE16 configuration in Xtensa Xplorer. The configuration should not already include user TIE.
2. Clone the configuration and attach TIE files. Select **bbe16_user_format.tie** and **bbe16_user_slots.tie**. Choose a name for the TDB.
3. Choose a name for the new config.
4. Compile the TDK for the new config with user TIE. **Note**: Generating cstub libraries requires more time than a standard TDK compile.
5. After compiling the TDK, use the new configuration to compile and run code.

7.4.2 User Warning to Ignore

Note: You can safely ignore the following TIE compiler warning:

Warning: (TIE_IMAP_ARG_NO_USE), In imap "imap_movi_int40_zero0", argument "o" is not used in output pattern

0 error(s), 1 warning(s) 0 message(s)

7.4.3 Name Space Restrictions for User TIE

There are restrictions in the name space that can be used in a customer user TIE file when using the ConnX BBE16 DSP Engine. The following TIE elements are reserved and cannot be used in TIE added to a configuration that includes ConnX BBE16. In general the prefix `BBE_`, and `xb_` are reserved for states, register files, ctypes, instructions and protos. More specifically:

State Names:

- `VSAR`
- `ROUND`
- `CBEGIN`
- `CEND`
- `TWIDDLEA`
- `TWIDDLEB`
- `TWIDDLEC`
- `FFTOP`
- `MODE`
- `RANGE`
- `BITREV_OFF`
- `BITREV_POS`

User Register Entries:

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

- 224-229
- 239
- 240-242
- 243
- 246-255

User Register Names:

- TWIDDLEA_0
- TWIDDLEA_1
- TWIDDLEA_2
- TWIDDLEA_3
- TWIDDLEA_4
- TWIDDLEB_0
- TWIDDLEB_1
- TWIDDLEB_2
- TWIDDLEB_3
- TWIDDLEB_4
- TWIDDLEC_0
- TWIDDLEC_1
- TWIDDLEC_2
- TWIDDLEC_3
- TWIDDLEC_4
- FFTCTRL
- VSAR
- ROUND_LO
- ROUND_HI
- CBEGIN
- CEND

Register File Names:

- vec
- valign
- sel

Coprocessor Number:

- 1

Ctype Names:

All ConnX BBE16 ctype names are reserved names.

They are all prefixed with "xb_" .

Operation Names:

All ConnX BBE16 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 BBE16 intrinsic (proto) names are reserved intrinsic names. These all have the prefix BBE_, or the prefix xb_ (for data type conversions protos).

TIE Function Names:

ConnX BBE16 uses a number of TIE functions, which you should not use for your own TIE functions. The following list of TIE function names should be avoided for user TIE:

add20a0, add20a1, add20a2, add20a3, add20b0, add20b1, add20b2, add20b3, add20c0, add20c1, add20c2, add20c3,

vec_add160, vec_shift0, vec_shift1, vec_shift2, vec_shift3, vec_shift4, vec_shift5, vec_shift6, vec_shift7

vec_xor, vec_cmp_to_0, vec_caddr_update,

bbe_sf_sel, bbe_pgen_bit, bbe_pgen_bit16, bbe_sf_mul0, bbe_sf_mul1, bbe_sf_mul2, bbe_sf_mul3, bbe_sf_mul4, bbe_sf_mul5, bbe_sf_mul6, bbe_sf_mul7, bbe_sf_mul8, bbe_sf_mul9, bbe_sf_mul10, bbe_sf_mul11, bbe_sf_mul12, bbe_sf_mul13, bbe_sf_mul14, bbe_sf_mul15, bbe_sf_mulacc0, bbe_sf_mulacc1, bbe_sf_mulacc2, bbe_sf_mulacc3, bbe_sf_mulacc4, bbe_sf_mulacc5, bbe_sf_mulacc6, bbe_sf_mulacc7

addradd, incr_bitrev, storesat20to16, storeusat20to16, load4x20, load2x40, store4x20, store2x40

s36_40, s35_40, s20_22, s36_37, s18_22, sr37to20, sr36to20, sr22to20

roundsr36, addroundsr36, range8x8, max3, max9x3, nsa20, srmask20to4, pack, sat20_16, pack40_20, vse16_20, abs40

div16x3_0, div16x3_1, div16x3_2, div16x3_3, div16x3_4, div16x3_5, div16x3_6, div16x3_7

despread_inv_fn, despread_elem_fn, cxadd5_fn, despread_fn, despread4_fn, despread4x4_fn

Semantic Names:

ConnX BBE16 uses a number of semantic names, which you should not use within your own TIE semantics. In particular, the semantic name `divide` is used by ConnX BBE16 and should not be used in user TIE.

7.5 XPG Configuration Options and Capabilities

The following tables list the configuration options and any constraints for the ConnX BBE16 DSP Engine.

Table 7–51. Simulation Modeling Capabilities

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

Table 7–52. Instruction Extensions Configuration Options and Constraints

Option	Allowed?
16AR	Yes
CLAMPS	Mandatory
MAC16	Yes
Extended L32R	No
MUL32 Implementation Selection	▪ None ▪ Iterative ▪ Fully pipelined ▪ 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
Density instructions supported (16-bit)	Yes

Table 7–53. Allowed Architecture Definition Configuration Options and Constraints

Option	Constraints
Load/Stores	Two
Action when handling an unaligned load/store	Take exception
Supported instruction widths	16, 24, & 64 required

Table 7–54. Instruction Width* Configuration Options and Constraints

Option	Allowed	Constraints
Pipeline length	Yes	Both five and seven 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 7–55. 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 7–56. Local Memories Configuration Options

Option	Allowed?
D-Cache supported	No
I-Cache supported	Yes
CAMMU	Yes
CAXLT	Yes
Full MMU (4KB page)	No
XLMI	No

Table 7–57. 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

7.6 Optional Configuration Templates for ConnX BBE16

There are three optional configuration templates provided in the XPG for ConnX BBE16, called XRC_B16LP, XRC_B16PM and XRC_B16SA. These optional templates contain a complete starting set of configuration options including ConnX BBE16 options for three reference cores, which are also referred to as XRC_B16LP, XRC_B16PM and XRC_B16SA. Effectively, these three reference cores bracket a wide range of configuration possibilities for ConnX BBE16. You can start with either of these three templates and modify configuration options, within the constraints defined by XPG, until you have a configuration you wish to build.

The XRC_B16LP, a low-end version of ConnX BBE16, only has the base BBE16 instruction option. This template does not select the 8-way vector divide, the 4-way reciprocal square root, or the 16-way despreader options in Xplorer. It does not have a 32 bit scalar multiplier or divider, and no instruction cache. It uses one 64 KB instruction memory and two 64 KB data memories. XRC_B16LP has a 64-bit PIF interface. It also has a seven-stage pipeline. For complete configuration information, see Section 7.6.1.

The XRC_B16PM, a high-end version of ConnX BBE16, selects all four Xplorer configuration options. Thus it has the 8-way vector divide, the 4-way reciprocal square root and the 16-way despreader. It also has:

- A high performance 32-bit scalar multiplier
- A 32-bit scalar divider,
- A 32KB instruction cache
- 128KB instruction memory
- Two 256KB data memories
- A128-bit PIF interface
- 7-stage pipeline

For complete configuration information for the XRC_B16PM, see Section 7.6.2

The XRC_B16SA is an even lower-end version of ConnX BBE16 than the XRC_B16LP. It has a 5-stage pipeline, one 32 KB instruction memory, one 64 KB data memory (with two load-store units, but no connection box (cbox)), and a reduced set of interrupts and vectors. It also has only the base BBE16 with none of the options. It also does not have a PIF interface, using local memories only.

For complete configuration information for the XRC_B16SA, see

You 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 your ConnX BBE16 configuration within the constraints and assumptions defined in XPG and the build process.

7.6.1 XRC_B16LP Configuration

The following tables contain configuration and instruction details for the XRC_B16LP configuration, which uses Xtensa ISA version LX4.0 and Xtensa Exception Architecture 2 (XEA2).

Table 7–58. XRC_B16LP Instruction Details

Feature	Configured	Details
16-bit MAC with 40-bit accumulator	No	
MUL16	Yes	
MUL32	No	
32-bit integer divider	No	
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)	---	Three
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	
Count of Load/Store units	---	Two

Table 7–58. XRC_B16LP Instruction Details (continued)

Feature	Configured	Details
Max instruction width (bytes)	---	Eight
L32R hardware support	Yes	Normal L32R
Pipeline length	---	Seven
FLIX3: 3-way FLIX	No	

Table 7–59. Configuration Options

Feature	Allowed?	Details
Vectra LX DSP coprocessor instruction family	No	No extra DSP (VectraVMB) instructions
ConnX D2 DSP	No	
HiFi2 Audio Engine DSP coprocessor instruction family	No	
SSP16 TIE Package	No	
BSP3 TIE Package	No	
Turbo16 TIE Package	No	
ConnX SSP TIE package	No	
ConnX BSP3 TIE package	No	
ConnX Turbo16 TIE package	No	
ConnX mimo	No	
ConnX 16tran_vec	No	
ConnX Viterbi_accelerator	No	
ConnX cbge	No	
ConnX VMB	No	
8-way vector divide	No	
4-way reciprocal square root	No	
16-way despreader	No	
Thread pointer	No	
GPIO32: 32-bit GPIO interface	No	
QIF32: 32-bit Queue interface	No	
Interrupts enabled	Yes	
Interrupt count	---	17 (see Table 7–60 for Interrupt details)
High-priority interrupts	---	Interrupt level count: Three
Medium-level interrupts	---	Highest medium-interrupt level: Two

Table 7–59. Configuration Options (continued)

Feature	Allowed?	Details
Timer count	---	Timer count: Two - Timer 0: 6 - Timer 1: 10
Byte ordering (endianness)	Yes	Little endian
Address registers available for call windows	---	32 available
Miscellaneous Special Register count	Yes	Maximum of two
Generate exception on unaligned load/store address	Yes	Take exception
Enable Processor Interface (PIF)	Yes	See Table 7–61 for PIF option details
Cache and memory interface widths		See Table 7–62 for details
Instruction cache selected	No	
Data Cache selected	No	
CBOX selected	Yes	
Debug	Yes	See Table 7–63 for details
Memory protection/MMU	Yes	Region protection (see Table 7–64 for details)
Inbound PIF request buffer depth	---	Eight
Local memory	--	See Table 7–65 for details
Vector configurations	--	See Table 7–66 for details
Relocatable vectors	Yes	See Table 7–67 for details
Target and CAD options	--	See Table 7–68 and Table 7–69 for details
Software target options	--	See Table 7–70 for details
Compatibility checking	--	See Table 7–71 for details
Variant	No	

Table 7–60. Interrupt Number Details

Interrupt Type	Priority Level
Int 0	ExtLevel / 1
Int 1	ExtLevel / 1
Int 2	ExtLevel / 1
Int 3	ExtLevel / 1
Int 4	ExtLevel / 1
Int 5	ExtLevel / 1
Int 6	Timer / 1
Int 7	Software / 1

Table 7–60. Interrupt Number Details (continued)

Interrupt Type	Priority Level
Int 8	ExtLevel / 2
Int 9	ExtLevel / 2
Int 10	Timer / 2
Int 11	Software / 2
Int 12	ExtEdge / 1
Int 13	ExtEdge / 1
Int 14	ExtEdge / 2
Int 15	ExtEdge / 2
Int 16 l	NMI / 4

Table 7–61. PIF Option Details

PIF Options	Allowed?	Details
Write buffer entries	Yes	Amount allowed: Eight
Enable PIF write responses	Yes	
Prioritize load before store	No	

Table 7–62. Cache and Memory Interface Widths

Feature	Width
Instruction fetch	64 bits
Data memory/cache	128 bits
PIF	64 bits

Table 7–63. Debug Details

Feature	Allowed?	Details
Data address breakpoint registers	Yes	Two
Instruction address breakpoint registers	Yes	Two
Debug interrupt level	Yes	Three
Trace port (address trace & pipeline status)	Yes	Cannot add data trace
On-Chip debug (OCD)	Yes	- Can use array of four Debug Instruction Registers (DIRs) - External Debug Interrupt available
Full scan	Yes	Cannot make latches transparent

Table 7–64. Memory Protection/MMU (Region Protection)

Feature	Start Address	Size
System RAM	0x60000000	64M
System ROM	0x50000000	16M

Table 7–65. Local Memory Details

Feature	Start Address	Size	Details
Instruction RAM	0x40000000	64K	[inbound PIF] [busy]
Instruction ROM	---	---	Not selected
Data RAM [0]	0x3fff0000	64K	[inbound PIF] [busy]
Data RAM [1]	0x3ffe0000	64K	[inbound PIF] [busy]
Data ROM	---	---	Not selected
XLMI	---	---	Not selected

Table 7–66. Vector Configuration Details

Feature	Start Address	Size
Reset vector	0x50000000	0x300
Kernel (stacked) exception vector	0x40000300	0x38
User (program) exception vector	0x40000340	0x38
Double-exception vector	0x400003c0	0x40
Window register overflow vector	0x40000000	0x178
Level 2 interrupt vector	0x40000180	0x38
Level 3 interrupt vector	0x400001c0	0x38
Level 4 interrupt vector (NMI vector)	0x400002c0	0x38

Table 7–67. Relocatable Vector Details

Feature	Details
Selected static vector set	Primary
Primary static vector group base address	0x50000000
Alternate static vector group base address	0x40000400
Alternate reset vector address	0x40000400
Default dynamic vector group base address (VECBASE)	0x40000000

Table 7–68. Target and CAD Options

Option	Allowed?	Details
RTL description	---	Verilog
Synthesis / P&R flow	---	Physical Synthesis, Route
Geometry / Process	---	45gs / Worst
Core speed	---	871 MHz
User-defined estimator library	---	Default (see Table 7–69 for scaling factor details)
Functional unit clock gating	Yes	
Global clock gating	Yes	
Register file implementation block	---	Flip-flops (Latches are deprecated)
Asynchronous reset	No	

Table 7–69. User-Defined Estimator Library Scaling Factors

Feature	Scaling Factor
User area	1.0
User speed	1.0
User dynamic power	1.0
User leak power	1.0
User area-to-gate	1.0

Table 7–70. Software Target Options

Option	Details
Extended L32R	Xtensa Tools should NOT use Extended L32R
Software ABI	Windowed
C libraries	newlib

Table 7–71. Compatibility Checking Features

Feature	Compatible?
Generic RTOS compatibility	No
Xtensa Model for Seamless compatibility	No
Target Linux compatibility	No

7.6.2 XRC_B16PM Configuration

The following tables contain configuration and instruction details for the XRC_B16PM reference core, which uses Xtensa ISA version LX4.0 and Xtensa Exception Architecture 2 (XEA2).

Table 7–72. Instruction Details

Feature	Allowed	Details
16-bit MAC with 40-bit accumulator	No	
MUL16	Yes	
MUL32	Yes	UHLH
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)	---	Three
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	
Count of Load/Store units	---	Two
Max instruction width (bytes)	---	Eight
L32R hardware support	Yes	Normal L32R
Pipeline length	---	Seven
FLIX3: 3-way FLIX	No	

Table 7–73. Configuration Options

Feature	Allowed?	Details
Vectra LX DSP coprocessor instruction family	No	No extra DSP (VectraVMB) instructions
ConnX D2 DSP	No	
HiFi2 Audio Engine DSP coprocessor instruction family	No	
SSP16 TIE Package	No	
BSP3 TIE Package	No	
Turbo16 TIE Package	No	
ConnX fft-fir	No	
ConnX SSP TIE package	No	
ConnX BSP3 TIE package	No	
ConnX Turbo16 TIE package	No	
ConnX mimo	No	
ConnX 16tran_vec	No	
ConnX Viterbi_accelerator	No	
ConnX cbge	No	
ConnX vmb	No	
8-way vector divide	Yes	
4-way reciprocal square root	Yes	
16-way despreader	Yes	
Thread pointer	No	
GPIO32: 32-bit GPIO interface	No	
QIF32: 32-bit Queue interface	No	
Interrupts enabled	Yes	
Interrupt count	---	17 (see Table 7–74 for details)
High-priority interrupts	---	Interrupt level count: Three
Medium-level interrupts	---	Highest Medium Interrupt Level: Two
Timer count	---	Timer count: Two - Timer 0: 6 - Timer 1: 10
Byte ordering (endianness)	Yes	Little endian
Address registers available for call windows	---	32 available
Miscellaneous Special Register count	Yes	Maximum of two
Generate exception on unaligned load/store address	Yes	Take exception
Enable Processor Interface (PIF)	Yes	See Table 7–75 for PIF option details
Cache and memory interface widths		See Table 7–76

Table 7–73. Configuration Options (continued)

Feature	Allowed?	Details
Instruction cache	Yes	32768 /64 2-way associativity No line locking No ICache memory error
Data cache	No	Not Selected
CBOX selected?	Yes	
Debug	Yes	See Table 7–77 for details.
Memory protection/MMU	Yes	Region Protection (see Table 7–78 for details)
Inbound PIF request buffer depth	---	Eight
Local memory	--	See Table 7–79 for details
Vector configurations	--	See Table 7–80 for details
Relocatable vectors	Yes	See Table 7–81 for details
Target and CAD options	--	See Table 7–82 and Table 7–83 for details
Software target options	--	See Table 7–84 for details
Compatibility checking	--	See Table 7–85 for details
Variant	No	

Table 7–74. Interrupt Details

Interrupt Type	Priority Level
Int 0	ExtLevel / 1
Int 1	ExtLevel / 1
Int 2l	ExtLevel / 1
Int 3	ExtLevel / 1
Int 4	ExtLevel / 1
Int 5	ExtLevel / 1
Int 6	Timer / 1
Int 7	Software / 1
Int 8	ExtLevel / 2
Int 9	ExtLevel / 2
Int 10	Timer / 2
Int 11	Software / 2
Int 12	ExtEdge / 1
Int 13	ExtEdge / 1

Table 7–74. Interrupt Details (continued)

Interrupt Type	Priority Level
Int 14	ExtEdge / 2
Int 15	ExtEdge / 2
Int 16	NMI / 4

Table 7–75. PIF Option Details

PIF Options	Allowed?	Details
Write buffer entries	Yes	Amount allowed: Eight
Enable PIF write responses	Yes	
Prioritize load before store	No	

Table 7–76. Cache and Memory Interface Widths

Feature	Width
Instruction fetch	64 bits
Data memory/cache	128 bits
PIF	64 bits
Interface to instruction cache	64 bits

Table 7–77. Debug Details

Feature	Allowed?	Details
Data address breakpoint registers	Yes	Two
Instruction address breakpoint registers	Yes	Two
Debug interrupt level	Yes	Three
Trace port (address trace & pipeline status)	Yes	Cannot add data trace
On-Chip debug (OCD)	Yes	- Can use array of four Debug Instruction Registers (DIRs) - External Debug Interrupt available
Full scan	Yes	Cannot make latches transparent

Table 7–78. Memory Protection/MMU (Region Protection)

Feature	Start Address	Size
System RAM	0x60000000	64M
System ROM	0x50000000	16M

Table 7–79. Local Memory Details

Feature	Start Address	Size	Details
Instruction RAM	0x40000000	64K	[inbound PIF] [busy]
Instruction ROM	---	---	Not selected
Data RAM [0]	0x3fff0000	64K	[inbound PIF] [busy]
Data RAM [1]	0x3ffe0000	64K	[inbound PIF] [busy]
Data ROM	---	---	Not selected
XLMI	---	---	Not selected

Table 7–80. Vector Configuration Details

Feature	Start Address	Size
Reset vector	0x50000000	0x300
Kernel (stacked) exception vector	0x40000300	0x38
User (program) exception vector	0x40000340	0x38
Double-exception vector	0x400003c0	0x40
Window register overflow vector	0x40000000	0x178
Level 2 interrupt vector	0x40000180	0x38
Level 3 interrupt vector	0x400001c0	0x38
Level 4 interrupt vector (NMI vector)	0x400002c0	0x38

Table 7–81. Relocatable Vector Details

Feature	Details
Selected static vector set	Primary
Primary static vector group base address	0x50000000
Alternate static vector group base address	0x40000400
Alternate reset vector address	0x40000400
Default dynamic vector group base address (VECBASE)	0x40000000

Table 7–82. Target and CAD Options

Option	Allowed?	Details
RTL description	---	Verilog
Synthesis / P&R flow	---	Physical Synthesis, Route
Geometry / Process	---	45gs / Worst
Core speed	---	871 MHz
User-defined estimator library	---	Default (see Table 7–83 for scaling factor details)
Functional unit clock gating	Yes	
Global clock gating	Yes	
Register file implementation block	---	Flip-flops (Latches are deprecated)
Asynchronous reset	No	

Table 7–83. User Defined Estimator Library Scaling Factors

Feature	Scaling Factor
User area	1.0
User speed	1.0
User dynamic power	1.0
User leak power	1.0
User area-to-gate	1.0

Table 7–84. Software Target Options

Option	Details
Extended L32R	Xtensa Tools should NOT use Extended L32R
Software ABI	Windowed
C libraries	newlib

Table 7–85. Compatibility Checking Features

Feature	Compatible?
Generic RTOS compatibility	Yes
Xtensa Model for Seamless compatibility	No
Target Linux compatibility	No

7.6.3 XRC_B16SA Configuration

The following tables contain configuration and instruction details for the XRC_B16SA configuration, which uses Xtensa ISA version LX4.0 and Xtensa Exception Architecture 2 (XEA2).

Table 7–86. XRC_B16SA Instruction Details

Feature	Configured	Details
16-bit MAC with 40-bit accumulator	No	
MUL16	Yes	
MUL32	No	
32-bit integer divider	No	
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)	---	Three
Enable density instructions	Yes	
Enable processor ID	Yes	
Zero-overhead loop instructions	Yes	
Synchronize instruction	Yes	
Conditional store synchronize instruction	No	
TIE arbitrary byte enables	Yes	
Count of Load/Store units	---	Two
Max instruction width (bytes)	---	Eight
L32R hardware support	Yes	Normal L32R
Pipeline length	---	Five
FLIX3: 3-way FLIX	No	

Table 7–87. Configuration Options

Feature	Allowed?	Details
Vectra LX DSP coprocessor instruction family	No	No extra DSP (VectraVMB) instructions
ConnX D2 DSP	No	
HiFi2 Audio Engine DSP coprocessor instruction family	No	
SSP16 TIE Package	No	
BSP3 TIE Package	No	
Turbo16 TIE Package	No	
ConnX SSP TIE package	No	
ConnX BSP3 TIE package	No	
ConnX Turbo16 TIE package	No	
ConnX mimo	No	
ConnX 16tran_vec	No	
ConnX Viterbi_accelerator	No	
ConnX cbge	No	
ConnX VMB	No	
8-way vector divide	No	
4-way reciprocal square root	No	
16-way despreader	No	
Thread pointer	No	
GPIO32: 32-bit GPIO interface	No	
QIF32: 32-bit Queue interface	No	
Interrupts enabled	Yes	
Interrupt count	---	10 (see Table 7–88 for Interrupt details)
High-priority interrupts	---	Interrupt level count: Two
Medium-level interrupts	No	
Timer count	---	Timer count: One - Timer 0: 9
Byte ordering (endianness)	Yes	Little endian
Address registers available for call windows	---	16 available
Miscellaneous Special Register count	Yes	Maximum of two
Generate exception on unaligned load/store address	Yes	Take exception
Enable Processor Interface (PIF)	No	No PIF
Cache and memory interface widths		See Table 7–89 for details
Instruction cache selected	No	

Table 7–87. Configuration Options (continued)

Feature	Allowed?	Details
Data Cache selected	No	
CBOX selected	No	
Debug	Yes	See Table 7–90 for details
Memory protection/MMU	Yes	Region protection
Inbound PIF request buffer depth	No	Not Applicable: no PIF
Local memory	--	See Table 7–91 for details
Vector configurations	--	See Table 7–92 for details
Relocatable vectors	No	
Target and CAD options	--	See Table 7–93 and Table 7–94 for details
Software target options	--	See Table 7–95 for details
Compatibility checking	--	See Table 7–96 for details
Variant	No	

Table 7–88. Interrupt Number Details

Interrupt Type	Priority Level
Int 0	ExtLevel / 1
Int 1	ExtLevel / 1
Int 2	ExtLevel / 1
Int 3	ExtLevel / 1
Int 4	ExtEdge / 1
Int 5	ExtEdge / 1
Int 6	ExtEdge / 1
Int 7	ExtEdge / 1
Int 8	Software /12
Int 9	Timer / 1

Table 7–89. Cache and Memory Interface Widths

Feature	Width
Instruction fetch	64 bits
Data memory/cache	128 bits
PIF	NONE

Table 7–90. Debug Details

Feature	Allowed?	Details
Data address breakpoint registers	Yes	One
Instruction address breakpoint registers	Yes	One
Debug interrupt level	Yes	Two
Trace port (address trace & pipeline status)	No	Cannot add data trace
On-Chip debug (OCD)	Yes	- Cannot use array of four Debug Instruction Registers (DIRs) - External Debug Interrupt available
Full scan	Yes	Cannot make latches transparent

Table 7–91. Local Memory Details

Feature	Start Address	Size	Details
Instruction RAM	0x5fff0000	32K	[busy]
Instruction ROM	---	---	Not selected
Data RAM	0x5ffe0000	64K	[busy]
Data ROM	---	---	Not selected
XLMI	---	---	Not selected

Table 7–92. Vector Configuration Details

Feature	Start Address	Size
Reset vector	0x5fff0000	0x2e0
Kernel (stacked) exception vector	0x5fff0304	0x1c
User (program) exception vector	0x5fff0324	0x1c
Double-exception vector	0x5fff0344	0x1c
Level 2 interrupt vector	0x5fff0600	0x1c

Table 7–93. Target and CAD Options

Option	Allowed?	Details
RTL description	---	Verilog
Synthesis / P&R flow	---	Physical Synthesis, Route
Geometry / Process	---	45gs / Worst

Table 7–93. Target and CAD Options

Option	Allowed?	Details
Core speed	---	725 MHz
User-defined estimator library	---	Default (see Table 7–94 for scaling factor details)
Functional unit clock gating	Yes	
Global clock gating	Yes	
Register file implementation block	---	Flip-flops (Latches are deprecated)
Asynchronous reset	No	

Table 7–94. User-Defined Estimator Library Scaling Factors

Feature	Scaling Factor
User area	1.0
User speed	1.0
User dynamic power	1.0
User leak power	1.0
User area-to-gate	1.0

Table 7–95. Software Target Options

Option	Details
Extended L32R	Xtensa Tools should NOT use Extended L32R
Software ABI	Call0
C libraries	newlib

Table 7–96. Compatibility Checking Features

Feature	Compatible?
Generic RTOS compatibility	No
Xtensa Model for Seamless compatibility	No
Target Linux compatibility	No

7.7 Synthesis and Place-and-Route

When the ConnX BBE16 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 Implementation and Verification Guide*.

For timing closure between synthesis and place-route, Cadence recommends using a layout aware synthesis tool. Because of the data path-intensive nature of the ConnX BBE16 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.

7.7.1 CadSetup.file Variable Selection for Optimal Performance

The CadSetup file is the primary interface to the Cadence EDA flows. In its out-of-the-box configuration, it is set to achieve high frequency performance, while still allowing for successful RTL-to-gates formal verification. As a result, some performance-enhancing features are disabled. To achieve the highest possible post-route frequency when using a Synopsys-based flow, the following variables should be modified (read the documentation provided above each variable for further explanation):

- Target_MaxTransition 0.2 (should be aggressive - example is for 45gs)
- Target_ClockTransition 0.2
- Syn_UseAdvancedFeatures 1
- DC_CriticalRange 0.9 (set to 75% of clock period)
- DC_CompileUltraAdaptiveRetiming 1
- ICC_CriticalRange 0.9 (set to 75% of clock period)
- ICC_PinOpt 1
- ICC_UseZroute 1
- ICC_WriteSPEF 1
- Fplan_CoreUtilization 0.60 (could be set even higher)

The other variables may be left at their default settings.

7.7.2 DRC Fanout Constraint Suggestions

As process technologies continue to scale, the ratio of wire delay to cell delay continues to increase. This is particularly acute at the 45nm node and below. The advanced physically-aware flow supported by Cadence (using Design Compiler (DC) Topographical for synthesis, followed by IC Compiler (ICC) for placement and routing) does a good job of correlating timing between synthesis and P&R. However, the ConnX BBE16 design has sections of very high fanout; this causes greater than expected performance degrada-

tion between DC and ICC. As a result, we recommend adding the additional DRC constraint of maximum fanout to both the DC and ICC tool scripts. A fanout constraint within the range of 16-24 has been experimentally determined to be optimal for the ConnX BBE16 configuration in the TSMC 45gs process technology.

To apply the DRC fanout constraint, edit the following files prior to running DC or ICC (the following uses an example of 24):

1. In the `xda/scripts/dc/DC_generic_cons.tcl` file, after the set_max_transition command, add the following command:

```
set_max_fanout 24.0 [current_design]
```

2. Additionally, in the `xda/scripts/icc/ICC_flow.cmd` file, after the suppress_errors command, add the following commands:

```
set_max_fanout 24.0 [current_design]
set psynopt_high_fanout_legality_limit 24
set_buffer_opt_strategy -effort high
```

Experiment to find the optimal fanout constraint for your particular deep-sub-micron library. Conversely, this fanout constraint does not seem to be required at the larger (that is, 65nm and above) process nodes.

7.8 ConnX BBE16 and Memory Floorplanning Suggestions

Cadence provides an automated, push-button, standard cells layout flow for ConnX BBE16. Before proceeding with floorplanning ConnX BBE16 with memory, it is imperative that out-of-the-box push-button flow is exercised for ConnX BBE16. The advantage is that it will provide accurate measure of flop-to-flop timing inside ConnX BBE16 without extraneous factors such as specific memory placement or floorplan dimensions that may cause layout tool to severely degrade the performance of ConnX BBE16. 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 BBE16 size to determine the dimensions of floorplan for ConnX BBE16 with its associated memories.

ConnX BBE16 interfaces with instruction and data memories. The instruction memories interface with the Program Counter and Instruction Fetch (PCIF) unit in ConnX BBE16. The data memories interface with the Load/Store (LS) unit in ConnX BBE16. 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 floorplan, 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 BBE16 may be configured to 5 or 7. With large memories, it is advisable to configure ConnX BBE16 with 7-pipeline stages so that a full cycle is available to register the memory data. For 5-stage ConnX BBE16, the access paths (data from memory to ConnX BBE16) are likely to be timing critical. This will likely result in insertion of buffers and sizing up of gates on the access paths. 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 BBE16 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 BBE16, 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 floorplan 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 guideline for interleaving the banks is important for 5-stage pipeline as memory access paths are likely to be timing critical. The data from ConnX BBE16 to memory (memory write data) and memory address signals need to be routed to each memory bank. For a 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 BBE16 floorplan.

In addition to considering memory interfaces, it is important to allocate sufficient floorplan space for ConnX BBE16. Our experience suggests that a starting utilization of 60% is a reasonable preliminary estimate for floorplanning experiments. The starting utilization refers to the floorplan area allocated to ConnX BBE16 standard cells divided by the post-synthesis cell area. The starting utilization ratio need to be adjusted down if floorplan 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 floorplan creation, the design of ConnX BBE16 floorplan is partly art and partly science. The first level consideration must take into account the following:

- Size and aspect ratio of floorplan space allocated to ConnX BBE16 and associated memories
- Data and instruction memory configuration
- Relative size of ConnX BBE16 and its associated memories

At this stage it is important to estimate the memory sizes and floorplan real estate available to place ConnX BBE16 standard cells. It is important that sufficient notch-free and channel-free space is available for placement of ConnX BBE16 standard cells.

The next level of detail must evaluate banking structure to implement ConnX BBE16 memory configuration. The banking may be along "data bits" (for example, 4 x 32-bits banks to implement a 128-bit interface) 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 floorplan dimensions.

The logical connectivity of instruction and data memories to ConnX BBE16 units must be exploited to place the memory instances. It is important to differentiate type of memories from ConnX BBE16 perspective and place memory of a specific type contiguously in the floorplan. The interleaving of memory bank instances must be considered to improve the congestion characteristics of the floorplan.

For optimal congestion-free ConnX BBE16 floorplan that meets the timing, the importance of early, floorplan exploration experiments can not be overemphasized.

7.9 Mapping ConnX BBE16 to FPGA

Experiments have been carried out mapping ConnX BBE16 to FPGA. For a particular use of a Xilinx Virtex 6 LX760, using the ConnX_BBE16_XRC_B16PM configuration (the reference core with all three ConnX BBE16 options plus other options as discussed in this chapter), the following results were obtained:

Number of Slice Registers:	26,769 out of 948,480	2%
Number of Slice LUTs:	108,841 out of 474,240	22%
Number of occupied Slices:	33,358 out of 118,560	28%
Number of RAMB36E1/FIFO36E1s:	82 out of 720	11%
Number of RAMB18E1/FIFO18E1s:	252 out of 1,440	23%

Due to heavy routing constraints, the achievable frequency is around 40Mhz.

A ConnX BBE16 configuration with fewer or different options may occupy fewer LUTs and slices, but this is something that your need to experiment with. ConnX BBE16 may map to other FPGAs with lower resources, but you will need to estimate or experiment with that using their own particular configuration.

A. On-Line ISA, Protos, and Other ConnX BBE16 Information

Xtensa Xplorer offers a number of on-line Instruction Set Architecture (ISA) documentation references in HTML format for Xtensa configurations, including BBE16. 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")
- 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.

A.1 Protos

As part of its programming model, ConnX BBE16 defines a number of instruction prototypes (protos) for use by the compiler in code generation, for use by programmers with other data types than the generic instructions support, and to provide compatibility with related programming models and DSPs such as ConnX Vectra LX (see Appendix B). For programmers, the most important use of protos is to provide alternative instruction mappings for various data types. If the data types are compatible, one instruction can support several variations without any extra hardware cost. For example, protos allow vector operations to support xb_vec8x20 (integer), xb_vec8xq4_15 (fixed-point), xb_4xc20 (complex integer) and xb_4xcq4_15 (complex fixed-point). Thus a single BBE_ADD8X20 or BBE_ADD4X40 instruction can support many different operations including both data type variants and larger sizes of operands (e.g. 8X40, which is mapped into two BBE_ADD4X40 instructions).

ConnX BBE16 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. All protos can be accessed via the online Xplorer documentation described above.

A.1.1 Extract Protos

Extract protos are used to do casting in the BBE16 programming model. Following is a list of the extract protos. 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 BBE16 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).

```
proto BBE_EXTRACT2XC40_FROM4XC40_HI { out xb_vec2xc40 hi, in xb_vec4xc40 b }{}{
BBE_MOV160 hi, b->hi;
}
proto BBE_EXTRACT2XC40_FROM4XC40_LO { out xb_vec2xc40 lo, in xb_vec4xc40 b }{}{
BBE_MOV160 lo, b->lo;
}
proto BBE_EXTRACT2XCQ9_30_FROM4XCQ9_30_HI { out xb_vec2xcq9_30 hi, in xb_vec4xcq9_30 b }{}{
BBE_MOV160 hi, b->hi;
}
proto BBE_EXTRACT2XCQ9_30_FROM4XCQ9_30_LO { out xb_vec2xcq9_30 lo, in xb_vec4xcq9_30 b }{}{
BBE_MOV160 lo, b->lo;
}
proto BBE_EXTRACT4X40_FROM8X40_HI { out xb_vec4x40 hi, in xb_vec8x40 b }{}{
BBE_MOV160 hi, b->hi;
}
proto BBE_EXTRACT4X40_FROM8X40_LO { out xb_vec4x40 lo, in xb_vec8x40 b }{}{
BBE_MOV160 lo, b->lo;
}
proto BBE_EXTRACT4XQ9_30_FROM8XQ9_30_HI { out xb_vec4xq9_30 hi, in xb_vec8xq9_30 b }{}{
BBE_MOV160 hi, b->hi;
}
proto BBE_EXTRACT4XQ9_30_FROM8XQ9_30_LO { out xb_vec4xq9_30 lo, in xb_vec8xq9_30 b }{}{
BBE_MOV160 lo, b->lo;
}
proto BBE_EXTRACTB2B4 { out xtbool2 bb2, in xtbool4 bb4, in immediate imm1x1 }{}{
BBE_EXTRACTB2B4 bb2, bb4, imm1x1 + 0;
}
proto BBE_EXTRACTB4B8 { out xtbool4 ba4, in xtbool8 ba8, in immediate imm1x1 }{}{
BBE_EXTRACTB4B8 ba4, ba8, imm1x1 + 0;
}
```

```
proto BBE_EXTRACTI_FROM2X4XC20 { out xb_vec8x20 im, in xb_vec4xc20 hi, in xb_vec4xc20 lo
 }{xb_vec8x20 tmp, vsel extractoddx20}{
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 extractoddx20, tmp, 12;
BBE_SEL8X20 im, hi, lo, extractoddx20;
}
proto BBE_EXTRACTI_FROM2X4XCQ4_15 { out xb_vec8xq4_15 im, in xb_vec4xcq4_15 hi, in
 xb_vec4xcq4_15 lo }{xb_vec8x20 tmp, vsel extractoddx20}{
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 extractoddx20, tmp, 12;
BBE_SEL8X20 im, hi, lo, extractoddx20;
}
proto BBE_EXTRACTI_FROM4XC40 { out xb_vec4x40 im, in xb_vec4xc40 c }{xb_vec4x40 tmp, vsel
 extractoddx40}{
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 extractoddx40, tmp, 12;
BBE_SEL8X20 im, c->hi, c->lo, extractoddx40;
}
proto BBE_EXTRACTI_FROM4XCQ9_30 { out xb_vec4xq9_30 im, in xb_vec4xcq9_30 c }{xb_vec4x40 tmp,
 vsel extractoddx40}{
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 extractoddx40, tmp, 12;
BBE_SEL8X20 im, c->hi, c->lo, extractoddx40;
}
proto BBE_EXTRACTI_FROMC20 { out xb_int20 im, in xb_c20 c }{}{
BBE_REP8X20 im, c, 1;
}
proto BBE_EXTRACTI_FROMC40 { out xb_int40 im, in xb_c40 c }{}{
BBE_REP4X40 im, c, 1;
}
proto BBE_EXTRACTI_FROMCQ4_15 { out xb_q4_15 im, in xb_cq4_15 c }{}{
BBE_REP8X20 im, c, 1;
}
proto BBE_EXTRACTI_FROMCQ9_30 { out xb_q9_30 im, in xb_cq9_30 c }{}{
BBE_REP4X40 im, c, 1;
}
proto BBE_EXTRACTR_FROM2X4XC20 { out xb_vec8x20 re, in xb_vec4xc20 hi, in xb_vec4xc20 lo
 }{xb_vec8x20 tmp, vsel extractevenx20}{
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 extractevenx20, tmp, 8;
BBE_SEL8X20 re, hi, lo, extractevenx20;
}
proto BBE_EXTRACTR_FROM2X4XCQ4_15 { out xb_vec8xq4_15 re, in xb_vec4xcq4_15 hi, in
 xb_vec4xcq4_15 lo }{xb_vec8x20 tmp, vsel extractevenx20}{
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 extractevenx20, tmp, 8;
BBE_SEL8X20 re, hi, lo, extractevenx20;
}
proto BBE_EXTRACTR_FROM4XC40 { out xb_vec4x40 re, in xb_vec4xc40 c }{xb_vec4x40 tmp, vsel
 extractevenx40}{
BBE_MOVVI8X20 tmp, 49;
```

```
BBE_SEL4V8X20 extractevenx40, tmp, 8;
BBE_SEL8X20 re, c->hi, c->lo, extractevenx40;
}
proto BBE_EXTRACTR_FROM4XCQ9_30 { out xb_vec4xq9_30 re, in xb_vec4xcq9_30 c }{xb_vec4x40 tmp,
  vsel extractevenx40}{
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 extractevenx40, tmp, 8;
BBE_SEL8X20 re, c->hi, c->lo, extractevenx40;
}
proto BBE_EXTRACTR_FROMC20 { out xb_int20 re, in xb_c20 c }{}{
BBE_REP8X20 re, c, 0;
}
proto BBE_EXTRACTR_FROMC40 { out xb_int40 re, in xb_c40 c }{}{
BBE_REP4X40 re, c, 0;
}
proto BBE_EXTRACTR_FROMCQ4_15 { out xb_q4_15 re, in xb_cq4_15 c }{}{
BBE_REP8X20 re, c, 0;
}
proto BBE_EXTRACTR_FROMCQ9_30 { out xb_q9_30 re, in xb_cq9_30 c }{}{
BBE_REP4X40 re, c, 0;
}
```

Extract Protos 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];  // Q12.20
xb_c40 hcxabs = hcin[i] * ~ hcin[i];  // Q12.20
// Extract real magnitude squared for channel
xb_int40 habs = BBE_EXTRACTR_FROMC40(hcxabs);   // Q12.20
...........
```

Note that BBE16 scalar types are used here, as the compiler does automatic vectorization and inference.

A.1.2 Combine Protos

Combine protos are used to do casting in the BBE16 programming model. Following is a list of the combine protos. 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 immedi-ates.

There are some special source type names used that are not normal BBE16 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.

```
proto BBE_COMBINE2X4XC20_FROMIR { out xb_vec4xc20 chi, out xb_vec4xc20 clo, in xb_vec8x20 im, in
  xb_vec8x20 re }{xb_vec8x20 tmp, vsel itlv20lower, vsel itlv20upper}{
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 clo, im, re, itlv20lower;
BBE_SEL4V8X20 itlv20upper, tmp, 0;
BBE_SEL8X20 chi, im, re, itlv20upper;
}
proto BBE_COMBINE2X4XC20_FROMIZ { out xb_vec4xc20 chi, out xb_vec4xc20 clo, in xb_vec8x20 im
  }{xb_vec8x20 tmpzero, xb_vec8x20 tmp, vsel itlv20lower, vsel itlv20upper}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 clo, im, tmpzero, itlv20lower;
BBE_SEL4V8X20 itlv20upper, tmp, 0;
BBE_SEL8X20 chi, im, tmpzero, itlv20upper;
}
proto BBE_COMBINE2X4XC20_FROMZR { out xb_vec4xc20 chi, out xb_vec4xc20 clo, in xb_vec8x20 re
  }{xb_vec8x20 tmpzero, xb_vec8x20 tmp, vsel itlv20lower, vsel itlv20upper}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 clo, tmpzero, re, itlv20lower;
BBE_SEL4V8X20 itlv20upper, tmp, 0;
BBE_SEL8X20 chi, tmpzero, re, itlv20upper;
}
proto BBE_COMBINE2X4XCQ4_15_FROMIR { out xb_vec4xcq4_15 chi, out xb_vec4xcq4_15 clo, in
  xb_vec8xq4_15 im, in xb_vec8xq4_15 re }{xb_vec8x20 tmp, vsel itlv20lower, vsel itlv20upper}{
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 clo, im, re, itlv20lower;
BBE_SEL4V8X20 itlv20upper, tmp, 0;
BBE_SEL8X20 chi, im, re, itlv20upper;
}
proto BBE_COMBINE2X4XCQ4_15_FROMIZ { out xb_vec4xcq4_15 chi, out xb_vec4xcq4_15 clo, in
  xb_vec8xq4_15 im }{xb_vec8x20 tmpzero, xb_vec8x20 tmp, vsel itlv20lower, vsel itlv20upper}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 clo, im, tmpzero, itlv20lower;
BBE_SEL4V8X20 itlv20upper, tmp, 0;
BBE_SEL8X20 chi, im, tmpzero, itlv20upper;
}
```

```
proto BBE_COMBINE2X4XCQ4_15_FROMZR { out xb_vec4xcq4_15 chi, out xb_vec4xcq4_15 clo, in
  xb_vec8xq4_15 re }{xb_vec8x20 tmpzero, xb_vec8x20 tmp, vsel itlv20lower, vsel itlv20upper}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 clo, tmpzero, re, itlv20lower;
BBE_SEL4V8X20 itlv20upper, tmp, 0;
BBE_SEL8X20 chi, tmpzero, re, itlv20upper;
}
proto BBE_COMBINE4XC40_FROMIR { out xb_vec4xc40 c, in xb_vec4x40 im, in xb_vec4x40 re
  }{xb_vec4x40 tmp, vsel itlv40lower, vsel itlv40upper}{
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c->lo, im, re, itlv40lower;
BBE_SEL4V8X20 itlv40upper, tmp, 0;
BBE_SEL8X20 c->hi, im, re, itlv40upper;
}
proto BBE_COMBINE4XC40_FROMIZ { out xb_vec4xc40 c, in xb_vec4x40 im }{xb_vec4x40 tmp, xb_vec4x40
  tmpzero, vsel itlv40lower, vsel itlv40upper}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c->lo, im, tmpzero, itlv40lower;
BBE_SEL4V8X20 itlv40upper, tmp, 0;
BBE_SEL8X20 c->hi, im, tmpzero, itlv40upper;
}
proto BBE_COMBINE4XC40_FROMZR { out xb_vec4xc40 c, in xb_vec4x40 re }{xb_vec4x40 tmp, xb_vec4x40
  tmpzero, vsel itlv40lower, vsel itlv40upper}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c->lo, tmpzero, re, itlv40lower;
BBE_SEL4V8X20 itlv40upper, tmp, 0;
BBE_SEL8X20 c->hi, tmpzero, re, itlv40upper;
}
proto BBE_COMBINE4XCQ9_30_FROMIR { out xb_vec4xcq9_30 c, in xb_vec4xq9_30 im, in xb_vec4xq9_30
  re }{xb_vec4x40 tmp, vsel itlv40lower, vsel itlv40upper}{
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c->lo, im, re, itlv40lower;
BBE_SEL4V8X20 itlv40upper, tmp, 0;
BBE_SEL8X20 c->hi, im, re, itlv40upper;
}
proto BBE_COMBINE4XCQ9_30_FROMIZ { out xb_vec4xcq9_30 c, in xb_vec4xq9_30 im }{xb_vec4x40 tmp,
  xb_vec4x40 tmpzero, vsel itlv40lower, vsel itlv40upper}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c->lo, im, tmpzero, itlv40lower;
BBE_SEL4V8X20 itlv40upper, tmp, 0;
BBE_SEL8X20 c->hi, im, tmpzero, itlv40upper;
```

```
}
proto BBE_COMBINE4XCQ9_30_FROMZR { out xb_vec4xcq9_30 c, in xb_vec4xq9_30 re }{xb_vec4x40 tmp,
  xb_vec4x40 tmpzero, vsel itlv40lower, vsel itlv40upper}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c->lo, tmpzero, re, itlv40lower;
BBE_SEL4V8X20 itlv40upper, tmp, 0;
BBE_SEL8X20 c->hi, tmpzero, re, itlv40upper;
}
proto BBE_COMBINEC20_FROMIR { out xb_c20 c, in xb_int20 im, in xb_int20 re }{xb_vec8x20 tmp,
  vsel itlv20lower}{
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 c, im, re, itlv20lower;
}
proto BBE_COMBINEC20_FROMIZ { out xb_c20 c, in xb_int20 im }{xb_vec8x20 tmp, xb_vec8x20 tmpzero,
  vsel itlv20lower}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 c, im, tmpzero, itlv20lower;
}
proto BBE_COMBINEC20_FROMZR { out xb_c20 c, in xb_int20 re }{xb_vec8x20 tmp, xb_vec8x20 tmpzero,
  vsel itlv20lower}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 c, tmpzero, re, itlv20lower;
}
proto BBE_COMBINEC40_FROMIR { out xb_c40 c, in xb_int40 im, in xb_int40 re }{xb_vec4x40 tmp,
  vsel itlv40lower}{
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c, im, re, itlv40lower;
}
proto BBE_COMBINEC40_FROMIZ { out xb_c40 c, in xb_int40 im }{xb_vec4x40 tmp, xb_vec4x40 tmpzero,
  vsel itlv40lower}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c, im, tmpzero, itlv40lower;
}
proto BBE_COMBINEC40_FROMZR { out xb_c40 c, in xb_int40 re }{xb_vec4x40 tmp, xb_vec4x40 tmpzero,
  vsel itlv40lower}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c, tmpzero, re, itlv40lower;
}
```

```
proto BBE_COMBINECQ4_15_FROMIR { out xb_cq4_15 c, in xb_q4_15 im, in xb_q4_15 re }{xb_vec8x20
 tmp, vsel itlv20lower}{
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 c, im, re, itlv20lower;
}
proto BBE_COMBINECQ4_15_FROMIZ { out xb_cq4_15 c, in xb_q4_15 im }{xb_vec8x20 tmp, xb_vec8x20
 tmpzero, vsel itlv20lower}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 c, im, tmpzero, itlv20lower;
}
proto BBE_COMBINECQ4_15_FROMZR { out xb_cq4_15 c, in xb_q4_15 re }{xb_vec8x20 tmp, xb_vec8x20
 tmpzero, vsel itlv20lower}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 48;
BBE_SEL4V8X20 itlv20lower, tmp, 4;
BBE_SEL8X20 c, tmpzero, re, itlv20lower;
}
proto BBE_COMBINECQ9_30_FROMIR { out xb_cq9_30 c, in xb_q9_30 im, in xb_q9_30 re }{xb_vec4x40
 tmp, vsel itlv40lower}{
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c, im, re, itlv40lower;
}
proto BBE_COMBINECQ9_30_FROMIZ { out xb_cq9_30 c, in xb_q9_30 im }{xb_vec4x40 tmp, xb_vec4x40
 tmpzero, vsel itlv40lower}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c, im, tmpzero, itlv40lower;
}
proto BBE_COMBINECQ9_30_FROMZR { out xb_cq9_30 c, in xb_q9_30 re }{xb_vec4x40 tmp, xb_vec4x40
 tmpzero, vsel itlv40lower}{
BBE_MOVVI8X20 tmpzero, 0;
BBE_MOVVI8X20 tmp, 49;
BBE_SEL4V8X20 itlv40lower, tmp, 4;
BBE_SEL8X20 c, tmpzero, re, itlv40lower;
}
```

Combine Protos 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.

```
// 8-way inverse for scaling, note divide is integer division
// Also convert to complex multiplicand to use in later steps
```

```
xb_vec8x20 hinv = BBE_DIV8X32S(numinv, numinv, hsabs);    // Q5.10 (2^20/2^15)
xb_vec4xc20 hcxhi, hcxlo;
BBE_COMBINE2X4XC20_FROMZR(hcxhi, hcxlo, hinv);
```

A.2 Operator Protos

The on-line Xplorer based proto 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. Consider BBE_ADD8X20: This intrinsic expects arguments of type xb_vec8x20, but sometimes it's more convenient to use variables of type xb_vec8x16.

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. Therefore, to support, for example the "+" operator on one variable of type xb_vec8x20 and one variable of type xb_vec8x16, it is necessary to define such a proto. The resultant intrinsic is necessary for operator overloading, but there is no advantage in calling it directly.

B. Vectra LX Compatibility

ConnX BBE16 subsumes ConnX Vectra LX, however there are two differences: First, a Vectra LX multiply operation will multiply two variables of type vec8x20, to produce a vec8x40. Each of two vec8x40 registers will contain the even half or the odd half of the results. For complex multiplication, the results will be split into real and imaginary registers. ConnX BBE16 multiply operations produce xb_vec8x40 results, with all data in normal, interleaved order.

Note that native BBE16 multiplies will give better performance than Vectra LX operations, since they produce all 8 results in one operation.

Secondly, in Vectra LX, 160-bit wide register types are stored to memory in a different data order than ConnX BBE16.

ConnX BBE16 includes the Vectra LX instruction set, but with a set of BBE_ instruction names. These operations are provided for backwards compatibility with Vectra LX code.To simplify migration, a Vectra LX compatibility set of protos are included in the header file: <xtensa/tie/xt_vectralx.h>.

Therefore, our recommendation for users migrating to ConnX BBE16 is to convert their Vectra LX code to BBE16 data types and instructions and, except for very short term work, avoid using the compatibility layers. After using the compatibility layers to test their code, users should switch to native BBE16 data types and instructions.

The following table provides mappings between Vectra LX instruction names and the proto names supporting these in ConnX BBE16.

Table B-97. Vectra LX Instructions Mapped to ConnX BBE16

Vectra LX Inst	BBE16 Proto	Vectra LX Inst	BBE16 Proto
ABS20	BBE_ABS8X20	IMULS18	BBE_MULS18.I
ABS40	BBE_ABS4X40	LALIGN.I	BBE_LALIGN128.I
ABS8X40	BBE_ABS8X40	LE20	BBE_LE8X20
ADD20	BBE_ADD8X20	LE40	BBE_LE4X40
ADD40	BBE_ADD4X40	LSEL.I	BBE_LSEL8X4.I
ADD8X40	BBE_ADD8X40	LSS16.I	BBE_LS8X16S.I
AND160	BBE_AND160	LSS16.IU	BBE_LS8X16S.IU
AND20	BBE_AND8X20	LSS16.X	BBE_LS8X16S.X
AND40	BBE_AND4X40	LSS16.XU	BBE_LS8X16S.XU
AND8X40	BBE_AND8X40	LSS32.I	BBE_LS4X32S.I
EQ20	BBE_EQ8X20	LSS32.IU	BBE_LS4X32S.IU

Table B-97. Vectra LX Instructions Mapped to ConnX BBE16 (continued)

Vectra LX Inst	BBE16 Proto
EQ40	BBE_EQ4X40
ESHFT20	BBE_ESHFT8X20
ESHFT40	BBE_ESHFT4X40
IMUL18	BBE_MUL18.I
IMULA18	BBE_MULA18.I
IMULR18	BBE_MULR18.I
LSU32.I	BBE_LS4X32U.I
LSU32.IU	BBE_LS4X32U.IU
LSU32.X	BBE_LS4X32U.X
LSU32.XU	BBE_LS4X32U.XU
LT20	BBE_LT8X20
LT40	BBE_LT4X40
LVA.CP	BBE_LA128.CP
LVA.P	BBE_LA128.P
LVH.I	BBE_LVH8X4.I
LVS16.CU	BBE_LV8X16S.CU
LVS16.I	BBE_LV8X16S.I
LVS16.IU	BBE_LV8X16S.IU
LVS16.X	BBE_LV8X16S.I
LVS16.XU	BBE_LV8X16S.XU
LVS16A.CP	BBE_LA8X16S.CP
LVS16A.CP	BBE_LA8X16S.CP
LVS16A.CU	BBE_LA8X16S.CU
LVS16A.I	BBE_LA8X16S.I
LVS16A.IU	BBE_LA8X16S.IU
LVS16A.P	BBE_LA8X16S.P
LVS16A.P	BBE_LA8X16S.P
LVS16A.X	BBE_LA8X16S.X
LVS16A.XU	BBE_LA8X16S.XU
LVS32.CU	BBE_LV4X32S.CU
LVS32.I	BBE_LV4X32S.I
LVS32.IU	BBE_LV4X32S.IU
LVS32.X	BBE_LV4X32S.X
LVS32.XU	BBE_LV4X32S.XU
LSS32.X	BBE_LS4X32S.X
LSS32.XU	BBE_LS4X32S.XU
LSU16.I	BBE_LS8X16U.I
LSU16.IU	BBE_LS8X16U.IU
LSU16.X	BBE_LS8X16U.X
LSU16.XU	BBE_LS8X16U.XU
LVS32A.P	BBE_LA4X32S.P
LVS32A.X	BBE_LA4X32S.X
LVS32A.XU	BBE_LA4X32S.XU
LVU16.CU	BBE_LV8X16U.CU
LVU16.I	BBE_LV8X16U.I
LVU16.IU	BBE_LV8X16U.IU
LVU16.X	BBE_LV8X16U.X
LVU16.XU	BBE_LV8X16U.XU
LVU16A.CP	BBE_LA8X16U.CP
LVU16A.CP	BBE_LA8X16U.CP
LVU16A.CU	BBE_LA8X16U.CU
LVU16A.I	BBE_LA8X16U.I
LVU16A.IU	BBE_LA8X16U.IU
LVU16A.P	BBE_LA8X16U.P
LVU16A.P	BBE_LA8X16U.P
LVU16A.X	BBE_LA8X16U.X
LVU16A.XU	BBE_LA8X16U.XU
LVU32.CU	BBE_LV4X32U.CU
LVU32.ILVU32.IU	BBE_LV4X32U.IU
LVU32.X	BBE_LV4X32U.X
LVU32.XU	BBE_LV4X32U.XU
LVU32A.CP	BBE_LA4X32U.CP
LVU32A.CP	BBE_LA4X32U.CP
LVU32A.CU	BBE_LA4X32U.CU
LVU32A.I	BBE_LA4X32U.I
LVU32A.IU	BBE_LA4X32U.IU
LVU32A.P	BBE_LA4X32U.P
LVU32A.P	BBE_LA4X32U.P

Table B-97. Vectra LX Instructions Mapped to ConnX BBE16 (continued)

Vectra LX Inst	BBE16 Proto
LVS32A.CP	BBE_LA4X32S.CP
LVS32A.CP	BBE_LA4X32S.CP
LVS32A.CU	BBE_LA4X32S.CU
LVS32A.I	BBE_LA4X32S.I
LVS32A.IU	BBE_LA4X32S.IU
LVS32A.P	BBE_LA4X32S.P
MAXB20	BBE_MAXB8X20
MAXB40	BBE_MAXB4X40
MERGE_VEC8X40	BBE_JOINV8X40V4X40
MIN20	BBE_MIN8X20
MIN40	BBE_MIN4X40
MIN8X40	BBE_MIN8X40
MINB20	BBE_MINB8X20
MINB40	BBE_MINB4X40
MOV160	BBE_MOV160
MOV20	BBE_MOV8X20
MOV40	BBE_MOV4X40
MOV8X40	BBE_MOV8X40
MOV8XAR32	BBE_MOVAV8X40
MOV8XVR40	BBE_MOVVA8X40
MOVAB4	BBE_MOVBA4
MOVAB8	BBE_MOVBA8
MOVAR16	BBE_MOVAV16
MOVAR32	BBE_MOVAV32
MOVBA4	BBE_MOVAB4
MOVBA8	BBE_MOVAB8
MOVF20	BBE_MOV8X20F
MOVF40	BBE_MOV4X40F
MOVT20	BBE_MOV8X20T
MOVT40	BBE_MOV4X40T
MOVVR20	BBE_MOVVA20
MOVVR40	BBE_MOVVA32
MSEL	BBE_MSEL8X4
MUL18.0	BBE_MUL18.0

Vectra LX Inst	BBE16 Proto
LVU32A.X	BBE_LA4X32U.X
LVU32A.XU	BBE_LA4X32U.XU
MALIGN	BBE_MALIGN128
MAX20	BBE_MAX8X20
MAX40	BBE_MAX4X40
MAX8X40	BBE_MAX8X40
MULS18.1	BBE_MULS18.1
MULSGN20	BBE_MULSGN8X20
MULSGN40	BBE_MULSGN4X40
MULSGN8X40	BBE_MULSGN8X40
NAND160	BBE_NAND160
NAND20	BBE_NAND8X20
NAND40	BBE_NAND4X40
NAND8X40	BBE_NAND8X40
NEG20	BBE_NEG8X20
NEG40	BBE_NEG4X40
NEG8X40	BBE_NEG8X40
OR160	BBE_OR160
OR20	BBE_OR8X20
OR40	BBE_OR4X40
OR8X40	BBE_OR8X40
PACK40	BBE_PACK8X40.01
PACK8FX40	BBE_PACKQ8XQ9_30
PACK8IX40	BBE_PACKS8X40
PACK8X40	BBE_PACK8X40.01
RADD20	BBE_RADD8X20
RADD40	BBE_RADD4X40
RADD8X40	BBE_RADD8X40
REDADD16	BBE_RADDA16V8X20
REDADD32	BBE_RADDA32V4X40
REDADD8X32	BBE_RADDA32V8X40
REDMAX16	BBE_RMAXA16V8X20
REDMAX32	BBE_RMAXA32V4X40
REDMIN16	BBE_RMINA16V8X20

Table B-97. Vectra LX Instructions Mapped to ConnX BBE16 (continued)

Vectra LX Inst	BBE16 Proto
MUL18.1	BBE_MUL18.1
MULA18.0	BBE_MULA18.0
MULA18.1	BBE_MULA18.1
MULR18.0	BBE_MULR18.0
MULR18.1	BBE_MULR18.1
MULS18.0	BBE_MULS18.0
RMULS18	BBE_MULS18.R
SALIGN.I	BBE_SALIGN128.I
SEL	BBE_SEL8X20
SEL40	BBE_SEL4X40
SELI	BBE_SEL8X20I
SELI40	BBE_SEL4X40I
SLL20	BBE_SLL8X20
SLL40	BBE_SLL4X40
SLLI20	BBE_SLLI8X20
SLLI40	BBE_SLLI4X40
SLLI8X40	BBE_SLLI8X40
SLS20	BBE_SLS8X20
SLS40	BBE_SLS4X40
SLS8X40	BBE_SLS8X40
SRA20	BBE_SRA8X20
SRA40	BBE_SRA4X40
SRAI20	BBE_SRAI8X20
SRAI40	BBE_SRAI4X40
SRAI8X40	BBE_SRAI8X40
SSEL.I	BBE_SSEL8X4.I
SSS16.I	BBE_SS8X16S.I
SSS16.IU	BBE_SS8X16S.IU
SSS16.X	BBE_SS8X16S.X
SSS16.XU	BBE_SS8X16S.XU
SSS32.I	BBE_SS4X32S.I
SSS32.IU	BBE_SS4X32S.IU
SSS32.X	BBE_SS4X32S.X
SSS32.XU	BBE_SS4X32S.XU

Vectra LX Inst	BBE16 Proto
REDMIN32	BBE_RMAXA32V4X40
REP20	BBE_REP8X20
REP40	BBE_REP4X40
RMUL18	BBE_MUL18.R
RMULA18	BBE_MULA18.R
RMULR18	BBE_MULR18.R
SSU32.X	BBE_SS4X32U.X
SSU32.XU	BBE_SS4X32U.XU
SUB20	BBE_SUB8X20
SUB40	BBE_SUB4X40
SUB8X40	BBE_SUB8X40
SVA.F	BBE_SA128.F
SVH.I	BBE_SVH8X4.I
SVL.I	BBE_SVL8X16.I
SVS16.CU	BBE_SV8X16S.CU
SVS16.I	BBE_SV8X16S.I
SVS16.IU	BBE_SV8X16S.IU
SVS16.X	BBE_SV8X16S.X
SVS16.XU	BBE_SV8X16S.XU
SVS16A.CU	BBE_SA8X16S.CU
SVS16A.F	BBE_SA8X16S.F
SVS16A.F	BBE_SA8X16S.F
SVS16A.I	BBE_SA8X16S.I
SVS16A.IU	BBE_SA8X16S.IU
SVS16A.X	BBE_SA8X16S.X
SVS16A.XU	BBE_SA8X16S.XU
SVS32.CU	BBE_SV4X32S.CU
SVS32.I	BBE_SV4X32S.I
SVS32.IU	BBE_SV4X32S.IU
SVS32.X	BBE_SV4X32S.X
SVS32.XU	BBE_SV4X32S.XU
SVS32A.CU	BBE_SA4X32S.CU
SVS32A.F	BBE_SA4X32S.F
SVS32A.F	BBE_SA4X32S.F

Table B-97. Vectra LX Instructions Mapped to ConnX BBE16 (continued)

Vectra LX Inst	BBE16 Proto	Vectra LX Inst	BBE16 Proto
SSU16.I	BBE_SS8X16U.I	SVS32A.I	BBE_SA4X32S.I
SSU16.IU	BBE_SS8X16U.IU	SVS32A.IU	BBE_SA4X32S.IU
SSU16.X	BBE_SS8X16U.X	SVS32A.X	BBE_SA4X32S.X
SSU16.XU	BBE_SS8X16U.XU	SVS32A.XU	BBE_SA4X32S.XU
SSU32.I	BBE_SS4X32U.I	SVU16.CU	BBE_SV8X16U.CU
SSU32.IU	BBE_SS4X32U.IU	SVU16.I	BBE_SV8X16U.I
SVU16.IU	BBE_SV8X16U.IU	SVU32A.CU	BBE_SA4X32U.CU
SVU16.X	BBE_SV8X16U.X	SVU32A.F	BBE_SA4X32U.F
SVU16.XU	BBE_SV8X16U.XU	SVU32A.F	BBE_SA4X32U.F
SVU16A.CU	BBE_SA8X16U.CU	SVU32A.I	BBE_SA4X32U.I
SVU16A.F	BBE_SA8X16U.F	SVU32A.IU	BBE_SA4X32U.IU
SVU16A.F	BBE_SA8X16U.F	SVU32A.X	BBE_SA4X32U.X
SVU16A.I	BBE_SA8X16U.I	SVU32A.XU	BBE_SA4X32U.XU
SVU16A.IU	BBE_SA8X16U.IU	VZERO20	BBE_ZERO8X20
SVU16A.X	BBE_SA8X16U.X	VZERO40	BBE_ZERO4X40
SVU16A.XU	BBE_SA8X16U.XU	VZERO8X40	BBE_ZERO8X40
SVU32.CU	BBE_SV4X32U.CU	XOR160	BBE_XOR160
SVU32.I	BBE_SV4X32U.I	XOR20	BBE_XOR8X20
SVU32.IU	BBE_SV4X32U.IU	XOR40	BBE_XOR4X40
SVU32.X	BBE_SV4X32U.X	XOR8X40	BBE_XOR8X40
SVU32.XU	BBE_SV4X32U.XU	ZALIGN	BBE_ZALIGN128

B.1 Vectra LX Specific Complex and Real Vector Multiplies and Multiply-Adds

The Vectra LX specific partitioned multiplies for real and complex (for example MUL18.0, MUL18.1, IMUL18, IMUL18, etc.) can still be used in ConnX BBE16 under their new names or via protos for the old names (subject to the advice in the previous section). However, BBE16 offers via its complex and real multiply set, better methods of doing multi-way real and complex multiplies in many cases. Therefore, consult the BBE16 multiply instruction set before continuing to use an older Vectra LX SIMD multiply as it may not be as efficient. The ISA HTML comments that these instructions are for Vectra LX compatibility and that users should consult the newer instructions first. This section contains some details on these older, now discouraged, instructions.

For some real operations, there are dual instructions for the even elements and the odd elements of the source vectors. The operations to compute even elements have a `.0` suffix, while the operations for the odd elements have a `.1` suffix. For example, the even and odd multiplies would operate on the source elements as in the following figures.

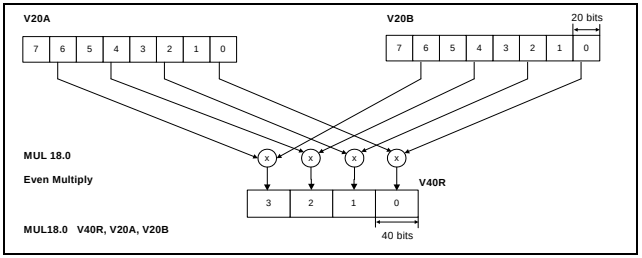


Figure B–20. MUL18 Even Data Path (MUL18.0)

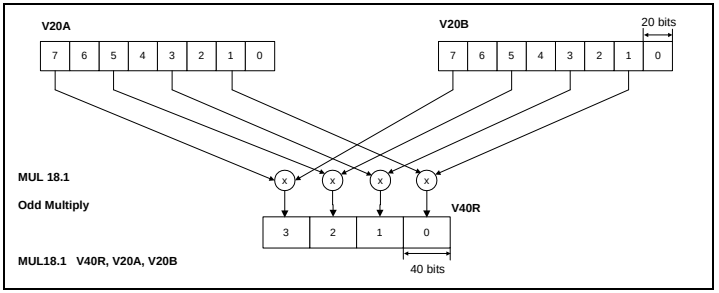


Figure B–21. MUL18 Odd Data Path (MUL18.1)

If necessary, the PACK40 instruction converts the 40-bit precision elements into 20-bit precision elements for the algorithm being implemented. It will also interleave the vector elements, thus the inputs are 4x40 vectors and the output is a single 8x20 vector. ConnX Vectra did not interleave the results, whereas the native ConnX BBE16 multiplies do.

Complex vectors have interleaved real and imaginary elements. A ConnX BBE16 register vec8x20 will accommodate four complex elements. For complex operations, in the ConnX Vectra compatibility ISA, there are dual instructions, one of which computes the real part of the operation and the other that computes the imaginary part of the operation. Figure B–22 and Figure B–23 show the real and imaginary multiply, and operations which compute real and complex results on four complex pairs as a single operation.

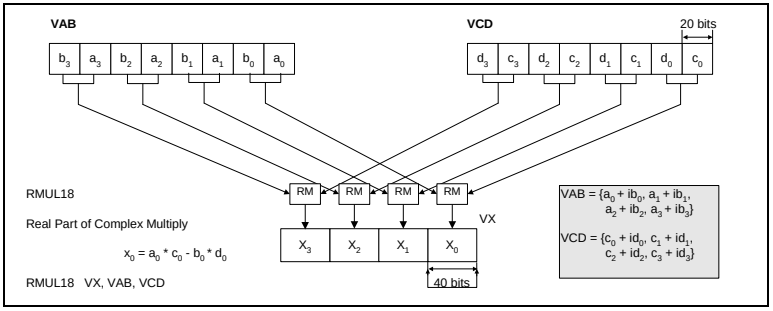


Figure B–22. RMUL18 Data Path

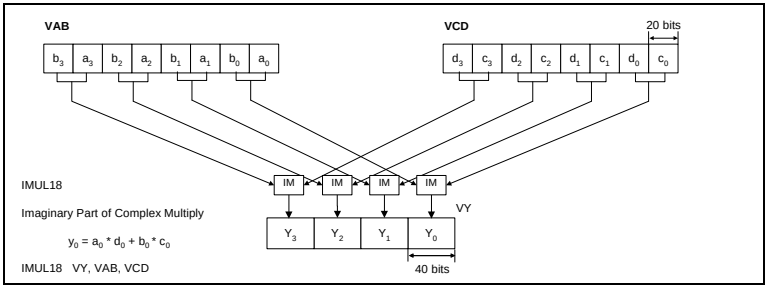


Figure B–23. IMUL18 Data Path

Figure B–24 shows the details of the computation of the real portion of the complex multiply using RMUL18. This figure also includes some of the details of the computational units' schedule at the vector element level.

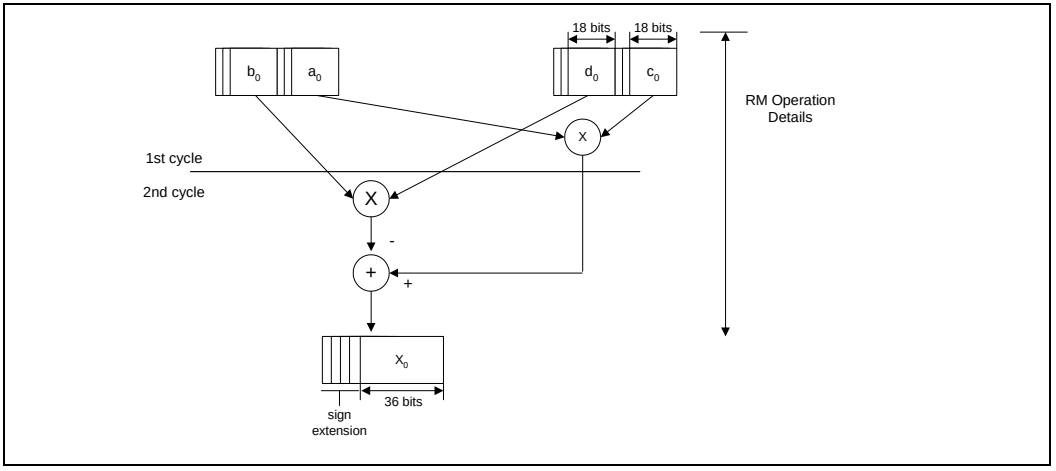


Figure B–24. RMUL18 Details

Figure B–25 shows the details of the computation of the imaginary portion of the complex multiply using IMUL18. It includes some details of the computational units' schedule.

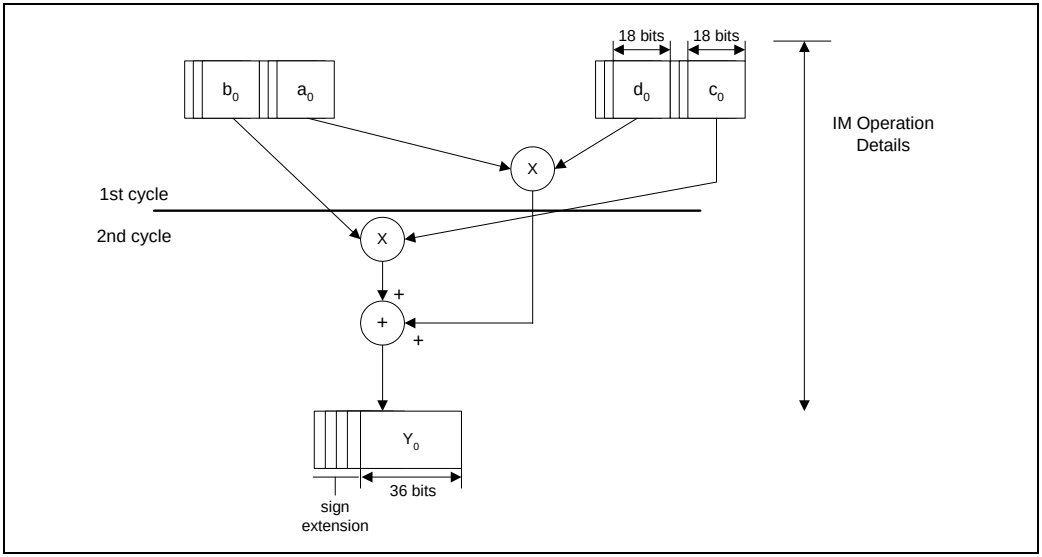


Figure B–25. IMUL18 Details

The PACK40 instruction is used to interleave the results back into the normal complex (imaginary, real) order.

Index

Numerics
32-Bit Select operand 100

A
Addressing
  bit-reversed 94
Addressing modes 122
Addressing, circular 122
Aligning Loads and Stores 123
Architecture
  behavior 9
  overview 2
Arithmetic
  real and complex multiplies 30
Automatic type conversion 60
Automatic vectorization 64, 65

B
BFR4 operation 87
Bit-reversed addressing 94
Block floating point 38

C
C type names 134
CadSetup.file 151
Circular addressing 122
Compatibility with Vectra LX 167
Complex 2x2 matrix multiply multi mode 109
Complex conjugate multiply instructions 39
Configuration options 136
Configuration templates 138
Configuring a ConnX BBE16 127
Converting
  guarded types to unguarded 61
  int16 and int32 60
  unguarded types to guarded 61
Cosine computation 45

D
Data types 57
  details 20
  mapped to the vector register file 17
Definitions
  notations xi
  terms xi
Despread operation 50
Display formats for Xtensa Xplorer 63
Divide instructions 54
DRC fanout 151

E
Explicit instrinsics 72

F
Fanout 151
Features of ConnX BBE16 9
FFT
  acceleration 81
  control register 81
  dynamic range 86
  indexed store with update 96
  inner loop C code 86
  instruction usage 92
  normalization of results 85
  operations and instructions 84
  structure 82
FFT_ADD3MUL operation 89
Fixed point values 15
  arithmetic 17
FLIX slots and formats 55
Floating point, block 38
Floorplanning 152
FPGA, mapping ConnX BBE16 to 154

G
Guard bits 9
Guarded types
  converting from unguarded 60
  converting to unguarded 61

I
Include files 2
Inferencing and operator overloading 66
Inner loop C code 86
Installing a ConnX BBE16 configuration 2
Instruction formats 3
Instruction name categories 10
Instruction naming conventions 10
Instruction set
  concepts 10
  HTML page 13
  overview 4
int16 and int32, converting 60
Intrinsics/protos 10

L
Load /Store
intrinsic names....121
Load/Store
units....73
Loads/Stores
aligning....123
Local data RAMs....73

M
Manual vectorization....71
-mcoproc....76
Memory floorplanning....152
Memory storing data....59
MODE shift....86
Move instruction....35
Multiply instructions....32, 97
real and complex arithmetic....30
Multiply-accumulate instructions....97

N
Name categories for instructions....10
Name space restrictions....133
Naming conventions for instructions....10

O
Operator overloading....64
and inferencing....66

P
Place-and-route....151
Preface....xi
Programming
ConnX BBE16....57
models....6
styles....65, 72
Protos
list for ConnX BBE16....157
Protos/intrinsics....10

R
Radix-4
DIF FFT result order....93
Real 2x2 matrix multiply mode....107
Reciprocal square root instruction....54
Related Documents....xii
Restrictions, name space for TIE....133

S
Scalar data types
mapped to vector register file....17
Select instructions....34
Setup polynomial evaluation instruction....40
Synthesis....151

T
TI C6x intrinsics....76
TIE....130
compiling....133
name space restrictions....133
Timing closure....151

U
Unguarded types
converting from guarded....61
converting to guarded....60

V
Vector data types
mapped to vector register file....19
Vector initialization....35
Vector polynomial evaluation instructions....39
Vector register file and data types....17
Vectorization
automatic....6, 64, 65
manual....71
Vectra LX compatibility....167

X
XCC vectorization....6
XRC_B16PM configuration....145
Xtensa Xplorer
display formats....63
xtensa_root....1
xt-xcc compiler....65