



Vision P6

User's Guide

For Cadence Tensilica DSP IP Cores

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

© 2016 Cadence Design Systems, Inc.
All Rights Reserved

This publication is provided "AS IS." Cadence Design Systems, Inc. (hereafter "Cadence") does not make any warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Information in this document is provided solely to enable system and software developers to use our processors. Unless specifically set forth herein, there are no express or implied patent, copyright or any other intellectual property rights or licenses granted hereunder to design or fabricate Cadence integrated circuits or integrated circuits based on the information in this document. Cadence does not warrant that the contents of this publication, whether individually or as one or more groups, meets your requirements or that the publication is error-free. This publication could include technical inaccuracies or typographical errors. Changes may be made to the information herein, and these changes may be incorporated in new editions of this publication.

Cadence, the Cadence logo, Allegro, Assura, Broadband Spice, CDNLIVE!, Celtic, Chipestimate.com, Conformal, Connections, Denali, Diva, Dracula, Encounter, Flashpoint, FLIX, First Encounter, Incisive, Incyte, InstallScape, NanoRoute, NC-Verilog, OrCAD, OSKit, Palladium, PowerForward, PowerSI, PSpice, Purespec, Puresuite, Quickcycles, SignalStorm, Sigrity, SKILL, SoC Encounter, SourceLink, Spectre, Specman, Specman-Elite, SpeedBridge, Stars & Strikes, Tensilica, TripleCheck, TurboXim, Vectra, Virtuoso, VoltageStorm Xplorer, Xtensa, and Xtreme are either trademarks or registered trademarks of Cadence Design Systems, Inc. in the United States and/or other jurisdictions.

OSCI, SystemC, Open SystemC, Open SystemC Initiative, and SystemC Initiative are registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission. All other trademarks are the property of their respective holders.

Issue Date:10/2016

RG-2016.4

Build 380292

PD-16-2104-10-01

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

Contents

List of Tables.....	vii
List of Figures.....	ix
1 Introduction.....	11
Purpose of this User's Guide.....	12
Installation Overview.....	12
Changes from the Previous Version	12
New Features.....	12
Changes to the Document.....	12
Vision P6 Architecture Overview.....	13
System Interfaces.....	16
2 Features.....	17
2.1 Register Files and States.....	18
2.2 Vision P6 Instruction Formats.....	20
Vision P6 Instruction Format Structure.....	21
Instruction Formats.....	21
2.3 Architecture of Vision P6 Operation.....	28
2.4 Operation Naming Convention.....	29
Load/Store Operations.....	29
ALU Operations.....	30
Multiply Operations.....	31
Shorthand Notation.....	32
2.5 Vision P6 C Types.....	33
2.6 Gather/Scatter.....	37
Gather/Scatter Error Handling.....	38
Gather/Scatter ECC Support.....	40
Gather/Scatter ECC Error Logging.....	41
Gather/Scatter Debug Data Breakpoints.....	42
Gather/Scatter and Wait Mode.....	43
2.7 Option Packages.....	43
Vision P6 Single-Precision Vector Floating Point Package.....	43
Vision P6 Half-Precision Vector Floating Point Package.....	44
Vision P6 Histogram Package.....	44
3 Programming and Optimization Techniques.....	47
3.1 Programming Overview.....	48
3.2 Programming in Prototypes.....	50
Move Protos.....	51
Operator Overload Protos.....	51
3.3 Xtensa Xplorer Display Format Support.....	52
3.4 Programming Styles.....	52

Auto-vectorization.....	53
Operator Overloading.....	54
Auto-vectorization Examples.....	54
Manual Vectorization Example.....	56
N-way Programming Model.....	57
3.5 Using the Two Local Data RAMs and Two Load/Store Units.....	58
3.6 Other Compiler Switches.....	59
3.7 Guidelines for Optimizing Code for Vision P6.....	60
Optimizing Gather/Scatter.....	60
3.8 Guidelines for Using iDMA in Vision P6.....	61
3.9 Linker Support Package (LSP) Considerations for Vision P6 Software Development.....	62
4 Standard DSP Operations by Type.....	65
4.1 DSP Operations by Type.....	66
Load & Store Operations.....	66
Multiplication Operations.....	74
Division Operations.....	81
Vector Compression and Expansion.....	83
Arithmetic Operations.....	85
Bitwise Logical Operations.....	88
Bit Manipulation Operations.....	90
Comparison Operations.....	91
Shifts.....	92
Rotations.....	94
NSA Operations.....	94
Reduction Operations.....	94
Pack Operations.....	95
Unpack Operations.....	96
Select Operations	96
Dual Select Operations.....	121
Shuffle Operations.....	121
4.2 Move Operations.....	122
Moves between (narrow) Vector Registers and Address Registers.....	122
Moves between (narrow) Vector Registers.....	123
Move between Alignment Registers.....	123
Move between Vector Boolean Registers.....	123
Moves between AR or Core Boolean Register (BR) and Vector Boolean Register (vbool).....	123
Moves from Immediate.....	124
Moves between Wide Vector Register and Narrow Vector Register.....	124
Moves between Wide Vector Register and Narrow Vector Register.....	128
Move between Wide Vector Registers.....	131
5 Gather/Scatter Operations.....	133
5.1 Gather/Scatter Operations Overview.....	134

5.2 Gather Registers.....	136
5.3 Gather/Scatter Request Ordering.....	136
5.4 Gather/Scatter Address Aliases.....	138
5.5 Gather/Scatter Operation Descriptions.....	138
5.6 Gather/Scatter Programming Interface.....	141
6 Floating-Point Operations.....	143
6.1 Vision P6 Floating-Point Features.....	145
Vision P6 Register Files Related to Floating-Point.....	145
Vision P6 Architecture Behavior Related to Floating-Point.....	145
Binary Floating-Point Values.....	146
Half-Precision Data.....	146
Single-Precision Data.....	147
Maximum Possible Error in the Unit in the Last Place (ULP).....	147
Encodings of Infinity and Not a Number (NaN).....	148
Scalar Float Data-types Mapped to the Vector Register File.....	148
Vector Float Data-types Mapped to the Vector Register File.....	149
Floating-Point Data Typing.....	149
Floating-Point Rounding and Exception Controls.....	149
Floating-Point Status/Exception Flags.....	150
Floating-Point Status and Control Flags Alternate Access.....	151
6.2 Vision P6 Floating-Point Operations.....	152
Half/Single Arithmetic Operations.....	153
Floating-Point Conversion Operations.....	153
Integer to and from Half/Single Conversion Operations.....	153
Half/Single to Integral Value Rounding Operations.....	154
Classification, Comparison and Min/Max Operations.....	155
Half/Single Division and Square Root Operations.....	156
Reciprocal, Reciprocal Square Root, and Fast Square Root Operations.....	156
Notes on Not a Number (NaN) Propagation.....	157
6.3 Vision P6 Floating-Point Programming.....	157
General Guidelines for Floating-Point Programming.....	157
Invalid Floating-Point Operation Examples.....	158
Arithmetic Exception and Exception Reduction.....	158
Associative and Distributive Rules as well as Expression Transformations.....	158
Half/Single Fused-Multiply-Add (FMA).....	159
Division, Square Root, Reciprocal, and Reciprocal Square Root Intrinsic Functions.....	159
Sources of NaN and Potential Changes of NaN Sign and Payload.....	160
Auto-Vectorizing Scalar Floating-Point Programs.....	160
Compiler Switches Related to Floating-point Programs.....	160
Domain Expertise, Vector Data-types, C Intrinsics, and Libraries.....	161
6.4 Accuracy and Robustness Optimizations on Vision P6.....	161
Some Old Beliefs	162
Comparisons and Measurements against Infinitely Precise Results.....	162

Effects of Rounding Mode and Standards Conforming	163
Computer Algebra, Algorithm Selections, Default Substitutions, and Exception Flags.....	164
Error Bounds, Interval Arithmetic, and Directed Rounding.....	164
FMA, Split Number, Double-Precision, and Multi-Precision.....	164
6.5 Floating-Point Operations List.....	165
Vector Floating-Point Operations List.....	165
Scalar Floating-Point Operations List.....	170
6.6 Floating Point References.....	175
7 Histogram Calculation Acceleration Operations.....	177
7.1 Algorithms and Terminology.....	178
7.2 Histogram Operations.....	179
Histograms Distribution Calculation Operations.....	180
Cumulative Histograms Calculation Operations.....	180
8 On-Line Information	183
Appendix A: Vision P6 ISA Enhancements over Vision P5.....	185
New Operations.....	186
New Operation Variants.....	187
Enhanced Operations.....	188
Half-Precision Vector Floating-Point Option.....	191
Slotting Changes	191
Appendix B: Configuration Options.....	193
Appendix C: User TIE Extension.....	195
User Formats.....	196
Naming Restrictions.....	197
User Registers.....	199

List of Tables

Table 1: Register File Port Usage in Each Format.....	14
Table 2: Vision P6 Register Files.....	18
Table 3: VFPU Exceptions/State Flags.....	20
Table 4: Groups of Operations in Format F0.....	21
Table 5: Port Usage in Format F0.....	22
Table 6: Groups of Operations in Format F1.....	22
Table 7: Port Usage in Format F1.....	22
Table 8: Groups of Operations in Format F2.....	23
Table 9: Port Usage in Format F2.....	23
Table 10: Groups of Operations in Format F3.....	24
Table 11: Port Usage in Format F3.....	24
Table 12: Groups of Operations in Format F4.....	25
Table 13: Port Usage in Format F4.....	25
Table 14: Groups of Operations in Format F5.....	25
Table 15: Port Usage in Format F5.....	25
Table 16: Groups of Operations in Format F11.....	26
Table 17: Port Usage in Format F11.....	26
Table 18: Groups of Operations in Format N0.....	27
Table 19: Port Usage in Narrow Format N0.....	27
Table 20: Groups of Operations in Format N1.....	27
Table 21: Port Usage in Narrow Format N1.....	27
Table 22: Groups of Operations in Format N2.....	28
Table 23: Port Usage in Narrow Format N2.....	28
Table 24: Scalar C Types.....	33
Table 25: Vector C Types.....	35
Table 26: Scalar Boolean C Types.....	36
Table 27: Format of the GSERR Special Register.....	39
Table 28: Format of the GSMES Special Register.....	42
Table 29: Format of the GSMEA Special Register.....	42
Table 30: Enable Fields in DBREAKC[i].....	43
Table 31: Integer and Floating-point Types which Have no Guard Bits.....	49
Table 32: Integer and Floating-point Types which Have Guard Bits.....	49
Table 33: Wide Vector Primary Types.....	50
Table 34: Multiplies with the Output to Wide Vector Register.....	76
Table 35: Multiplies with the Output to Narrow Vector Register.....	80
Table 36: Divide Operations.....	81
Table 37: SELI ROTATES.....	98
Table 38: SELI ROTATE Patterns.....	98

Table 39: SELI INTERLEAVES.....	107
Table 40: SELI INTERLEAVE Patterns.....	107
Table 41: SELI EXTRACT Patterns.....	112
Table 42: SELI PACKS.....	118
Table 43: SELI PACK Patterns.....	119
Table 44: Gather Operations List.....	139
Table 45: Scatter Operations List.....	140
Table 46: GATHER Operations.....	141
Table 47: Components of a Binary Floating-point Value.....	146
Table 48: Half Precision Data.....	146
Table 49: Single Precision Data.....	147
Table 50: Encodings of Infinity and Not a Number (NaN).....	148
Table 51: Vector Float Data-types Mapped to the Vector Register File.....	149
Table 52: FCR Fields.....	150
Table 53: FCR Fields and Meaning.....	150
Table 54: FSR Fields.....	151
Table 55: FSR Fields and Meaning.....	151
Table 56: Floating-Point Status and Control Flags in 32-bit Vec Scalar.....	152
Table 57: List of Vector Floating-Point Single Precision Operations.....	165
Table 58: List of Vector Floating-Point Half Precision Operations.....	168
Table 59: List of Scalar Floating-Point Single Precision Operations.....	170
Table 60: List of Scalar Floating-Point Half Precision Operations.....	173
Table 61: New SELI Operation Patternns.....	188
Table 62: New SHFLI Operation Patternns.....	189
Table 63: Specialized SELI Operation Patternns.....	189
Table 64: Specialized SHFLI Operation Patternns.....	190
Table 65: Register File and State Names.....	197
Table 66: User Register Entries.....	199

List of Figures

Figure 1: Vision P6 Architecture.....	15
Figure 2: SELI ROTATE Diagram.....	97
Figure 3: SELI INTERLEAVEs Diagram.....	105
Figure 4: SELI PACK Diagram.....	118

1. Introduction

Topics:

- [*Purpose of this User's Guide*](#)
- [*Installation Overview*](#)
- [*Changes from the Previous Version*](#)
- [*Vision P6 Architecture Overview*](#)

The Vision P6 vision and imaging DSP is a high-performance embedded digital signal processor (DSP) optimized for vision, image, and video processing. The instruction set architecture and memory subsystem provide easy programmability in C/C++ and deliver the high sustained pixel processing performance required for advanced vision, image and video processing and analysis applications.

Vision P6 is a vector (SIMD) DSP supporting 64-way 8-bit, 32-way 16-bit and 16-way 32-bit signed and unsigned integer and fixed-point types and operations. Vision P6 optionally supports 16-way single-precision and 32-way half-precision floating point types and operations. An integral multiply-accumulate (MAC) unit supports two-hundred-fifty-six 8x8 or one-hundred-twenty-eight 8x16 MACs, sixty-four 16x16 MACs or sixteen 16x32 MACs, accumulating 24-bits, 48-bits or 64-bits per element. Integral MAC results can be scaled/rounded/saturated according to application requirements. Optionally sixteen single-precision floating point MACs and thirty two half-precision floating point MACs are supported.

Vision P6 is a 5-way VLIW architecture. Up to two 64-byte loads or one 64-byte load and one 64-byte store can be included in each instruction, reading up to 128 bytes or reading and writing up to 64 bytes each per cycle. Vision P6 also supports vector gather and scatter operations to enable efficient vector processing of disparate data in local data memory. Gather and scatter operations read or write up to 32 16-bit or 8-bit vector elements per cycle to arbitrary local data memory locations.

In addition to standard arithmetic and logical vector units, specialized units support rearranging data across vector lanes through data select, shuffle and interleave patterns and shifting and normalization of data.

Purpose of this User's Guide

The *Vision P6 User's Guide* provides an overview of the Vision P6 architecture and its instruction set. It provides guidelines to help programmers achieve high performance software through the use of appropriate Vision P6 operations, prototypes and ctypes.

This guide will help you quickly understand and start using Vision P6 for your Vision and Image processing applications. To use this guide most effectively, a basic level of familiarity with the Xtensa software development flow is highly recommended. For more details, refer to the *Xtensa Software Development Toolkit User's Guide*.

Installation Overview

To install a Vision P6 configuration for application software development, follow the procedure outlined for installing a newly-built core configuration in the *Xtensa Development Tools Installation Guide*.

Throughout this guide, the symbol `<xtensa_root>` refers to the installation directory of your Xtensa configuration. For example, `<xtensa_root>` might refer to the directory `/usr/xtensa/<user>/<s1>` (`C:\usr\xtensa\<user>\<s1>` on Windows) if `<user>` is the username and `<s1>` is the name of your Xtensa configuration. For all examples in this guide, replace `<xtensa_root>` with the path to the installation directory of your Xtensa distribution.

Changes from the Previous Version

The following changes were made to this document for the Cadence RG-2016.4 release of Xtensa processors. Subsequent releases may contain updates for features in this release or additional features may be added.

New Features

- Vision P6 general release.

Changes to the Document

- *Vision P6 User's Guide* is no longer preliminary
- Updated descriptions of IVP_SEL2NX8I patterns
 - Added summary overview tables for INTERLEAVE and PACK patterns
 - Changed pattern tables to show actual input elements selected by the pattern instead of byte indices, for clarity
 - Added three figures to clarify SELI ROTATE, INTERLEAVE, and PACK patterns
- Described rounding done by IVP_PACKVR operations

- Clarified the Arithmetic Exception interface is required when either the Vision P6 Single-Precision Vector Floating Point Package or the Vision P6 Half-Precision Vector Floating Point Package is selected
- Added details regarding CLSFY operations
- Added description of floating-point status and control flag access with IVP_MOVSCFV and IVP_MOVVSCF operations and bundling restriction on IVP_MOVSCFV
- Added appendix describing Vision P6 differences compared to Vision P5
- Added appendix describing how to extend Vision P6 with user TIE
- General cleanup and clarifications

Vision P6 Architecture Overview

Vision P6 is a multi-way SIMD processor where each operation works on multiple data elements in parallel. A single Vision P6 SIMD operation can support 64-way operations on 8-bit integer and fixed-point types, 32-way operations on 16-bit integer and fixed-point types, and 16-way operations on 32-bit integer and fixed-point types. With the Vision P6 Single Precision Vector Floating Point option it also supports 16-way operations on single-precision floating-point types, and with the Vision P6 Half Precision Vector Floating Point option it supports 32-way operations on half-precision floating-point types.

Vision P6 inherits the 24-bit and 16-bit instructions of the base Xtensa architecture. See the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for further information on the base Xtensa architecture. Vision P6 is also a VLIW processor which bundles multiple operations into an instruction bundle. A single instruction can bundle up to five SIMD (vector) operations, or a mix of vector and scalar operations (for example, base Xtensa operations). Instruction bundles are encoded in either 128- or 64-bit formats, with the former containing four or five operation slots and the latter containing two operation slots. There are seven different formats for 128-bit instruction bundles and three different formats for 64-bit instruction bundles. While the exact combinations of operations available in the different slots varies according to instruction format, the following general categories of operations are available in each slot:

Slot 0: base operations, vector load, vector store, vector gather/scatter, vector alu

Slot 1: base operations, vector load, vbool alu, wvec packs

Slot 2: base operations, vector multiply/multiply-accumulate, vector divide, vector alu, vector shift imm

Slot 3: vector alu, vector select, vector shift

Slot 4: vector alu

A slot can have a SIMD (vector) operation or a scalar operation (for example, base Xtensa operation). If all the operations could be combined or paired with any other operations from the other four slots, this would require a huge amount of bits to encode the instruction bundles, require an extremely high number of ports in the register files, and create routing

congestion. Multiple FLIX formats, which define operations which are paired with each other, is the mechanism which is used to alleviate this problem. Operations which are more likely to improve the overall performance of the machine by being executed in parallel are encoded in the same format, while those operations which do not give significant performance improvement by being executed in parallel are encoded in different formats and cannot be executed in parallel. In some cases fewer than five operations can be combined in an instruction bundle due to program dependencies. To reduce code size in such cases a set of FLIX formats with two effective operation slots is provided, and these formats use a second narrower encoding width. Different operations have different numbers of operands (both in and out). The different formats have also been designed so that operations with the same number of operands are put into slots together, and slots containing operations with more operands are combined with slots containing operations with fewer operands in a given format. This is done so that register file ports can be shared between slots across the different formats, with the goal of minimizing the total number of ports required for each register file. The restrictions placed by the formats result in the following number of read/write ports of each register file in Vision P6.¹

Table 1: Register File Port Usage in Each Format

Format	vec read/ write	wvec read/ write	AR read/write	align read/ write	vbool read/ write	gvr read/write
F0	7/4	2/1	7/4	2/2	4/3	1/1
F1	7/4	2/1	7/4	2/2	5/4	1/1
F2	7/4	2/1	7/4	2/2	5/3	1/1
F3	7/4	2/1	7/4	2/2	5/4	1/1
F4	7/4	2/1	6/3	2/2	5/2	1/0
F5	0/0	0/0	7/4	0/0	0/0	0/0
F11	6/4	1/1	7/4	2/2	3/4	0/0
N0	4/2	0/0	4/2	1/1	1/1	0/0
N1	4/2	1/1	4/2	1/1	2/1	0/1
N2	2/2	1/0	4/2	2/2	3/2	1/1

For more information, see the section [Vision P6 Instruction Formats](#) on page 20 and refer to the Instruction Formats page of the online Vision P6 ISA HTML documentation.

A block diagram of Vision P6 is shown in the following figure.

¹ In a few cases the port usage depends on the option packages configured. The table is for a maximal configuration with all option packages.

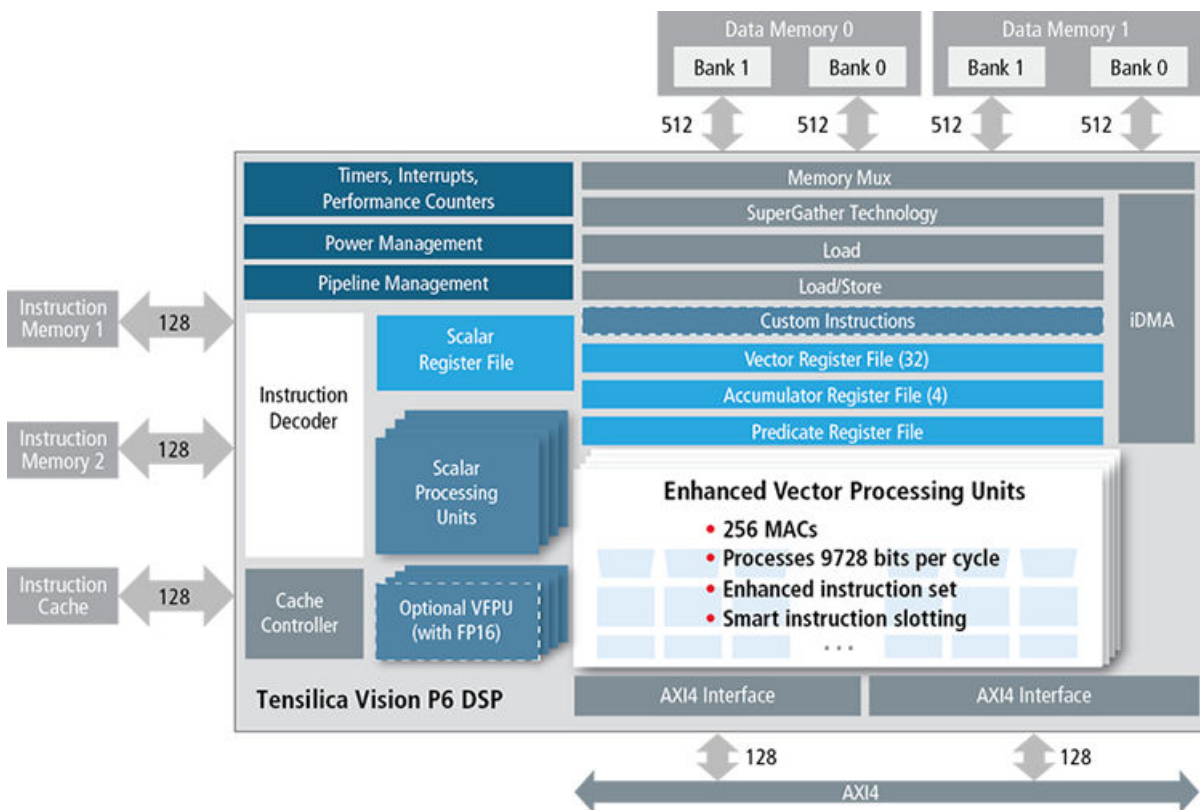


Figure 1: Vision P6 Architecture

Vision P6 computation is organized around three register files: the vec register file (Vector Register File), the wvec register file (Accumulator Register File), and the vbool register file (Predicate Register File). A set of vector processing units work on data contained in these register files. See [Register Files and States](#) on page 18 for further details on the register files and the data types and formats they contain. See [Standard DSP Operations by Type](#) on page 65 for details on the operations supported by the vector processing units.

Vision P6 computation is supported by tightly-coupled local memories. Two local data RAMs (Data Memory 0 and Data Memory 1) support the traditional X/Y memory processing model. Each data RAM is partitioned into **two 512-bit wide banks** (Bank 0 and Bank 1) interleaved on a low-order address bit to provide additional bandwidth and a statistical reduction in bank contention when data placement cannot be explicitly managed (as it is between the X/Y memories). Each 512-bit wide bank is **further partitioned into either four or eight sub-banks** (not shown in the diagram). The sub-banks are separately addressable to support gather/scatter operations (see [Gather/Scatter](#) on page 37 for further details). The local data RAM sizes are **customer-configurable**.

Local instruction memories, either an instruction cache (Instruction Cache) and/or one or more local memories (Instruction Memory 1, Instruction Memory 2) hold Vision P6 instructions. An instruction cache can provide good performance in many applications while freeing the programmer from needing to explicitly manage local instruction memory. Local instruction memories provide deterministic access for latency critical code and are typically smaller and lower power than the same-size instruction cache. The size and organization of an instruction cache and the size of local instruction memories are all **customer-configurable**.

System Interfaces

Vision P6 configurations include the Xtensa Integrated DMA (iDMA) controller. The iDMA is a tightly-coupled DMA engine for copying data between the **local data RAMs and external system memory (and also within the local data RAMs themselves)**. It works in parallel with Vision P6 processing, executing commands from a list of descriptors in the local data RAMs. The iDMA supports 1-D and 2-D (tile) transfers, arbitrary alignment of source and destination addresses, and arbitrary row pitch for 2-D transfers. The iDMA supports multiple outstanding requests to achieve high throughput with large external system memory latencies, and the number of outstanding requests and the burst size are both programmable. Full details of the iDMA are available in the *Xtensa LX7 Microprocessor Data Book*.

The **iDMA is the primary channel** for moving data between Vision P6 and the rest of the system. It has its own dedicated **128-bit AXI4 master interface** for accessing system resources. The iDMA can be used to pull data for processing, for example from system memory. It can also be used to push results, for example to the local data memory of another Vision P6 for subsequent processing.

Vision P6 configurations also include a 128-bit AXI4 slave interface that allows system access to internal Vision P6 resources, including local data RAMs and local instruction memories. This can be used by the system for various purposes, for example to implement a command and control channel for Vision P6 applications through shared local data RAM locations.

The Vision P6 processor can directly access system resources using its own 128-bit AXI4 master interface. This is used, for example, by instruction cache line fills. Vision P6 can directly access system resources with loads and stores, but care should be used when directly accessing long-latency resources as this can stall processing.

2. Features

Topics:

- *Register Files and States*
- *Vision P6 Instruction Formats*
- *Architecture of Vision P6 Operation*
- *Operation Naming Convention*
- *Vision P6 C Types*
- *Gather/Scatter*
- *Option Packages*

The architectural features of Vision P6 are described in this chapter. This includes descriptions of the data types provided for the programmer and the register files in which they reside, the naming conventions for the operations, and an overview of the computational model. Also covered is the Vision P6 gather/scatter extension to the Xtensa architecture and an overview of the Vision P6 option packages.

2.1 Register Files and States

Vision P6 uses multiple register files to obtain high operand bandwidth from a modest number of ports in each register file. The relatively large register files permit deep software pipelining and reduce memory traffic.

The different register files are shown in [Table 2: Vision P6 Register Files](#) on page 18. Included are both the base Xtensa register files (AR and BR) and the Vision P6-specific register files.

The first **Vision-specific register file (vec)** consists of a set of 32 512-bit general purpose narrow vector registers that hold the operands and results of SIMD vector operations. Each vec register can hold either thirty two 16-bit elements, sixteen 32-bit elements, or sixty-four 8-bit elements depending on how the register is used by the software.

The **second Vision-specific register file (wvec)** consists of a set of four 1536-bit wide vector registers each of which can hold either thirty two 48-bit elements, sixteen 64-bit elements², or sixty-four 24-bit elements. Direct loads/stores to/from wide vector registers are not supported. All loads and stores for wide wvec registers take place through intermediate moves into the narrow 512-bit vec registers (the interface to each data memory is 512-bits wide).

Note that narrow vector registers are also referred to as just vector registers (omitting narrow), while wide vector registers are always called wide: (wide vector registers, sometimes wide vectors or wide registers).

Table 2: Vision P6 Register Files

Register File Name	Typical Usage	Width	Number of Entries
AR	Addresses and scalar C types (int8,int16,int32)	32 bits	32
BR	Scalar Boolean values	1 bit	16
vec	Vector data, shift/select amount	512 bits (64 8b, 32 16b or 16 32b)	32
wvec	Vector intermediate values	1536 bits (64 24b, 32 48b or 16 64b)	4
vbool	Vector Boolean predicates	64 bits (64 1b, 32 2b or 16 4b)	8
valign	Load store alignment data	512 bits	4
gvr	Gather data	512 bits	4

² 64-bit elements are stored in the least-significant 64 bits of a 96-bit field.

The Vision-specific 8-entry vector Boolean register file (vbool) supports predicated vector operations. Each entry contains 64 bits. When used for predication, each lane in the vector register file needs to have one bit in the vector Boolean register per each lane of the narrow register file to control predication. The maximum number of elements in the narrow vector register is 64 (for 8-bit types). For 16- and 32-bit vector register types the corresponding number of lanes in vbool is 16 and 32 respectively. In the latter cases, the vector Boolean register lanes are 2- and 4-bits wide, respectively, and each vbool lane contains replicated bits.

The Vision-specific 4-entry vector alignment register file (valign) supports vector loads/stores from/to unaligned addresses. Unaligned addresses are addresses whose values are not a multiple of the load/store word width. Each entry contains 512 bits, the maximum width of data which can be stored or loaded via one load/store interface. Alignment registers are used to buffer data for aligning load and aligning store operations. In addition, for predicated aligning store operations, entries in the vector Boolean register file are used for buffering the predicates.

The Vision-specific 4-entry gather vector register file (gvr) holds data assembled by gather operations. Each entry is 512 bits. The gather register file is further described in the section [Gather Registers](#) on page 136.

All of the Vision P6 register files are fully bypassed, with the exception of the vec register file. With full bypass an operation's result is made available from the "def" pipeline stage onwards, to minimize the latency to subsequent dependent operations. The vec register file is bypassed, but it does not implement the late bypass from the "def" stage. Operation results written to the vec register file are thus available from the pipeline stage following the "def" stage onwards. This increases the latency to subsequent dependent operations by one cycle compared to a fully bypassed register file.³ Programming considerations for the absence of late bypass are discussed in [Guidelines for Optimizing Code for Vision P6](#) on page 60.

The Vision P6 architecture is built on the Xtensa platform where address registers from the address register file (AR) are used for most scalar computation and memory addressing. A limited set of operations move data between the AR and vector register files. The Vision P6 architecture also uses the 16-entry 1-bit scalar boolean register file (BR) of the Xtensa platform, which supports scalar predication and branching operations.

The Vision P6 Single Precision Vector Floating Point and Vision P6 Half Precision Vector Floating Point option packages (VFPU) add the following floating point exception/state flags.

³ Note the Implementation Pipeline section of Vision P6 operations in the ISA HTML does not reflect the presense or absence of late bypass on the result register files. For Vision P6 operations which write results to the vec register file this section shows the pipeline stage in which the vec register file results are defined but the results are only available to subsequent instructions starting from the following stage.

Table 3: VFPU Exceptions/State Flags

Name	Width (bits)
RoundMode	2
InvalidEnable	1
DivZeroEnable	1
OverflowEnable	1
UnderflowEnable	1
InexactEnable	1
InvalidFlag	1
DivZeroFlag	1
OverflowFlag	1
UnderflowFlag	1
InexactFlag	1

These exception/state flags are accessible via the `RUR/WUR` and the `MOVSCFV/MOVVSCF` operations. VFPU operations use the `vec` and `vbool` register files for holding input operands and results. See the section [Floating-Point Operations](#) for further details.

2.2 Vision P6 Instruction Formats

Vision P6 is a **VLIW machine**, capable of executing **up to five operations in parallel**, encoded as one VLIW (**five-slot**) instruction. These operations can be five vector (SIMD) DSP operations or they can be a mixture of vector DSP and Xtensa base scalar operations (for example, three vector and two scalar operations encoded into one instruction). Instructions which consist of five scalar (base Xtensa) operations are not supported.

Operations which are in different slots can be executed in parallel (from the hardware standpoint each has its own copy of hardware), while operations in the same slot cannot be executed in parallel (they may be sharing hardware). If there are no further limiting mechanism on executing operations from different slots in parallel, the number of VLIW instructions which need to be supported is the product of operations in Slot 0 x Slot 1 x Slot 2 x Slot 3 x Slot 4. If, for example, there are 500 operations in each slot, the number of VLIW instructions would be 500 to the power of 5 or over 31 trillion instructions.

In addition, operations are non-symmetrical with respect to internal resource usage such as the number of input and output ports. If several operations requiring a large amount of the same resource, are allowed to execute in parallel we may end up increasing that resource just for this instruction, and the resource will be unused most of them time. The typical resources which are at stake here are: the number of ports in register files and the instruction encoding bits.

The mechanism which limits operations which can be encoded into one VLIW instruction is the use of formats. Operations which are in **the same format** (and different slots) can be executed in parallel (these are also called pairable operations), while operations not in the same format, even if they are in different slots, cannot be executed in parallel.

Vision P6 Instruction Format Structure

Vision P6 has 10 FLIX formats. Out of these 10, 7 formats use 128-bit encoding; they are also referred to as wide formats, and support a large number of parallel operations. The remaining 3 formats are called narrow formats; they use 64-bit encoding, encode fewer parallel operations and serve to improve code density.

Out of the 7 wide (**128-bit instruction size**) formats, 2 formats support all the five slots, while the remaining 5 support four slots. The narrow formats support two slots.

Instruction Formats

The 10 instruction formats of Vision P6 have the following names: F0, F1, F2, F3, F4, F5, F11, N0, N1, N2.

F0 is a 4-slot wide format. Its main features are: pairing dual select operations in Slot 3 (3 vec input, 2 vec output operations) with multiplies in Slot 3. The multiply operations include 64-way 16-bit by 8-bit vector by vector multiplication (and accumulation) which require 3 input vectors. All the multiplications are into wvec; multiplications into vec (multiply and pack) are not supported in this format. These operations can be paired with PACK operations (moving data from wvec to vec) in Slot 1 or two loads (in Slot 0 and Slot1) or store and load (in Slot 0 and Slot1) as well as base Xtensa operations. Two-input ALU operations are only available in Slot 3.

Table 4: Groups of Operations in Format F0

Slot 0	Slot 1	Slot 2	Slot 3
scalar loads,	scalar loads (L32l only),	scalar ALU,	scalar ALU (limited to 1 reg input ops),
scalar stores,	scalar ALU,	vector ALU (limited to ALU to wvec and vbool ops),	vector ALU,
scalar ALU,	single vector loads (no predication),	vector MAC,	vector selects,
single vector loads,	GatherD.	vector MAC pairs (limited to 2 vec input ops),	dual selects,
vector stores,		vector MAC quad (limited to 2 vec input ops).	vector shifts.
vector ALU (limited to 1 vec input ops),			
vector shifts (limited to 16b and 8b right shifts with imm),			
GatherA.			

Table 5: Port Usage in Format F0

Register file	S0 read/write	S1 read/write	S2 read/write	S3 read/write
vec	1/1	0/1	3/0	3/2
wvec		1/0	1/1	
AR	2/1	2/1	2/1	1/1
valign	1/1	1/1		
vbool	1/0	2/1	0/1	1/1

F1 is a 4-slot wide format. Its main features are: pairing 2-input 3 ALU operations in Slots 0,2 and 3 and single select operations in Slot 3. Multiplication (and accumulation) is also supported in Slot 2, but 64-way 16-bit by 8-bit vector by vector multiplication (and accumulation) and multiplications into vec (multiply and pack) are not supported in this format. Multiplication can be paired with PACK operations (moving data from wvec to vec) in Slot 1 or two loads (in Slot 0 and Slot1) or store and load (in Slot 0 and Slot1) as well as base Xtensa operations.

Table 6: Groups of Operations in Format F1

Slot 0	Slot 1	Slot 2	Slot 3
scalar loads, scalar stores, scalar ALU, single vector loads, vector stores, vector ALU, vector shifts (limited to 16b and 8b right shifts with imm), GatherA, Scatter.	scalar loads (L32I only), scalar ALU, single vector loads (no predication), GatherD.	scalar ALU, vector ALU, vector MAC, vector MAC pairs (limited to 2 vec input ops), vector MAC quad (limited to 2 vec input ops), vector shifts (limited to shifts with imm),	scalar ALU (limited to 1 reg input ops), vector ALU, vector selects, vector shifts.

Table 7: Port Usage in Format F1

Register file	S0 read/write	S1 read/write	S2 read/write	S3 read/write
vec	2/1	0/1	2/1	3/1
wvec		1/0	1/1	
AR	2/1	2/1	2/1	1/1

Register file	S0 read/write	S1 read/write	S2 read/write	S3 read/write
valign	1/1	1/1		
vbool	1/1	2/1	1/1	1/1

F2 is a 4-slot wide format. Its main features are: pairing integer and fixed point divide operations in Slot 2 with 2-input ALU operations in Slot 3 and single select operations in Slot 3. This format also has a full set of multiplication (and accumulation) operations, which includes both 64-way 16-bit by 8-bit vector by vector multiplication (and accumulation) and multiplications (and accumulation) into vec (multiply and pack). Unlike F0 and F1 format F2 does not support pairing with two load operations. The operations can only be paired with loads in Slot 1 and stores in Slot 0. Loads in Slot 0 are not supported.

Table 8: Groups of Operations in Format F2

Slot 0	Slot 1	Slot 2	Slot 3
scalar loads, scalar stores, scalar ALU, vector stores, GatherA.	scalar loads (L32I only), scalar ALU, single vector loads (no prediction). GatherD.	scalar ALU, vector ALU, vector MAC, vector MAC pairs (limited to 2 vec input ops), vector MAC quad (limited to 2 vec input ops), vector divide, vector shifts (limited to shifts with imm).	scalar ALU (limited to 1 reg input ops), vector ALU, vector selects, vector shifts.

Table 9: Port Usage in Format F2

Register file	S0 read/write	S1 read/write	S2 read/write	S3 read/write
vec	1/0	0/1	3/2	3/1
wvec		1/0	1/1	
AR	2/1	2/1	2/1	1/1
valign	1/1	1/1		
vbool	1/0	2/1	1/1	1/1

F3 is a 5-slot wide format. Its main features are: pairing of basic multiplication (and accumulation) in Slot 2 with two ALU operations in Slots 3 and 4. Neither 64-way 16-bit by 8-bit vector by vector multiplication (and accumulation) nor multiplications into vec (multiply and pack) is supported. Multiplication can be paired with PACK operations (moving data from

wvec to vec) in Slot 1 or two loads (in Slot 0 and Slot1) or store and load (in Slot 0 and Slot1) as well as base Xtensa operations.

Table 10: Groups of Operations in Format F3

Slot 0	Slot 1	Slot 2	Slot 3	Slot 4
scalar loads, scalar stores, scalar ALU, single vector loads, vector stores, vector ALU (limited to 1 vec input ops), vector shifts (limited to 16b and 8b right shifts with imm), GatherA.	scalar stores (L32I only), scalar ALU, single vector loads (no predication), GatherD.	scalar ALU, vector ALU (limited to ALU to wvec and vbool ops), vector MAC, vector MAC pairs (limited to 2 vec input ops), vector MAC quad (limited to 2 vec input ops).	scalar ALU (limited to 1 reg input ops), vector ALU, vector selects (limited to select with imm), vector shifts.	vector ALU (no predication).

Table 11: Port Usage in Format F3

Register file	S0 read/write	S1 read/write	S2 read/write	S3 read/write	S4 read/write
vec	1/1	0/1	2/0	2/1	2/1
wvec		1/0	1/1		
AR	2/1	2/1	2/1	1/1	
valign	1/1	1/1			
vbool	1/0	2/1	0/1	2/1	0/1

F4 is a 4-slot wide format. This format is designed to support pairing of 4 vector input multiply (and accumulate) pairs operations in Slot 2. These operations can be paired with dual select operations in Slot 3 or 2-input ALU operations in Slot 3. No other multiplication operations are supported in the format. The 4-input multiplication (and accumulation) operations can be paired with PACK operations (moving data from wvec to vec) in Slot 1 or two loads (in Slot 0 and Slot1) as well as base Xtensa operations. Stores from vec are not supported in this format.

Table 12: Groups of Operations in Format F4

	Slot 0	Slot 1	Slot 2	Slot 3
scalar loads	scalar loads, scalar stores, scalar ALU, single vector loads, vector stores, GatherD	scalar loads (L32I only), single vector loads (no predication).	vector MAC pairs (limited to 4 vec input ops), vector MAC quad (limited to 4 vec input ops).	scalar ALU(limited to 1 reg input ops), vector ALU, vector selects (limited to select scalar from vec), vector shifts.

Table 13: Port Usage in Format F4

Register file	S0 read/write	S1 read/write	S2 read/write	S3 read/write
vec	0/1	0/1	4/0	3/2
wvec		1/0	1/1	
AR	2/1	2/1	1/0	1/1
valign	1/1	1/1		
vbool	1/0	2/1		2/1

F5 is a 4-slot wide format. It is designed to have the richest pairing of base Xtensa operations among all the formats. This format does not support vector (SIMD) DSP operations.

Table 14: Groups of Operations in Format F5

Slot 0	Slot 1	Slot 2	Slot 3
scalar loads, scalar stores, scalar ALU.	scalar loads (L32I only), scalar ALU.	scalar ALU	scalar ALU (limited to 1 reg input ops).

Table 15: Port Usage in Format F5

Register file	S0 read/write	S1 read/write	S2 read/write	S3 read/write
vec				
wvec				
AR	2/1	2/1	2/1	1/1
valign				
vbool				

F11 is a 5-slot wide format. It was designed to pair dual load operations (operations which use both load from TCM interfaces). The format does not support any other load operations, nor does it support store operations. These dual load operations can be paired with 2-input ALU operations in Slot 3 and 4. The format supports basic multiplication operations in Slot 2. Neither 64-way 16-bit by 8-bit vector by vector multiplication (and accumulation) nor multiplications into vec (multiply and pack) is supported. PACK operations (moving data from wvec to vec) are not supported in this format. The **vector DSP operations can be paired with base Xtensa operations in Slot 0,1,2, and 3.**

Table 16: Groups of Operations in Format F11

Slot 0	Slot 1	Slot 2	Slot 3	Slot 4
scalar loads, scalar stores, scalar ALU.	scalar loads (L32I only), scalar ALU, dual vector loads.	scalar ALU, vector ALU (limited to ALU to wvec and vbool ops), vector MAC, vector MAC pairs (limited to 2 vec input ops), vector MAC quad (limited to 2 vec input ops).	scalar ALU (limited to 1 reg input ops, vector selects (limited to shuffle with imm), vector shifts.	vector ALU (no predication).

Table 17: Port Usage in Format F11

Register file	S0 read/write	S1 read/write	S2 read/write	S3 read/write	S4 read/write
vec		0/2	2/0	2/1	2/1
wvec		1/0	1/1		
AR	2/1	2/1	2/1	1/1	
valign	1/1	1/1			
vbool		1/1	0/1	1/1	0/1

N0, N1, and N2 are narrow 2-slot formats.⁴ Each one of them is a subset of the wide formats. These narrow instructions are encoded in 64 bits and are intended to improve code density. Each of these formats has operations in Slot 0, which allows them to have load and store operations in Slot 0, although N0 has a very limited subset of load and store operations. N2 can also have 2 loads in Slot 0 and 1. Loads in Slot 1 are limited. The three formats are

⁴ In practice the narrow formats generally have more than two slots, but only two of the slots contain operations other than NOP (No Operation). This is done so that the operations are in the same slots in both the wide and narrow formats.

designed to “cover” a four-slot format: Slot 0,1,2,3. Each of the narrow format has Slot 0 for load and store, the other Slot depends on the format.

Table 18: Groups of Operations in Format N0

Slot 0	Slot 3
scalar loads (L32l.N only), scalar stores (S32l.N only), single vector loads, vector stores (no predication).	scalar ALU, vector ALU, vector selects.

Table 19: Port Usage in Narrow Format N0

Register file	S0 read/write	S3 read/write
vec	1/1	3/1
wvec		
AR	2/1	2/1
valign	1/1	
vbool		1/1

Table 20: Groups of Operations in Format N1

Slot 0	Slot 2
scalar loads, scalar stores, scalar ALU, single vector loads, vector stores.	scalar ALU (limited to ALU to wvec and vbool ops), vector MAC, vector MAC pairs (limited to 2 vec input ops), vector MAC quad (limited to 2 vec input ops).

Table 21: Port Usage in Narrow Format N1

Register file	S0 read/write	S2 read/write
vec	1/1	3/1
wvec		1/1
AR	2/1	2/1
valign	1/1	
vbool	1/0	1/1

Table 22: Groups of Operations in Format N2

Slot 0	Slot 1
scalar loads, scalar stores, scalar ALU, single vector loads, vector stores, special select and shuffle ops, GatherA, Scatter.	scalar loads, scalar ALU, single vector loads (no predication), vector ALU (limited to ALU to vbool ops), GatherD.

Table 23: Port Usage in Narrow Format N2

Register file	S0 read/write	S1 read/write
vec	2/1	0/1
wvec		1/0
AR	2/1	2/1
valign	1/1	1/1
vbool	1/1	2/1

2.3 Architecture of Vision P6 Operation

The Vision P6 architecture provides guard bits in data paths and the wide register file to avoid overflow of ALU and MAC operations. The **typical data flow** on this machine is:

1. Load data into the unguarded narrow vector register file.
2. Compute and accumulate into the wide vector register file with guard bits.
3. Store data in a narrow format by packing down with truncation or saturation (via the vec register file).

Vision P6 has sixteen 32bx16b, sixty-four 16bx16b, one-hundred-twenty-eight 16bx8b and two-hundred-fifty-six 8bx8b multipliers, each of which produces a 64/48/24/24-bit result to be stored in wide vector register elements. This allows up to sixteen, sixteen and eight guard bits for accumulation in the wide vector register elements.

The first load/store unit supports all load and store operations present in the Vision P6 ISA and the base Xtensa RISC ISA. The second load unit supports the load operations present in the Vision P6 ISA and the base Xtensa RISC ISA, but does not support any **store** operations.

Additionally, Vision P6 offers operations that pack results. For instance, as an example of the 'PACK' category of operations, one of them extracts 16-bits from 48-bit **wide** data in wvec register elements and saturate results to the range $[-2^{15} .. 2^{15}-1]$. Most 'MOVE' operations that move 16-bit or 32-bit results from high-precision to low-precision data widths saturate. Unlike some architectures, the Vision P6 architecture does not set flags or take exceptions when operations overflow their wvec range, or if saturation or truncation occurs on packing down: for example from a 48-bit wide vector element to a 16-bit narrow vector element.

2.4 Operation Naming Convention

Vision P6 uses certain naming conventions for greater consistency and predictability in determining names of Ctypes, operations, and protos. In general, Vision P6 operation names follow this pattern:

<PREFIX>_<OP>[<SIMD>]X[<DATA-TYPE>][_<SUFFIX>]?

Where,

- PREFIX := IVP
- OP := LOADOP | STOREOP | ALUOP | MULOP | SELOP
- SIMD := 2N | N | N_2 where **N = 32 (2N = 64, N_2 = 16)**
- DATA-TYPE := 8 | 16 | 32 | 24 | 48 | 64

The following section describes the naming convention of each OP group in detail.

Load/Store Operations

IVP_<OP>[<D>]?[<OPTYPE>][<SIMD>][X<DATA-TYPE>]?[<SIGN>]?[<PREDICATE>]?
[_<LDST_SUFFIX>]?

Where,

- OP := L|S
- D := 2

Operation uses both the load units.

- OPTYPE:= one or more letters specifying a sub-class of operation(s)
 - B:= vector boolean register
 - S:= scalar load/store on lane 0 of vec register
 - SR:=scalar load/store replicating to all lanes of vec register
 - A:= Unaligned vector load/store
 - AV:= Unaligned load/store with variable number of bytes
 - U:= Unigned load without using the valign register
 - V:= Aligned vector load/store
 - ALIGN:= load/store the alignment register

- SAPOS:= Flush alignment register to memory
- SIMD:= $2N \mid N \mid N_2$ where $N = 32$

Number of elements of the following DATA-TYPE

- DATA-TYPE:= $8 \mid 16 \mid 32$

Bit-width of each element

- SIGN:= $S \mid U$

For loads, sign or zero extend each element. For stores, saturate or store LSBs.

- PREDICATE:= T

Predicated load/stores. Only load/store elements whose corresponding vbool register bits are TRUE.

For unaligned predicated load (IVP_LAT2NX8_XP), this conditionally updates the vector register. [More detailed explanation](#) is in the Load and Store chapter.

- LDST_SUFFIX:= $I \mid IP \mid X \mid XP \mid PP \mid FP$

Addressing modes – ([More detailed explanation](#) is in the Load and Store chapter)

$I \mid X$ – immediate or indexed

$IP \mid XP$ – immediate or indexed with post-increment updating

PP – aligning load primes

FP – flush for aligning stores

ALU Operations

$IVP_ [<OP_PREFIX>]? [<OPTYPE>] [<OP_SUFFIX>]? [<SIMD>] [X<DATA-TYPE>]? [<SIGN>]? [IMM]? [<PREDICATE>]? [_SPLIT]?$

Where,

- OP_PREFIX:= $R \mid B \mid O \mid U$
- Reduction (R) for ADD, MIN, MAX
- With Boolean Output (B) for MIN, MAX, ADDNORM, SUBNORM
- Unordered (U), Ordered (O) for floating point comparisons that return different results when inputs are NaN
- OPTYPE := ABS | ABSSUB | ADD | ADDEXP | ADDEXPM | ADDMOD | ADDNORM | AND | ANDNOT | AVG | AVGR | CONST | EQ | EXT | EXTRACT | FICEIL | FIFLOOR | FIRINT | FIROUND | FITRUNC | FLOAT | INJ | LE | LT | LTR | MAX | MIN | MKDADJ | MKSADJ | MOV | MULSGN | NEG | NEQ | NEXP01 | NOT | NSA | OR | ORNOT | REP | ROTR | SEL | SELS | SEQ | SHFL | SL | SLA | SLL | SQZ | SR | SRA | SRL | SUB | TRUNC | UEQ | ULE | ULEQ | ULT | ULTQ | UNEQ | UN | UNPK | UNSQZ | XOR

- OP_SUFFIX:= I | H | L

Immediate (I) for EXT, INJ, SL,SLL, SLA, SR, SRL, SRA

- SIMD:= 1 | 2N | N | N_2 where N = 32

Number of elements of the following DATA-TYPE

- DATA-TYPE:= 8 | 16 | 32 | F32

Bit-width of each element

- SIGN:= S | U

Saturating (S), Non-saturating (none) for ABSSUB, ADD, MULSGN, NEG, SUB, SL, SR

Unsigned (U) for ABSSUB, AVG, LE, LT, MAX, MIN

- IMM:= I

Immediate (I) operand for LTR, SEL,DSEL,SHFL

- PREDICATE:= T

For reduction operations, input only TRUE elements.

For all other operations, output only TRUE elements.

Multiply Operations

IVP_[<OPTYPE>][<PROPERTIES>]?[<SIMD>][X?<TYPE1>]?[X<TYPE2>][PACK-TYPE]?
[<PREDICATE>]?[_SPLIT]?

Where,

- OPTYPE := MUL

- PROPERTIES:= [UU|US|SU] [A|S] [I] [H]

Unsigned x Unsigned (UU), Unsigned x Signed (US), Signed x Unsigned (SU)

with Accumulation (A) OR with Subtraction (S)

Interleaved (I) input - TYPE2 input is interleaved from two arguments.

IVP_MULI2NR8X16 – the 16b multiplicand is input from even/odd elements of two vec registers.

High Precision (H) – To generate intermediate results for high precision 32x32 multiplies

- SIMD:= 1 | 2N | N | N_2 where N = 32

Number of elements of the following DATA-TYPE

- TYPE1:= R8 | 8 | 16

Bit-width of each element. R indicates a scalar operand from the AR register

- TYPE2:= 8 | 16 | R16 | 32 | F32

Bit-width of each element. R indicates a scalar operand from the AR register

- PACK-TYPE:= PACKL | PACKP | PACKQ

Saturating (S), Non-saturating (none) for ABSSUB, ADD, MULSGN, NEG, SUB, SL, SR

Unsigned (U) for ABSSUB, AVG, LE, LT, MAX, MIN

- PREDICATE:= T

Output only TRUE elements. IVP_MULANX16PACKLT

- SPLIT := 0|1

even elements (0), odd elements (1)

usually used when the number of input elements is 1/2 the number of output elements, or the number of output elements is 1/2 the number of input elements.

Shorthand Notation

A shorthand notation is used in this User's Guide to refer to a set of related operations. The shorthand notation is designed to describe a group of related operations using one extended name, similar to a regular expression. For example, to refer to following variants of the vector add operation:

```
{ IVP_ADD2NX8, IVP_ADD2NX8T, IVP_ADDNX16, IVP_ADDNX16T, IVP_ADDN_2X32, IVP_ADDN_2X32T }
```

The following shorthand notation is used:

```
IVP_ADD[2NX8|NX16|N_2X32][T]
```

This notation is convenient because the name before the square brackets clearly shows the category of the operations and the bracketed quantities show the variants and their structure.

Below is an explanation on how the shorthand notation works.

IVP_NAME[|A|B] is interpreted as [nothing or A or B], which expands to:

```
{ IVP_NAME, IVP_NAMEA, IVP_NAMEB }
```

Two or more [] brackets in the same name represent an outer product of all [], for example:

IVP_NAME[|A|B]US_[0|1] expands to six names:

```
{ IVP_NAMEUS_0, IVP_NAMEAUS_0, IVP_NAMEBUS_0, IVP_NAMEUS_1, IVP_NAMEAUS_1, IVP_NAMEBUS_1 }
```

Note an empty or blank alternative within the brackets indicates the empty string is an alternative, as in the earlier example IVP_NAME[|A|B]. When there are just two alternatives,

one of which is the empty string, the notation can be abbreviated to just the alternative that is not the empty string. For example, in `IVP_ADD[2NX8|NX16|N_2X32][T]` the `[T]` is equivalent to `[|T]`.

2.5 Vision P6 C Types

The following tables present all of the floating point, integer, and Boolean ctypes for the Vision P6.

- The Register File column indicates where the ctype resides.
- The Bitwidth column indicates the number of bits of useful data in the ctype. For constrained types⁵, this can be less than the bit width of the register file.
- The Memory Bitwidth column indicates the number of bits the ctype occupies in memory.
- The Register Format and Memory Format columns give an encoded description of the layout of the data type in the register and in memory respectively.
 - The data in the type is referred to as "int Constant". Zeros are written in Verilog-like notation e.g. `16'b0`.
 - Sign extensions of the next field are written "se#" where # is the number of sign extended bits.
 - The order of fields is in Verilog order. The least-significant field is listed to the right.
 - Vector types are specified with the layout of 1 element (the most-significant one) followed by ", ...". For example, the type `"xb_vec2Nx32w"` is a vector of 32-bit signed integer values. Each value is sign extended into 48-bit elements in the wide vector register file. Its description is written as `"{se8, int32, ...}"`.
 - Ctypes that take more than one register are specified with the values in each register in `{}`
- The "Description" columns are not unique. For example, there are multiple register files where a 32-bit scalar ctype can exist -- AR, vec, and wvec. When a "C" program refers to an "int", the builtin datatype that lives in the AR is used. If a user want to explicitly use a 32-bit scalar in the vector or wide vector register files, the `"xb_int32"` or `"xb_int32w"` datatypes should be used. The most common usage for scalar datatype is as an output for reduction operations. They can also be used for refining scalar C code to permit autovectorization of non-standard operations.

Table 24: Scalar C Types

Ctype	Register File	Bitwidth	Memory Bitwidth	Register Format	Memory Format	Description
xb_int8	vec	8	8	{504'b0,int8}	{int8}	scalar 8-bit signed integer

⁵ A constrained type is a type for which not all bit patterns within a word, which represent the value of the type, are legal.

Ctype	Register File	Bitwidth	Memory Bitwidth	Register Format	Memory Format	Description
xb_int8U	vec	8	8	{504'b0,uint8}	{uint8}	scalar 8-bit unsigned integer
xb_int16	vec	16	16	{496'b0,int16}	{int16}	scalar 16-bit signed integer
xb_int16U	vec	16	16	{496'b0,uint16}	{uint16}	scalar 16-bit unsigned integer
xb_int32v	vec	32	32	{480'b0,int32}	{int32}	scalar 32-bit signed integer
xb_int32Uv	vec	32	32	{480'b0,uint32}	{uint32}	scalar 32-bit unsigned integer
xb_int24	wvec	24	32	{1512'b0,int24}	{se8,int24}	scalar 24-bit signed integer
xb_int48	wvec	48	64	{1488'b0,int48}	{se16,int48}	scalar 48-bit signed integer
xb_int64w	wvec	64	64	{1472'b0,int64}	{int64}	scalar 64-bit signed integer
xb_f16**	vec	16	16	{496'b0,float16}	{float16}	scalar 16-bit floating point
uint16	AR	16	16	{16'b0, uint16}	{uint16}	ANSI C scalar 16-bit unsigned integer
uint32*	AR	32	32	{uint32}	{uint32}	ANSI C scalar 32-bit unsigned integer
uint64	AR x 2	64	64	{uint32} {uint32}		ANSI C scalar 64-bit unsigned integer
uint8	AR	8	8	{24'b0, uint8}	{uint8}	ANSI C scalar 8-bit unsigned integer
valign*	valign	512	512	{valign}		Vector Alignment

Table 25: Vector C Types

Ctype	Register File	Bitwidth	Memory Bitwidth	Register Format	Memory Format	Description
xb_vec2Nx8*	vec	512	512	{int8,...}		64-way vector 8-bit signed integer
xb_vec2Nx8U	vec	512	512	{uint8,...}		64-way vector 8-bit unsigned integer
xb_vecNx8	vec	256	256	{se8,int8,...}	{int8,...}	32-way vector 8-bit signed integer
xb_vecNx8U	vec	256	256	{8'b0,uint8,...}	{uint8,...}	32-way vector 8-bit unsigned integer
xb_vecNx16	vec	512	512	{int16,...}		32-way vector 16-bit signed integer
xb_vecNx16U	vec	512	512	{uint16,...}		32-way vector 16-bit unsigned integer
xb_vecN_2x16	vec	256	256	{se16,int16,...}	{int16,...}	16-way vector 16-bit signed integer
xb_vecN_2x16U	vec	256	256	{16'b0,uint16,...}	{uint16,...}	16-way vector 16-bit unsigned integer
xb_vecN_2x32v	vec	512	512	{int32,...}	{int32,...}	16-way vector 32bit signed integer
xb_vecN_2x32Uv	vec	512	512	{uint32,...}	{uint32,...}	16-way vector 32-bit unsigned integer
xb_vecNxf16**	vec	512	512	{float16,...}		32-way vector 16-bit floating point
xb_vecN_2xf16**	vec	256	256	{16'b0,float16,...}	{float16,...}	16-way vector 16-bit floating point
xb_vecN_2xf32**	vec	512	512	{float32,...}		16-way vector 32-bit floating point

Ctype	Register File	Bitwidth	Memory Bitwidth	Register Format	Memory Format	Description
xb_vec2Nx16w	wvec	1024	1024	{se8,int16,...}	{int16,...}	64-way vector 16-bit signed integer
xb_vec2Nx16Uw	wvec	1024	1024	{8'b0,uint16,...}	{uint16,...}	64-way vector 16-bit unsigned integer
xb_vec2Nx24*	wvec	1536	2048	{int24,...}	{se8,int24,...}	64-way vector 24-bit signed integer
xb_vecNx32	wvec	1024	1024	{se16,int32,...}	{int32,...}	32-way vector 32-bit signed integer
xb_vecNx32U	wvec	1024	1024	{16'b0,int32,...}	{uint32,...}	32-way vector 32-bit unsigned integer
xb_vecNx48	wvec	1536	2048	{int48,...}	{se16,int48,...}	64-way vector 16-bit signed integer
xb_vecN_2x64w***	wvec	1024	1024	{32'b0,int64,...}	{int64,...}	16-way vector 64-bit signed integer
vbool1	vbool	1	8	{7'b0, bool}	{7'b0,bool}	Scalar Boolean
vbool2N	vbool	64	64	{bool...}		64-way vector Boolean
vboolN	vbool	32	32	{se1, bool...}		32-way vector Boolean
vboolN_2	vbool	16	16	{se3, bool...}		16-way vector Boolean

Table 26: Scalar Boolean C Types

Ctype	Register File	Bitwidth	Memory Bitwidth	Register Format	Memory Format	Description
xtbool*	BR	1	8	{bool}	{7'b0,bool}	Scalar boolean
xtbool2	BR2	2	8	{bool,...}	{6'b0,bool,...}	Scalar boolean pair
xtbool4	BR4	4	8	{bool,...}	{4'b0,bool,...}	Scalar boolean quad

Ctype	Register File	Bitwidth	Memory Bitwidth	Register Format	Memory Format	Description
xtbool8	BR8	8	8	{bool,...}	{bool,...}	Scalar boolean oct
Xtbool16	BR16	16	16	{bool,...}	{bool,...}	Scalar boolean hex

*default ctype

**Available in Single Precision and Half Precision VFPU configuration option package

***Each 64b element resides in a 96b container with zero-extension in the 32 most-significant bits.

2.6 Gather/Scatter

Vision P6 supports vector gather and scatter operations that improve the efficiency of **vector** processing applications where data is stored in disparate locations in local data memory. Vector gather operations load data from arbitrary addresses in local data memory to consecutive elements (lanes) of a vector in a processor register file. Vector scatter operations store data from consecutive elements of a vector in a processor register file to arbitrary addresses in local data memory. Data in disparate locations can thus be processed in vector form in the processor registers and efficiently moved between local data memory and the processor registers using gather and scatter operations.

Vision P6 gather and scatter operations make use of the Xtensa SuperGather™ core instruction option to perform the local memory reads and writes done by the operations. (Note the SuperGather option is named “Scatter Gather” and is found in the Processors configuration page in Xtensa Xplorer.) The Xtensa SuperGather option implements a tightly-coupled engine that operates in parallel with the Xtensa core pipeline. Vision P6 gather and scatter operations offload their memory accesses to the SuperGather engine. The SuperGather engine independently arbitrates for use of the local data memory and performs the accesses, utilizing the bank and sub-bank structure of local data memory to load or store multiple lanes of vector data in parallel each cycle. Vision P6 gather and scatter operations synchronize as necessary with the SuperGather engine.

The Xtensa SuperGather option uses a separate gather register file to hold the vectors assembled by gather operations. Vision P6 gather operations are split into two separate operations:

- **gatherA operations** (A for address) compute a vector of addresses which are read and assembled into a vector in the gather register file.
- **gatherD operations** (D for data) move vectors from the gather register file to the vec register file.

GatherA operations offload the gather to the SuperGather engine, sending it the vector of addresses to read and assemble. Vision P6 instruction execution then continues in parallel with the SuperGather engine, which generally requires multiple cycles to perform the gather⁶. GatherD operations wait if necessary for a prior gatherA to complete. Separate gatherA and gatherD operations allow the variable latency of the SuperGather engine to be overlapped with other Vision P6 processing.

Vision P6 scatter operations compute a vector of addresses to write, and store individual vector elements to those addresses (both done by a single operation). The vector stored is contained in the vec register file. Scatters are internally buffered by the **SuperGather engine**; this buffer is similar to a store buffer and not architecturally visible. Scatter operations normally execute in one cycle, but wait if necessary for space in the internal buffer. Vision P6 instruction execution then continues in parallel with the SuperGather engine performing the scatter.

Details on the specific Vision P6 gather and scatter operations can be found in the [Gather/Scatter Operations](#) chapter.

Gather/Scatter Error Handling

Vision P6 gather and scatter operations check the validity of the addresses they compute prior to accessing local data memory. These checks are similar to the address checks done by load and store operations.

The gatherA and scatter operations compute addresses by adding a base address from an AR register to a vector of offsets from the vec register file. **The base address must be contained in a local data RAM.** If the base address is not contained in a local data RAM the gatherA or scatter operation is not performed and an exception is taken with EXCCAUSE set to LoadStoreErrorCause.

Vision P6 requires the Xtensa Memory Protection Unit (MPU). See the *Xtensa LX7 Microprocessor Data Book* for further information on the MPU. Vision P6 gatherA operations check that the MPU region containing the base address is readable, and scatter operations check that the region is writable. If the region is not accordingly accessible, the gatherA or scatter operation is not performed and an exception is taken with EXCCAUSE set to LoadProhibitedCause.

Other error checks done by the Vision P6 gatherA and scatter operations are completed too late in the processor pipeline to cause the gatherA or scatter operation to take an exception. The Xtensa SuperGather option adds the Gather/Scatter error interrupt for reporting such errors and the GSERR special register to record error details. When a gatherA or scatter

⁶ The number of cycles depends upon the number of elements being gathered that reside in different addresses within the same local data memory sub-bank. Typically a few cycles are required. In the best case where all the elements gathered are either in different sub-banks or within the same sub-bank address as another element, the gather completes in a single cycle. In the worst case where all the elements are in different addresses in the same sub-bank, one cycle per element gathered is required.

operation detects one of these errors (enumerated below), the operation completes normally, but the read or write of any element whose offset or address is in error is not done. For gatherA operations, elements whose offset or address is in error are set to 0 in the gather register file. Bits corresponding to the type of error detected are set in the GSERR register and the Gather/Scatter error interrupt is raised. The `EXCW` instruction can be used to wait until these asynchronously reported errors have been reported.


- The locations accessed by Vision P6 gatherA and scatter operations must be in the same data RAM as the base address. Elements beyond the end of the data RAM containing the base address will not be read or written and for gathers the corresponding gather result will be zero.
- The locations accessed by Vision P6 gatherA and scatter operations must be within the MPU region containing the base address. Elements beyond the end of the region will not be read or written and for gathers the corresponding gather result will be zero. This check is only done when the MPU region ends within the data RAM containing the base pointer. (If not, note any address beyond the end of the MPU region is also beyond the end of data RAM and will be detected as such.)
- The locations accessed by Vision P6 gatherA and scatter operations must be aligned to the size of the elements being gathered or scattered. Elements whose address is not so aligned will not be read or written and for gathers the corresponding gather result will be zero.
- The offsets for Vision P6 gatherA and scatter operations that **operate on 32-bit elements must be less than 65536**. Elements whose offset is greater than or equal to 65536 will not be read or written and for gathers the corresponding gather result will be zero. An invalid offset also inhibits checking the corresponding element for address beyond the end of data RAM and address beyond the end of MPU region.

The Gather/Scatter Error (GSERR) special register (#116) contains five flag bits – one for each of the errors above, plus a bit for uncorrectable memory ECC errors as described in [Gather/Scatter ECC Support](#) on page 40. Each bit is set to 1 when the corresponding error condition is detected in a gatherA or scatter operation. Once set to 1, each bit remains 1 until cleared by writing a 0 to the corresponding bit position in the register. These bits can also be set by writing a 1 to the corresponding bit position (this does not cause a Gather/Scatter error interrupt).

The format of the GSERR special register (#116) is shown in [Table 27: Format of the GSERR Special Register](#) on page 39. Bits not specified in the table are reserved and read-only with a value of zero.

Table 27: Format of the GSERR Special Register

Bit Position	Flag	Description
0	OffsetErr	One or more 32-bit element offsets were ≥ 65536
1	AddressErr	One or more element addresses were beyond the end of data RAM
2	UnalignedErr	One or more element addresses were not aligned to the element size

Bit Position	Flag	Description
3	ProtectErr	One or more element addresses were beyond the end of the MPU region containing the base address
8	UncorrectableErr	Uncorrectable memory error encountered by gatherA or scatter operation
		 Important: This bit is only defined when the Data Memory or Instruction Memory error type has been configured to word-wide ECC; otherwise this bit is undefined.

The internal Gather/Scatter error interrupt must be configured for Vision P6. This interrupt is raised when any of the errors logged in GSERR are detected by a gatherA or scatter operation. The Gather/Scatter error interrupt is subsequently cleared by writing to INTCLEAR. Note the flags in GSERR are independent of the Gather/Scatter error interrupt, though they are set to 1 by the same conditions that raise the interrupt.

Gather/Scatter ECC Support

The SuperGather engine supports word-wide ECC on the local data RAMs.

When ECC is configured the gatherA and scatter operations both check for correctable and uncorrectable errors in the local data RAM words containing the elements being read and written. Checking is only done on the ECC words that contain elements being accessed by the gatherA and scatter operations and not on any other ECC words in the same sub-bank locations. Predicated false elements do not access local data RAM and thus do not result in any ECC checking. Checking for correctable and uncorrectable errors is controlled by the MESR.ErrEnab and MESR.ErrTest bits that also control memory error checking by other processor memory accesses and by iDMA memory accesses. (See the *Xtensa LX7 Microprocessor Data Book* for further details on the MESR register.) The gatherA and scatter operations check for memory errors when MESR.ErrEnab is 1. When MESR.ErrEnab is 0 no checks for memory errors are made but the same values (i.e. including the computed check bits) are written to Data RAM. The behavior of gatherA and scatter operations is undefined when MESR.ErrTest is 1.

When a correctable error is encountered by a gatherA operation the value read from local data RAM is corrected and the corrected value used for the gather result. When a correctable error is encountered by a scatter operation (as part of the read-modify-write (RMW) sequence needed to write the scatter elements) the value read from local data RAM is corrected and the corrected value is merged with the scatter data being written back to the local data RAM. The detection of a correctable error is logged (see [Gather/Scatter ECC Error Logging](#) on page 41) but otherwise the gatherA and scatter operations proceed normally when correctable errors are encountered. Correctable ECC errors never generate the internal Gather/Scatter error interrupt. Software must poll the log to determine whether any correctable ECC errors were encountered by gatherA or scatter operations.

When an uncorrectable error is encountered by a gatherA operation the value read from local data RAM is invalid but still written to the corresponding element of the gather result. When an uncorrectable error is encountered by a scatter operation that element is not written to local data RAM and the containing ECC word is left unchanged. In each case the uncorrectable error is logged (see [Gather/Scatter ECC Error Logging](#) on page 41) and the internal Gather/Scatter error interrupt (see [Gather/Scatter Error Handling](#)) is raised. The gatherA or scatter operation and subsequent operations continue after an uncorrectable error is detected. If additional uncorrectable errors are encountered they are handled in the same way. The EXCW operation can be used as a barrier; provided the Gather/Scatter error interrupt is not masked it will be taken before EXCW completes if any uncorrectable errors are encountered by prior gatherA or scatter operations. (It may also be desirable for the Gather/Scatter error interrupt to be the highest priority interrupt in the system so that uncorrectable errors are handled ahead of all other interrupts.)

Gather/Scatter ECC Error Logging

Correctable and uncorrectable ECC errors detected by the gatherA and scatter operations are logged in special registers specific to the gatherA and scatter operations. These registers are similar to the MESR/MECR/MEVADDR registers that log detected errors for other processor memory accesses. The GSMES (Gather Scatter Memory Error Status #112) register indicates the presence of a logged error and contains information about the error, including its syndrome. The GSMEA (Gather Scatter Memory Error Address #113) register contains the address of the error. Note the GSERR.UncorrectableErr bit indicates the presence of an uncorrectable error in the log and it must be checked in addition to GSMES.RCE. The GSMES and GSMEA registers and the GSERR.UncorrectableErr bit are only present when the Data Memory or Instruction Memory error type has been configured to word-wide ECC. When not present the GSERR.UncorrectableErr bit position is undefined.

The gatherA and scatter operations can encounter more than one memory ECC error in the same cycle. When more than one correctable error is detected in the same cycle the one logged is implementation specific, and similarly when more than one uncorrectable error is detected in the same cycle the one logged is implementation specific. When a correctable error and an uncorrectable error are detected in the same cycle the details of the uncorrectable error are logged. A memory error can also be detected subsequent to earlier memory errors being detected and logged. When this occurs the log is updated when the log contains details for a correctable error and the subsequent error is an uncorrectable error, otherwise the log is unchanged. When an error is detected but not logged the GSMES.DLCE bit is set if the unlogged error is correctable and the GSMES.DLUE bit is set if the unlogged error is uncorrectable. GSMES.DLCE is also set when a logged correctable error is overwritten by a subsequent uncorrectable error.

The format of the GSMES register is shown in [Table 28: Format of the GSMES Special Register](#) on page 42. Bits not specified in the table are reserved and read-only with a value of zero.

Table 28: Format of the GSMES Special Register

Bit Position	Flag	Description
1	DLUE	Data Lost Uncorrectable Error. Set when one or more uncorrectable errors have been detected by gatherA or scatter operations that were not logged in GSMES.Syn / GSMES.Memory_Type / GSMEA because the log was already full. Set but never cleared by hardware. Software reads and writes DLUE normally.
2	Memory Type	Memory type where the error was detected: 0 – Data Ram 0 1 – Data Ram 1 This bit is written by the hardware whenever a correctable or uncorrectable error is logged. Software reads and writes Memory Type normally.
4	RCE	Recorded Correctable Error. Set when GSMES.Syn / GSMES.Memory_Type / GSMEA contain the details of a correctable memory error encountered by a gatherA or scatter operation. This bit is cleared by the hardware when an uncorrectable error replaces the correctable error in GSMES.Syn / GSMES.Memory_Type / GSMEA (i.e. if this bit is set when the uncorrectable error is logged). Software reads and writes RCE normally.
5	DLCE	Data Lost Correctable Error. Set when one or more correctable errors have been detected by gatherA or scatter operations that were not logged in GSMES.Syn / GSMES.Memory_Type / GSMEA because the log was already full. Set but never cleared by hardware. Software reads and writes DLCE normally.
30:24	Syn	Syndrome for the 32-bit word containing a correctable or uncorrectable error. The location of the word is in GSMEA.Offset. Software reads and writes Syn normally.

The format of the GSMEA register is shown in [Table 29: Format of the GSMEA Special Register](#) on page 42. Bits not specified in the table are reserved and read-only with a value of zero.

Table 29: Format of the GSMEA Special Register

Bit Position	Flag	Description
23:0	Offset	Byte offset of the 32-bit word containing a correctable or uncorrectable error. The offset is in the Data Ram specified by GSMES.Memory_Type. The syndrome of the error is in GSMES.Syn. Software reads and writes normally the least-significant bits of Offset needed to address the configured Data Ram size. The remaining most-significant bits of Offset ignore writes and always read as 0s.

Gather/Scatter Debug Data Breakpoints

Vision P6 gatherA and scatter operations have basic hardware support for data breakpoints. When enabled, gatherA and scatter operations break when a data breakpoint register

contains an address in data RAM greater than or equal to the base address of the gather or scatter. The exception handler is responsible for computing the gather or scatter addresses from the operands and comparing them to the data breakpoint registers.

There are separate enables for reads (gatherA) and writes (scatter), and these are located in the DBREAKC register associated with the breakpoint.

The enable fields in DBREAKC[i] are as follows (other fields not shown):

Table 30: Enable Fields in DBREAKC[i]

Bit Position	Enable	Description
29	scatter match enable	1 to enable data breakpoint exception
28	gather match enable	1 to enable data breakpoint exception

See the *Xtensa Instruction Set Architecture (ISA) Reference Manual* for further information on the DBREAKC registers and the Debug Option.

Gather/Scatter and Wait Mode

Before entering wait mode (PWaitMode asserted) the processor waits for all outstanding gather and scatter operations to complete. See the *Xtensa LX7 Microprocessor Data Book* for further information on Wait Mode.

2.7 Option Packages

Vision P6 option packages are operation sets that enhance performance on specific computations and data types. They provide significant performance benefit for the targeted computations and data types. The packages are optional so that customers can trade off the package's performance benefit against its cost (in gate count / die area) based on their specific application requirements. Option packages are selected as part of Vision P6 configuration in Xtensa Explorer.

Brief descriptions of the option packages available for Vision P6 are given below, along with pointers to detailed descriptions.

Vision P6 Single-Precision Vector Floating Point Package

The Vision P6 Single-Precision Vector Floating Point package adds a set of vector and scalar single-precision floating point operations and associated data types. This package supports applications which require high-performance single-precision floating-point calculations. The operations are fully integrated into Vision P6, operating on floating-point values in the vec register file and floating-point predicates in the vbool register file. Full details of the Vision P6 Single-Precision Vector Floating Point package are in the [Floating Point Operations](#) chapter. Brief descriptions of the operations are in the [floating point operations list](#) section.

The Vision P6 Single-Precision Vector Floating Point Package is independent of, but compatible with, the Vision P6 Half-Precision Vector Floating Point package. Either package can be configured separately or together with the other.

Note the Xtensa Scalar Floating Point coprocessors (single-precision scalar or single and double precision scalar options) are also supported for Vision P6 configurations, for applications requiring more modest floating-point performance. The Vision P6 Single-Precision Vector Floating Point package provides a full set of scalar single-precision operations so the two are mutually exclusive (that is, you cannot select the Xtensa Scalar Floating Point coprocessor and either the Vision P6 Single-Precision Vector Floating Point package or Vision P6 Half-Precision Vector Floating Point package).

Vision P6 Half-Precision Vector Floating Point Package

The Vision P6 Half-Precision Vector Floating Point package adds a set of vector and scalar half-precision floating point operations and associated data types. This package supports applications which require high-performance half-precision floating-point calculations. The operations are fully integrated into Vision P6, operating on floating-point values in the vec register file and floating-point predicates in the vbool register file. Full details of the Vision P6 Half-Precision Vector Floating Point package are in the [Floating Point Operations](#) chapter. Brief descriptions of the operations are in the [floating point operations list](#) section.

The Vision P6 Half-Precision Vector Floating Point Package is independent of, but compatible with, the Vision P6 Single-Precision Vector Floating Point package. Either package can be configured separately or together with the other.

The Vision P6 Half-Precision Vector Floating Point Package is not compatible with and cannot be selected together with the Xtensa Scalar Floating Point coprocessors (single-precision scalar or single and double precision scalar options).

Vision P6 Histogram Package

Histogram operations are common to many image and video processing application. It is possible to achieve a degree of acceleration in histogram computation by SIMD and VLIW architecture of Vision P6 alone.

However, Vision P6 architecture goes one step further and offers SIMD operations specifically designed to accelerate histogram computations.

Histogram operations are not part of the standard DSP Vision P6 ISA. Histogram calculation operations are supported only if the optional Vision P6 Histogram package is enabled. This package is a Vision-specific customer configurable option.

The Vision P6 Histogram package adds a set of operations that accelerate computing histograms containing a modest number of bins.

Histograms are computed by using the histogram count operations to count input values in the ranges of interest and accumulating the resulting counts using a separate ADD operation. N histogram bin values are updated for 4N input values per cycle using the count operations

in this package. The throughput is thus proportional to the number of bins in the histogram. For histograms containing a modest number of bins significant speedup is achieved . For histograms containing a large number of bins, it may be more effective to only update the bins containing the input values using either scalar or gather/scatter operations instead of using the vector count operations in the Histogram package. Typically, histograms containing a few thousand bins are the crossover point where the two approaches have equivalent performance.

An overview of the histogram count operations is in the [Histogram Calculation Acceleration Operations](#) chapter. Detailed descriptions of the count operations are in the Vision P6 ISA HTML documentation. Examples of usage are available in the Vision P6 Software Examples Package.

3. Programming and Optimization Techniques

Topics:

- [*Programming Overview*](#)
- [*Programming in Prototypes*](#)
- [*Xtensa Xplorer Display Format Support*](#)
- [*Programming Styles*](#)
- [*Using the Two Local Data RAMs and Two Load/Store Units*](#)
- [*Other Compiler Switches*](#)
- [*Guidelines for Optimizing Code for Vision P6*](#)
- [*Guidelines for Using iDMA in Vision P6*](#)
- [*Linker Support Package \(LSP\) Considerations for Vision P6 Software Development*](#)

Vision P6 is based on SIMD/VLIW (Single Instruction Multiple Data / Very Long Instruction Word) techniques for parallel processing. It is typical for programmers to do some work to fully exploit the available performance on such architectures. This work may be as simple as recognizing that an existing implementation of an application is already in the right form for vectorization, or it may require a complete reordering of the algorithm's computations to bring together those that can be done in parallel.

This chapter describes several approaches to programming Vision P6. This chapter also explores the capabilities of automated instruction inference and vectorization and includes example cases where the use of intrinsic-based programming is appropriate.

3.1 Programming Overview

Cadence® recommends two important Xtensa manuals to read and become familiar with before attempting to obtain optimal results when programming Vision P6:

- *Xtensa C Application Programmer's Guide*
- *Xtensa C and C++ Compiler User's Guide*

Note that this chapter does not attempt to duplicate the material in either of these guides.

The Vision P6 ISA HTML is also a useful resource for Vision P6 programming. Refer to the chapter [On-Line Information](#) on page 183 for details on where this and other documents can be found.

To use Vision P6 data types and intrinsics in C or C++ programs, the appropriate top level header file must be included by using the following preprocessor directive in the source code file:

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

The `xt_ivpn.h` include file is auto-generated through the Vision P6 core build process and includes a lower-level include file (`xt_ivp32.h`) which has core specific ISA information. Note that the header file `xt_ivpn_verification.h` contains additional `#defines`, but as these are used only for ISA verification purposes they are not documented for programmers' use.

To conditionalize C or C++ programs based on specific Xtensa processor configuration parameters the `core.h` header file is typically included as follows:

```
#include <xtensa/config/core.h>
```

The `core.h` header file includes a lower-level header file (`core-isa.h`) which contains a number of Vision-specific `#defines` that can be used to conditionalize C or C++ programs:

```
#define XCHAL_HAVE_VISION          1 /* 1 for VP5/6 */
#define XCHAL_VISION_TYPE          6 /* 5 for VP5, 6 for VP6 */
#define XCHAL_VISION_SIMD16        32 /* simd16 for VP5/6 */
#define XCHAL_VISION_QUAD_MAC_TYPE 1 /* 0 for none, 1 for quad_mac type on VP6 */
#define XCHAL_HAVE_VISION_HISTOGRAM 1 /* 1 for histogram option on VP5/6 */
#define XCHAL_HAVE_VISION_SP_VFPV 1 /* 1 for sp_vfpv option on VP5/6 */
#define XCHAL_HAVE_VISION_HP_VFPV 1 /* 1 for hp_vfpv option on VP6 */
```

- `XCHAL_HAVE_VISION` is `#defined` to 1 for Vision processors and 0 otherwise.
- `XCHAL_VISION_TYPE` is `#defined` to a value indicating the version of Vision processor, where same version processors support the same standard DSP operation set.
- `XCHAL_VISION_SIMD16` is `#defined` to the Vision processor's vector SIMD width in units of 16-bit-wide elements.

- `XCHAL_VISION_QUAD_MAC_TYPE` is #defined to 1 when the Quad 8x8 MAC operations of Vision P6 are supported and 0 otherwise.⁷
- `XCHAL_HAVE_VISION_HISTOGRAM`, `XCHAL_HAVE_VISION_SP_VFPU` and `XCHAL_HAVE_VISION_HP_VFPU` are #defined to 1 when the corresponding feature is present (configured) and 0 otherwise.

Vision P6 processes integer data and, when either or both of the Vision P6 Half Precision Vector Floating Point or Vision P6 Single Precision Vector Floating Point option packages are enabled, also floating point data. (Note the Vision P6 Histogram option package adds no data types and is not discussed further in this section.) The Vision P6 vec register file data types are listed in the following table.

Table 31: Integer and Floating-point Types which Have no Guard Bits

Description	Scalar Type	Vector Type	Width of Vector
8b signed/unsigned integer	<code>xb_int8</code>	<code>xb_vec2Nx8</code>	64 elements, 512 bits
	<code>xb_int8U</code>	<code>xb_vec2Nx8U</code>	
16b signed/unsigned integer	<code>xb_int16</code>	<code>xb_vecNx16</code>	32 elements, 512 bits
	<code>xb_int16U</code>	<code>xb_vecNx16U</code>	
32b signed/unsigned integer	<code>xb_int32v</code>	<code>xb_vecN_2x32v</code>	16 elements, 512 bits
	<code>xb_int32Uv</code>	<code>xb_vecN_2x32Uv</code>	
16b floating point	<code>xb_f16</code>	<code>xb_vecNxf16</code>	32 elements, 512 bits
32b floating point	<code>xtfloat</code>	<code>xb_vecN_2xf32</code>	16 elements, 512 bits

In addition to the above types, which are unguarded (no guard bits in registers) Vision P6 has additional memory data types shown below which, when loaded into vec register file registers, have guard bits.

Table 32: Integer and Floating-point Types which Have Guard Bits

Description	Scalar Type	Memory Type	Register Type
8b data in memory promoted to 16b data in registers	<code>xb_int8</code>	<code>xb_vecNx8</code>	<code>xb_vecNx16</code>
	<code>xb_int8U</code>	<code>xb_vecNx8U</code>	<code>xb_vecNx16U</code>
16b data in memory promoted to 32b data in registers	<code>xb_int16</code>	<code>xb_vecN_2x16U</code>	<code>xb_vecN_2x32Uv</code>
	<code>xb_int16U</code>	<code>xb_vecN_2x16</code>	<code>xb_vecN_2x32v</code>
16b floating point ⁸	<code>xb_f16</code>	<code>xb_vecN_2xf16</code>	<code>xb_vecN_2xf32</code>

⁷ Future Vision processors may support different variants of Quad MAC operations.

The above scalar Vision P6 data types use element zero of the corresponding vector type. Vision P6 also has the wider wvec register file data types as shown below.

Table 33: Wide Vector Primary Types

Vec Data Type	Wvec Data Type	Description
xb_vecNx16	xb_vecNx48	32 elements of 48 bits each
xb_vecNx16U		
xb_vecN_2x32Uv	xb_vecN_2x64w	16 elements of 64 bits each
xb_vecN_2x32v		
xb_vec2Nx8	xb_vec2Nx24	64 elements of 24 bits each
xb_vec2Nx8U		

Vision P6 ALU/MAC operations that produce results in wvec wrap on overflow. Pack and convert operations that move data from wvec registers to vec registers either truncate without rounding or saturate with rounding. Note there are no direct loads into or stores from the wvec register file. All wvec loads/stores take place through vec registers.

Automatic type conversion is also supported between vector types and the associated scalar types, for example between `xb_vecNx16` and `xb_int16`. 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.

Vision P6 does not explicitly support any fixed-point C data types; it is left up to programmers to use integer data types for fixed-point data. There are MUL operations in Vision P6 that make it easy to process two fixed-point formats: Q15 and Q5.10. Note that these are data formats and not C types. These formats use 16-bit integers.

3.2 Programming in Prototypes

As part of its programming model, Vision P6 defines a number of operation protos (prototypes or intrinsics) for use by the compiler in code generation, for use by programmers with data types other than the generic operation support, and to provide compatibility with related programming models and DSPs. For programmers, the most important use of protos is to provide alternative operation mappings for various data types. If the data types are

⁸ 16-bit floating point is both a storage format and a computation format in Vision P6. It is supported as a storage format when either or both of the Vision P6 Half Precision Vector Floating Point or Vision P6 Single Precision Vector Floating Point option packages are enabled. It is supported as a computation format only when the Vision P6 Half Precision Vector Floating Point option package is enabled.

compatible, one operation can support several variations without any extra hardware cost. For example, protos allow vector select operations to support all types of integer and floating point data types. The same select hardware operation can support `xb_vecN_2x32v`, `xb_vecN_2x32Uv` and `xb_vecN_2xf32` data types using different protos. In some cases, protos also encapsulate a commonly used set of operations into a single proto - for example for implementing divides using divide steps.

Some protos are meant for compiler usage only, although advanced programmers may find a use for them in certain algorithms (for example `IVP_OPERATOR*`, `*rtor*`, `*mtor*` protos fall in this category). The complete list of protos can be accessed via the ISA HTML.

Move Protos

Vision P6 additionally provides a “`IVP_MOV`” category of protos used to cast (coerce) between data types. For vector inputs, Vision P6 `IVP_MOV` protos cast from one data type to another data type without changing the size/content of the vectors. They allow the programmer to override the type system by interpreting a representation in one type as a different type. These protos, which do not have a cycle cost, are helpful in multi-type programming environments. While the data is interpreted differently, the content and length of the vector remains the same within the register file. This category of protos follows the format:

```
IVP_MOV<after_type>_FROM<before_type>
```

For example, `IVP_MOVNx16_FROM2Nx8` changes the type from `xb_vec2Nx8` to `xb_vecNx16` without changing the underlying representation.

Note that a full set of `IVP_MOV` protos to coerce between arbitrary types is not provided. `IVP_MOV` protos are provided between all C types and the default C type of a register file. In general type coercion between two arbitrary types requires going through the default C type. As a convenience `IVP_MOV` protos are provided for some common coercions where neither type is the default C type.

Operator Overload Protos

The proto HTML documentation also includes the Operator protos for compiler usage. Note that Cadence recommends usage by advanced programmers only. When using intrinsics, you may pass variables of different types than what is expected, as long as there is a defined conversion from the variable type to the type expected by the intrinsic. Operator overloading is only supported if there is an intrinsic with types that exactly match the variables. Implicit conversion is not allowed, since with operator overloading you are not specifying the intrinsic name, and the compiler does not guess which intrinsics might match. The resultant intrinsic is necessary for operator overloading, but there is no advantage in calling it directly.

3.3 Xtensa Xplorer Display Format Support

Xtensa Xplorer provides support for a wide variety of display formats, which makes using and debugging the varied data types easier. 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, by default, always display their entire content as a single hexadecimal number. However, you can change the format to any other format.

For example if we load these 64 bytes:

0,4,8,0xc,...,0xfc

to a vector of type `xb_vec2Nx8U`, then they will be displayed as

(252,248,...,0)

But the same bytes in a vector of type `xb_vec2Nx8` will be displayed as

(-4,-8,-12,...,0)

3.4 Programming Styles

Typically, programmers have to spend some effort on their code, especially legacy code, to make it run efficiently on a vector 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.

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

- Auto-vectorizing scalar C code
- C code with vector data types and operator overloading (manual vectorization)
- Use of C intrinsic functions along with vector data types (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 the Vision P6 special scalars - `xb_int16`, `xb_int8`, etc). Once the correctness of code using the scalar data-types is determined, the first step is to attempt to automatically vectorize critical portions of the code. Automatically vectorized code can then be further manually optimized and code that failed to automatically vectorize can be manually vectorized using vector data types and operator overloading. 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 automatic vectorization and refining only the most computationally intensive portions of code manually, the engineering effort can be directed to where it has the most effect, which is discussed in the next sections.

Auto-vectorization

Auto-vectorization of scalar C code using Vision P6 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. Gather/scatter operations (see [Gather/Scatter Operation Descriptions](#) section) are not currently supported by the auto-vectorizer.

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*, particularly 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 keep intermediate results, respectively.
- For best performance, load/store data should be aligned to the vector size. The XCC compiler will naturally align arrays to start, 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`. This option tells the compiler that it can assume data is always aligned.
- Pointer aliasing causes problems with vectorization. The `__restrict` attribute for pointer declarations (e.g. `short * __restrict cp;`) tells the compiler that the pointer does not alias. There are global compiler aliasing options, but these can sometimes be dangerous.
- Subtle C/C++ semantics in loops may make them impossible to vectorize. The `-LNO:simd_v` feedback can assist in identifying small changes that allow effective vectorization.
- Irregular or non-unity strides in data array accessing can be a problem for vectorization. Changing data array accesses to regular unity strides can improve results, even if some "unnecessary computation" is necessary.
- Outer loops can be simplified wherever possible to allow inner loops to be more easily vectorized. Sometimes trading outer and inner loops can improve results.
- Loops containing function calls and conditionals may prevent vectorization. It may be better to duplicate code and perform a little "unnecessary computation" to produce better results.

- Array references, rather than pointer dereferencing, can make code (especially mathematical algorithms) both easier to understand and easier to vectorize.

Operator Overloading

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
- & bitwise AND
- ^ bitwise XOR
- | bitwise OR
- << bitwise left shift
- >> bitwise right shift
- ~ bitwise NOT or one's complement
- < less than
- <= less than or equal to
- > greater than
- >= greater than or equal to
- == equal to

The next section illustrates how they work in conjunction with automatic and manual vectorization.

Auto-vectorization Examples

Consider the following scalar C-code:

```
void Filt3TapVectorAuto(int16_t *pcoeff, int16_t *pvecin, int16_t *pvecout, int32_t veclen)
{
    int16_t(*__restrict pc) = pcoeff;
    int16_t(*__restrict pv) = pvecin;
    int16_t(*__restrict pf) = pvecout;
    int16_t temp;
    int32_t indx;

    for (indx = 0; indx < veclen; indx++)
    {
        temp = (pv[indx] * pc[0]);
        temp += (pv[indx + 1] * pc[1]);
        temp += (pv[indx + 2] * pc[2]);
        pf[indx] = temp;
    }

    return;
}
```

When compiled with the `-O3 -LNO:simd` compiler options, the main processing loop vectorizes. The compiler unrolls the loop by a factor of 4, and schedules it in 12 cycles. So effectively we achieve 3 cycles for processing the loop 32 times. Note that we have used the qualifier `__restrict` to inform the compiler that data accessed via each of the pointers `pcoeff`, `pvecin`, and `pvecout` can be loaded and stored independent of any other data in the code.

```
{ivp_la2nx8_ip v0,u0,a2; nop; ivp_mulnx16packl v3,v12,v1;}
{ivp_la2nx8_ip v5,u0,a2; nop; ivp_mulanx16packl v10,v13,v4;}
{ivp_sa2nx8_ip v7,u1,a3; ivp_la2nx8_ip v8,u0,a2; ivp_mulanx16packl v9,v14,v6;}
{nop; nop; ivp_mulnx16packl v7,v12,v0; ivp_selnx16i v2,v0,v1,0;}
{nop; ivp_la2nx8_ip v1,u0,a2; ivp_mulanx16packl v10,v14,v11; ivp_selnx16i v4,v0,v1,1;}
{ivp_sa2nx8_ip v9,u1,a3; nop; ivp_mulnx16packl v9,v12,v5; ivp_selnx16i v6,v5,v0,0;}
{nop; nop; ivp_mulanx16packl v3,v13,v2; ivp_selnx16i v0,v5,v0,1;}
{ivp_sa2nx8_ip v10,u1,a3; nop; ivp_mulnx16packl v10,v12,v8; ivp_selnx16i v2,v8,v5,0;}
{nop; nop; ivp_mulanx16packl v3,v14,v4; ivp_selnx16i v4,v1,v8,0;}
{nop; nop; ivp_mulanx16packl v7,v13,v6; ivp_selnx16i v6,v8,v5,1;}
{nop; nop; ivp_mulanx16packl v9,v13,v2; ivp_selnx16i v11,v1,v8,1;}
{ivp_sa2nx8_ip v3,u1,a3; nop; ivp_mulanx16packl v7,v14,v0;}
```

Next, consider the same function implemented in floating point scalar C-code:

```
void Filt3TapVectorAuto(float *pcoeff, float *pvecin, float *pvecout, int32_t veclen)
{
    float(*__restrict pc) = pcoeff;
    float(*__restrict pv) = pvecin;
    float(*__restrict pf) = pvecout;
    float temp;
    int32_t indx;

    for (indx = 0; indx < veclen; indx++)
    {
        temp = (pv[indx] * pc[0]);
        temp += (pv[indx + 1] * pc[1]);
        temp += (pv[indx + 2] * pc[2]);
        pf[indx] = temp;
    }

    return;
}
```

When compiled with `-O3 -LNO:simd` compiler options, the main processing loop vectorizes. The compiler unrolls the loop by a factor of 8, and schedules it in 24 cycles. So effectively we achieve 3 cycles for processing the loop 32 times. Note that the unrolling factor is two times larger than the unrolling used in integer code. This is because the pipeline latency of the floating point operations is larger than that of the integer operations. Hence, additional software pipelining is needed to achieve the optimal processing throughput.

```
{ivp_lv2nx8_ip v0,a2,192; nop; ivp_mulan_2xf32 v14,v22,v8;}
{ivp_lv2nx8_i v4,a2,-128; ivp_lv2nx8_i v8,a2,-64; ivp_muln_2xf32 v2,v20,v1; nop;}
{ivp_sv2nx8_ip v9,a14,64; ivp_lv2nx8_ip v11,a2,192; ivp_mulan_2xf32 v10,v22,v15; nop;}
{ivp_lv2nx8_i v13,a2,-128; ivp_lv2nx8_i v15,a2,-64; ivp_muln_2xf32 v5,v20,v0; ivp_seln_2x32 v3,v0,v1,v24;}
```

```

{ivp_sv2nx8_ip v12,a14,64; ivp_lv2nx8_ip v18,a2,128; ivp_muln_2xf32 v9,v20,v4; ivp_seln_2x32
v6,v4,v0,v24;}
{ivp_sv2nx8_ip v14,a14,64; nop; ivp_muln_2xf32 v12,v20,v8; ivp_seln_2x32 v7,v4,v0,v23;}
{ivp_sv2nx8_ip v16,a14,64; ivp_lv2nx8_i v1,a2,-64; ivp_mulan_2xf32 v2,v21,v3; ivp_seln_2x32
v0,v0,v1,v23;}
{ivp_sv2nx8_ip v10,a14,64; nop; ivp_muln_2xf32 v14,v20,v11; ivp_seln_2x32 v10,v8,v4,v24;}
{nop; nop; ivp_mulan_2xf32 v5,v21,v6; ivp_seln_2x32 v3,v8,v4,v23;}
{nop; nop; ivp_muln_2xf32 v16,v20,v13; ivp_seln_2x32 v4,v11,v8,v24;}
{nop; nop; ivp_mulan_2xf32 v9,v21,v10; ivp_seln_2x32 v6,v11,v8,v23;}
{nop; nop; ivp_mulan_2xf32 v2,v22,v0; ivp_seln_2x32 v8,v13,v11,v24;}
{nop; nop; ivp_mulan_2xf32 v19,v22,v17; ivp_seln_2x32 v0,v18,v15,v24;}
{nop; nop; ivp_mulan_2xf32 v5,v22,v7; ivp_seln_2x32 v17,v15,v13,v24;}
{nop; nop; ivp_muln_2xf32 v10,v20,v15; ivp_seln_2x32 v7,v1,v18,v24;}
{nop; nop; ivp_mulan_2xf32 v12,v21,v4; ivp_seln_2x32 v4,v15,v13,v23;}
{nop; nop; ivp_mulan_2xf32 v14,v21,v8; ivp_seln_2x32 v8,v13,v11,v23;}
{ivp_sv2nx8_ip v19,a14,64; nop; ivp_mulan_2xf32 v16,v21,v17; ivp_seln_2x32 v15,v18,v15,v23;}
{ivp_sv2nx8_ip v2,a14,64; nop; ivp_muln_2xf32 v19,v20,v18; ivp_seln_2x32 v17,v1,v18,v23;}
{ivp_sv2nx8_ip v5,a14,64; nop; ivp_mulan_2xf32 v10,v21,v0;}
{nop; nop; ivp_mulan_2xf32 v9,v22,v3;}
{nop; nop; ivp_mulan_2xf32 v12,v22,v6;}
{nop; nop; ivp_mulan_2xf32 v16,v22,v4;}
{nop; nop; ivp_mulan_2xf32 v19,v21,v7;}

```

Manual Vectorization Example

In the above integer multiply example we achieved vectorization using the packL flavor of the mulNX16 instruction. If we want to use the packQ or packR versions of the mulNX16 operation, or use the accumulator followed by explicit, user-specified “pack” operation, we need to use manual vectorization. Consider a similar multiply example with manual vectorization using the vecNX48 accumulator, and explicit packvr instruction:

```

void xvMulAcc(int16_t *pvec1, int16_t *pvec2, int16_t *psum, int32_t vecLen)
{
    xb_vecNx16 vecData1;
    xb_vecNx16 vecData2;
    xb_vecNx16 vecResult;
    xb_vecNx16 * __restrict pvecData1;
    xb_vecNx16 * __restrict pvecData2;
    xb_vecNx16 * __restrict pvecResult;
    xb_vecNx48 accResult;
    int32_t ix;

    pvecData1 = (xb_vecNx16 *) pvec1;
    pvecData2 = (xb_vecNx16 *) pvec2;
    pvecResult = (xb_vecNx16 *) psum;
    for (ix = 0; ix < (vecLen / IVP_SIMD_WIDTH); ix++)
    {
        vecData1 = *pvecData1++;
        vecData2 = *pvecData2++;
        accResult = vecData1 * vecData2;
        vecResult = IVP_PACKVRNX48(accResult, 7);
        *pvecResult++ = vecResult;
    }

    return;
}

```


Note that we have used a mix of operator overloading, and direct use of intrinsics in the code above. The compiler unrolls the processing loop in the above code by a factor of two, and the compiled output looks like:

```
{ivp_lv2nx8_ip v0,a7,128; ivp_packvrnx48 v2,wv0,a9; ivp_mulnx16 wv0,v1,v0; nop;}
{ivp_sv2nx8_i v5,a4,-64; ivp_lv2nx8_ip v1,a3,128; nop; nop;}
{ivp_lv2nx8_i v3,a7,-64; ivp_packvrnx48 v5,wv1,a9; ivp_mulnx16 wv1,v4,v3; nop;}
{ivp_sv2nx8_ip v2,a4,128; ivp_lv2nx8_i_n v4,a3,-64;}
```

Note that when we load / store data using the overloaded indirection operator “*”, the compiler assumes that the pointer and the load / store operation is aligned to the vector width. Hence, the compiler uses the aligned load / store operations and achieves a better code packing efficiency (2 cycles for processing 32 elements, compared to the 3 cycles achieved earlier.)

N-way Programming Model

Programmers are strongly encouraged to use the **Vision N-way programming model** on Vision P6. N-way programming is a way to write code that is portable across current and potential future SIMD width alternatives in the Vision family.

N-way programming is supported through the use of Vision's generic N-way vector C types and "XCHAL" variables for machine-specific parameters such as SIMD width. On Vision processors the generic N-way vector C types correspond to one SIMD vector of data where the number of elements depends on the specific machine's SIMD vector width. For example, the `xb_vecNx16` type is a vector of 32 16-bit (integer) elements on Vision P6. The SIMD width measured in 16-bit elements is defined by `XCHAL_VISION_SIMD16` in the `core-isa.h` header file.⁹ For Vision P6 this definition is as follows:

```
#define XCHAL_VISION_SIMD16 32
```

The following example illustrates the use of the generic vector types and `XCHAL_VISION_SIMD16` with the N-way programming model:

```
xb_vecNx16 vin[VIN_LEN / XCHAL_VISION_SIMD16];
xb_vecNx16 vout = 0;
for (i = 0; i < (VIN_LEN / XCHAL_VISION_SIMD16); i++)
    vout += vin[i] * vin[i];
*out_p = IVP_RADDNX16(vout);
```

⁹ In some earlier imaging machines this was defined by `XCHAL_IVPN_SIMD_WIDTH` in `xt_ivpn.h` instead. This earlier definition is still provided for backwards compatibility, but new code should use `XCHAL_VISION_SIMD16`.

This code is portable to another Vision core with a different N, e.g., N=8, N=16. (This example assumes `VIN_LEN` is always a multiple of `XCHAL_VISION_SIMD16`; handling arbitrary `VIN_LEN` would complicate this simple example.)

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

Vision P6 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 Instruction Set Simulator (ISS) options. If a single instruction issues two loads to the same bank of local memory (and to two different addresses within the bank), the processor will stall for one cycle to do the second load. Stores are buffered by the hardware so it can often sneak into a cycle that does not access the same memory. This access contention is not modeled by the ISS unless you select the `--mem_model` simulation parameter. Thus, it is important to select memory modeling when studying the code performance.

If you are using the standard set of LSPs (Linker Support Packages) provided with your Vision P6 configuration, and do not have your own LSP, you may consider using the "sim-stacklocal" LSP. The "sim-stacklocal" LSP is very similar to the "sim" LSP, with the only difference being that "sim-stacklocal" places the stack in the local memory, while "sim" LSP places the stack in the system memory. Alternately, you can start with the "sim" LSP and create a custom LSP, as per the requirements. 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[NSAMP][ARR_SIZE][ARR_SIZE] __attribute__((section(".dram1.data")));
```

This code declares a short 3-dimensional array `ar`, and places it in DataRAM 1. 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 C-Box to access the two data RAMs. There are two controls that provide this further level of control. These are documented in Chapter 4 of the *Xtensa C and C++ Compiler User's Guide*. 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 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.
- If your configuration does not have a C-Box, you should not use `-mcbox` as you are constraining the compiler to avoid an effect that does not apply. (Note a C-Box is required for Vision P6.)

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

3.6 Other Compiler Switches

The following two other compiler switches are important:

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

3.7 Guidelines for Optimizing Code for Vision P6

It is recommended to try the following general optimization techniques at the C level for getting performance improvement:

- **Loop unrolling:** This technique gives more opportunities for scheduling.
- **Loop splitting:** This technique helps in reducing register pressure. Generates simple loops which can be either auto-vectorized or scheduled better by the compiler in case of manual vectorization.
- **Swapping of inner and outer loops:** To get a higher loop count for the innermost loop. Improve data fetch for each of vectorization. Note that horizontal loop iterations in Vision P6 can be quite small because they increment in multiples of the vector size, while vertical iteration counts are higher because they often increment by one.
- **Unrolling of the outer loop:** To have vectorizable code in inner loop.
- **Code restructuring to have linear data access:** This is less of an issue for Vision P6 because of gather/scatter.
- **Simplify data storage:** In some cases separating interleaved data to separate buffers might help.
- **Use of compiler pragmas and restrict keyword**
- **Function inlining**
- Effective use of dual load/stores, for example, by proper data placement in data RAMs
- Use of aligned data when possible
- Effectively hiding gather/scatter latency
- Reducing gather/scatter conflicts

Optimizing Gather/Scatter

Gather/scatter is a key feature of Vision P6 that allows efficient vectorization of functions that **access data in non-linear fashion**. There are two main considerations to keep in mind for efficient use of the Vision P6 gather/scatter functionality:

- Effective hiding of gather/scatter latency
- Reducing or avoiding sub-bank conflicts

Gather/scatter operations have relatively long latency, as described in the [Gather/Scatter Operations](#) chapter. To use them effectively, **the programmer must try to hide the latency by doing useful work in those cycles**. Several techniques can be used for this:

- Make sure gather/scatter loops have enough other operations in the loop that can be performed while gather/scatter operations are being performed. For example, gather/scatter address calculations for the next set of gather/scatters or processing of gathered data or generation of scattered data. In some cases, combining loops allows use of the gather/scatter latency cycles.

- Queuing multiple gather/scatters at a time. Since gather/scatters are pipelined and also the hardware will try to optimize access across multiple gather/scatters, this will generally perform better.
- Rearranging the data or access pattern to reduce **sub-bank conflicts**. This can be done by controlling the tile stride during DMA, using local memory to local memory DMA to change the tile stride of data in local memory, or changing the function that produces data for gathers or consumes data generated by scatters.
- Gather/scatter protos have a version (`_v`) that allows the programmer to specify the number of access cycles expected for any gather/scatter. This gives the compiler more information on how to schedule gather/scatters to hide the latency when possible. If the typical number of accesses for a gather/scatter are known, then it is recommended to use this interface. For example, the code fragment below tells the compiler that this gather is expected to need three cycles of accesses for every call:

```
vecD = IVP_GATHERNX8U_V(ptr, vecI, 3);
```

- Gather/scatter operations have predicated versions and if it is known that some of the lanes' data is don't-care then using predicated gather/scatters can potentially reduce sub bank conflicts.
- Just like normal pointers, the `__restrict` keyword tells the compiler that gather/scatter pointers do not alias with other pointers. This allows for better scheduling.
- Gather/scatters can also conflict with load/stores if they access the same data RAM banks. So, just like for load/stores, data placement as described in the [Using the Two Local Data RAMs and Two Load/Store Units](#) section is also important for gather/scatter to avoid data RAM bank conflicts with load/stores.

We recommend profiling critical loops using gather/scatter to get an idea of the number of stalls due to gather/scatter, and then apply the above techniques to optimize the code further. A further step is to analyze the addresses in gather/scatter to get an idea of sub-bank conflicts expected. If this number is high, then data layout or the access pattern can be changed.

3.8 Guidelines for Using iDMA in Vision P6

Vision P6 configurations normally include iDMA (integrated-DMA) for transferring data between external memory and local data RAM independent of, and **in parallel** with, Vision P6 processor execution. Refer to the *Xtensa LX7 Microprocessor Data Book* for a complete description of the iDMA.

iDMA transfers to/from local data RAM can be lower priority than Vision P6 local data RAM accesses. Still, in use cases with high load/store and DMA bandwidth requirements, a PING-PONG buffering scheme where processor and iDMA buffers are kept in different data RAMs is recommended. Thus, when iDMA is transferring data to/ from PING buffer in one data

RAM, Vision P6 core is processing data from/into PONG buffer in another data RAM and vice-versa.

A general recommended guideline to be adopted during application software design / architecture is to explore opportunities to reduce the data transfer bandwidth for iDMA as much as possible. For example, consider a scenario where there is a requirement of applying three functions fnA, fnB, and fnC successively on an image frame. Here, instead of transferring the image data to and from the system memory and data RAM three times separately to apply each of these functions, the programmer should explore the possibility of applying two or even all three functions on the data transferred to the local memory. It is possible that to achieve this, more data may need to be transferred initially compared to the case where we are applying only a single function. This can be the case for example where we need additional edge data to process an image tile. However this additional upfront data transfer will help reduce the overall data transfer bandwidth.

iDMA programming is supported by the Integrated DMA Library (iDMAlib) software library described in detail in the *Xtensa System Software Reference Manual*. Vision P6 also supports a higher level software layer called the Tile Manager which provides a higher level interface for image tile transfer using iDMA. Refer to the *Vision P6 Software Package User's Guide* for more details on the Tile Manager.

3.9 Linker Support Package (LSP) Considerations for Vision P6 Software Development

A Linker Support Package (LSP) provides information to the linker about object placement within the memory map of a target system. For detailed information on Xtensa LSPs, refer to the *Xtensa Linker Support Packages (LSP) Reference Manual*.

This section highlights key considerations for LSP use with Vision P6. Xtensa processor configurations include standard LSPs as well the ability to customize LSPs. Vision and Imaging applications almost always require the stack to be placed in the local memory. This is achieved by using the "sim-stacklocal" LSP which is very similar to the "sim" LSP, except that the stack is placed in the local memory.

Some LSP guidelines for Vision P6 software development are:

- When the ISS with `--mem-model` is used for simulation, accurate modelling of accesses to local and system memory is performed. In this case, it is critical to use an LSP that, like `simApp`, places frequently-accessed data and the stack in local data RAM. Additionally, if instruction RAM is available, critical code may be placed there. This is particularly important when developing gather/scatter Vision P6 kernels because accurate modelling of their performance requires `--mem-model`.
- Non-gather/scatter Vision P6 kernel development can use ISS without `--mem_model`, at least during initial development. In this case, programs can be linked with the standard `sim` LSP since no special care needs to be taken to locate data in local data RAM.

- When developing complete application software, in addition to efficient kernel development, one is also interested in making efficient use of limited local memory resources. A key design feature is dividing the image to be processed into image segments (tiles), and transferring the tiles in and out of local data RAM using iDMA so that the iDMA transfer cycles are masked by the Vision P6 processing cycles. In this case, it is also critical to use an LSP that places the stack and other data in local data RAM appropriately.
- Other important parameters are the memory wait state (latency) parameters specified to ISS. These model the delay encountered in accessing external memory and are useful to understand performance with a realistic memory system. When external memory system performance is not being modelled it is reasonable to set the latency numbers to 0.

4. Standard DSP Operations by Type

Topics:

- [*DSP Operations by Type*](#)
- [*Move Operations*](#)

This section serves as a quick overview for groups of Vision operations which are not part of an optional package.

Detailed descriptions of each operation are provided in the Vision P6 ISA HTML.

Most operations are described with unevaluated N, which is equal to 32 for Vision P6. This is done on purpose to get used to the N-way programming model.

4.1 DSP Operations by Type

This section serves as a quick overview for groups of Vision operations which are not part of an optional package.

Detailed descriptions of each operation are provided in the Vision P6 ISA HTML.

Most operations are described with unevaluated N, which is equal to 32 for Vision P6. This is done on purpose to get used to the N-way programming model.

Load & Store Operations

Vision P6 is well balanced between load-store bandwidth and computational power. With two 512-bit each wide data paths between the local data memories and register files, the machine is rarely load/store bound.

Overview

In addition to the basic 32-bit load/store operations, Vision P6 features a rich set of vector load/store instructions which is used in vector data processing. Combined with two 512 bits (64 bytes) wide load store units these instructions can sustain very high throughput between SIMD registers and the local data memories.

These are some capabilities offered by load and store operations:

1. Up to two 512-bit (each) loads or one 512-bit load and one 512-bit store operations can be performed in parallel with one cycle throughput
2. A single load operation which loads 1024 bits of data with one cycle throughput
3. Unaligned load/store operations (can load from byte addresses not multiple of 64 bytes)
4. Load and store operations which load/store a variable number of elements
5. Predicated load and store operations
6. Scalar in vector register load and store operations, and scalar in vector register load operations replicating one loaded element to all the lanes of the output vector register.
7. Promoting loads with sign extension and zero extension, demoting stores with sign saturation and truncation.
8. Support for four addressing modes (I,IP,X,XP) (not every operation supports all four addressing modes)

Convention and terminology used in describing load and store operations

Predication:

T – Predicated on TRUE for loads: not all the elements of the output vector are written to registers with new elements from memory. Only predicated TRUE lanes of the vector register are filled with data fetched from memory. All bits are set to 0 in register lanes which are not predicated TRUE. Non-predicated elements which are not written to register are “skipped”

and not “squeezed” in memory. For example, if Elements 0,1,2 of a vector register corresponding to memory addressed 0x1020, 0x1022, 0x1024 are loaded but only Elements 0 and 2 are predicated, they are loaded from addresses 0x1020 and 0x1024, while nothing is loaded from address 0x1022 (“this address is skipped”) and all bits in the vector register lane corresponding to Elements 1 are set 0.

T – Predicated on TRUE for stores: not all the elements of the input vector are written to memory. Only predicated TRUE lanes of the vector register are written to memory. Register lanes which are not predicated TRUE are not written to memory. Non-predicated elements which are not written to memory are “skipped” and not “squeezed” in memory. For example, if Elements 0,1,2 corresponding to memory addressed 0x1020, 0x1022, 0x1024 are stored to memory but only Elements 0 and 2 are predicated, they are written to addresses 0x1020 and 0x1024, while address 0x1022 is not written to (“skipped”).

Load and store addressing modes:

_I the load/store address is the sum of the input base address in an address register and the offset encoded in an input immediate. The base address in the address register is not updated (not written by the operation)

_X the load/store address is the sum of the input base address in an address register and the offset in another input address register. The base address in the address register is not updated (not written by the operation)

_IP the load/store address is the input base address in an address register. The sum of the input base address in the address register and the offset encoded in an input immediate is written back to the address register which contains the base address after loading or storing data.

_XP the load/store address the input base address in an address register. The sum of the input base address in the address register and the offset stored in another, input, address register is written back to the base address register (same address register that contained the input base address) after loading or storing data.

Load and Store Exceptions

When attempting to load or store, under certain conditions, an exception can be triggered. Below is the list of these conditions:

1. If a store or load operation which requires an aligned address, tries to access an unaligned address a `LoadStoreAlignmentCause` exception is raised
2. A memory protection violation will raise a `LoadProhibitedCause` for a load or a `StoreProhibitedCause` for a store exception
3. Attempting to access non-existent memory will not raise a load/store exception unless the MPU attributes for the memory space corresponding to the non-existing memory are set to read/write protected. In this case accessing non-existing memory will result in a memory protection violation and a `LoadProhibitedCause` exception will be raised for a load or `StoreProhibitedCause` exception will be raised for a store.

Note that load/store exception behavior of Vision P6 is the same as that of the base Xtensa core.

Non-aligning Loads to Vector Register

These load operations load data from data memory to narrow vector registers and require aligned addresses in data memory. Unaligned addresses will raise a `LoadStoreAlignmentCause` exception.

IVP_LV2NX8[T]_[I|IP|X|XP]

The Load Vector operation loads 64 bytes from an aligned data memory location to a narrow vector register. The element size (8 bits) does not have particular significance; these operations simply load a vector of elements (512 bits) regardless of element size. Note the representation of vbool values allows vboolN, vbool2N and vboolN_2 values to be directly used as byte-level predicates in Load Vector operations.

IVP_LV[N_2X16|NX8][S|U][T]_[I|IP|X|XP]

These promoting Load Vector operations load N/2X16 or NX8 packed bits (from contiguous memory addresses without gaps) to N/2 X32 and NX16 representations in a vec register by either sign extending or zero extending individual elements, e.g. 16 bits to 32 bits. [16U] operations zero extend the 16-bit values from memory, while [16S] sign extend the 16-bit elements to 32 bits. [8S] operations sign extend 8-bit elements from memory to 16 bits.

The alignment requirement is with respect to the packed data size in memory, i.e. 32 bytes.

Non-aligning Stores of a Vector Register

These store operations store data from narrow vector registers to data memory and require aligned addresses in data memory otherwise `LoadStoreAlignmentCause` exception is raised.

IVP_SV2NX8[T]_[I|IP|X|XP]

Stores the content of a vector register (excluding non-predicated lanes for predicated operations variants).

IVP_SV[N_2X16][NX8]U[T]_[I|IP|X|XP]

These operations truncate N/2 X 32-bits to N/2X16-bits or NX16-bits to NX8-bits elements before storing them. The alignment requirement is with respect to packed data size in memory, i.e. 32 bytes.

IVP_SV[N_2X16][NX8]S[T]_[I|IP|X|XP]

These operations saturates signed data, N/2 X 32-bits to N/2X16-bits or NX16-bits to NX8-bits elements before storing them. The alignment requirement is with respect to packed data size in memory, i.e. 32 bytes.

Alignment Registers

The Vision P6 memory interface does not allow direct unaligned loads or stores. Operations which perform unaligned (or aligning load) of a word (e.g. loading 64 bytes from an address which is not a multiple of 64 bytes) rely on loading two aligned words, one containing the upper part of the required word and the other – the lower part, then concatenating the upper and lower parts to construct the final requested unaligned word.

The purpose of alignment registers is to store an aligned word which has one part (typically the lower part) of the unaligned word. When an aligning load is invoked, the aligning load operation performs an aligned load from memory and then concatenates part of the just loaded data with data from an alignment register. In order to do so, the alignment register needs to be loaded with proper data first. This “pre-loading” of the required data to an alignment register is also known as “priming” the alignment register. Alignment registers are used in loading narrow vector registers, their width is equal to that of narrow vector registers.

Loading Alignment Register

The following operation is used to load (prime) an alignment register

IVP_LA_PP

The operation takes the input address from AR, clears 6 lsbs (least-significant bits) of the address value, and uses the new address value to load 64 bytes into 512-bit-wide alignment register. This operation is also called priming of an alignment register. This operation is typically used before an unaligned load operation.

IVP_LALIGN_[I|IP]

LALIGN stands for Load Alignment (register). Unlike **IVP_LA_PP**, this operation does not clear lsbs of the address value and requires an aligned (a multiple of 64 bytes) address. Unlike **IVP_LA_PP** it is typically used for restoring an alignment register content after a spill to memory.

This is NOT an Aligning Load (LA) and will raise an exception if a non-aligned address is passed.

These operations load 512 bits from memory into an output alignment vector register.

Storing Alignment Register

IVP_SALIGN

SALIGN stands for Store Alignment (register). This is NOT an Aligning Store (SA) and will raise an exception if a non-aligned address is passed.

These operations store the entire 512-bit content of an alignment register to memory. If the memory address is not a multiple of 64 bytes, (i.e. 512 bits), an exception is raised.

IVP_SAPPOS_FP

Flushes data in an alignment register to memory. The amount of flushed data may be less than the size of the register. After storing data the content of the register is set to 0. This operation is used at the end of a sequence of aligning store(s) to memory to guarantee that all data ends up in memory and is not held in the alignment register.

Clearing an Alignment Register

IVP_ZALIGN

Clears all bits (sets to 0) of the alignment register. This is not a load/store operation, it does not access memory, but this operation is essential for performing aligning stores.

Aligning Loads

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

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

Note the Xtensa C Compiler (XCC) for Vision P6 always aligns arrays with at least 2N bytes of data to 2N-byte alignment, where N is the SIMD length for 16-bit data. If the C code development is not in a XCC environment, users may need to explicitly force alignment appropriate to the environment used.

Aligning Stores

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

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

Start with the IVP_ZALIGN operation to store an array of vectors beginning at an aligning memory address. This operation initializes the alignment register's contents and clears the flag bit. A series of aligning stores following the IVP_ZALIGN operation will store one vector to memory per operation. Note that when using stores that write 512-bits to memory the first store operation of this series will perform a partial store because the flag bit was cleared by the IVP_ZALIGN operation. Each subsequent store will perform a full 512-bit store because the first (and subsequent) store operations set the flag bit. Finally, a flush operation flushes out the last remaining bytes in the alignment register.

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

Aligning Load Operations

Aligning load operations fill the entire vector register and do not require aligned addresses, but an alignment register must be primed in order for these operations to complete a load from an unaligned address.

IVP_LA2NX8_[IP|XP]

Loads 64 bytes from an unaligned (or aligned) address from contiguous memory addresses into a 64-byte vector register. These loads require a primed alignment register unless the address to memory is aligned.

IVP_LA[N_2X16|NX8][S|U]_[IP|XP]

These are promoting aligning loads, 8 and 16 bit values read from memory are promoted (through zero or sign extension depending on whether the operation signed or unsigned) to 16 and 32-bit elements before being written to the output vector register. N (32) 8-bit or N/2 (16) 16-bit elements come from memory (if the address is aligned) or from memory and the alignment register if $\log_2(2N) \text{ lsb} + N > N/2$.

If $\log_2(2N) \text{ lsb} + N \leq N/2$ then a memory read is not performed and all the elements are loaded onto vector register from the alignment register (with zero or sign extension).

IVP_LAT2NX8_XP

Predicated TRUE load from an unaligned (or aligned) address from contiguous memory addresses into a vector register. Only the vector register lanes corresponding to vector Boolean lanes with values 1 (TRUE) are written, all the bits in other lanes are set 0. These load requires a primed alignment register unless the address in memory is a multiple of 64 bytes.

Aligning Load Operations which Use both Load Units

These operations are “the widest” load operations which can load up to 128 bytes of data to fill two vector registers. All the operations utilize both load units and as the result have a usage restriction— they cannot be issued in a FLIX bundle with another load operation, or with a store operation.

IVP_L2A4NX8_IP loads 128 consecutive bytes from an unaligned address in AR into two narrow vector registers. The address is post-incremented by an implicit 128 bytes and is written back to AR. The data comes from an aligning register and two load units. The alignment register must be primed for this operation. This operation targets streaming type of sequential loads.

IVP_L2U2NX8_XP loads 64 consecutive bytes from an unaligned address into one vector register. Although it is an aligning load, this operation does not require an alignment register, it reads 128 bytes from an aligned address in memory and loads a 64 byte unaligned portion of the 128 bytes into vec. This operation targets unaligned vector loads with disparate addresses.

IVP_L2AU2NX8_IP loads 64 consecutive bytes from an unaligned address in AR into one vec register. The operation is similar to IVP_L2U2NX8_XP but differs in two ways:

1. At the end of the operation the 64 MSBytes are written to an alignment register.
2. The addressing mode is IP not XP, the address is post increment by implicit 64 bytes and written back to AR.

This operation targets the first load in a streaming load sequence.

Aligning Stores from a Vector Register

IVP_SA2NX8_IP

Stores to an unaligned (or aligned) address 64 bytes from a vector register. In order to achieve aligned stores to actual memory, an aligning store uses an alignment register as a buffer between the input data and the physical memory. Thus after executing this type of operation the input data can end up in the alignment buffer, memory, or both. To guarantee that all data is written to memory after a sequence of one or more aligning stores, the IVP_SAPPOS_FP operation is used.

IVP_SA[N_2X16|NX8]U_IP

Demoting unsigned SA operations are similar to **IVP_SA2NX8_IP**, but they truncate each element in the source vector register to the next size down, i.e. 32-bit elements to 16 bits for

N_2X16 versions and 16-bit elements to 8 bits for NX8 versions. Demoting unsigned SA operations store only 32 bytes to memory.

IVP_SA[N_2X16|NX8]S_IP

Demoting signed SA operations are similar to **IVP_SA2NX8_IP**, but they sign saturate each element in the source vector register to the next size down, i.e. 32-bit elements to 16 bits for N_2X16 versions and 16-bit elements to 8 bits for NX8 versions. Demoting signed SA operations store only 32 bytes to memory.

Variable Aligning Loads to Vector Register

Aligning variable load operations (LAV) do not require aligned addresses, but an alignment register must be primed in order for these operations to complete a load from an unaligned address. Unlike non-variable (non-predicated) aligning loads (LA) aligning variable load operations (LAV) do not always write 64 bytes from memory to the vector registers. The number of elements which are loaded to the vector register is variable and is specified through a user defined argument. Just as in the case of an LA load, an actual memory load is not always performed, but in LAV operations avoid extra memory accesses not only based on the address, but also on the number of bytes specified by the user. There are no predicated versions of variable aligning loads

IVP_LAV2NX8_XP

Load aligning variable, LAV, operations write the number of elements specified by the user to the lsbs of the destination vector register. The rest of the elements in the register is set to 0

IVP_LAV[N_2X16][NX8][U]S_XP

Promoting load aligning variable, LAV, operations write the number of bytes specified by the user to the lsbs of the destination vector register. The rest of the elements in the register are set to 0

Just as in the case of LA operations promoting LAV for N_2X16 and NX8, sign [S] or zero [U] extends 16 and 8-bit data loaded from memory to 32 and 16 bits respectively.

Variable Aligning Stores from a Vector Register

IVP_SAV[N_2X16|NX8]U_XP

Demoting unsigned store aligning variable (SAV) operations truncate and store a variable number of contiguous least significant elements in a vector register. The number is specified by the user as an input operand. Since these are unsigned demoting operations, they truncate elements in the source vector register to the next size down, i.e. 32-bit elements to 16 bits for N_2X16 version and 16-bit elements to 8 bits for NX8 versions, before storing.

Note that the AR input specifies the number of bytes in memory, the corresponding number of bytes in the source register is twice that number.

IVP_SAV[N_2X16|NX8]S_XP

Demoting signed store aligning variable (SAV) operations sign saturate and store a variable number of contiguous least significant elements from a vector register. The number is specified by the user as an input operand. Since these are signed demoting operations, they sign saturate elements in the source vector register to the next size down, i.e. 32-bit elements to 16 bits for N_2X16 version and 16-bit elements to 8 bits for NX8 versions, before storing.

Note that the AR input specifies the number of bytes in memory, the corresponding number of bytes in the source register is twice that number.

Loading Vector Boolean Register (Vbool)

These load operations load data from data memory to vector Boolean register. These load operations require aligned addresses to data memory otherwise an exception is raised.

IVP_LB[N|2N|N_2]_[I|IP]

These operations load a 2N-bit vector Boolean register. If the number of lanes are equal to the number of bits in Vbool, IVP_LB2N, all 64 bits are loaded from memory. For N and N/2 cases, only 32 or 16 bits are read from memory respectively and then replicated to fill each lane of the vbool register. These operations require aligned addresses, otherwise a LoadStoreAlignmentCause exception is raised. The alignment requirement is with respect to the memory read, e.g. 4 –byte alignment is required for IVP_LBN.

Storing Vector Boolean Register

These store operations store data to data memory from the vector Boolean register. These store operations require aligned addresses in data memory otherwise an exception is raised.

IVP_SB[N|2N|N_2]_[I|IP]

Stores 1 bit (lsb) from each lane of a vector Boolean register to memory. Since a vector Boolean (vbool) register is $2*N$ bit wide (each bit represent one lane), IVP_SB2N stores all the bits of the vbool registers. This is not an aligning store, if the memory address is not a multiple of M bytes: M is the number of bytes stored in memory, e.g. M=8 for IVP_SB2N, M=4 for IVP_SBN, M=2 for IVP_SBN_2.

Multiplication Operations

Vision supports SIMD multiply, multiply-add (multiply-accumulate), and multiply-subtract operations. Vision P6 can sustain the following throughput per cycle:

- Up to one-hundred-twenty-eight 8-bit x 8-bit or 8-bit x 16-bit multiply or multiply-add pair operations
- Up to two-hundred-fifty-six 8-bit x 8-bit multiply or multiply-add quad operations
- Up to sixty-four 16-bit x 16-bit multiply, multiply-add or multiply-subtract operations
- Up to sixteen 16-bit x 32-bit multiply, multiply-add or multiply-subtract operations

There are also “H” high 16-bit x 32-bit multiply operations which shift the result by 16-bits to the left, these are present to make more efficient software implementations of 32x32 multiplies.

Operations with the Output to Wide Vector Register (wvec)

For [UU] operations the result is zero extended to the corresponding output lane width; for [US|SU] operations the result is sign extended to the corresponding output lane width.

$a[i]=b[i]*c[i]$ or $a[i]=b[i]*c$ - regular SIMD multiplication, where i is a SIMD lane index, $c[i]$ is a value in a SIMD lane and c is a scalar from AR (the same for all lanes)

$a[i]+=b[i]*c[i]$ or $a[i]+=b[i]*c$ - regular SIMD multiply-accumulate, where i is a SIMD lane index, $c[i]$ is a value in a SIMD lane and c is a scalar from AR (the same for all lanes)

A note on pair and quad multiplications (and multiply-accumulates). A multiplication pair operation performs the number of multiplications, which is twice the size of its SIMD output. For example, a 64-way multiplication pair produces 64 results, but performs 128 multiplications. Similarly, a 64-way quad multiplication produces also 64 results, but performs 256 multiplications. This is achieved by a reduction sum performed across corresponding lanes of different vectors (not along the lanes of the same vector). The expressions below illustrate this:

$a[i]=b[i]*c[i]+d[i]*e[i]$ or $a[i]=b[i]*c+d[i]*e$ - SIMD multiplication pair, where i is a SIMD lane index, $c[i]$, $e[i]$ are values in SIMD lanes and c and e are scalars from AR (the same for all lanes of one vector)

$a[i]+=b[i]*c[i]+d[i]*e[i]$ or $a[i]+=b[i]*c+d[i]*e$ - SIMD multiply-accumulate pair, where i is a SIMD lane index, $c[i]$, $e[i]$ are values in SIMD lanes and c and e are scalars from AR (the same for all lanes of one vector)

$a[i]=b[i]*f+c[i]*g+d[i]*h+e[i]*p$ - quad SIMD multiplication, where i is a SIMD lanes index, and f , g , h , p are scalars from AR (the same for all lanes of one vector)

$a[i]+=b[i]*f+c[i]*g+d[i]*h+e[i]*p$ - quad SIMD multiply-accumulate, where i is a SIMD lanes index, and f , g , h , p are scalars from AR (the same for all lanes of one vector)

$a[i]=bc[i]*d+bc[i+1]*e+bc[i+2]*f+bc[i+3]*g$ - a 4 tap 8-bit convolution start (quad SIMD multiplication).

$a[i]+=bc[i]*d+bc[i+1]*e+bc[i+2]*f+bc[i+3]*g$ - continuation of an 8-bit convolution (quad SIMD multiplication and accumulation).

Here d,e,f,g are scalars from AR - the four taps of the filter, and bc is a vector, which is formed by the concatenation of the first input vector b and second input vector c , i.e. $bc[4*N-1]=b[2*N-1]$, $bc[2*N]=b[0]$, $bc[2*N-1]=c[2*N-1]$, $bc[0]=c[0]$. IVP_MUL4T[A]2N8XR8 are the operations which perform these computations.

Table 34: Multiplies with the Output to Wide Vector Register

Operation Name	Description
IVP_MUL[[UU US SU]N_2X16X32_[0 1]	32x16 bit multiplication with 64 bit result in wvec
IVP_MUL[[UU US SU]AN_2X16X32_[0 1]	32x16 bit multiply-accumulate with 64 bit accumulator in wvec
IVP_MUL[[UU US SU]SN_2X16X32_[0 1]	32x16 bit multiply-subtract with 64 bit accumulator in wvec
IVP_MUL[[UU US]SNX16	16x16 bit multiply-subtract with 48 bit accumulator in wvec
IVP_MUL[[UU US SU]HN_2X16X32_1	32x16 bit multiplication with 64 bit result in wvec shifted by 16 to the left. 16-bit multiplier comes from odd lanes of vec
IVP_MUL[[UU US SU]AHN_2X16X32_1	32x16 bit multiply-accumulate with 64 bit accumulator in wvec. Result is shifted by 16 before addition in wvec. 16-bit multiplier comes from odd lanes of vec
IVP_MUL[[UU US SU]SHN_2X16X32_1	32x16 bit multiply-subtract with 64 bit accumulator in wvec Result is shifted by 16 before subtraction in wvec. 16-bit multiplier comes from odd lanes of vec
IVP_MUL[US]2N8XR16	64-way 8-bit vector x 16-bit address register (AR) scalar scalar-vector multiply - producing wide (24-bit) results
IVP_MUL[[UU US]2NX8	64-way 8-bit multiply - producing wide (24-bit) results with sign-extension or zero extension for [UU]
IVP_MUL[US]A2N8XR16	64-way 8-bit vector x 16-bit address register (AR) scalar scalar-vector multiply-accumulate - producing wide (24-bit) results
IVP_MUL[[UU US]A2NX8	64-way 8-bit multiply-accumulate - producing wide (24-bit) results with sign-extension
IVP_MUL[US]AI2NR8X16	64-way 8-bit address register (AR) scalar x 16-bit interleaved vector scalar-vector multiply-accumulate - producing wide (24-bit) results
IVP_MUL[US]AI2NX8X16	64-way 8-bit x 16-bit multiply-accumulate - producing wide (24-bit) results
IVP_MUL[[UU US]ANX16	32-way 16-bit multiply-accumulate - producing wide (48-bit) results with sign-extension or zero extension for [UU]

Operation Name	Description
IVP_MUL[US]I2NR8X16	64-way 8-bit address register (AR) scalar x 16-bit interleaved vector scalar-vector multiply - producing wide (24-bit) results
IVP_MUL[US]I2NX8X16	64-way 8-bit x 16-bit multiply - producing wide (24-bit) results
IVP_MUL[[UU]US]NX16	32-way 16-bit multiply - producing wide (48-bit) results with sign-extension or zero extension for [UU]
IVP_MUL[[UU]US]P2NX8	64-way 8-bit x 8-bit vector by vector multiply pair. This operation performs per element vector by vector multiplication of one pair of input vectors and also per element vector by vector multiplication of another pair of input vectors, adds the corresponding elements of the two products together producing a 64-way 24-bit wide result.
IVP_MUL[[UU]US]PA2NX8	64-way 8-bit x 8-bit vector by vector multiply-accumulate pair. This operation performs per element vector by vector multiplication of one pair of input vectors and also per element vector by vector multiplication of another pair of input vectors, adds the corresponding elements of the two products together producing an intermediate 64-way 24-bit wide result. Finally, intermediate result is added to the values in the inout wide vector and written back to the same vector overwriting the wide input values.
IVP_MUL[US]PI2NR8X16	64-way 8-bit scalar x 16-bit vector multiplication pair with interleaved outputs. The product of AR[15:8] and the first input vector is added to the product of the second input vector and AR[7:0] and the sums are written to 32 odd lanes of output wide vector. The product of AR[15:8] and the third input vector is added to the product of the fourth input vector and AR[7:0] and the sums are written to 32 even lanes of the output wide vector.
IVP_MUL[US]PAI2NR8X16	64-way 8-bit scalar x 16-bit vector multiply-accumulate pair with interleaved outputs. The product of AR[15:8] and the first input vector is added to the product of the second input vector and AR[7:0] and the sum is added to the content of 32 odd lanes of inout wide vector. The sum is written back to these lanes overwriting the wide vector input. The product of AR[15:8] and the third input vector is added to the product of the fourth input vector

Operation Name	Description
	and AR[7:0] and the sum is added to the content of the 32 even lanes of the inout wide vector. The sum is written back to these lanes overwriting the wide vector input
IVP_MUL[US]P2N8XR16	64-way 8-bit vector x 16-bit scalar multiplication pair. The product of the first input vector and AR[31:16] is added to the product of the second input vector and AR[15:0] and 64 per element sums are written to the output wide vector.
IVP_MUL[US]PA2N8XR16	64-way 8-bit vector x 16-bit scalar multiply-accumulate pair. The product of the first input vector and AR[31:16] is added to the product of the second input vector and AR[15:0] and 64 per element sums are added to the elements of inout wide vector. The results are written back to the inout wide vector overwriting the input values.
IVP_MUL[US]4T2N8XR8	4-tap 8-bit coefficient convolution start with 64-element 8-bit vector. The two 64 element (each) input vectors are concatenated. The concatenated vector is shifted by 0,1,2, and 3 bytes. In each shifted vector 64 least significant bytes are extracted to form four 64element vectors. Each vector is then multiplied by a different scaler: AR[31:24], AR[23:16], AR[15:8], and AR[7:0]. The four products are added and the 64 sums are written to the output wide vector.
IVP_MUL[US]4TA2N8XR8	4-tap 8-bit coefficient convolution continuation with 64-element 8-bit vector. The two 64 element (each) input vectors are concatenated. The concatenated vector is shifted by 0,1,2, and 3 bytes. In each shifted vector 64 least significant bytes are extracted to form four 64element vectors. Each vector is then multiplied by a different scaler: AR[31:24], AR[23:16], AR[15:8], and AR[7:0]. The four products are added and their 64 sums added to the inout wide vector. The results are written back to the inout wide vector overwriting the input values.
IVP_MUL[US]Q2N8XR8	64-way 8-bit vector by 8-bit scalar quad multiplication. The elements of the first, second, third, and fourth input vectors are multiplied by scalars in AR[31:24], AR[23:16], AR[15:8], and AR[7:0] respectively. The

Operation Name	Description
IVP_MUL[US]QA2N8XR8	<p>corresponding elements of the four 64-element products are added together and the 64 sums are written to the output wide vector.</p> <p>64-way 8-bit vector by 8-bit scalar quad multiply-accumulate. The elements of first, second, third, and fourth input vectors are multiplied by scalars in AR[31:24], AR[23:16], AR[15:8], and AR[7:0] respectively. The corresponding elements of the four 64-element products are added together and the 64 sums added to the elements of inout wide vector. The results are written back to the inout wide vector overwriting the input values.</p>
IVP_MUL[[UU]US]PNX16	<p>32-way 16-bit vector by 16-bit vector multiplication pair. The product of the first and the second input vectors is added to the product of the third and fourth output vectors, the 32 sums are written to the output wide vector.</p>
IVP_MUL[[UU]US]PANX16	<p>32-way 16-bit vector by 16-bit vector multiply-accumulate pair. The product of the first and the second input vectors is added to the product of the third and fourth output vectors, the 32 sums are added to the inout wide vector. The results are written back to the inout wide vector overwriting the 48-bit input values.</p>
IVP_MUL[[UU]US]PN16XR16	<p>32-way 16-bit vector by 16-bit scalar multiplication pair. The product of the first input vector with AR[31:16] and the second input vectors with AR[15:0] are added together, the 32 sums are written to the output wide vector</p>
IVP_MUL[[UU]US]PAN16XR16	<p>32-way 16-bit vector by 16-bit scalar multiply-accumulate pair. The product of the first input vector with AR[31:16] and the second input vectors with AR[15:0] are added together and added to the 32 values in the inout wide vector. The results are written back to the inout wide vector overwriting the input values.</p>

Operations with the Output to (narrow) Vector Register (vec)

Table 35: Multiplies with the Output to Narrow Vector Register

Vision P name	Description
IVP_MULANX16PACKL[T]	32-way 16-bit signed multiply-accumulate - producing narrow (16-bit) integer results with the lower 16-bits of result. 16-bits are obtained by truncation before the accumulation.
IVP_MULANX16PACKP[T]	32-way 16-bit signed multiply-accumulate - producing narrow (16-bit) fractional results with the shifted high-order 16 bits of result, without saturation. 16 bits are obtained before the accumulation.
IVP_MULANX16PACKQ[T]	32-way 16-bit signed multiply-accumulate - producing narrow (16-bit) fractional results with the shifted high-order 16 bits of result, without saturation
IVP_MULNX16PACKL	32-way 16-bit real signed multiply - producing narrow (16-bit) integer results with the lower 16-bits (truncated) of result.
IVP_MULNX16PACKP	32-way 16-bit signed multiply - producing narrow (16-bit) fractional results with the shifted high-order 16 bits of result, without saturation
IVP_MULNX16PACKQ	32-way 16-bit signed multiply - producing narrow (16-bit) fractional results with the shifted high-order 16 bits of result, without saturation
IVP_MULSNX16PACKL	32-way 16-bit signed multiply-subtract - producing narrow (16-bit) integer results with the lower 16-bits of result. 16 bits are obtained before the subtraction
IVP_MULSNX16PACKP	32-way 16-bit signed multiply-subtract - producing narrow (16-bit) fractional (Q5.10) results with the shifted high-order 16 bits of result, without saturation. 16 bits are obtained before the subtraction.
IVP_MULSNX16PACKQ	32-way 16-bit signed multiply-subtract - producing narrow (16-bit) fractional (Q15) results with the shifted high-order 16 bits of result, without saturation. 16 bits are obtained before the subtraction.

Division Operations

Vision P6 has a powerful hardware support for SIMD divide operations. When a divide operation is implemented in software it typically takes dozens of cycles of a recursive algorithm to complete the divide. Vision P6 can sustain 32 16-bit by 16-bit integer and fixed-point divisions every eight cycles, and 16 32-bit by 16-bit integer and fixed-point divisions every eight cycles.

Features

Vision P6 supports divide operations intended to perform the following:

1. 8-step signed and unsigned N/2-way 32-bit/16-bit integer divisions
2. 4-step signed and unsigned N/2-way 16-bit/16-bit integer divisions.
3. 4-step signed and unsigned N/2-way 16-bit/16-bit fractional divisions.

List of Operation

Table 36: Divide Operations

Vision P name	Description
IVP_DIVN_2X32X16S_4STEP0	First 4 steps of 32bit/16bit signed integer divide
IVP_DIVN_2X32X16U_4STEP0	First 4 steps of 32bit/16bit unsigned integer divide
IVP_DIVN_2X32X16S_4STEP	Intermediate 4 steps of 32bit/16bit signed integer divide
IVP_DIVN_2X32X16U_4STEP	Intermediate 4 steps of 32bit/16bit unsigned integer divide
IVP_DIVN_2X32X16S_4STEPN	Last 4 steps of 32bit/16bit signed integer divide
IVP_DIVN_2X32X16U_4STEPN	Last 4 steps of 32bit/16bit unsigned integer divide
IVP_DIVNX16S_4STEP0	First 4 steps of 16bit/16bit signed integer divide
IVP_DIVNX16U_4STEP0	First 4 steps of 16bit/16bit unsigned integer divide
IVP_DIVNX16S_4STEP	Intermediate 4 steps of 16bit/16bit signed integer divide
IVP_DIVNX16U_4STEP	Intermediate 4 steps of 16bit/16bit unsigned integer divide
IVP_DIVNX16S_4STEPN	Last 4 steps of 16bit/16bit signed integer divide
IVP_DIVNX16U_4STEPN	Last 4 steps of 16bit/16bit unsigned integer divide
IVP_DIVNX16SQ_4STEP0	First 4 steps of 16bit/16bit signed fractional divide
IVP_DIVNX16Q_4STEP0	First 4 steps of 16bit/16bit unsigned fractional divide

Special Case Handling

The following are special cases. They arise when one or more input element takes on an illegal value. Vision P6 behavior in these special cases is described here for debugging purposes only. Programmers must NEVER rely on this behavior when writing code for execution under normal conditions. The behavior of Vision P6 when handling special conditions is subject to change.

In all these cases, the remainder returned by the operation is 0.

1. signed integer division: smallest value divided by -1 returns the largest signed positive value (0x7fff for 16-bit/16-bit, 0x7fffffff for 32-bit/16 bit)
2. signed integer division: divide by 0 returns either the largest signed positive or negative value depending on the sign of the dividend (0x7fff or 0x8000 for 16-bit/16-bit, 0x7fffffff or 0x80000000 for 32-bit/16-bit for a positive and a negative dividend respectively)
3. unsigned integer division: divide by 0 returns the largest unsigned value (0xffff for 16-bit/16-bit, 0xffffffff for 32-bit/16-bit)
4. unsigned fractional division: divide by 0 returns the largest unsigned value 0xffff
5. signed fractional division: returns either the largest signed positive or negative value (0x7fff or 0x8000) depending on the sign of the dividend
6. unsigned fractional division: if the dividend is greater or equal to the divisor, then the operation returns the largest unsigned value (0xffff)
7. signed fractional division: if the magnitude of the dividend is greater or equal to that of the divisor, either the largest signed positive value (0x7fff) if the signs of the dividend and the divisor are the same, or smallest negative value (0x8000) if the signs are different is returned.

Usage

All divide operations implement only one step of a complete division. For example, to complete a 16-bit by 16-bit signed division, a series of division operations are executed: IVP_DIVNX16S_4STEP0, followed by two IVP_DIVNX16S_4STEP operations, followed by one IVP_DIVNX16S_4STEPN.

A number of protos are provided which encapsulate these operations to accomplish end-to-end divides.

These protos are listed below:

IVP_DIVNX16()

Performs a complete 32-way 16-bit by 16-bit signed integer divide

IVP_DIVNX16U()

Performs complete 32-way 16-bit by 16-bit unsigned integer divide

IVP_DIVNX16SQ()

Performs a complete 32-way 16-bit by 16-bit signed fractional divide

The magnitude of the dividend must be smaller than that of the divisor, the result is in a shifted to the left by 15 (signed) fractional format, i.e., the MSB (Bit 15) is the sign bit, Bit 14 is 1/2 , Bit 13 1/4, Bit 12 1/8, etc.

IVP_DIVNX16Q()

Performs a complete 32-way 16-bit by 16-bit unsigned fractional divide

The dividend must be smaller than the divisor, the result is in a shifted to the left by 16 (unsigned) fractional format, i.e., the MSB (Bit 15) is 1/2, Bit 14 is 1/4, Bit 13 1/8, Bit 12 1/16, etc.

IVP_DIVN_2X32X16()

Performs a complete 16-way 32-bit by 16-bit signed integer divide

IVP_DIVN_2X32X16U()

Performs a complete 16-way 32-bit by 16-bit unsigned integer divide

The programmer is discouraged from using individual division step intrinsics explicitly (and encouraged to use protos above which implement complete division instead) in Vision P6 source code, since individual instructions which implement division steps are not guaranteed to be same in possible future variants of the Vision processor.

Vector Compression and Expansion

Vision P6 is a very **wide SIMD machine**.

When processing a wide “swath” of pixels or other data items, it is efficient to process “needed” data elements in the “swath” and ignore unneeded elements in one step. Predicated operations address this issue. Often we can increase the processing efficiency by combining/packing needed elements into one register from multiple registers and then processing a single register. This is the idea behind vector compression.

After the compressed vector has been processed it is often required to put the data elements back to their “original” places, and this is the idea behind vector expansion.

Vector Compression

Vector compression is typically performed in two steps.

We start by receiving a vector Boolean register with lanes corresponding to elements which we need to process set to TRUE. In the first step we apply a SQUEEZE operations to put the vector Boolean register lane numbers which are set to TRUE in consecutive elements of a narrow vector register. In the second step we use a *shuffle operation* to put all the elements which we need to process in contiguous elements of a narrow vector. We can repeat the steps above to put more elements into the same narrow vector until it only contains elements which need to be processed and none that don't.

IVP_SQZN

Squeeze generates a vector of element indices, to be used by a subsequent [IVP_SHFLNX16](#) operation, for the purpose of squeezing vector elements according to a Boolean mask.

The operation reads a Boolean vector from a vbool register file, containing 32 logical Boolean values. It then creates a packed series of element values, packed to the least significant elements of the vector, containing the position of any Boolean TRUE values, starting from the least significant position and appending bit positions until one value is created for each Boolean one. The remaining elements above the elements corresponding to Boolean ones are filled with the select value 32. The vector of select values is written to a narrow vector register

The operation also counts the number of Boolean ones in the input vector, multiplies the count by 2, to create a count of the bytes squeezed and writes that count to an address register.

For example, suppose that N=8 elements (N=8 and 1 bit per vector Boolean lane for illustration purposes only, N=32 for Vision P6), and the input vector Boolean register lanes are 0b01001001. Then the output vector the register would contain (0,0,0,0,0,6,3,0) and the output address register would contain $(2*3) = 6$.

As another example, suppose N=8 elements (N=8 and 1 bit per vector Boolean lane for illustration purposes only, for Vision P6 N=32), and the input Boolean register contained 0b01001000. Then the output vector register would contain (0,0,0,0,0,0,6,3) and the output address register would contain $(2*2) = 4$.

Vector Expansion

Vector expansion takes elements in a narrow vector corresponding to ones in vector Boolean register, copies it to a certain element index in another result narrow vector and fills the gaps in the result narrow vector by replicating the elements. Vector expansion is typically performed in two steps. We start by receiving a vector Boolean register with lanes corresponding to elements which we need to expand set to TRUE. In the first step we apply an UNSQUEEZE operations. In the second step we use a [shuffle operation](#) to put the input elements narrow vector elements with the computed indices in a narrow vector and replicate them. Shuffle operation find use in many computations beyond vector compression and expansion, it easy to comprehend and is described in the ALU operations section. Unsqueeze is not that straightforward and is described here

IVP_UNSQZN

Unsqueeze operation generates a vector of element numbers, to be used by an IVP_SHFLNX16 operation, for the purpose of unsqueezing vector elements according to a Boolean mask. The shuffle selection values are output as a set of N (N=32 for Vision P6) elements to vector register.

The operation reads a vector Boolean register containing N logical Boolean values. It counts the number of ones in the Boolean vector (count). The operation creates a set of up to N selection values to move the count lowest elements to the positions corresponding to the ones in the vector Boolean register. The lowest order element (element zero) in output vector

register is always set to zero. Each higher order element (from one up to N-1) will contain a count of the number of *Shuffle Operations* on page 121 ones in the input vector Boolean register, starting at the next lower element and going down to element zero. See below for two examples. In other words, element i of output vector register will always contain a count of the number of ones in elements 0..(i-1) of input vector Boolean register.

The vector of select values is written to the output vector register.

The operation also counts the number of Boolean ones in the input vector Boolean register, multiplies the count by 2, to create a count of the bytes unsqueezed and writes that count to an address register.

For example, suppose that N=8 elements (N=8 and 1 bit per vector Boolean lane for illustration purposes only, for Vision P6 N=32), and the input vector Boolean register contained 0b01001001. Then the output vector register would contain (3;2;2;2;1;1;1;0), and the output address register would contain $(2*3) = 6$.

As another example, suppose N=8 elements (N=8 and 1 bit per vector Boolean lane for illustration purposes only, for Vision P6 N=32), and the input Boolean register contained 0b01001000. Then the output vector register would contain (2;1;1;1;0;0;0;0), and the output address register would contain $(2*2) = 4$.

Arithmetic Operations

IVP_ABS[2NX8|NX16|N_2X32]

Computes absolute value non-saturating; the absolute value of the minimum signed integer is the minimum signed integer.

IVP_ABSSNX16

Computes absolute value with saturation: the absolute value of the minimum signed integer at the input saturates to the signed maximum value at the output.

IVP_ABSSUB[U][2NX8|NX16]

Absolute value of a difference, non-saturating. If the result is interpreted as an unsigned value (for both signed and unsigned inputs) overflow is not possible.

IVP_ABSSSUBNX16

Computes absolute value of a difference with saturation. If the absolute value of the difference is greater than the maximum signed integer the result saturates to the maximum signed positive value.

IVP_ADD[2NX8|NX16|N_2X32][T]

The sum of corresponding elements in two input vector registers. These operations are non-saturating and applicable to both signed and unsigned values.

IVP_ADDSNX16[T]

Saturating signed sum of corresponding elements in two input vector registers.

IVP_ADDMOD16U

This is a scalar operation, which computes the sum of the unsigned value of the least significant 16 bits of the second address register and the unsigned value contained in the least significant 16 bits ([15:0]) of the first of address register modulo the unsigned value specified in the most significant 16 bits ([31:16]) of the first address register. A 16 bit result is computed, zero-extended to 32 bits and written to the output address register.

This operation does not perform a complete modulo operation on the sum of the values contained in the least significant 16 bits of the two inputs. In the case where the input values are not in the correct range, the output may not be in the desired range.

IVP_ADDWS[2NX8|NX16]

Signed accumulating subtract. It sign extends two 2NX8 vec inputs values from 8 to 24 bits and then subtracts their sum from the inout 2NX24 wvec and writes the result back to the same register overwriting the wvec input.

IVP_ADDWUS[2NX8|NX16]

Unsigned accumulating subtract. It zero extends two 2NX8 vec inputs values from 8 to 24 bits and then subtracts their sum from the inout 2NX24 wvec and writes the result back to the same register overwriting the wvec input.

IVP_ADDW[2NX8|NX16]

IVP_ADDWU[2NX8|NX16]

Signed and unsigned add, sum written to wvec.

IVP_ADDWA[2NX8|NX16]

IVP_ADDWUA[2NX8|NX16]

Signed and unsigned accumulating add to wvec. These operations add two vec inputs to wvec input and write the result back to wvec overwriting wvec input.

IVP_SUBW[2NX8|NX16]

IVP_SUBWU[2NX8|NX16]

Signed and unsigned subtract, difference written to wvec.

IVP_SUBWA[2NX8|NX16]

IVP_SUBWUA[2NX8|NX16]

Signed and unsigned accumulating subtract to wvec. These operations subtract two vec inputs, add the difference to the wvec input and write the result back to wvec overwriting wvec input.

IVP_AVGR[U][2NX8|NX16]

Signed and unsigned average of two vec inputs with rounding. The operation averages elements of the vector as $av=(input1+input2)/2$ and rounds the result to the nearest integer: if non-integer part of av is < 0.5 the result rounds down, otherwise it rounds up. The result is written to output vec retaining its input width, i.e. $2NX8 \rightarrow 2NX8$, $NX16 \rightarrow NX16$.

IVP_AVG[U][2NX8|NX16]

Signed and unsigned average of two vec inputs without rounding. The operation averages elements of the vector as $av=(input1+input2)>>1$, but does not perform any rounding. The result is written to output vec retaining its input width, i.e. $2NX8 \rightarrow 2NX8$, $NX16 \rightarrow NX16$.

IVP_BADDNORMNX16

Adds two corresponding lanes from two NX16 vec inputs, producing NX16 vec and vboolN outputs. If the result of the addition in a particular lane does not overflow 16 bits, this is just a regular 16-bit addition and the 16-bit result of the addition, $vec_lane[15:0]=result[15:0]$, is written to the corresponding output vec lane. If the result of adding two 16-bit numbers in a particular lane overflows, the results in that lane is shifted right by 1, forming a normalized vec output, $vec_lane[15:0]=result[16:1]$, bit 0 is dropped. This operation generates a Boolean mask: if the result in a particular vec lane overflows, the corresponding Boolean register lane is set to 1, otherwise it is set to 0.

IVP_BSUBNORMNX16

Subtracts corresponding lanes of the second NX16 vec input from the first, producing NX16 vec and vboolN outputs. If the result of the subtraction in a particular lane does not overflow 16 bits, this is just a regular 16-bit subtraction and the 16-bit result of the subtraction, $vec_lane[15:0]=result[15:0]$, is written to the corresponding output vec lane. If the result of subtracting two 16-bit numbers in a particular lane overflows, the results in that lane undergoes right arithmetic shift by 1 forming a normalized vec output, $vec_lane[15:0]=result[16:1]$, bit 0 is dropped. This operation generates a Boolean mask: if the result in a particular vec lane overflows, the corresponding Boolean register lane is set to 1, otherwise it is set to 0.

IVP_BMAX[U][2NX8|NX16|N_2X32]

IVP_BMIN[U][2NX8|NX16|N_2X32]

These are min/max operations that also generate a Boolean mask. The operations take two inputs and produce two outputs.

For MAX operation, the maximum of the two corresponding elements of the first and second input is written to the output vec. In addition, if the value of element i of the first input vector register is greater than the value of element i of the second input vector register, then element i of the output vector Boolean register is set to 1; otherwise it is set to 0.

For MIN operations, the minimum of the two corresponding elements of the first and second input is written to the output vec. In addition, if the value of element i of the first input vector register is less than the value of element i of the second input vector register, then element i of the output vector Boolean register is set to 1; otherwise it is set to 0.

IVP_MAX[U]2NX8

IVP_MAX[U]NX16[T]

IVP_MAX[U]N_2X32

IVP_MIN[U]2NX8

IVP_MIN[U]NX16[T]

IVP_MIN[U]N_2X32

These are min/max operations. The operations take two vector inputs and produce one vector output.

For MAX operation, the maximum of the two corresponding elements of the first and second input is written to the output vec. For MIN operations, the minimum of the two corresponding elements of the first and second input is written to the output vec.

IVP_MULSGN[NX16|N_2X32]

The multiply sign operation will multiply each element in the second input vector register by the sign of the corresponding element in the first input vector register. If the element in the first input vector register is zero, the output element will be zero.

IVP_NEG[2NX8|NX16|N_2X32]

IVP_NEGNX16T

Negative of the input – non-saturating

IVP_NEGSNX16[T]

Negative of the input – saturating

IVP_SUB[2NX8|NX16|N_2X32][T]

Calculates the differences between corresponding elements in two input vector registers (elements in the second vector are subtracted from elements in the first vector). These operations are non-saturating and applicable to both signed and unsigned values. Overflow and underflow are not detected.

Bitwise Logical Operations

IVP_AND2NX8

Bitwise AND operations between two input vector registers.

This operation can be used with datatypes of multiple widths because it is bit-parallel.

IVP_ANDB

Bitwise AND operations between two input vector Boolean registers

This operation can be used with datatypes of multiple widths because it is bit-parallel.

IVP_ANDNOTB

Bitwise Boolean AND of the first input vector Boolean register with the NOT of the second input vector Boolean register. The result is written to the output vector Boolean register.

This operation can be used with datatypes of multiple widths because it is bit-parallel.

IVP_NOTB

Calculates bitwise Boolean NOT for the input vector Boolean register

This operation can be used with datatypes of multiple widths because it is bit-parallel.

IVP_NOT2NX8

Calculates bitwise Boolean NOT for the input vector register

This operation can be used with datatypes of multiple widths because it is bit-parallel.

IVP_ORB

Bitwise OR operation between two input vector Boolean registers

This operation can be used with datatypes of multiple widths because it is bit-parallel.

IVP_OR2NX8

Calculates bitwise Boolean OR for the input vector register

This operation can be used with datatypes of multiple widths because it is bit-parallel.

IVP_ORNOTB

Bitwise Boolean OR of the first input vector Boolean register with the NOT of the second input vector Boolean register. The result is written to the output vector Boolean register.

IVP_ANDNOTB1

The bitwise ANDNOT of the least significant bit of the two input vector Boolean registers is computed and placed into the least significant bit of output vector Boolean register making other bits 0. The ANDNOT of bits A_bit0 and B_bit0 is computed as (A_bit0 & ~B_bit0).

IVP_ORNOTB1

The bitwise ORNOT of the least significant bit of the two input vector Boolean registers is computed and placed into the least significant bit of the output vector Boolean register making all other bits 0. The ORNOT of bits A_bit0 and B_bit0 is computed as (A_bit0 | ~B_bit0).

IVP_NOTB1

The bitwise NOT of the least significant bit of the input vector Boolean register is computed and placed into the least significant bit of the output vector Boolean register making all other bits 0.

IVP_XOR2NX8

The bitwise XOR of each bit of the two input narrow vector registers is computed and placed into the output narrow vector register. This operation can be used with datatypes of multiple widths because it is bit-parallel.

IVP_XORB

The bitwise XOR of each bit of the two input vector Boolean registers computed and placed into the output vector Boolean registers. This operation can be used with datatypes of multiple widths because it is bit-parallel

IVP_RANDB[2N|N|N_2]

The AND reduction of the elements of the input vector. The result is a vbool1.

IVP_RORB[2N|N|N_2]

The OR reduction of the elements of the input vector. The result is a vbool1.

Bit Manipulation Operations

IVP_EXTBI2NX8

Takes 2NX8 input lanes from vec a register and one 3-bit immediate. The immediate values are interpreted as a bit position of a bit with a vec lane. Immediate value 0 corresponds to the lsb bit, 1 to the next bit after lsb; 7 to msb. This operation copies the bit in the bit position specified by the immediate from each vec lane, to the corresponding lane of the vector Boolean register. In Vision P6 vbool registers are 64-bit wide, so each vbool lane for this operation is 1 bit.

IVP_EXTR[NX16|2NX8|N_2X32]

Extracts one element of narrow vector input register and writes it into output address register. The element to be extracted is determined by the input immediate. For the 16-bit and 8-bit element variants, the element is sign-extended to 32 bits before being written to output address register.

IVP_EXTRVRN_2X32

Copies consecutive 32 bits from the input narrow vector register to the output address register. The 32 bits to be copied are specified as a byte offset in the input AR register. For example, the offset value of 3 indicates that bits $[3 \cdot 8 + 31 : 3 \cdot 8]$ from the input narrow vector register are to be copied to $[31 : 0]$ of the output AR. The offset value in AR is saturated to 0..64 inclusive. Virtual bytes 64, 65, 66, 67 in input vec all have values 0. As stated above the operation extracts 32 bits, but these bits are byte aligned, not 32-bit aligned.

IVP_EXTRACTBH

Calculates a 1-bit split of the upper half of input Boolean register into an output Boolean register by sign-extending each input bit Boolean to 2 bits for each of the N elements of the upper half of the Boolean simd register input. The output elements are written out in exactly

the same order as the input elements, with element N/2 of input vector register being mapped to the lowest two bits (1,0) of output Boolean register, and element N-1 of input vector register being mapped to the highest two bits of output Boolean register. The result elements are written to output vector Boolean register.

IVP_EXTRACTBL

Calculates a 1-bit split of the lower half of input Boolean register into an output Boolean register by sign-extending each input bit Boolean to 2 bits for each of the N elements of the lower half of the Boolean simd register input. The output elements are written out in exactly the same order as the input elements, with element 0 of input vector register being mapped to the lowest two bits (1,0) of output Boolean register, and element N/2-1 of input vector register being mapped to the highest two bits of output Boolean register. The result elements are written to output vector Boolean register.

IVP_INJBI2NX8

Takes 2N input bits from the vector Boolean register and writes them to the same position within each lane of the vector register. The position is determined by the input immediate. Only 2N bits in the output of the vector register are changed, the rest of the bits retain the previous values.

IVP_JOINB

Takes N even inputs from the first and the second input Boolean register and places them in the low and upper half of the output Boolean register.

IVP_LTR[2N|N|N_2][I]

IVP_LTRS[2N|N|N_2]

The predicate loop remainder computation operations take the input operand and set groups of Boolean bits in the output vector Boolean operand to replicated ones for each group with an index less than the input operand. This operation is designed to efficiently generate a bit-mask for predicated remainder loops for vectorized loops where the scalar loop count is not known to be a multiple of the factorization length. For a saturating LTR if the signed input operand is less than 0, then all output bits are set to 0. If the signed input operand is greater than the number of Boolean lanes, then all Boolean output bits are set to 1.

Comparison Operations

IVP_EQ[NX16|N_2X32|2NX8]

IVP_LE[U][NX16|N_2X32|2NX8]

IVP_LT[U][NX16|N_2X32|2NX8]

IVP_NEQ[NX16|N_2X32|2NX8]

Equal, less than or equal, less than, and not equal vector comparisons. These operations take two vector inputs and produce vector Boolean output. Comparison is performed on corresponding lanes/elements of the input vectors. If the comparison condition evaluates to

TRUE for a specific element, the corresponding element/lane in the output vector Boolean register is set to 1 (TRUE), else it is set to 0 (FALSE).

Shifts

IVP_SLA[NX16|N_2X32]

The source of the shift amount is an input vector register. In this case, every element in the input vector register can have a different shift amount applied to it. The shift amounts are saturated to the range -16 to 16, and -32 to 32, for IVP_SLANX16 and IVP_SLAN_2X32 respectively.

These are bidirectional shifts. Positive shift amounts will shift in the direction implied by the name. Negative shift amounts will shift in the direction opposite to that implied by the name. Zero shifts pass input through to the output unchanged. Note that shift left arithmetic in the context of this operation implies that a negative shift amount results in an arithmetic shift. For positive shift amount the shift is logical.

IVP_SLLI[NX16|2NX8|N_2X32]

The source of the shift amount is an immediate, one value for all the input elements. In this case, every element in the input vector register is shifted by the same amount. These are unidirectional shifts. Zero shifts pass input through to the output unchanged. The type of shift that is applied, and its direction, is shift left logical.

IVP_SLL[NX16|N_2X32]

The source of the shift amount is an input vector register. In this case, every element in the input vector register can have a different shift amount applied to it. The shift amounts are saturated to the range -16 to 16, and -32 to 32, for IVP_SLANX16 and IVP_SLAN_2X32 respectively.

These are bidirectional shifts. Positive shift amounts will shift in the direction implied by the name. Negative shift amounts will shift in the direction opposite to that implied by the name. Zero shifts pass input through to the output unchanged. The type of shift that is applied, and its direction, is shift left logical.

IVP_SLS[NX16|N_2x32]

The input vector is a vector register. The output results are written out to vector register. The source of the shift amount is a second input vector register. In this case, every element in input vector register can have a different shift amount applied to it. The shift amounts are saturated to range -16 to 16, and -32 to 32, for IVP_SLSNX16 and IVP_SLSN_2X32 respectively. These are bidirectional shifts. Positive shift amounts will shift in the direction implied by the name. Negative shift amounts will shift in the direction opposite to that implied by the name. Zero shifts pass input through to the output unchanged. The type of shift that is applied, and its direction, is shift left saturating.

IVP_SLSI[NX16|N_2x32]

The input vector is a vector register. The output results are written out to vector register. The source of the shift is an immediate. In this case, every element in input vector register can only be shifted by the same amount. The shift amounts are saturated to range -16 to 16, and -32 to 32, for IVP_SLSNX16 and IVP_SLSN_2X32 respectively. These are bidirectional shifts. Positive shift amounts will shift in the direction implied by the name. Negative shift amounts will shift in the direction opposite to that implied by the name. Zero shifts pass input through to the output unchanged. The type of shift that is applied, and its direction, is shift left saturating.

IVP_SRA[NX16|N_2X32]

For the shift right arithmetic the source of the shift amount is an input vector register. In this case, every element in input vector register can have a different shift amount applied to it. The shift amounts are saturated to the range -16 to 16 and -32 to 32 IVP_SRANX16 and IVP_SRAN_2X32 respectively. These are bidirectional shifts. Positive shift amounts will shift in the direction implied by the name. Negative shift amounts will shift in the direction opposite to that implied by the name. Zero shifts pass input through to the output unchanged. The type of shift that is applied, and its direction, is shift right arithmetic.

IVP_SRAI[NX16|2NX8|N_2X32]

For the shift right arithmetic with an immediate the source of the shift amount is an input immediate. In this case, every element in input vector register is shifted by one amount (immediate value). These are unidirectional shifts. Zero shifts pass input through to the output unchanged. The type of shift that is applied, and its direction, is shift right arithmetic.

IVP_SRL[NX16|N_2X32]

For the shift right logical the source of the shift amount is an input vector register. In this case, every element in input vector register can have a different shift amount applied to it. The shift amounts are saturated to the range -16 to 16 and -32 to 32 IVP_SRANX16 and IVP_SRAN_2X32 respectively. These are bidirectional shifts. Positive shift amounts will shift in the direction implied by the name. Negative shift amounts will shift in the direction opposite to that implied by the name. Zero shifts pass input through to the output unchanged. The type of shift that is applied, and its direction, is shift right logical.

IVP_SRLI[NX16|2NX8|N_2X32]

For the shift right logical with an immediate the source of the shift amount is an input immediate. In this case, every element in input vector register is shifted by one amount (immediate value). These are unidirectional shifts. Zero shifts pass input through to the output unchanged. The type of shift that is applied, and its direction, is shift right logical.

IVP_SRS[NX16|N_2X32]

For the shift right with saturation the source of the shift amount is an input vector register. In this case, every element in input vector register can have a different shift amount applied to it. The shift amounts are saturated to the range -16 to 16 and -32 to 32 IVP_SRANX16 and IVP_SRAN_2X32 respectively. These are bidirectional shifts. Positive shift amounts will shift in the direction implied by the name. Negative shift amounts will shift in the direction opposite

to that implied by the name. Zero shifts pass input through to the output unchanged. The type of shift that is applied, and its direction, is shift right saturating.

Rotations

IVP_ROTRI[NX16|N_2X32|2NX8]

IVP_ROTTR[NX16|N_2X32]

These operations rotate right each input vec lane by the immediate shift amount (ROTRI) or by the shift amounts defined by the corresponding lanes of the second input vec register (ROTTR). The rotate amounts are signed. If the rotate amount is positive - the rotation is right, if negative – the rotation is left by the same amount.

NSA Operations

IVP_NSA[U][NX16|N_2X32]

Signed and unsigned NSA (Normalizing Shift Amount computation operations) for narrow vector (vec) register. The signed operation returns the number of redundant sign bits in each lane. The unsigned variant returns the number of contiguous most-significant zero bits.

Reduction Operations

Reduction operations process lanes/elements of one input vector register and put the result in Lane 0/ the first element of the output vector register.

IVP_RADD[U][NX16|2NX8][T]

IVP_RADDN_2X32[T]

Reduction add operations sum elements of one input vector and place the result in Lane 0 (the first element) of the vector.

Vision supports 8-bit and 16-bit signed and unsigned, with and without TRUE predication, reduction sum operations as well as 32-bit signed with and without TRUE predication. For 8-bit reductions the result is 16 bits, for 16-bit and 32-bit reductions the result is 32 bits. For 32-bit reduction adds the result can overflow undetected.

IVP_RBMAXNX16

IVP_RBMAXUNX16

IVP_RBMINNX16

IVP_RBMINUNX16

Vision supports reduction min and max operations which generate Boolean masks for 16-bit signed and unsigned data types. These operations set the Boolean lanes of output Vbool which correspond to the position(s) (vec lanes) of maximum or minimum value(s) (for MAX and MIN respectively).

IVP_RMAX[U]N_2X32

IVP_RMIN[U]N_2X32

IVP_RMAX[U]NX16[T]

IVP_RMIN[U]NX16[T]

Reduction MIN and MAX operations are similar to IVP_RBMAX and IVP_RBMIN but do not generate a Boolean mask.

Pack Operations

IVP_PACKMNX48

IVP_PACKHN_2X64

IVP_PACKLN_2X64

IVP_PACKL2NX24[|_0_|_1]

IVP_PACKVNR[NX48, 2NX24][|_0_|_1]

IVP_PACKVNRN_2X64

IVP_PACKVR2NX24[|_0_|_1]

IVP_PACKVRNR[NX48, 2NX24][|_0_|_1]

IVP_PACKVRNRN_2X64

IVP_PACKVR[NX48, 2NX24][|_0_|_1]

IVP_PACKVRN_2X64

IVP_PACKVRU2NX24[|_0_|_1]

Suffix interpretation:

- L - least-significant portion of wvec
- M - mid-significant portion of wvec
- H - most-significant portion of wvec
- VR - the same amount of shift for all lanes from AR
- NR - no rounding or saturation of result (rounding with saturation if not present)

Pack operations move a portion of bits in wvec lanes to corresponding vec lanes in such a way that the portion fits vec lanes appropriate for the wvec and vec data types involved. IVP_PACKH operations copy msb (upper part of the lane) of wvec to a corresponding vec lane. For example, IVP_PACKHN_2X64 copies upper 32 bits of a 64-bit wvec lane to a 32-bit vec lane. Likewise IVP_PACKL operations copy lsb (lower part of the lane) of wvec to a corresponding vec lane.

There are also operations which perform a signed right shift on a wvec before copying lower part of the wvec lane to a vec lane. IVP_PACKVR[NR] operations shift all the lanes by the

same amount, and the shift amount comes from an AR register. The maximum shift amount for wvec 24-bit data type is 23, i.e. actual shift amounts [0:23] inclusive are supported. The maximum shift amount for wvec 48-bit and 64-bit data types is 32, i.e. actual shift amounts [0:32] inclusive are supported.

The IVP_PACKVR operations round the shifted wvec value to the nearest value representable in the vec result, where the bits shifted off the end are considered the fractional bits of a fixed-point value. If the fixed-point value is exactly half way between two values representable in the result it is rounded to the higher value (i.e. towards $+\infty$). If the fixed-point value is outside the range representable in the result it is saturated to the maximum or minimum representable value, whichever it is closer to.

The IVP_PACKVRNR operations neither round nor saturate the shifted wvec value; the least-significant bits of the shifted result are copied to vec.

Pack operations cover all the supported wvec types to be complete/orthogonal with respect to the functionality, namely: operations with rounding and saturation vs. without rounding and saturation. For N/2x64-bit wide data type packing is supported only to N/2x32-bit output narrow format. For Nx48-bit two output formats are supported Nx32-bit and Nx16-bit. When the output format is Nx32-bit two separate operations are used to output odd or even lanes to a vec register. Similarly 2Nx24-bit wide data type could be output to 2Nx8-bit or 2Nx16-bit formats, in the latter case the odd and even lanes are packed using two separate operations.

Unpack Operations

IVP_UNPK[U|S]2NX8_[0|1]

Each operation unpacks NX8 elements from input vec to output vec by zero extending (IVP_UNPKU) or sign extending (IVP_UNPKS) each input 8-bit element to 16-bit vec element in the output. [0] operations process even input lanes, while [1] operations processes odd input lanes.

IVP_UNPKSNX16_[L|H]

Each operation Unpacks N/2X16 elements of high half of the narrow input vector register to 32 bits, by sign-extension. The unpacked elements are written out to the narrow (32-bit) output vector register.

Select Operations

IVP_SEL[NX16|2NX8|N_2X32][T]

The SElect operation takes 2N/N/N_2 elements out of 4N/2N/N elements in two vector inputs and copies them to 2N/N/N_2 elements in the output vector without changing the values of the elements. The width of input elements is equal to the width of the output elements (no promotion or de-motion). The order and the choice of the elements in the output vector are determined by the third input vector. kth lane of the third input vector contains the position of the element in the concatenated first two input vectors which will be put in the kth lane of the output vector. Note that for all the supported data types (NX16, 2NX8, N_2X32)

the actual position bits are in the lsbs of the third input vector lanes, the upper bits of each lane in this vector are ignored.

IVP_SEL2NX8I

IVP_SEL2NX8I_S[0|2|4]

The SElect Immediate operation, just like a select with a vector register, takes 2N elements out of 4N elements in two vector inputs and copies them to 2N elements in the output vector without changing the values of the elements. The width of input elements is equal to the width of the output elements (no promotion or de-motion). However, select with immediate, does not have a third input vector register which describe the positions of input elements. Instead these positions are stored in tables. The input immediate serves is an index into the table to retrieve the positions of input elements.

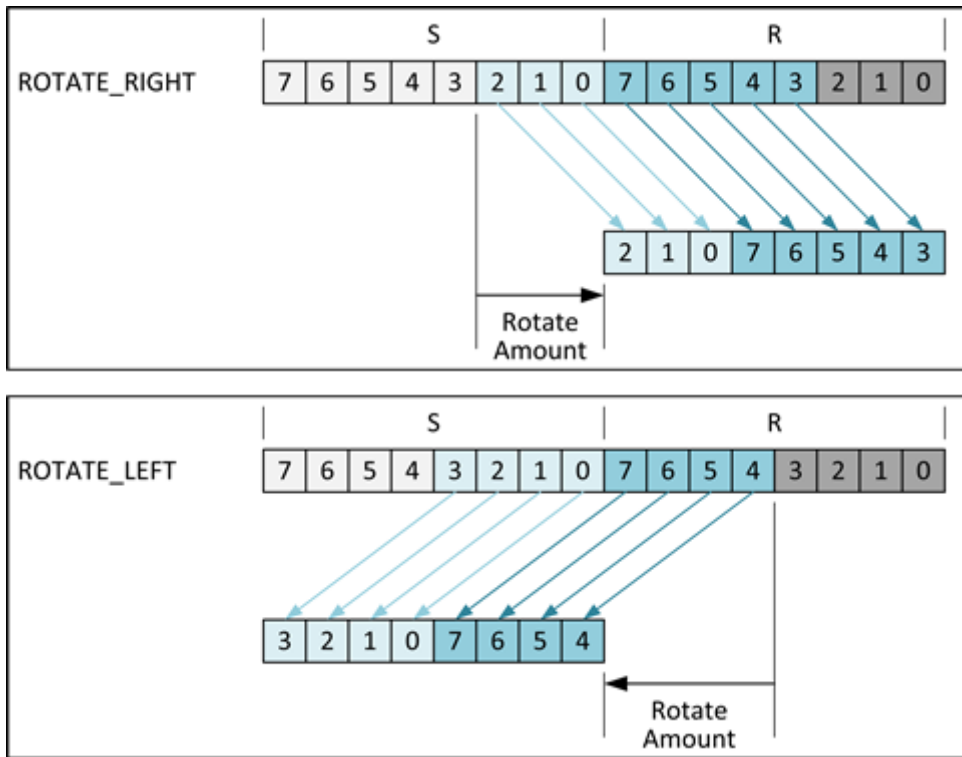
The **IVP_SEL2NX8I_S[0|2|4]** operations are specialized versions of **IVP_SEL2NX8I** in slot 0, slot 2 and slot 4 respectively. These operations support a subset of the patterns supported by **IVP_SEL2NX8I**. The C compiler will select these specialized operations as appropriate; they should not be explicitly invoked by the programmer.

While the operation itself selects 8-bit elements, the available patterns are meaningful for a variety of element sizes, including 8-bit, 16-bit, 32-bit and 64-bit sizes. The pattern tables below show the input elements selected by the pattern from the two input vectors S and R, where the size of the elements corresponds to the the size indicated in the pattern name. The patterns can be divided into several categories.

ROTATE_RIGHT, ROTATE_LEFT

These patterns shift the elements of the two concatenated input vectors (S, R) right or left by a fixed number of elements. The result is either the right-most (for ROTATE_RIGHT) or left-most (for ROTATE_LEFT) vector after the shift.

Figure 2: SELI ROTATE Diagram



The following shifts are available:

Table 37: SELI ROTATES

Element Size	ROTATE_RIGHT Amount	ROTATE_LEFT Amount
8-bit	1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 32	1, 2, 3, 4, 5, 6, 7, 8, 16, 24, 32
16-bit	1, 2, 3, 4, 6, 8, 10, 12, 16	1, 2, 3, 4, 8, 12, 16
32-bit	1, 2, 3, 4, 5, 6, 8	1, 2, 4, 6, 8
64-bit	1, 2, 3, 4	1, 2, 3, 4

Table 38: SELI ROTATE Patterns

Pattern Name	Input Elements
IVP_SELI_8B_ROTATE_LEFT_1	R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31,

Pattern Name	Input Elements
	S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55, S56, S57, S58, S59, S60, S61, S62
IVP_SEL1_8B_ROTATE_LEFT_2	R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55, S56, S57, S58, S59, S60, S61
IVP_SEL1_8B_ROTATE_LEFT_3	R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55, S56, S57, S58, S59, S60
IVP_SEL1_8B_ROTATE_LEFT_4	R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55, S56, S57, S58, S59
IVP_SEL1_8B_ROTATE_LEFT_5	R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55, S56, S57, S58
IVP_SEL1_8B_ROTATE_LEFT_6	R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55, S56, S57

Pattern Name	Input Elements
IVP_SEL1_8B_ROTATE_LEFT_7	R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55, S56
IVP_SEL1_8B_ROTATE_LEFT_8	R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55
IVP_SEL1_8B_ROTATE_LEFT_16	R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47
IVP_SEL1_8B_ROTATE_LEFT_24	R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39
IVP_SEL1_8B_ROTATE_LEFT_32	R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31
IVP_SEL1_8B_ROTATE_RIGHT_1	R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52,

Pattern Name	Input Elements
	R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0
IVP_SEL1_8B_ROTATE_RIGHT_2	R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1
IVP_SEL1_8B_ROTATE_RIGHT_3	R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2
IVP_SEL1_8B_ROTATE_RIGHT_4	R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3
IVP_SEL1_8B_ROTATE_RIGHT_5	R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4
IVP_SEL1_8B_ROTATE_RIGHT_6	R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5
IVP_SEL1_8B_ROTATE_RIGHT_7	R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27,

Pattern Name	Input Elements
	R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6
IVP_SEL1_8B_ROTATE_RIGHT_8	R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7
IVP_SEL1_8B_ROTATE_RIGHT_12	R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11
IVP_SEL1_8B_ROTATE_RIGHT_16	R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15
IVP_SEL1_8B_ROTATE_RIGHT_20	R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19
IVP_SEL1_8B_ROTATE_RIGHT_24	R24, R25, R26, R27, R28, R29, R30, R31, R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23

Pattern Name	Input Elements
IVP_SEL1_8B_ROTATE_RIGHT_32	R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31
IVP_SEL1_16B_ROTATE_LEFT_1	R31, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30
IVP_SEL1_16B_ROTATE_LEFT_2	R30, R31, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29
IVP_SEL1_16B_ROTATE_LEFT_3	R29, R30, R31, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28
IVP_SEL1_16B_ROTATE_LEFT_4	R28, R29, R30, R31, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27
IVP_SEL1_16B_ROTATE_LEFT_8	R24, R25, R26, R27, R28, R29, R30, R31, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23
IVP_SEL1_16B_ROTATE_LEFT_12	R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19
IVP_SEL1_16B_ROTATE_LEFT_16	R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15
IVP_SEL1_16B_ROTATE_RIGHT_1	R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0
IVP_SEL1_16B_ROTATE_RIGHT_2	R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0, S1
IVP_SEL1_16B_ROTATE_RIGHT_3	R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0, S1, S2

Pattern Name	Input Elements
IVP_SEL1_16B_ROTATE_RIGHT_4	R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0, S1, S2, S3
IVP_SEL1_16B_ROTATE_RIGHT_6	R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0, S1, S2, S3, S4, S5
IVP_SEL1_16B_ROTATE_RIGHT_8	R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0, S1, S2, S3, S4, S5, S6, S7
IVP_SEL1_16B_ROTATE_RIGHT_10	R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9
IVP_SEL1_16B_ROTATE_RIGHT_12	R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11
IVP_SEL1_16B_ROTATE_RIGHT_16	R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15
IVP_SEL1_32B_ROTATE_LEFT_1	R15, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14
IVP_SEL1_32B_ROTATE_LEFT_2	R14, R15, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13
IVP_SEL1_32B_ROTATE_LEFT_4	R12, R13, R14, R15, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11
IVP_SEL1_32B_ROTATE_LEFT_6	R10, R11, R12, R13, R14, R15, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9
IVP_SEL1_32B_ROTATE_LEFT_8	R8, R9, R10, R11, R12, R13, R14, R15, S0, S1, S2, S3, S4, S5, S6, S7
IVP_SEL1_32B_ROTATE_RIGHT_1	R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, S0
IVP_SEL1_32B_ROTATE_RIGHT_2	R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, S0, S1
IVP_SEL1_32B_ROTATE_RIGHT_3	R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, S0, S1, S2

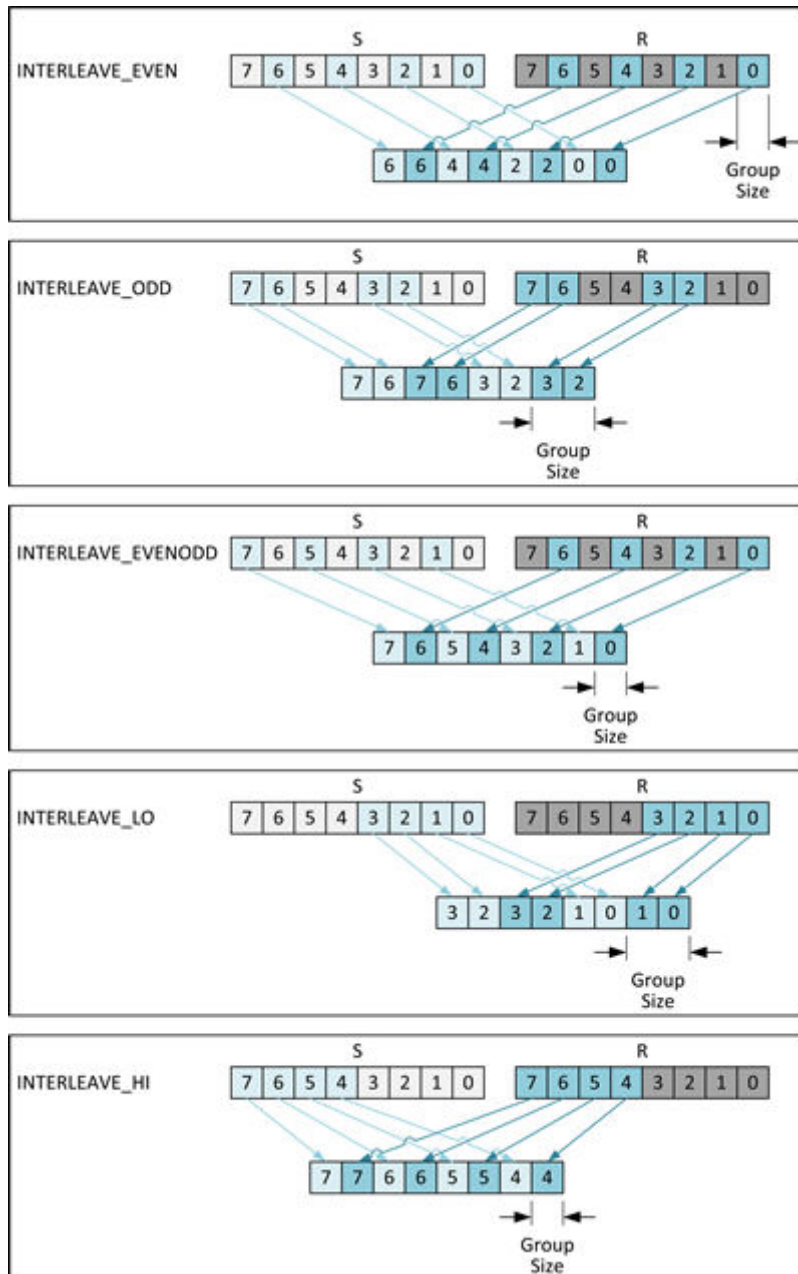
Pattern Name	Input Elements
IVP_SEL1_32B_ROTATE_RIGHT_4	R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, S0, S1, S2, S3
IVP_SEL1_32B_ROTATE_RIGHT_5	R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, S0, S1, S2, S3, S4
IVP_SEL1_32B_ROTATE_RIGHT_6	R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, S0, S1, S2, S3, S4, S5
IVP_SEL1_32B_ROTATE_RIGHT_8	R8, R9, R10, R11, R12, R13, R14, R15, S0, S1, S2, S3, S4, S5, S6, S7
IVP_SEL1_64B_ROTATE_LEFT_1	R7, S0, S1, S2, S3, S4, S5, S6
IVP_SEL1_64B_ROTATE_LEFT_2	R6, R7, S0, S1, S2, S3, S4, S5
IVP_SEL1_64B_ROTATE_LEFT_3	R5, R6, R7, S0, S1, S2, S3, S4
IVP_SEL1_64B_ROTATE_LEFT_4	R4, R5, R6, R7, S0, S1, S2, S3
IVP_SEL1_64B_ROTATE_RIGHT_1	R1, R2, R3, R4, R5, R6, R7, S0
IVP_SEL1_64B_ROTATE_RIGHT_2	R2, R3, R4, R5, R6, R7, S0, S1
IVP_SEL1_64B_ROTATE_RIGHT_3	R3, R4, R5, R6, R7, S0, S1, S2
IVP_SEL1_64B_ROTATE_RIGHT_4	R4, R5, R6, R7, S0, S1, S2, S3

INTERLEAVE_EVEN, INTERLEAVE_ODD, INTERLEAVE_EVENODD, INTERLEAVE_LO, INTERLEAVE_HI

These patterns interleave groups of elements from each of the two input vectors (S, R). Groups consist of 1, 2, 4 or 8 adjacent elements aligned to the number of elements.

- INTERLEAVE_EVEN interleaves the even element groups from each input vector
- INTERLEAVE_ODD interleaves the odd element groups from each input vector
- INTERLEAVE_EVENODD interleaves the even element groups from one input vector with the odd element groups from the other input vector
- INTERLEAVE_LO interleaves the element groups from the low halves of each input vector
- INTERLEAVE_HI interleaves the element groups from the high halves of each input vector

Figure 3: SEL1 INTERLEAVES Diagram



The following interleaves are available:

Table 39: SELI INTERLEAVES

Element Size	EVEN group size	ODD group size	EVENODD group size	LO group size	HI group size
8-bit	1, 2, 4, 8	1, 2, 4, 8	2, 4, 8	1, 2, 4, 8	1, 2, 4, 8
16-bit	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4	1, 2, 4
32-bit	1, 2	1, 2	1, 2	1, 2	1, 2
64-bit ¹⁰	1	1	1	1	1

Table 40: SELI INTERLEAVE Patterns

Pattern Name	Input Elements
IVP_SELI_8B_INTERLEAVE_1_EVEN	R0, S0, R2, S2, R4, S4, R6, S6, R8, S8, R10, S10, R12, S12, R14, S14, R16, S16, R18, S18, R20, S20, R22, S22, R24, S24, R26, S26, R28, S28, R30, S30, R32, S32, R34, S34, R36, S36, R38, S38, R40, S40, R42, S42, R44, S44, R46, S46, R48, S48, R50, S50, R52, S52, R54, S54, R56, S56, R58, S58, R60, S60, R62, S62
IVP_SELI_8B_INTERLEAVE_1_ODD	R1, S1, R3, S3, R5, S5, R7, S7, R9, S9, R11, S11, R13, S13, R15, S15, R17, S17, R19, S19, R21, S21, R23, S23, R25, S25, R27, S27, R29, S29, R31, S31, R33, S33, R35, S35, R37, S37, R39, S39, R41, S41, R43, S43, R45, S45, R47, S47, R49, S49, R51, S51, R53, S53, R55, S55, R57, S57, R59, S59, R61, S61, R63, S63
IVP_SELI_8B_INTERLEAVE_1_HI	R32, S32, R33, S33, R34, S34, R35, S35, R36, S36, R37, S37, R38, S38, R39, S39, R40, S40, R41, S41, R42, S42, R43, S43, R44, S44, R45, S45, R46, S46, R47, S47, R48, S48, R49, S49, R50, S50, R51, S51, R52, S52, R53, S53, R54, S54, R55, S55, R56, S56, R57, S57, R58, S58, R59, S59, R60, S60, R61, S61, R62, S62, R63, S63
IVP_SELI_8B_INTERLEAVE_1_LO	R0, S0, R1, S1, R2, S2, R3, S3, R4, S4, R5, S5, R6, S6, R7, S7, R8, S8, R9, S9, R10, S10, R11, S11, R12, S12, R13, S13, R14, S14, R15, S15, R16, S16, R17, S17, S18, S18, R19, S19, R20, S20, R21, S21, R22, S22, R23, S23, R24, S24, R25, S25, R26, S26, R27, S27, R28, S28, R29, S29, R30, S30, R31, S31, R32, S32, R33, S33, R34, S34, R35, S35, R36, S36, R37, S37, R38, S38, R39, S39, R40, S40, R41, S41, R42, S42, R43, S43, R44, S44, R45, S45, R46, S46, R47, S47, R48, S48, R49, S49, R50, S50, R51, S51, R52, S52, R53, S53, R54, S54, R55, S55, R56, S56, R57, S57, R58, S58, R59, S59, R60, S60, R61, S61, R62, S62, R63, S63

¹⁰ Interleave pattern names for 64-bit elements are not defined; use the equivalent 32-bit pattern name (IVP_SELI_32B_INTERLEAVE_2_XXX) instead.

Pattern Name	Input Elements
	S17, R18, S18, R19, S19, R20, S20, R21, S21, R22, S22, R23, S23, R24, S24, R25, S25, R26, S26, R27, S27, R28, S28, R29, S29, R30, S30, R31, S31
IVP_SEL1_8B_INTERLEAVE_2_EVEN	R0, R1, S0, S1, R4, R5, S4, S5, R8, R9, S8, S9, R12, R13, S12, S13, R16, R17, S16, S17, R20, R21, S20, S21, R24, R25, S24, S25, R28, R29, S28, S29, R32, R33, S32, S33, R36, R37, S36, S37, R40, R41, S40, S41, R44, R45, S44, S45, R48, R49, S48, S49, R52, R53, S52, S53, R56, R57, S56, S57, R60, R61, S60, S61
IVP_SEL1_8B_INTERLEAVE_2_ODD	R2, R3, S2, S3, R6, R7, S6, S7, R10, R11, S10, S11, R14, R15, S14, S15, R18, R19, S18, S19, R22, R23, S22, S23, R26, R27, S26, S27, R30, R31, S30, S31, R34, R35, S34, S35, R38, R39, S38, S39, R42, R43, S42, S43, R46, R47, S46, S47, R50, R51, S50, S51, R54, R55, S54, S55, R58, R59, S58, S59, R62, R63, S62, S63
IVP_SEL1_8B_INTERLEAVE_2_EVENODD	R0, R1, S2, S3, R4, R5, S6, S7, R8, R9, S10, S11, R12, R13, S14, S15, R16, R17, S18, S19, R20, R21, S22, S23, R24, R25, S26, S27, R28, R29, S30, S31, R32, R33, S34, S35, R36, R37, S38, S39, R40, R41, S42, S43, R44, R45, S46, S47, R48, R49, S50, S51, R52, R53, S54, S55, R56, R57, S58, S59, R60, R61, S62, S63
IVP_SEL1_8B_INTERLEAVE_2_HI	R32, R33, S32, S33, R34, R35, S34, S35, R36, R37, S36, S37, R38, R39, S38, S39, R40, R41, S40, S41, R42, R43, S42, S43, R44, R45, S44, S45, R46, R47, S46, S47, R48, R49, S48, S49, R50, R51, S50, S51, R52, R53, S52, S53, R54, R55, S54, S55, R56, R57, S56, S57, R58, R59, S58, S59, R60, R61, S60, S61, R62, R63, S62, S63
IVP_SEL1_8B_INTERLEAVE_2_LO	R0, R1, S0, S1, R2, R3, S2, S3, R4, R5, S4, S5, R6, R7, S6, S7, R8, R9, S8, S9, R10, R11, S10, S11, R12, R13, S12, S13, R14, R15, S14, S15, R16, R17, S16, S17, R18, R19, S18, S19, R20, R21, S20, S21, R22, R23, S22, S23, R24, R25, S24, S25, R26, R27, S26, S27, R28, R29, S28, S29, R30, R31, S30, S31
IVP_SEL1_8B_INTERLEAVE_4_EVEN	R0, R1, R2, R3, S0, S1, S2, S3, R8, R9, R10, R11, S8, S9, S10, S11, R16, R17, R18, R19, S16, S17, S18,

Pattern Name	Input Elements
	S19, R24, R25, R26, R27, S24, S25, S26, S27, R32, R33, R34, R35, S32, S33, S34, S35, R40, R41, R42, R43, S40, S41, S42, S43, R48, R49, R50, R51, S48, S49, S50, S51, R56, R57, R58, R59, S56, S57, S58, S59
IVP_SEL1_8B_INTERLEAVE_4_ODD	R4, R5, R6, R7, S4, S5, S6, S7, R12, R13, R14, R15, S12, S13, S14, S15, R20, R21, R22, R23, S20, S21, S22, S23, R28, R29, R30, R31, S28, S29, S30, S31, R36, R37, R38, R39, S36, S37, S38, S39, R44, R45, R46, R47, S44, S45, S46, S47, R52, R53, R54, R55, S52, S53, S54, S55, R60, R61, R62, R63, S60, S61, S62, S63
IVP_SEL1_8B_INTERLEAVE_4_EVENODD	R0, R1, R2, R3, S4, S5, S6, S7, R8, R9, R10, R11, S12, S13, S14, S15, R16, R17, R18, R19, S20, S21, S22, S23, R24, R25, R26, R27, S28, S29, S30, S31, R32, R33, R34, R35, S36, S37, S38, S39, R40, R41, R42, R43, S44, S45, S46, S47, R48, R49, R50, R51, S52, S53, S54, S55, R56, R57, R58, R59, S60, S61, S62, S63
IVP_SEL1_8B_INTERLEAVE_4_HI	R32, R33, R34, R35, S32, S33, S34, S35, R36, R37, R38, R39, S36, S37, S38, S39, R40, R41, R42, R43, S40, S41, S42, S43, R44, R45, R46, R47, S44, S45, S46, S47, R48, R49, R50, R51, S48, S49, S50, S51, R52, R53, R54, R55, S52, S53, S54, S55, R56, R57, R58, R59, S56, S57, S58, S59, R60, R61, R62, R63, S60, S61, S62, S63
IVP_SEL1_8B_INTERLEAVE_4_LO	R0, R1, R2, R3, S0, S1, S2, S3, R4, R5, R6, R7, S4, S5, S6, S7, R8, R9, R10, R11, S8, S9, S10, S11, R12, R13, R14, R15, S12, S13, S14, S15, R16, R17, R18, R19, S16, S17, S18, S19, R20, R21, R22, R23, S20, S21, S22, S23, R24, R25, R26, R27, S24, S25, S26, S27, R28, R29, R30, R31, S28, S29, S30, S31
IVP_SEL1_8B_INTERLEAVE_8_EVEN	R0, R1, R2, R3, R4, R5, R6, R7, S0, S1, S2, S3, S4, S5, S6, S7, R16, R17, R18, R19, R20, R21, R22, R23, S16, S17, S18, S19, S20, S21, S22, S23, R32, R33, R34, R35, R36, R37, R38, R39, S32, S33, S34, S35, S36, S37, S38, S39, R48, R49, R50, R51, R52, R53, R54, R55, S48, S49, S50, S51, S52, S53, S54, S55

Pattern Name	Input Elements
IVP_SEL1_8B_INTERLEAVE_8_ODD	R8, R9, R10, R11, R12, R13, R14, R15, S8, S9, S10, S11, S12, S13, S14, S15, R24, R25, R26, R27, R28, R29, R30, R31, S24, S25, S26, S27, S28, S29, S30, S31, R40, R41, R42, R43, R44, R45, R46, R47, S40, S41, S42, S43, S44, S45, S46, S47, R56, R57, R58, R59, R60, R61, R62, R63, S56, S57, S58, S59, S60, S61, S62, S63
IVP_SEL1_8B_INTERLEAVE_8_EVENODD	R0, R1, R2, R3, R4, R5, R6, R7, S8, S9, S10, S11, S12, S13, S14, S15, R16, R17, R18, R19, R20, R21, R22, R23, S24, S25, S26, S27, S28, S29, S30, S31, R32, R33, R34, R35, R36, R37, R38, R39, S40, S41, S42, S43, S44, S45, S46, S47, R48, R49, R50, R51, R52, R53, R54, R55, S56, S57, S58, S59, S60, S61, S62, S63
IVP_SEL1_8B_INTERLEAVE_8_HI	R32, R33, R34, R35, R36, R37, R38, R39, S32, S33, S34, S35, S36, S37, S38, S39, R40, R41, R42, R43, R44, R45, R46, R47, S40, S41, S42, S43, S44, S45, S46, S47, R48, R49, R50, R51, R52, R53, R54, R55, S48, S49, S50, S51, S52, S53, S54, S55, R56, R57, R58, R59, R60, R61, R62, R63, S56, S57, S58, S59, S60, S61, S62, S63
IVP_SEL1_8B_INTERLEAVE_8_LO	R0, R1, R2, R3, R4, R5, R6, R7, S0, S1, S2, S3, S4, S5, S6, S7, R8, R9, R10, R11, R12, R13, R14, R15, S8, S9, S10, S11, S12, S13, S14, S15, R16, R17, R18, R19, R20, R21, R22, R23, S16, S17, S18, S19, S20, S21, S22, S23, R24, R25, R26, R27, R28, R29, R30, R31, S24, S25, S26, S27, S28, S29, S30, S31
IVP_SEL1_16B_INTERLEAVE_1_EVEN	R0, S0, R2, S2, R4, S4, R6, S6, R8, S8, R10, S10, R12, S12, R14, S14, R16, S16, R18, S18, R20, S20, R22, S22, R24, S24, R26, S26, R28, S28, R30, S30
IVP_SEL1_16B_INTERLEAVE_1_ODD	R1, S1, R3, S3, R5, S5, R7, S7, R9, S9, R11, S11, R13, S13, R15, S15, R17, S17, R19, S19, R21, S21, R23, S23, R25, S25, R27, S27, R29, S29, R31, S31
IVP_SEL1_16B_INTERLEAVE_1_EVENODD	R0, S1, R2, S3, R4, S5, R6, S7, R8, S9, R10, S11, R12, S13, R14, S15, R16, S17, R18, S19, R20, S21, R22, S23, R24, S25, R26, S27, R28, S29, R30, S31
IVP_SEL1_16B_INTERLEAVE_1_HI	R16, S16, R17, S17, R18, S18, R19, S19, R20, S20, R21, S21, R22, S22, R23, S23, R24, S24, R25, S25,

Pattern Name	Input Elements
	R26, S26, R27, S27, R28, S28, R29, S29, R30, S30, R31, S31
IVP_SEL1_16B_INTERLEAVE_1_LO	R0, S0, R1, S1, R2, S2, R3, S3, R4, S4, R5, S5, R6, S6, R7, S7, R8, S8, R9, S9, R10, S10, R11, S11, R12, S12, R13, S13, R14, S14, R15, S15
IVP_SEL1_16B_INTERLEAVE_2_EVEN	R0, R1, S0, S1, R4, R5, S4, S5, R8, R9, S8, S9, R12, R13, S12, S13, R16, R17, S16, S17, R20, R21, S20, S21, R24, R25, S24, S25, R28, R29, S28, S29
IVP_SEL1_16B_INTERLEAVE_2_ODD	R2, R3, S2, S3, R6, R7, S6, S7, R10, R11, S10, S11, R14, R15, S14, S15, R18, R19, S18, S19, R22, R23, S22, S23, R26, R27, S26, S27, R30, R31, S30, S31
IVP_SEL1_16B_INTERLEAVE_2_EVENODD	R0, R1, S2, S3, R4, R5, S6, S7, R8, R9, S10, S11, R12, R13, S14, S15, R16, R17, S18, S19, R20, R21, S22, S23, R24, R25, S26, S27, R28, R29, S30, S31
IVP_SEL1_16B_INTERLEAVE_2_HI	R16, R17, S16, S17, R18, R19, S18, S19, R20, R21, S20, S21, R22, R23, S22, S23, R24, R25, S24, S25, R26, R27, S26, S27, R28, R29, S28, S29, R30, R31, S30, S31
IVP_SEL1_16B_INTERLEAVE_2_LO	R0, R1, S0, S1, R2, R3, S2, S3, R4, R5, S4, S5, R6, R7, S6, S7, R8, R9, S8, S9, R10, R11, S10, S11, R12, R13, S12, S13, R14, R15, S14, S15
IVP_SEL1_16B_INTERLEAVE_4_EVEN	R0, R1, R2, R3, S0, S1, S2, S3, R8, R9, R10, R11, S8, S9, S10, S11, R16, R17, R18, R19, S16, S17, S18, S19, R24, R25, R26, R27, S24, S25, S26, S27
IVP_SEL1_16B_INTERLEAVE_4_ODD	R4, R5, R6, R7, S4, S5, S6, S7, R12, R13, R14, R15, S12, S13, S14, S15, R20, R21, R22, R23, S20, S21, S22, S23, R28, R29, R30, R31, S28, S29, S30, S31
IVP_SEL1_16B_INTERLEAVE_4_EVENODD	R0, R1, R2, R3, S4, S5, S6, S7, R8, R9, R10, R11, S12, S13, S14, S15, R16, R17, R18, R19, S20, S21, S22, S23, R24, R25, R26, R27, S28, S29, S30, S31
IVP_SEL1_16B_INTERLEAVE_4_HI	R16, R17, R18, R19, S16, S17, S18, S19, R20, R21, R22, R23, S20, S21, S22, S23, R24, R25, R26, R27, S24, S25, S26, S27, R28, R29, R30, R31, S28, S29, S30, S31

Pattern Name	Input Elements
IVP_SEL1_16B_INTERLEAVE_4_LO	R0, R1, R2, R3, S0, S1, S2, S3, R4, R5, R6, R7, S4, S5, S6, S7, R8, R9, R10, R11, S8, S9, S10, S11, R12, R13, R14, R15, S12, S13, S14, S15
IVP_SEL1_32B_INTERLEAVE_1_EVEN	R0, S0, R2, S2, R4, S4, R6, S6, R8, S8, R10, S10, R12, S12, R14, S14
IVP_SEL1_32B_INTERLEAVE_1_ODD	R1, S1, R3, S3, R5, S5, R7, S7, R9, S9, R11, S11, R13, S13, R15, S15
IVP_SEL1_32B_INTERLEAVE_1_EVENODD	R0, S1, R2, S3, R4, S5, R6, S7, R8, S9, R10, S11, R12, S13, R14, S15
IVP_SEL1_32B_INTERLEAVE_1_HI	R8, S8, R9, S9, R10, S10, R11, S11, R12, S12, R13, S13, R14, S14, R15, S15
IVP_SEL1_32B_INTERLEAVE_1_LO	R0, S0, R1, S1, R2, S2, R3, S3, R4, S4, R5, S5, R6, S6, R7, S7
IVP_SEL1_32B_INTERLEAVE_2_EVEN	R0, R1, S0, S1, R4, R5, S4, S5, R8, R9, S8, S9, R12, R13, S12, S13
IVP_SEL1_32B_INTERLEAVE_2_ODD	R2, R3, S2, S3, R6, R7, S6, S7, R10, R11, S10, S11, R14, R15, S14, S15
IVP_SEL1_32B_INTERLEAVE_2_EVENODD	R0, R1, S2, S3, R4, R5, S6, S7, R8, R9, S10, S11, R12, R13, S14, S15
IVP_SEL1_32B_INTERLEAVE_2_HI	R8, R9, S8, S9, R10, R11, S10, S11, R12, R13, S12, S13, R14, R15, S14, S15
IVP_SEL1_32B_INTERLEAVE_2_LO	R0, R1, S0, S1, R2, R3, S2, S3, R4, R5, S4, S5, R6, R7, S6, S7

EXTRACT_N_OF_M_OFF_O

These patterns extract groups of N elements every M elements starting from offset O from the two concatenated input vectors (S, R). They de-interleave elements from the input.

Table 41: SEL1 EXTRACT Patterns

Pattern Name	Input Elements
IVP_SEL1_8B_EXTRACT_1_OF_2_OFF_0	R0, R2, R4, R6, R8, R10, R12, R14, R16, R18, R20, R22, R24, R26, R28, R30, R32, R34, R36, R38, R40, R42, R44, R46, R48, R50, R52, R54, R56, R58, R60, R62, S0, S2, S4, S6, S8, S10, S12, S14, S16, S18, S20, S22, S24, S26, S28, S30, S32, S34, S36, S38,

Pattern Name	Input Elements
	S40, S42, S44, S46, S48, S50, S52, S54, S56, S58, S60, S62
IVP_SEL1_8B_EXTRACT_1_OF_2_OFF_1	R1, R3, R5, R7, R9, R11, R13, R15, R17, R19, R21, R23, R25, R27, R29, R31, R33, R35, R37, R39, R41, R43, R45, R47, R49, R51, R53, R55, R57, R59, R61, R63, S1, S3, S5, S7, S9, S11, S13, S15, S17, S19, S21, S23, S25, S27, S29, S31, S33, S35, S37, S39, S41, S43, S45, S47, S49, S51, S53, S55, S57, S59, S61, S63
IVP_SEL1_8B_EXTRACT_2_OF_4_OFF_0	R0, R1, R4, R5, R8, R9, R12, R13, R16, R17, R20, R21, R24, R25, R28, R29, R32, R33, R36, R37, R40, R41, R44, R45, R48, R49, R52, R53, R56, R57, R60, R61, S0, S1, S4, S5, S8, S9, S12, S13, S16, S17, S20, S21, S24, S25, S28, S29, S32, S33, S36, S37, S40, S41, S44, S45, S48, S49, S52, S53, S56, S57, S60, S61
IVP_SEL1_8B_EXTRACT_2_OF_4_OFF_2	R2, R3, R6, R7, R10, R11, R14, R15, R18, R19, R22, R23, R26, R27, R30, R31, R34, R35, R38, R39, R42, R43, R46, R47, R50, R51, R54, R55, R58, R59, R62, R63, S2, S3, S6, S7, S10, S11, S14, S15, S18, S19, S22, S23, S26, S27, S30, S31, S34, S35, S38, S39, S42, S43, S46, S47, S50, S51, S54, S55, S58, S59, S62, S63
IVP_SEL1_8B_EXTRACT_2_OF_6_OFF_0	R0, R1, R6, R7, R12, R13, R18, R19, R24, R25, R30, R31, R36, R37, R42, R43, R48, R49, R54, R55, R60, R61, S2, S3, S8, S9, S14, S15, S20, S21, S26, S27, S32, S33, S38, S39, S44, S45, S50, S51, S56, S57, S62, S63, R4, R5, R10, R11, R16, R17, R22, R23, R28, R29, R34, R35, R40, R41, R46, R47, R52, R53, R58, R59
IVP_SEL1_8B_EXTRACT_2_OF_6_OFF_2	R2, R3, R8, R9, R14, R15, R20, R21, R26, R27, R32, R33, R38, R39, R44, R45, R50, R51, R56, R57, R62, R63, S4, S5, S10, S11, S16, S17, S22, S23, S28, S29, S34, S35, S40, S41, S46, S47, S52, S53, S58, S59, R0, R1, R6, R7, R12, R13, R18, R19, R24, R25, R30, R31, R36, R37, R42, R43, R48, R49, R54, R55, R60, R61
IVP_SEL1_8B_EXTRACT_2_OF_6_OFF_4	R4, R5, R10, R11, R16, R17, R22, R23, R28, R29, R34, R35, R40, R41, R46, R47, R52, R53, R58, R59, S0, S1, S6, S7, S12, S13, S18, S19, S24, S25, S30, S31, S36,

Pattern Name	Input Elements
	S37, S42, S43, S48, S49, S54, S55, S60, S61, R2, R3, R8, R9, R14, R15, R20, R21, R26, R27, R32, R33, R38, R39, R44, R45, R50, R51, R56, R57, R62, R63
IVP_SEL1_8B_EXTRACT_2_OF_8_OFF_0	R0, R1, R8, R9, R16, R17, R24, R25, R32, R33, R40, R41, R48, R49, R56, R57, S0, S1, S8, S9, S16, S17, S24, S25, S32, S33, S40, S41, S48, S49, S56, S57, R0, R1, R8, R9, R16, R17, R24, R25, R32, R33, R40, R41, R48, R49, R56, R57, S0, S1, S8, S9, S16, S17, S24, S25, S32, S33, S40, S41, S48, S49, S56, S57
IVP_SEL1_8B_EXTRACT_2_OF_8_OFF_2	R2, R3, R10, R11, R18, R19, R26, R27, R34, R35, R42, R43, R50, R51, R58, R59, S2, S3, S10, S11, S18, S19, S26, S27, S34, S35, S42, S43, S50, S51, S58, S59, R2, R3, R10, R11, R18, R19, R26, R27, R34, R35, R42, R43, R50, R51, R58, R59, S2, S3, S10, S11, S18, S19, S26, S27, S34, S35, S42, S43, S50, S51, S58, S59
IVP_SEL1_8B_EXTRACT_2_OF_8_OFF_4	R4, R5, R12, R13, R20, R21, R28, R29, R36, R37, R44, R45, R52, R53, R60, R61, S4, S5, S12, S13, S20, S21, S28, S29, S36, S37, S44, S45, S52, S53, S60, S61, R4, R5, R12, R13, R20, R21, R28, R29, R36, R37, R44, R45, R52, R53, R60, R61, S4, S5, S12, S13, S20, S21, S28, S29, S36, S37, S44, S45, S52, S53, S60, S61
IVP_SEL1_8B_EXTRACT_2_OF_8_OFF_6	R6, R7, R14, R15, R22, R23, R30, R31, R38, R39, R46, R47, R54, R55, R62, R63, S6, S7, S14, S15, S22, S23, S30, S31, S38, S39, S46, S47, S54, S55, S62, S63, R6, R7, R14, R15, R22, R23, R30, R31, R38, R39, R46, R47, R54, R55, R62, R63, S6, S7, S14, S15, S22, S23, S30, S31, S38, S39, S46, S47, S54, S55, S62, S63
IVP_SEL1_8B_EXTRACT_4_OF_6_OFF_0	R0, R1, R2, R3, R6, R7, R8, R9, R12, R13, R14, R15, R18, R19, R20, R21, R24, R25, R26, R27, R30, R31, R32, R33, R36, R37, R38, R39, R42, R43, R44, R45, R48, R49, R50, R51, R54, R55, R56, R57, R60, R61, R62, R63, S2, S3, S4, S5, S8, S9, S10, S11, S14, S15, S16, S17, S20, S21, S22, S23, S26, S27, S28, S29
IVP_SEL1_8B_EXTRACT_4_OF_6_OFF_2	R2, R3, R4, R5, R8, R9, R10, R11, R14, R15, R16, R17, R20, R21, R22, R23, R26, R27, R28, R29, R32, R33, R34, R35, R38, R39, R40, R41, R44, R45, R46, R47, R50, R51, R52, R53, R56, R57, R58, R59, R62, R63, S0, S1, S4, S5, S6, S7, S10, S11, S12, S13, S16,

Pattern Name	Input Elements
	S17, S18, S19, S22, S23, S24, S25, S28, S29, S30, S31
IVP_SEL1_8B_EXTRACT_4_OF_6_OFF_4	R4, R5, R6, R7, R10, R11, R12, R13, R16, R17, R18, R19, R22, R23, R24, R25, R28, R29, R30, R31, R34, R35, R36, R37, R40, R41, R42, R43, R46, R47, R48, R49, R52, R53, R54, R55, R58, R59, R60, R61, S0, S1, S2, S3, S6, S7, S8, S9, S12, S13, S14, S15, S18, S19, S20, S21, S24, S25, S26, S27, S30, S31, S32, S33
IVP_SEL1_8B_EXTRACT_4_OF_8_OFF_0	R0, R1, R2, R3, R8, R9, R10, R11, R16, R17, R18, R19, R24, R25, R26, R27, R32, R33, R34, R35, R40, R41, R42, R43, R48, R49, R50, R51, R56, R57, R58, R59, S0, S1, S2, S3, S8, S9, S10, S11, S16, S17, S18, S19, S24, S25, S26, S27, S32, S33, S34, S35, S40, S41, S42, S43, S48, S49, S50, S51, S56, S57, S58, S59
IVP_SEL1_8B_EXTRACT_4_OF_8_OFF_4	R4, R5, R6, R7, R12, R13, R14, R15, R20, R21, R22, R23, R28, R29, R30, R31, R36, R37, R38, R39, R44, R45, R46, R47, R52, R53, R54, R55, R60, R61, R62, R63, S4, S5, S6, S7, S12, S13, S14, S15, S20, S21, S22, S23, S28, S29, S30, S31, S36, S37, S38, S39, S44, S45, S46, S47, S52, S53, S54, S55, S60, S61, S62, S63
IVP_SEL1_8B_EXTRACT_8_OF_16_OFF_0	R0, R1, R2, R3, R4, R5, R6, R7, R16, R17, R18, R19, R20, R21, R22, R23, R32, R33, R34, R35, R36, R37, R38, R39, R48, R49, R50, R51, R52, R53, R54, R55, S0, S1, S2, S3, S4, S5, S6, S7, S16, S17, S18, S19, S20, S21, S22, S23, S32, S33, S34, S35, S36, S37, S38, S39, S48, S49, S50, S51, S52, S53, S54, S55
IVP_SEL1_8B_EXTRACT_8_OF_16_OFF_8	R8, R9, R10, R11, R12, R13, R14, R15, R24, R25, R26, R27, R28, R29, R30, R31, R40, R41, R42, R43, R44, R45, R46, R47, R56, R57, R58, R59, R60, R61, R62, R63, S8, S9, S10, S11, S12, S13, S14, S15, S24, S25, S26, S27, S28, S29, S30, S31, S40, S41, S42, S43, S44, S45, S46, S47, S56, S57, S58, S59, S60, S61, S62, S63
IVP_SEL1_8B_EXTRACT_HI_HALVES	R32, R33, R34, R35, R36, R37, R38, R39, R40, R41, R42, R43, R44, R45, R46, R47, R48, R49, R50, R51, R52, R53, R54, R55, R56, R57, R58, R59, R60, R61, R62, R63, S32, S33, S34, S35, S36, S37, S38, S39,

Pattern Name	Input Elements
	S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55, S56, S57, S58, S59, S60, S61, S62, S63
IVP_SELI_8B_EXTRACT_LO_HALVES	R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31
IVP_SELI_16B_EXTRACT_1_OF_2_OFF_0	R0, R2, R4, R6, R8, R10, R12, R14, R16, R18, R20, R22, R24, R26, R28, R30, S0, S2, S4, S6, S8, S10, S12, S14, S16, S18, S20, S22, S24, S26, S28, S30
IVP_SELI_16B_EXTRACT_1_OF_2_OFF_1	R1, R3, R5, R7, R9, R11, R13, R15, R17, R19, R21, R23, R25, R27, R29, R31, S1, S3, S5, S7, S9, S11, S13, S15, S17, S19, S21, S23, S25, S27, S29, S31
IVP_SELI_16B_EXTRACT_1_OF_3_OFF_0	R0, R3, R6, R9, R12, R15, R18, R21, R24, R27, R30, S1, S4, S7, S10, S13, S16, S19, S22, S25, S28, S31, R2, R5, R8, R11, R14, R17, R20, R23, R26, R29
IVP_SELI_16B_EXTRACT_1_OF_3_OFF_1	R1, R4, R7, R10, R13, R16, R19, R22, R25, R28, R31, S2, S5, S8, S11, S14, S17, S20, S23, S26, S29, R0, R3, R6, R9, R12, R15, R18, R21, R24, R27, R30
IVP_SELI_16B_EXTRACT_1_OF_3_OFF_2	R2, R5, R8, R11, R14, R17, R20, R23, R26, R29, S0, S3, S6, S9, S12, S15, S18, S21, S24, S27, S30, R1, R4, R7, R10, R13, R16, R19, R22, R25, R28, R31
IVP_SELI_16B_EXTRACT_1_OF_4_OFF_0	R0, R4, R8, R12, R16, R20, R24, R28, S0, S4, S8, S12, S16, S20, S24, S28, R0, R4, R8, R12, R16, R20, R24, R28, S0, S4, S8, S12, S16, S20, S24, S28
IVP_SELI_16B_EXTRACT_1_OF_4_OFF_1	R1, R5, R9, R13, R17, R21, R25, R29, S1, S5, S9, S13, S17, S21, S25, S29, R1, R5, R9, R13, R17, R21, R25, R29, S1, S5, S9, S13, S17, S21, S25, S29
IVP_SELI_16B_EXTRACT_1_OF_4_OFF_2	R2, R6, R10, R14, R18, R22, R26, R30, S2, S6, S10, S14, S18, S22, S26, S30, R2, R6, R10, R14, R18, R22, R26, R30, S2, S6, S10, S14, S18, S22, S26, S30
IVP_SELI_16B_EXTRACT_1_OF_4_OFF_3	R3, R7, R11, R15, R19, R23, R27, R31, S3, S7, S11, S15, S19, S23, S27, S31, R3, R7, R11, R15, R19, R23, R27, R31, S3, S7, S11, S15, S19, S23, S27, S31

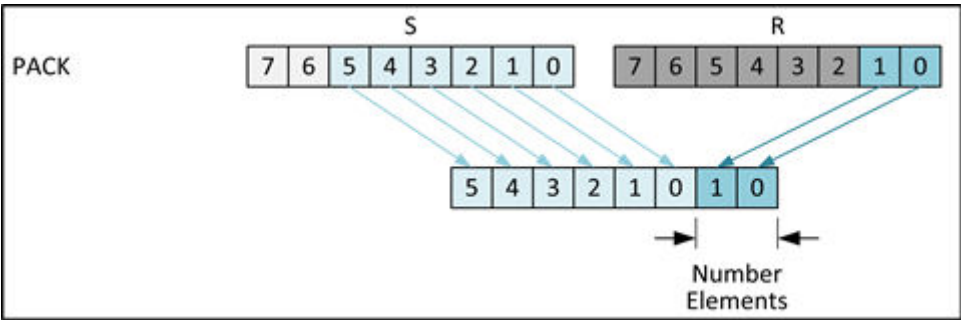
Pattern Name	Input Elements
IVP_SEL1_16B_EXTRACT_2_OF_3_OFF_0	R0, R1, R3, R4, R6, R7, R9, R10, R12, R13, R15, R16, R18, R19, R21, R22, R24, R25, R27, R28, R30, R31, S1, S2, S4, S5, S7, S8, S10, S11, S13, S14
IVP_SEL1_16B_EXTRACT_2_OF_3_OFF_1	R1, R2, R4, R5, R7, R8, R10, R11, R13, R14, R16, R17, R19, R20, R22, R23, R25, R26, R28, R29, R31, S0, S2, S3, S5, S6, S8, S9, S11, S12, S14, S15
IVP_SEL1_16B_EXTRACT_2_OF_3_OFF_2	R2, R3, R5, R6, R8, R9, R11, R12, R14, R15, R17, R18, R20, R21, R23, R24, R26, R27, R29, R30, S0, S1, S3, S4, S6, S7, S9, S10, S12, S13, S15, S16
IVP_SEL1_16B_EXTRACT_2_OF_4_OFF_0	R0, R1, R4, R5, R8, R9, R12, R13, R16, R17, R20, R21, R24, R25, R28, R29, S0, S1, S4, S5, S8, S9, S12, S13, S16, S17, S20, S21, S24, S25, S28, S29
IVP_SEL1_16B_EXTRACT_2_OF_4_OFF_2	R2, R3, R6, R7, R10, R11, R14, R15, R18, R19, R22, R23, R26, R27, R30, R31, S2, S3, S6, S7, S10, S11, S14, S15, S18, S19, S22, S23, S26, S27, S30, S31
IVP_SEL1_16B_EXTRACT_4_OF_8_OFF_0	R0, R1, R2, R3, R8, R9, R10, R11, R16, R17, R18, R19, R24, R25, R26, R27, S0, S1, S2, S3, S8, S9, S10, S11, S16, S17, S18, S19, S24, S25, S26, S27
IVP_SEL1_16B_EXTRACT_4_OF_8_OFF_4	R4, R5, R6, R7, R12, R13, R14, R15, R20, R21, R22, R23, R28, R29, R30, R31, S4, S5, S6, S7, S12, S13, S14, S15, S20, S21, S22, S23, S28, S29, S30, S31
IVP_SEL1_16B_EXTRACT_HI_HALVES	R16, R17, R18, R19, R20, R21, R22, R23, R24, R25, R26, R27, R28, R29, R30, R31, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31
IVP_SEL1_16B_EXTRACT_LO_HALVES	R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15
IVP_SEL1_32B_EXTRACT_1_OF_2_OFF_0	R0, R2, R4, R6, R8, R10, R12, R14, S0, S2, S4, S6, S8, S10, S12, S14
IVP_SEL1_32B_EXTRACT_1_OF_2_OFF_1	R1, R3, R5, R7, R9, R11, R13, R15, S1, S3, S5, S7, S9, S11, S13, S15
IVP_SEL1_32B_EXTRACT_2_OF_4_OFF_0	R0, R1, R4, R5, R8, R9, R12, R13, S0, S1, S4, S5, S8, S9, S12, S13
IVP_SEL1_32B_EXTRACT_2_OF_4_OFF_2	R2, R3, R6, R7, R10, R11, R14, R15, S2, S3, S6, S7, S10, S11, S14, S15

Pattern Name	Input Elements
IVP_SEL1_32B_EXTRACT_HI_HALVES	R8, R9, R10, R11, R12, R13, R14, R15, S8, S9, S10, S11, S12, S13, S14, S15
IVP_SEL1_32B_EXTRACT_LO_HALVES	R0, R1, R2, R3, R4, R5, R6, R7, S0, S1, S2, S3, S4, S5, S6, S7

PACK

These patterns prepend a number of elements from the beginning of one input vector (R) to the beginning of the other input vector (S).

Figure 4: SELI PACK Diagram



The following packs are supported:

Table 42: SELI PACKs

Element Size	PACK Number of Elements
8-bit	2, 4, 6, 8, 10, 12, 14, 16
16-bit	1, 2, 3, 4, 5, 6, 7, 8
32-bit	1, 2, 3, 4
64-bit ¹¹	1, 2

¹¹ Pack pattern names for 64-bit elements are not defined; use the equivalent 32-bit pattern name (IVP_SEL1_32B_PACK_2 or IVP_SEL1_32B_PACK_4) instead.

Table 43: SELI PACK Patterns

Pattern Name	Input Elements
IVP_SELI_8B_PACK_2	R0, R1, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55, S56, S57, S58, S59, S60, S61
IVP_SELI_8B_PACK_4	R0, R1, R2, R3, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55, S56, S57, S58, S59
IVP_SELI_8B_PACK_6	R0, R1, R2, R3, R4, R5, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55, S56, S57
IVP_SELI_8B_PACK_8	R0, R1, R2, R3, R4, R5, R6, R7, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53, S54, S55
IVP_SELI_8B_PACK_10	R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51, S52, S53
IVP_SELI_8B_PACK_12	R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49, S50, S51
IVP_SELI_8B_PACK_14	R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47, S48, S49
IVP_SELI_8B_PACK_16	R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18,

Pattern Name	Input Elements
	S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30, S31, S32, S33, S34, S35, S36, S37, S38, S39, S40, S41, S42, S43, S44, S45, S46, S47
IVP_SEL1_16B_PACK_1	R0, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30
IVP_SEL1_16B_PACK_2	R0, R1, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29
IVP_SEL1_16B_PACK_3	R0, R1, R2, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28
IVP_SEL1_16B_PACK_4	R0, R1, R2, R3, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27
IVP_SEL1_16B_PACK_5	R0, R1, R2, R3, R4, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25, S26
IVP_SEL1_16B_PACK_6	R0, R1, R2, R3, R4, R5, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25
IVP_SEL1_16B_PACK_7	R0, R1, R2, R3, R4, R5, R6, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24
IVP_SEL1_16B_PACK_8	R0, R1, R2, R3, R4, R5, R6, R7, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23
IVP_SEL1_32B_PACK_1	R0, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14
IVP_SEL1_32B_PACK_2	R0, R1, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13
IVP_SEL1_32B_PACK_3	R0, R1, R2, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12
IVP_SEL1_32B_PACK_4	R0, R1, R2, R3, S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11

IVP_SELS[NX16|2NX8|N_2X32]

Scalar selects copies one element of input narrow vector register and writes it into element 0 of output vector. All the rest of output vector register is filled with zeroes. The element to be selected is determined by input immediate.

Dual Select Operations

IVP_DSELNX16[T]

Dual select operations use all 2N elements in two vector inputs and copies them to 2N elements in the two output vector without changing the values of the elements. The width of input elements is equal to the width of the output elements (no promotion or de-motion).). The order of the elements in the two output vectors is determined by the third input vector. Even Lane number 2k of the third input vector contains a position of the element in the concatenated 4N8 input vector which will be copied to Lane k of the first output vector, while odd Lane 2k+1 of the third input vector contains a position of the element in the concatenated 4N8 input vector which will be copied in the Lane k of the second output vector. Note that the actual position bits are in the lsbs of the third input vector lanes, the upper bits of each lane in this vector are ignored.

IVP_DSEL2NX8I

Dual select operation with immediate, unlike the dual select operations with the vector register determining the elements' positions above, dual select with immediate uses tables to determine the elements' positions. The immediate serves as an index into the table

IVP_DSEL2NX8I_H

Dual select with immediate which interleaves or de-interleaves which uses a table with only two indices (two immediate values). Half of one of the output vectors remain unchanged.

Shuffle Operations

IVP_SHFL[NX16|2NX8|N_2X32]

Shuffle copies elements from one input vector register, to one output vector register. The width of input elements is equal to the width of the output elements (no promotion or de-motion). The order of the elements in the output vector is determined by the second input vector. kth lane of the second input vector contains the position of the element in input vector which will be copied to the kth lane of the output vector. Note that for all the supported data types (NX16, 2NX8, N_2X32) the actual position bits are in the lsbs of the second input vector lanes, the upper bits of each lane in this vector are ignored.

IVP_SHFL2NX8I

IVP_SHFL2NX8I_S[0|2|4]

Shuffle with an immediate copies 2N elements, each narrow (8 bits) wide from input vector register to one output vector register. The elements positions in the output register is defined by a table. The immediate value serves as an index into the table. If the immediate value is out of the supported range, a value of zero is used instead.

The IVP_SHFL2NX8I_S[0|2|4] operations are specialized versions of IVP_SHFL2NX8I in slot 0, slot 2 and slot 4 respectively. These operations support a subset of the patterns

supported by **IVP_SHFL2NX8I**. The C compiler will select these operations as appropriate; they should not be explicitly invoked by the programmer.

4.2 Move Operations

IMPORTANT NOTE: Most move operations are intended for use by the XCC compiler, thus, explicit use of move intrinsics in C/C++ code is discouraged.

Moves between (narrow) Vector Registers and Address Registers

IVP_MOVAV32

Copies 32-bit lsbs of a vector register to an address register

IVP_MOVAV16

1-way 32-bit (sign-extended 16-bit) vector register (element 0) to address register move

IVP_MOVAVU16

1-way 32-bit (zero-extended 16-bit) vector register (element 0) to address register move, unsigned

IVP_MOVAV8

1-way 32-bit (sign-extended 8-bit) vector register (element 0) to address register move

IVP_MOVAVU8

1-way 32-bit (zero-extended 8-bit) vector register (element 0) to address register move, unsigned

IVP_MOVVA16

1-way 16-bit address register to vector register move, truncating to 16-bits and replicating into all elements of vector register

IVP_MOVVA32

1-way 32-bit address register to vector register move, replicating into all elements of vector register

IVP_MOVVA8

1-way 8-bit address register to vector register move, truncating to 8-bits and replicating into all elements of vector register

IVP_MOVPA16

Moves a 16-bit move address register to vector register as fraction with saturation and replication. The low order 6 bits of the address register contents are moved to the upper 6 bits of each element of the narrow vector register with the bottom 10 bits zeroed, unless the value is saturated to 32 bits (i.e., 32 bit MAXINT or MININT), in which case the full 16 bit

signed-saturated values are moved to the narrow vector register. All elements are replicated in the output vector register.

Moves between (narrow) Vector Registers

IVP_MOV2NX8T

Moves 2NX8 elements from two input narrow vector register to one output narrow vector register according to the values of the corresponding lanes in the vector Boolean register. If a lane in the vector Boolean register is set to TRUE, the corresponding lane of the FIRST input register is copied to the same lane of the output register, otherwise (if the lane is FALSE), the corresponding lane of the SECOND input register is copied to the same lane of the output register

IVP_MOVVV

The content of the source vector register is copied to the destination vector register

Move between Alignment Registers

IVP_MALIGN

Copies the content of the sources alignment register (all 2NX8 bits) to the destination alignment register

Move between Vector Boolean Registers

IVP_MB

Copies the content of the sources vector Boolean register (all 2N bits) to the destination vector Boolean register.

Moves between AR or Core Boolean Register (BR) and Vector Boolean Register (vbool)

IVP_MOVBRBV

Copies one bit from a vector Boolean register to a core BR Boolean register. Since the core BR Boolean register is one bit, this move moves only the low order (0) bit from the vector Boolean register.

IVP_MOVBVBR

Copies 1 bit from a core BR Boolean register file to a vector Boolean register and replicates it. The single bit Boolean from the core BR Boolean register is replicated into all the elements of the vector Boolean register.

IVP_MOVAB1

Moves 1 least significant bit from a vector Boolean register to the least significant bit of an address register. The upper 31 bits of the address register are set to 0.

IVP_MOVB A1

Moves 1 least significant bit from an address register to the least significant bit of a vector Boolean register. The upper 63 bits of the vector Boolean register are set to 0.

Moves from Immediate

IVP_MOVVINT8

1-way 8-bit move from immediate value (truncated to 8-bit) to vector register with replication

IVP_MOVVINT16

1-way 16-bit move from immediate value (truncated to 16-bit) to vector register with replication

IVP_MOVVINX16

Move a 32-bit value from a table to a vector register with replication. Input immediate serves as an index into the table.

IVP_MOVPINT16

Moves a 16-bit move immediate to vector register as fraction with saturation and replication. The input is an immediate. The low order 6 bits of the immediate are moved to the upper 6 bits of each element of the narrow vector register with the bottom 10 bits zeroed, unless the value is saturated to 32 bits (i.e., 32 bit MAXINT or MININT), in which case the full 16 bit signed-saturated values are moved to the narrow vector register. All elements are replicated in the output vector register.

IVP_MOVQINT16

Moves a 16-bit move immediate to vector register as fraction with saturation and replication. The input is an immediate. The low order 1 bit of the immediate are moved to the upper 1 bit of each element of the narrow vector register with the bottom 15 bits zeroed, unless the value is saturated to 32 bits (i.e., 32 bit MAXINT or MININT), in which case the full 16 bit signed-saturated values are moved to the narrow vector register. All elements are replicated in the output vector register.

IVP_SEQ[2NX8|NX16|N_2X32]

Sequence operations generates a constant output where each lane of the output is set to its lane number. Lane numbers start from 0 in the least significant lane.

Moves between Wide Vector Register and Narrow Vector Register

Wide vectors do not have a direct load/store support. Saving a wide vector to memory is performed by sign extending the wide elements to the nearest size narrow elements which is greater or equal to the size of the wide element and then executing a store operation from the narrow vector register to memory. For example to store 24-bit elements, the elements are

sign extended to the next size narrow 32-bit size, and 32-bit elements are stored from narrow vector to memory.

Restoring a wide vector from memory happens in the opposite order: the narrow element is loaded to a narrow vector register, truncated to the original size (e.g. 32-bit to 24-bit element) and moved to the wide vector.

The primary purpose of move operations between narrow and wide vector registers is to save and restore wide vector registers. These move operations are intended for use by the XCC compiler, explicit use of these move intrinsics in C/C++ code is discouraged.

Note: in the descriptions below “second quarter” of a register refers to the quarter adjacent to the bottom quarter in the low half of the register, while “third quarter” of a register refers to the quarter adjacent to the top quarter in the upper half of the register.

IVP_CVT96UN_2X64

Move N/2 64-bit elements from two narrow vector registers to N/2 64-bit element in a wide vector register. The upper (96-64) parts of physical 96-bit lanes in wvec are set to 0.

IVP_CVT64UN_2X96H

Move from msb (upper) half of a wide vector register N/4 64 bit elements to N/4 64 bit elements in a narrow register. The upper (96-64) parts of physical 96-bit lanes in wvec are not copied.

IVP_CVT64UN_2X96L

Move from lsb (lower) half of a wide vector register N/4 64 bit elements to N/4 64 bit elements in a narrow register. The upper (96-64) parts of physical 96-bit lanes in wvec are not copied.

IVP_CVT48UNX32

Move N 32b elements from 2 vecs to wvec after zero-extending to 48b

IVP_CVT48UNX32L

Move N_2 32b elements from vec to lower half of wvec after zero-extending to 48b. Set upper half of wvec to 0.

IVP_CVT48SNX32L

Move N_2 32b elements from vec to lower half of wvec after sign-extending to 48b. Set upper half of wvec to 0.

IVP_CVT48SNX32

Move N 32b elements from 2 vecs to wvec after sign-extending to 48b.

IVP_CVT24U2NX16

Move 2N 16b elements from 2 vecs to wvec after zero-extending to 24b.

IVP_CVT24S2NX16

Move 2N 16b elements from 2 vecs to wvec after sign-extending to 24b.

IVP_MOVVLL

Move N_4 64b elements from vec to 1st quarter of wvec after truncating to 48b. Set rest of wvec to 0.

IVP_CVT48UN_2X64L

Move N_2 64b elements from 2 vecs to lower half of wvec after truncating to 48b. Set upper half of wvec to 0.

IVP_CVT48UN_2X64H

Move N_2 64b elements from 2 vecs to upper half of wvec after truncating to 48b. Preserves the lower half of wvec.

IVP_CVT24UNX32L

Move N 32b elements from 2 vecs to lower half of wvec after truncating to 24b. Set upper half of wvec to 0.

IVP_CVT24UNX32H

Move N 32b elements from 2 vecs to upper half of wvec after truncating to 24b. Preserves the lower half of wvec.

IVP_CVT32UNX48L

From the lower half of a NX48 wvec, move N_2 48b elements to vec, after truncating to 32b

IVP_CVT32UNX48H

From the upper half of a NX48 wvec, move N_2 48b elements to vec, after truncating to 32b

IVP_CVT32SNX48L

From the lower half of a NX48 wvec, move N_2 48b elements to vec, after signed saturating to 32b

IVP_CVT32SNX48H

From the upper half of a NX48 wvec, move N_2 48b elements to vec, after signed saturating to 32b

IVP_CVT64SNX48LL

From the 1st quarter of a NX48 wvec, move N_4 48b elements to vec, after sign extending to 64b

IVP_CVT64SNX48LH

From the 2nd quarter of a NX48 wvec, move N_4 48b elements to vec, after sign extending to 64b

IVP_CVT64SNX48HL

From the 3rd quarter of a NX48 wvec, move N_4 48b elements to vec, after sign extending to 64b

IVP_CVT64SNX48HH

From the 4th quarter of a NX48 wvec, move N_4 48b elements to vec, after sign extending to 64b

IVP_CVT16U2NX24L

From the lower half of a 2NX24 wvec, move N 24b elements to vec, after truncating to 16b

IVP_CVT16U2NX24H

From the upper half of a 2NX24 wvec, move N 24b elements to vec, after truncating to 16b

IVP_CVT16S2NX24L

From the lower half of a 2NX24 wvec, move N 24b elements to vec, after signed saturating to 16b

IVP_CVT16S2NX24H

From the upper half of a 2NX24 wvec, move N 24b elements to vec, after signed saturating to 16b

IVP_CVT32S2NX24LL

From the 1st quarter of a 2NX24 wvec, move N_2 24b elements to vec, after sign extending to 32b

IVP_CVT32S2NX24LH

From the 2nd quarter of a 2NX24 wvec, move N_4 24b elements to vec, after sign extending to 32b

IVP_CVT32S2NX24HL

From the 3rd quarter of a 2NX24 wvec, move N_2 24b elements to vec, after sign extending to 32b

IVP_CVT32S2NX24HH

From the 4th quarter of a 2NX24 wvec, move N_2 24b elements to vec, after sign extending to 32b

IVP_CVT24U32

Scalar (1-way) move in vector from vec to wvec, conversion from 32-bit lsbs in vec to 24-bit lsbs in wvec by truncation, the rest of the wide vector register is set to 0.

IVP_CVT32S24

Scalar (1-way) in vector move from wvec to vec, conversion from 24-bit lsbs in wvec to 32-bit lsbs in vec by sign extension, the rest of the narrow vector register is set to 0.

IVP_CVT48U64

Scalar (1-way) move in vector from vec to wvec, conversion from 64-bit lsbs in vec to 48-bit lsbs in wvec by truncation, the rest of the wide vector register is set to 0.

IVP_CVT64S48

Scalar (1-way) in vector move from wvec to vec, conversion from 48-bit lsbs in wvec to 64-bit lsbs in vec by sign extension, the rest of the narrow vector register is set to 0.

IVP_CVT64U96

Scalar (1-way) in vector move from wvec to vec, copying from 64-bit lsbs in wvec to 64-bit lsbs in vec (bits 95 to 64 are zero in wvec), the rest of the narrow vector register is set to 0.

IVP_CVT96U64

Scalar (1-way) move in vector from vec to wvec, copying from 64-bit lsbs in vec to 64-bit lsbs in wvec (bits 95 to 64 are zero in wvec), the rest of the wide vector register is set to 0.

Moves between Wide Vector Register and Narrow Vector Register

Wide vectors do not have a direct load/store support. Saving a wide vector to memory is performed by sign extending the wide elements to the nearest size narrow elements which is greater or equal to the size of the wide element and then executing a store operation from the narrow vector register to memory. For example to store 24-bit elements, the elements are sign extended to the next size narrow 32-bit size, and 32-bit elements are stored from narrow vector to memory.

Restoring a wide vector from memory happens in the opposite order: the narrow element is loaded to a narrow vector register, truncated to the original size (e.g. 32-bit to 24-bit element) and moved to the wide vector.

The primary purpose of move operations between narrow and wide vector registers is to save and restore wide vector registers. These move operations are intended for use by the XCC compiler, explicit use of these move intrinsics in C/C++ code is discouraged.

Note: in the descriptions below “second quarter” of a register refers to the quarter adjacent to the bottom quarter in the low half of the register, while “third quarter” of a register refers to the quarter adjacent to the top quarter in the upper half of the register.

IVP_CVT96UN_2X64

Move N/2 64-bit elements from two narrow vector registers to N/2 64-bit element in a wide vector register. The upper (96-64) parts of physical 96-bit lanes in wvec are set to 0.

IVP_CVT64UN_2X96H

Move from msb (upper) half of a wide vector register N/4 64 bit elements to N/4 64 bit elements in a narrow register. The upper (96-64) parts of physical 96-bit lanes in wvec are not copied.

IVP_CVT64UN_2X96L

Move from lsb (lower) half of a wide vector register N/4 64 bit elements to N/4 64 bit elements in a narrow register. The upper (96-64) parts of physical 96-bit lanes in wvec are not copied.

IVP_CVT48UNX32

Move N 32b elements from 2 vecs to wvec after zero-extending to 48b

IVP_CVT48UNX32L

Move N_2 32b elements from vec to lower half of wvec after zero-extending to 48b. Set upper half of wvec to 0.

IVP_CVT48SNX32L

Move N_2 32b elements from vec to lower half of wvec after sign-extending to 48b. Set upper half of wvec to 0.

IVP_CVT48SNX32

Move N 32b elements from 2 vecs to wvec after sign-extending to 48b.

IVP_CVT24U2NX16

Move 2N 16b elements from 2 vecs to wvec after zero-extending to 24b.

IVP_CVT24S2NX16

Move 2N 16b elements from 2 vecs to wvec after sign-extending to 24b.

IVP_MOVVLL

Move N_4 64b elements from vec to 1st quarter of wvec after truncating to 48b. Set rest of wvec to 0.

IVP_CVT48UN_2X64L

Move N_2 64b elements from 2 vecs to lower half of wvec after truncating to 48b. Set upper half of wvec to 0.

IVP_CVT48UN_2X64H

Move N_2 64b elements from 2 vecs to upper half of wvec after truncating to 48b. Preserves the lower half of wvec.

IVP_CVT24UNX32L

Move N 32b elements from 2 vecs to lower half of wvec after truncating to 24b. Set upper half of wvec to 0.

IVP_CVT24UNX32H

Move N 32b elements from 2 vecs to upper half of wvec after truncating to 24b. Preserves the lower half of wvec.

IVP_CVT32UNX48L

From the lower half of a NX48 wvec, move N_2 48b elements to vec, after truncating to 32b

IVP_CVT32UNX48H

From the upper half of a NX48 wvec, move N_2 48b elements to vec, after truncating to 32b

IVP_CVT32SNX48L

From the lower half of a NX48 wvec, move N_2 48b elements to vec, after signed saturating to 32b

IVP_CVT32SNX48H

From the upper half of a NX48 wvec, move N_2 48b elements to vec, after signed saturating to 32b

IVP_CVT64SNX48LL

From the 1st quarter of a NX48 wvec, move N_4 48b elements to vec, after sign extending to 64b

IVP_CVT64SNX48LH

From the 2nd quarter of a NX48 wvec, move N_4 48b elements to vec, after sign extending to 64b

IVP_CVT64SNX48HL

From the 3rd quarter of a NX48 wvec, move N_4 48b elements to vec, after sign extending to 64b

IVP_CVT64SNX48HH

From the 4th quarter of a NX48 wvec, move N_4 48b elements to vec, after sign extending to 64b

IVP_CVT16U2NX24L

From the lower half of a 2NX24 wvec, move N 24b elements to vec, after truncating to 16b

IVP_CVT16U2NX24H

From the upper half of a 2NX24 wvec, move N 24b elements to vec, after truncating to 16b

IVP_CVT16S2NX24L

From the lower half of a 2NX24 wvec, move N 24b elements to vec, after signed saturating to 16b

IVP_CVT16S2NX24H

From the upper half of a 2NX24 wvec, move N 24b elements to vec, after signed saturating to 16b

IVP_CVT32S2NX24LL

From the 1st quarter of a 2NX24 wvec, move N_2 24b elements to vec, after sign extending to 32b

IVP_CVT32S2NX24LH

From the 2nd quarter of a 2NX24 wvec, move N_4 24b elements to vec, after sign extending to 32b

IVP_CVT32S2NX24HL

From the 3rd quarter of a 2NX24 wvec, move N_2 24b elements to vec, after sign extending to 32b

IVP_CVT32S2NX24HH

From the 4th quarter of a 2NX24 wvec, move N_2 24b elements to vec, after sign extending to 32b

IVP_CVT24U32

Scalar (1-way) move in vector from vec to wvec, conversion from 32-bit lsbs in vec to 24-bit lsbs in wvec by truncation, the rest of the wide vector register is set to 0.

IVP_CVT32S24

Scalar (1-way) in vector move from wvec to vec, conversion from 24-bit lsbs in wvec to 32-bit lsbs in vec by sign extension, the rest of the narrow vector register is set to 0.

IVP_CVT48U64

Scalar (1-way) move in vector from vec to wvec, conversion from 64-bit lsbs in vec to 48-bit lsbs in wvec by truncation, the rest of the wide vector register is set to 0.

IVP_CVT64S48

Scalar (1-way) in vector move from wvec to vec, conversion from 48-bit lsbs in wvec to 64-bit lsbs in vec by sign extension, the rest of the narrow vector register is set to 0.

IVP_CVT64U96

Scalar (1-way) in vector move from wvec to vec, copying from 64-bit lsbs in wvec to 64-bit lsbs in vec (bits 95 to 64 are zero in wvec), the rest of the narrow vector register is set to 0.

IVP_CVT96U64

Scalar (1-way) move in vector from vec to wvec, copying from 64-bit lsbs in vec to 64-bit lsbs in wvec (bits 95 to 64 are zero in wvec), the rest of the wide vector register is set to 0.

Move between Wide Vector Registers

IVP_MOVWW

The content of the source wide vector register is copied to the destination wide vector register.

5. Gather/Scatter Operations

Topics:

- [*Gather/Scatter Operations Overview*](#)
- [*Gather Registers*](#)
- [*Gather/Scatter Request Ordering*](#)
- [*Gather/Scatter Address Aliases*](#)
- [*Gather/Scatter Operation Descriptions*](#)
- [*Gather/Scatter Programming Interface*](#)

Gather/scatter operations are part of the standard DSP Vision P6 ISA. Details of the various gather and scatter operations are described in this chapter.

Vision P6 gather/scatter operations use the required Xtensa SuperGather option as described in [*Gather/Scatter*](#) on page 37.

5.1 Gather/Scatter Operations Overview

Vision P6 gather operations are split into separate gatherA (A for Address) and gatherD (D for Data) operations. Here, gatherA and gatherD refer to separate categories of operations; each category contains operation variants on the supported gather data formats (Nx16, Nx8, N_2x32 and 2Nx8) that are otherwise equivalent. An overview of the gatherA and gatherD operation categories is given in this section, and details of the specific individual operations can be found in [Gather/Scatter Operation Descriptions](#) on page 138.

GatherA operations compute a vector of addresses whose corresponding values are read and collected into a gather register. The vector of addresses is computed by adding a 32-bit scalar base address from the AR register file to a vector of 16-bit or 32-bit¹² unsigned byte offsets from the vec register file (the base address is added to each offset). The base address is an arbitrary byte address, but it must be an address in local data RAM, and all the locations gathered must be in that same local data RAM. The base address, offsets, and resulting element addresses are checked for validity before reading the corresponding locations. The following summarizes the checks performed:

- The scalar base address must be in a local data RAM. If the scalar base address is not contained in any data RAM, the entire gather is not performed.
- The MPU region containing the scalar base address must be readable. If the region is not readable, the entire gather is not performed.
- For 32-bit offsets, the offset must be less than 65536 (that is, the offset must fit in 16 bits). Any element whose offset is greater than or equal to 65536 is not read and the gather result for that element is zero.
- All locations gathered must be in the same local data RAM as the scalar base address. Any element whose address is not in the same data RAM (that is, the address is beyond the end of the data RAM) is not read and the gather result for that element is zero.
- All locations gathered must be in the MPU region containing the scalar base address. Any element whose address is not within the region (that is, the address is beyond the end of the region) is not read and the gather result for that element is zero.
- Addresses of individual gather elements must be aligned to the size of the elements being gathered or that element is not read and the gather result for that element is zero. Only the computed element addresses must be aligned; the base address and offsets have no individual alignment requirements.

Full details of the handling and reporting of gatherA errors are found in [Gather/Scatter Error Handling](#).

Predicated versions of the gatherA operation only gather the elements for which a corresponding vbool element is TRUE; the corresponding location is not read and the gather result is zero for the other elements.

¹² 32-bit offsets are only supported for 32-bit gatherA operations

GatherA operations are non-blocking; they initiate the gather but do not wait for its completion. The SuperGather engine performs the individual element reads of the gather in parallel with subsequent Vision P6 instruction execution, and the ordering of these gather reads is not defined with respect to other (non-gather/scatter) loads and stores. The individual element reads of the gather are also performed in arbitrary order with respect to each other, and may be combined (merged) when element locations are the same or adjacent. The programmer is responsible for understanding the request ordering behavior of gathers and scatters and managing all required synchronization through the use of explicit barrier instructions – see [Gather/scatter request ordering](#) for further details.

The values collected by gatherA operations are placed into a register in the gather register file. The gather register is used as a buffer for gatherA results. If a new gatherA operation is executed before reading the result of the previous gatherA operation to the same gather register, the result of the previous gatherA operation is lost.

GatherD operations copy the value in a gather register to a vec register, for subsequent processing (no processing operations are available on gather registers). The gatherD operation stalls, if necessary, until the result of the most recent gatherA is available in the gather register. The `MEMW` and `EXTW` operations wait for all pending gathers to complete.

Scatter operations compute a vector of addresses to which a vector of corresponding data values are written. The vector of addresses is computed by adding a 32-bit scalar base address from the AR register file to a vector of 16-bit or 32-bit¹³ unsigned byte offsets from the vec register file (the base address is added to each offset). The vector of data values is from the vec register file. The base address is an arbitrary byte address, but it must be an address in local data RAM, and all the locations written must be in that same local data RAM. The base address, offsets, and resulting element addresses are checked for validity before writing the corresponding locations. The following summarizes the checks performed:

- The scalar base address must be in a local data RAM. If the scalar base address is not contained in any data RAM, the entire scatter is not performed.
- The MPU region containing the scalar base address must be writeable. If the region is not writeable, the entire scatter is not performed.
- For 32-bit offsets, the offset must be less than 65536 (that is, the offset must fit in 16 bits). Any element whose offset is greater than or equal to 65536 is not written.
- All locations scattered must be in the same local data RAM as the scalar base address. Any element whose address is not in the same data RAM (that is, the address is beyond the end of the data RAM) is not written.
- All locations scattered must be in the MPU region containing the scalar base address. Any element whose address is not within the region (that is, the address is beyond the end of the region) is not written.
- Addresses of individual gather elements must be aligned to the size of the elements being scattered or that element is not written. Only the computed element addresses must be aligned; the base address and offsets have no individual alignment requirements.

¹³ 32-bit offsets are only supported for 32-bit Scatter operations

Full details of the handling and reporting of scatter errors are found in [Gather/Scatter Error Handling](#).

Predicated versions of the scatter operations only store the elements for which a corresponding vbool element is TRUE.

If a scatter operation writes more than one element to the same address, that address will contain the value of the highest-index element on scatter completion.

Scatter operations are non-blocking; they initiate the scatter but do not wait for its completion. The scatter addresses and scatter values are held in an internal scatter buffer (similar to a store buffer) until the scatter is completed, and a scatter operation stalls if necessary to wait for available space in this buffer. The SuperGather engine performs the individual element writes of the scatter in parallel with subsequent Vision P6 instruction execution, and the ordering of these writes is not defined with respect to other (non-gather/scatter) loads and stores. The individual element writes of the scatter are also performed in arbitrary order with respect to each other, and may be combined (merged) when element locations are the same or adjacent. The programmer is responsible for understanding the request ordering behavior of gathers and scatters and managing all required synchronization through the use of explicit barrier instructions – see [Gather/scatter request ordering](#) for further details.

The scatterW operation waits for all pending scatters to complete, as do MEMW and EXTW.

5.2 Gather Registers

Gather registers hold the results of gatherA operations. Four gather registers are provided. This allows up to four outstanding gatherA operations, which improves gather throughput by overlapping the gather pipeline latencies of the individual operations.

Gather registers are read using gatherD operations, which stall as necessary until any pending gatherA operation to that gather register completes. GatherA results must be copied using gatherD for subsequent processing as there are no processing operations on gather registers.

Gather registers are saved using the IVP_GATHERDNX16 operation and restored using the IVP_MOVGATHERD operation. The value of a gather register is undefined after reset.

5.3 Gather/Scatter Request Ordering

Vision P6 gatherA and scatter operations are offloaded to the SuperGather engine; this is done in program order. The SuperGather engine optimizes the performance of these operations by overlapping element reads or writes to minimize latency and maximize throughput, and it uses a relaxed request ordering model to achieve this.

The SuperGather engine performs the individual element reads or writes of each gatherA or scatter operation in arbitrary order (including in parallel) with respect to each other – it

chooses an order that minimizes the number of cycles taken to perform the overall gather or scatter operation. When two or more elements are contained in the same data RAM location, the SuperGather engine may access that location multiple times or may combine (merge) the accesses into as few as one access to the location. The only defined ordering within an operation is for two or more scatter elements written to the same address. In this case, the highest-index element value is the final value written by the scatter operation. The SuperGather engine may perform multiple writes to this same location or combine them into as few as one write of the final value.

The SuperGather engine may also overlap and perform out of order the individual element reads or writes of subsequent gathers and scatters with those of the current gather or scatter operation – this behavior typically reduces the effective number of cycles taken to perform a given gather or scatter. When one or more elements in the current operation are contained in the same locations as one or more elements in subsequent gather or scatter operations, the SuperGather engine may access those locations multiple times or may combine the accesses into as few as one access per location. The only defined ordering between separate operations is for scatter elements written to the same addresses referenced by other gathers or scatters. In this case, the scatter writes will be performed before any reads to that same location are performed by a subsequent gather operation and after any reads to that same location are performed by prior gather operations. The scatter writes will also be performed before any writes to that same location performed by a subsequent scatter operation, or the SuperGather Engine may combine the writes from multiple scatter operations into as few as one write of the final value.

The intent of the request ordering within and across gather and scatter operations as defined above is to maximize the opportunities for request re-ordering and combining (merging) while preserving the appearance of in-order requests to a thread performing gathers and scatters on unshared data. The actual request order and request combining is generally visible to other threads along with the thread performing gathers and scatters on shared data. As such, gathers and scatters must not be used for synchronization purposes. For example, a basic mailbox scheme defines a location for data to be shared along with a flag location to indicate when the data is valid. A scatter operation cannot be used to write both the data and the flag since they can be written in any order – the valid flag might be written before the data is written.

The SuperGather engine operates in parallel with Vision P6 instruction execution and the ordering of all processor load and store operations (non-gather/scatter, for example, AR register loads and stores, vec register loads and stores) with respect to gatherA and scatter element reads and writes is not defined. Software is responsible for enforcing ordering between gather and scatter accesses and processor load and store operations by using the appropriate barrier instructions.

The `MEMW` instruction should be used to ensure all processor load and store operations are performed before any gather or scatter operations. For example, `MEMW` should be used before a gatherA operation that may read data written by processor store operations.

There are several ways to ensure gather and scatter operations have been performed. For an individual gatherA, the corresponding gatherD ensures the gatherA has been performed. (Note that although gatherAs are normally performed in program order, they can be performed out of order, and so only the corresponding gatherD can be used to ensure a given gatherA has been performed.) For example, a processor store that may write a location read by a prior gatherA must use gatherD (or a coarser barrier such as `MEMW`) before the store.

For scatters, the scatterW operation ensures all prior scatter operations are performed. For example, scatterW should be used before a processor load that may read data written by a prior scatter.

The `MEMW`, `EXTW` and `EXCW` instructions also wait for all prior gatherA and scatter operations to be performed.

5.4 Gather/Scatter Address Aliases

Vision P6 gatherA and scatter operations require unique addresses be used for any given location gathered or scattered. If two or more different addresses are used to refer to a single location, then hazards between those operations may not be detected and incorrect operation may result.

Note that aliasing of local memory locations is only possible when the system design external to Vision P6 enables it. For example, a system which implements a local memory smaller than the configured size for Vision P6 and ignores the extra local memory address bits enables aliasing. When aliases exist, only one of the aliases may be used for gatherA and scatter operations. The MPU can be used to enforce use of a single alias region.

5.5 Gather/Scatter Operation Descriptions

Vision P6 provides gather operations that directly support gathering Nx16, Nx8 and N_2x32 format data. There are also operations to facilitate software gathers of 2Nx8 format data, and to save and restore gather register values.

GatherA operations compute a vector of read addresses by adding a 32-bit base address from an AR register to a vector of offsets from a vec register (the base address is added to each offset). The corresponding memory locations are read and assembled into a gather register. GatherA operations are non-blocking.

The `IVP_GATHERANX16[T]` operations take pointers to 16-bit memory elements and Nx16U format offset vectors. They return Nx16 format data in the gather register. The predicated version takes an additional `vboolN` format predicate; predicated false elements are not read from memory and are set to zero in the gather register.

The `IVP_GATHERANX8U[T]` operations take pointers to 8-bit memory elements and Nx16U format offset vectors. They return Nx16 format data in the gather register by zero-extending

the 8-bit elements from memory to 16 bits. The predicated version takes an additional vboolN format predicate; predicated false elements are not read from memory and are set to zero in the gather register.

The `IVP_GATHERAN_2X32[T]` operations take pointers to 32-bit memory elements and `N_2x32U` format offset vectors. They return `N_2x32` format data in the gather register. The predicated version takes an additional `vboolN_2` format predicate; predicated false elements are not read from memory and are set to zero in the gather register.

GatherD operations copy from a gather register to a vec register, where subsequent processing can be done. GatherD operations are blocking and wait if necessary for a prior gatherA operation to the gather register to be finished. They may also reformat the gather register data as part of the copy.

The `IVP_GATHERDNX16` operation copies a gather register value unchanged to a vec register. It is used to access `Nx16`, `Nx8U` and `N_2x32` format data in the gather register, and is also used to save gather register values.

The `IVP_GATHERDNX8S` operation copies a gather register value to a vec register, sign-extending the even bytes to 16 bits as part of the copy. It is used to access `Nx8S` format data in the gather register file.

The `IVP_GATHERD2NX8_[L|H]` operations copy the even bytes in a gather register value to either the low- or high-half of a vec register. The copy to the low-half zeros the high-half of the vec register and the copy to the high-half preserves the low-half of the vec register. These operations are used to facilitate software gathers of `2Nx8` format data.

The gather registers are saved and restored by moving the gather register values through vec registers (there is no direct load/store to memory).

The `IVP_MOVGATHERD` operation copies a vec register value to a gather register. It is used to restore a gather register value.

Table 44: Gather Operations List

Category	Operation
gatherA	<code>IVP_GATHERANX16[T]</code>
	<code>IVP_GATHERANX8U[T]</code>
	<code>IVP_GATHERAN_2X32[T]</code>
gatherD	<code>IVP_GATHERDNX16</code>
	<code>IVP_GATHERDNX8S</code>
	<code>IVP_GATHERD2NX8_[L H]</code>
restore	<code>IVP_MOVGATHERD</code>

Vision P6 provides scatter operations that directly support Nx16, Nx8, N_2x32 and 2Nx8 format data. There is also an operation to wait for scatter results to be visible.

Scatter operations compute a vector of write addresses by adding a 32-bit base address from an AR register to a vector of offsets from a vec register (the base address is added to each offset). The individual elements of a vector of write data from a vec register are written to the corresponding memory locations. If more than one element is written to the same location, that location will contain the highest-index element value on completion of the scatter. Scatter operations are non-blocking but will stall as needed to wait for space in the internal scatter buffer.

The `IVP_SCATTERNX16[T]` operations take pointers to 16-bit memory elements, Nx16U format offset vectors and Nx16 format scatter data. The predicated version takes an additional `vboolN` format predicate; predicated false elements are not written to memory.

The `IVP_SCATTERNX8U[T]` operations take pointers to 8-bit memory elements, Nx16U format offset vectors and Nx8 format scatter data. The predicated version takes an additional `vboolN` format predicate; predicated false elements are not written to memory.

The `IVP_SCATTERN_2X32[T]` operations take pointers to 32-bit memory elements, N_2x32U format offset vectors and N_2x32 format scatter data. The predicated version takes an additional `vboolN_2` format predicate; predicated false elements are not written to memory.

The `IVP_SCATTER2NX8[T]_[L|H]` operations take pointers to 8-bit memory data, Nx16U format offset vectors and 2Nx8 format scatter data. The predicated version takes an additional `vbool2N` format predicate; predicated false elements are not written to memory. They scatter Nx8 elements, taken from either the low- or high-half of the vec register scatter data (and similarly the low- or high-half of the vbool register predicate).

Scatter operations are performed asynchronously with respect to other (non-gather/scatter) load and store operations (see [Gather/Scatter Request Ordering](#) on page 136). Instructions that create barriers must be used to ensure ordering between scatters and other load and store operations.

The `IVP_SCATTERW` operation waits for all prior scatter operations to be complete with respect to load and store operations. `IVP_SCATTERW` is a subset of `MEMW` and `EXTW`, both of which also wait for all prior scatter operations to complete.

Table 45: Scatter Operations List

Category	Operation
scatter	<code>IVP_SCATTERNX16[T]</code>
	<code>IVP_SCATTERNX8U[T]</code>
	<code>IVP_SCATTER2NX8[T]_[L H]</code>
	<code>IVP_SCATTERN_2X32[T]</code>
scatterW	<code>IVP_SCATTERW</code>

5.6 Gather/Scatter Programming Interface

Prototypes which implement the relevant ctype interfaces for the gatherA, gatherD and scatter operations are provided:

- GatherA prototypes take a pointer to the scalar type to gather and an offset of either `xb_vecNx16U` or `xb_vecN_2x32Uv`. Results are always `xb_gsr`. For example,
`IVP_GATHERANX16 {out xb_gsr a, in int16* b, in xb_vecNx16U c}`
- GatherD prototypes take an input of `xb_gsr`. Results correspond to the proto name. For example, `IVP_GATHERDNX16 {out xb_vecNx16 a, in xb_gsr b}`
- Scatter prototypes take a pointer to the scalar type to scatter, an offset of either `xb_vecNx16U` or `xb_vecN_2x32Uv`, and a vector of the type to scatter. For example,
`IVP_SCATTERNX16 {in xb_vecNx16 a, in int16* b, in xb_vecNx16U c}`.

Full details of the provided gatherA, gatherD and scatter prototypes can be found in the ISA HTML documentation.

Abstract GATHER operations are also provided. These return gather results to the vec register file, hiding the gather registers to simplify programming. GATHER is recommended for general use, with the individual gatherA and gatherD operations used only where independent scheduling is needed. Abstract GATHER operations exist for all the ctypes supported by gatherA/gatherD. For example, `IVP_GATHERNX16 {out xb_vecNx16 a, in int16* b, in xb_vecNx16U c}`¹⁴. The full list of abstract GATHER operations is shown in the following table.

Table 46: GATHER Operations

Name
IVP_GATHERNX16[_V]
IVP_GATHERNX16T[_V]
IVP_GATHERNX16U[_V]
IVP_GATHERNX16UT[_V]
IVP_GATHERNXF16[_V]*
IVP_GATHERNXF16T[_V]*
IVP_GATHERNX8U[_V]
IVP_GATHERNX8UT[_V]

¹⁴ The abstract GATHER operations are implemented with C preprocessor macros: the type interface is implemented by the underlying gatherA and gatherD prototypes.

Name
IVP_GATHERNX8S[_V]
IVP_GATHERNX8ST[_V]
IVP_GATHERN_2X32[_V]
IVP_GATHERN_2X32T[_V]
IVP_GATHERN_2X32U[_V]
IVP_GATHERN_2X32UT[_V]
IVP_GATHERN_2XF32[_V]*
IVP_GATHERN_2XF32T[_V]*

* Only present when the Vision Family Single Precision Vector Floating Point option is configured.

GatherA operations take a variable number of cycles to perform. The number of cycles depends on the addresses of the locations being gathered. Each location is contained in a particular sub-bank, and each location in a different word of a particular sub-bank takes one cycle to read. The number of cycles for an individual gatherA operation is the maximum, over all sub-banks, of the number of cycles taken to read each sub-bank. GatherA operations share access to local data RAM, and additional cycles are incurred when access to local data RAM is preempted. The SuperGather engine also overlaps the sub-bank reads of multiple outstanding gatherAs, which effectively reduces the number of cycles taken by an individual gatherA operation.

GatherA operations also have a fixed latency component in addition to the variable number of gather cycles. This fixed latency is added to the number of gather cycles for the overall latency from gatherA to gatherD.

The XCC compiler uses a nominal number of gather cycles when scheduling gatherD operations. This generally gives good results, but there may be cases where the number of cycles for a particular gatherA operation is known at compile time and differs from nominal. All of the gatherA and abstract GATHER operations have variants with an `_v` name suffix to address such cases, for example `IVP_GATHERNX16_V{out xb_vecNx16 a, in int16* b, in xb_vecNx16U c, in immediate d}`. The variants take an additional cycle-count argument whose value must be known at compile time. XCC uses this cycle count instead of the nominal cycle count for scheduling dependant gatherD operations. Discussion on use of these variants is discussed in [Optimizing Gather/Scatter](#) on page 60.

6. Floating-Point Operations

Topics:

- [Vision P6 Floating-Point Features](#)
- [Vision P6 Floating-Point Operations](#)
- [Vision P6 Floating-Point Programming](#)
- [Accuracy and Robustness Optimizations on Vision P6](#)
- [Floating-Point Operations List](#)
- [Floating Point References](#)

Vision P6 is a single instruction multiple data (SIMD) machine. The multiple data is represented with a 512-bit-wide vector register file. Each 512-bit register can represent either 16-way 32-bit single-precision float data or 32-way 16-bit half-precision float data. Vision P6 optionally offers 16 copies of single precision hardware to handle 16-way single-precision float data with a single operation. Vision P6 optionally offers 32 copies of half precision hardware to handle 32-way half-precision float data with a single operation. Both single-precision and half-precision float support scalar and vector data. For scalar float, Vision P6 deploys the single precision hardware on data lane 0 and operates on data on the same data lane. Operations are either non-predicated or predicated-true.

This portion of this User's Guide introduces programmers to the Vision P6 floating-point operations. Except where noted, the floating-point operations implement the IEEE 754 standards [IEEE]. [The Vision P6 Floating-Point Features](#) section specifies the precision formats; Error in the Unit in the Last Place (ULP); Signaling and Quiet Not a Number (NaN); and advises on how to avoid Signaling NaN (sNaN) all together.

The [Vision P6 Floating-Point Operations](#) section categorizes floating-point operations according to their functions and details their behaviors, to assist programmers to better utilize these operations.

The [Vision P6 Floating-Point Programming](#) section is a programming guide on floating-point, including examples of exception-causing operations. Programmers will find essential information on efficient floating-point programming and accurate floating-point computations, without relying on exception or status flags. This section also gives recommendations on Arithmetic Exception and compiler switches.

The [Accuracy and Robustness Optimizations on Vision P6](#) section clarifies some common misconceptions on floating-point, and provides detail on accuracy

measurements. This section is intended to aid programmers on numerical computations, especially when an accumulated error of a few Units in the Last Place (ULP) matters. This section also provides references on selecting algorithms and precisions. Some algorithms are more robust, being able to handle a wider range of input values. All references, algorithms, or examples discussed in this section are provided for explanatory purposes only. Thus, there is no implication that these references, algorithms, or examples are supported or qualified by Cadence.

6.1 Vision P6 Floating-Point Features

Vision P6 Register Files Related to Floating-Point

The Vision P6 floating-point operations share the same vec registers with the fixed-point operations. Floating-point operations also share with the fixed-point operations the same load, store, move, select, shuffle and vector Boolean operations. In addition, floating-point comparison operations write vbool registers and predicated floating-point operations read vbool registers. Floating-point control and status access operations use AR registers. No other register files are used.

Vision P6 Architecture Behavior Related to Floating-Point

Floating-point operations may signal five types of exceptions as specified by the IEEE 754 standards [\[IEEE\]](#):

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

Invalid exception is signaled if and only if there is no usefully definable result. For example, $\text{ADD}(+\infty, -\infty)$ signal Invalid exception since there is no usefully definable result. $\text{ADD}(+\infty, -\infty)$ returns Not a Number (NaN) since returning any number will provide unintended input to follow-up operations and result in incorrect values from the sequence of operations.

Division by Zero exception is signaled if and only if an exact infinite result is defined for an operation on finite operands. Be aware that $\text{DIV}(\pm 0.0, \pm 0.0)$ is equivalent to $\text{MUL}(\pm 0.0, \pm \infty)$, which is an invalid operation, instead of Division by Zero operation.

Overflow exception is signaled if and only if the destination format's largest finite number is exceeded in magnitude.

Underflow exception is signaled when both tinyness is detected and Inexact exception is signaled. The Vision P6 detects tinyness after rounding, as though the exponent range were unbounded, when the computed result is non-zero and with a magnitude smaller than the minimum normal number. Please see [Binary Floating-point Values](#) for the definition of normal numbers.

Inexact exception is signaled if the rounded result differs from what would have been computed if both the exponent range and the precision were unbounded.

Each type of exception always sets a corresponding exception/status flag. Invalid Flag, Division by Zero Flag, Overflow Flag, Underflow Flag, and Inexact Flag are all sticky. They remain as set (logic 1) once the corresponding exception occurs, until they are explicitly cleared by the user.

Optionally, programmers may enable any number of these floating-point exceptions to generate an Arithmetic Exception. The Arithmetic Exception is an asynchronous interrupt for functional safety, rather than a trap for alternate handling recommended by the IEEE 754 standards [\[IEEE\]](#).

Binary Floating-Point Values

The Vision P6 supports binary floating-point numbers in hardware. A binary floating-point value is a bit-string with 3 components: a sign, a biased exponent, and a trailing significand.

The significand consists of 1 implicit leading bit to the left of an implied binary point, and many explicit trailing bits as represented by the trailing significand to the right of the binary point. When the biased exponent is 0, the implicit leading bit of significand (aka hidden bit) is 0, and the represented number is a subnormal (also known as a denorm). Otherwise, the hidden bit is 1, and the represented number is a normal number.

When the biased exponent is all 0's, the unbiased exponent is equal to 1 minus a bias. When the biased exponent is all 1's, the floating-point represent either infinity or Not a Number (NaN) as described in [Encodings of Infinity and Not a Number \(NaN\)](#). Otherwise, the unbiased exponent is the biased exponent minus the bias. The floating-point numerical value is the signed product of the significand and 2 raised to the power of its unbiased exponent.

Table 47: Components of a Binary Floating-point Value

Sign	Biased Exponent	Trailing Significand
------	-----------------	----------------------

Half-Precision Data

The half-precision representation conforms to the binary16 format defined by the IEEE 754 standards. In half-precision, the sign is 1 bit. The biased exponent is 5 bits. The trailing significand is 10 bits. The bias is 0xf. The significand consists of 1 implicit leading bit to the left of an implied binary point, and 10 explicit trailing bits to the right of the binary point.

If half precision is selected, half precision is supported with all operations. If single precision is selected, half precision to/from single precision conversion operations are always available. The half precision operations are described in [Floating-Point Conversion Operations](#)

Table 48: Half Precision Data

Bit Field	From	To	Length
Sign	15	15	1
Biased Exponent	14	10	5
Trailing Significand	9	0	10

Single-Precision Data

The single-precision representation conforms to the single or binary32 format defined by the IEEE 754 standards. In single precision, the sign is 1 bit. The biased exponent is 8 bits. The trailing significand is 23 bits. The bias is 0x7f. The significand consists of 1 implicit leading bit to the left of an implied binary point, and 23 explicit trailing bits to the right of the binary point.

If selected, single precision is supported with all operations, described in the “[Vision P6 Floating-Point Operations](#)” section.

Table 49: Single Precision Data

Bit Field	From	To	Length
Sign	31	31	1
Biased Exponent	30	23	8
Trailing Significand	22	0	23

Maximum Possible Error in the Unit in the Last Place (ULP)

When measured against “correctly rounded” results as specified by the IEEE 754 standards, the following operations generate no errors in all rounding modes: Half/Single Arithmetic Operations, Floating-point Conversion Operations, Integer to/from Half/Single Conversion Operations, Half/Single to Integral Value Rounding Operations, and Half/Single Division and Square Root Operations. All these operations are described in [Vision P6 Floating-Point Operations](#).

When measured against an infinitely precise result, those operations may generate an error which may vary from a rounding mode to another. The infinitely precise result is from mathematic calculation with real numbers instead of floating point numbers, or with floating point numbers of unbounded range and unbounded precision. The error occurs if and only if the infinitely precise result is unrepresentable in a destination format and rounding is necessary to fit the format. A default rounding mode is available after reset, to round to the nearest representable number, and generates an error of [0.0, 0.5] Unit in the Last Place (ULP) in the destination format. Additional rounding modes are also available, to round to various directions, and generates an error of [0.0, 1.0) ULP. All available rounding modes are described in [Floating-Point Rounding and Exception Controls](#).

This User’s Guide defines the error as the maximum absolute distance between the infinitely precise result and the operation result given all legal operands. This User’s Guide measures the error with the Unit in the Last Place, aka ULP, of the destination format.

Encodings of Infinity and Not a Number (NaN)

In all binary float-point formats, the value is a bit-string with 3 components: a sign, a biased exponent, and a trailing significand. The significand consists of 1 implicit leading bit to the left of an implied binary point, and many explicit trailing bits to the right of the binary point.

When the biased exponent is all 1's, and the significand field is all 0's, the numerical value is either positive infinity or negative infinity depending on the sign bit.

When the biased exponent is all 1's, and the significand field is non-zero, the floating-point is Not a Number (NaN). There are two types of NaNs: Signaling Not a Number (sNaN), and Quiet Not a Number (qNaN). sNaN provide representations for uninitialized variables, with the most significant bit (MSB) of the trailing significand being 0. qNaNs provide retrospective diagnostic information inherited from invalid or unavailable data and results, with the MSB of the trailing significand being 1. The retrospective diagnostic information in the trailing significand is called the payload. The IEEE 754 standards do not interpret the sign bit of a NaN.

Floating-point instructions can only generate qNaN, when at least one operand causes an operation to be invalid. No floating-point instructions generate sNaN. All floating-point instructions support both sNaN and qNaN, conforming to the IEEE 754 standards.

Table 50: Encodings of Infinity and Not a Number (NaN)

Encoding	Sign	Biased Exponent	Trailing Significand
$+\infty$	0	All 1's	All 0's
$-\infty$	1	All 1's	All 0's
qNaN	0 or 1	All 1's	1 as the MSB, anything as the rest of bits
sNaN	0 or 1	All 1's	0 as the MSB, non zero as the rest of bits

Throughout this User's Guide, the term NaN include both sNaN and qNaN. For completeness, this User's Guide provides information pertaining both sNaN and qNaN, even though a well coded program should never read an uninitialized variable and therefore could never encounter sNaN unless its coded intentionally.

Scalar Float Data-types Mapped to the Vector Register File

The Vision P6 support real data-types.

The real float data-types supported are:

- **xb_f16 (half)** - A 16-bit half-precision value stored in a 16-bit vector register element.
- **float (float)** - A 32-bit single-precision value stored in a 32-bit vector register element.

Vector Float Data-types Mapped to the Vector Register File

Following is a list of the vector register files and their corresponding vector data-types. Note that vector floating point types follow the same N-way convention. Vision P6 supports two vector floating point types: N-way 16-bit element (xb_vecNxf16) and N_2 (N/2)-way 32-bit element (xb_vecN_2xf32).

Table 51: Vector Float Data-types Mapped to the Vector Register File

Register Vector	Half Precision	Single Precision
vec Registers	xb_vecNxf16	
vec Registers	xb_vecN_2xf16	xb_vecN_2xf32

Floating-Point Data Typing

The above data-type naming convention applies to both operations and protos (C-programming prototypes for operations). Alternatively, intrinsics are provided to give the same functionality of each operation mapped appropriately to the set of data-types.

The Vision P6 supports standard C floating-point data-types, referred to as memory data-types, for 32-bit float. These data-elements are directly mapped to the Vision P6 vec registers as register data-types. There are no standard C floating-point data types for 16-bit half-precision floating point, so these types are defined by Vision P6.

Floating-Point Rounding and Exception Controls

Programmers may access a Floating-point Control Register (FCR), to read/set rounding mode, and/or to enable/disable any of the five floating-point exceptions as a source of Arithmetic Exception. The five floating-point exceptions are described in [Vision P6 Architecture Behavior Related to Floating-Point](#). The Arithmetic Exception is an asynchronous interrupt for functional safety, rather than a trap for alternate handling recommended by the IEEE 754 standards. The Vision P6 maintains neither an order nor a count of exceptions.

The enabling or disabling of any exceptions as a source of Arithmetic Exception changes neither the result nor the floating-point status/exception flags committed by operations. RUR.FCR is the instruction for reading FCR. WUR.FCR is the instruction for writing FCR. FCR is initialized to 0 at reset. Accessing FCR does not trigger Arithmetic Exception.

FCR[1:0] is rounding control (a.k.a. RM). Refer to the FCR fields and Meaning table below for RM values interpretation.

FCR[2] is Inexact Exception Enable. FCR[3] is Underflow Exception Enable. FCR[4] is Overflow Exception Enable. FCR[5] is Division by Zero Exception Enable. FCR[6] is Invalid Exception Enable. A value of 0 disables the corresponding exception and 1 enables corresponding exception.

All other bits are read as 0, and ignored on write. Refer to following table:

Table 52: FCR Fields

Bits	State/unused
1:0	RM
2	Inexact Exception Enable
3	Underflow Exception Enable
4	Overflow Exception Enable
5	Divide by Zero Exception Enable
6	Invalid Exception Enable
11:7	Ignore
31:12	MBZ

Table 53: FCR Fields and Meaning

FCR Fields	Meaning
RM	0-> round to the nearest tied to even 1-> round towards 0 (TRUNC) 2-> round towards +Infinity (CEIL) 3-> round towards -Infinity (FLOOR)
AE	Bits[6:2] are for Arithmetic Exception control bits.
MBZ	Reads as 0, Must be written with zeros.
Ignore	Reads as 0, ignored on write. Allows a value also containing FSR bits to be written.

Floating-Point Status/Exception Flags

Programmers may access a Floating-point Status Register (FSR), to read/set/clear each floating-point status/exception flags. Regardless of the state of the FCR, the FSR always records five types of exceptions when detected, as the default response in the non-trapping situations specified by the IEEE 754 standards. The five floating-point exceptions are described in [Vision P6 Architecture Behavior Related to Floating-Point](#). RUR.FSR is the instruction for reading FSR. WUR.FSR is the instruction for writing FSR. FSR is initialized to 0 at reset. Accessing FSR does not trigger Arithmetic Exception.

FSR[7] is Inexact Flag. FSR[8] is Underflow Flag. FSR[9] is Overflow Flag. FSR[10] is Division by Zero Flag. FSR[11] is Invalid Flag. A value of 0 indicates that the corresponding exception has not occurred. Each flag is sticky and remains as logic 1 once the corresponding exception occurs, until being explicitly cleared.

All other bits are read as 0, and ignored on write.

Table 54: FSR Fields

Bits	State/unused
6:0	Ignore
7	InexactFlag
8	UnderflowFlag
9	OverflowFlag
10	DivZeroFlag
11	InvalidFlag
12:31	MBZ

Table 55: FSR Fields and Meaning

FSR Field	Meaning
MBZ	Reads as 0, Must be written with zeros
Ignore	Reads as 0, ignored on write. Allows a value also containing FCR bits to be written.

Floating-Point Status and Control Flags Alternate Access

Programmers may also access the Floating-point Status and Control Flag state contained in the FSR and FCR registers using move operations that copy values between the Floating-point Status and Control Flag state and scalar 32-bit values in vec registers. There is only one copy of the Floating-point Status and Control Flag state as described in [Floating-Point Status/Exception Flags](#) on page 150 and [Floating-Point Rounding and Exception Controls](#) on page 149; these move operations are simply an alternate way to access that state.

The IVP_MOVSCFV operation copies from vec to the Floating-point Status and Control Flag state and the IVP_MOVVSCF operation copies from the Floating-point Status and Control Flag state to vec. For both operations the format of data in the vec register is shown below:

Table 56: Floating-Point Status and Control Flags in 32-bit Vec Scalar

Bits	State
1:0	MBZ
2	InexactFlag
3	UnderflowFlag
4	OverflowFlag
5	DivZeroFlag
6	InvalidFlag
7	MBZ
9:8	RoundMode
10	InexactEnable
11	UnderflowEnable
12	OverflowEnable
13	DivZeroEnable
14	InvalidEnable
31:15	MBZ

The fields labeled MBZ in the table must be 0 for IVP_MOVSCFV operations and read as 0 for IVP_MOVVSCF operations.



Attention: The IVP_MOVSCFV operation must not be bundled with other operations which modify the Floating-point Status and Control Flag state (e.g. ADD.S). The compiler will not generate such bundles and the assembler will generate an error on such a bundle.

6.2 Vision P6 Floating-Point Operations

The Vision P6 floating point operations include instructions for essential arithmetic, format conversions, (unsigned) integer to/from Half/Single conversions, Half/Single to integral value rounding, classification, comparisons and min/max, as well as division and square root instruction sequences for half, single. All these operations conform to the IEEE 754 standards. Programmers may refer to the IEEE 754 standards for detailed behaviors in terms of exception signaling, status/exception flag setting, rounding control, as well as treatments of infinity, Not a Number (NaN) and signed zero.

Additional instructions and instruction sequences are available for reciprocal, reciprocal square root, and fast square root. These additional instructions and sequences are optimized for speed but not for precision. Though not mandated by the IEEE 754 standards, these operations are provided for optimizing common floating-point applications.

Half/Single Arithmetic Operations

The Vision P6 supports arithmetic operations on half, single precision data-type(s). These arithmetic operations include absolute value (ABS), negation (NEG), addition (ADD), subtraction (SUB), multiplication (MUL), fused-multiplication-addition (MADD), and fused-multiplication-subtraction (MSUB).

ABS and NEG return exact results, and never signal any exceptions even for sNaN. All other operations normalize an infinitely precise result, round the normalized result according to a currently-set rounding mode in the FCR, and may signal Invalid, Overflow, Underflow and/or Inexact exception(s), conforming to the IEEE 754 standards.

Each of these operations has multiple protos, each of which supports a specific data-type related to the operation. Refer to the ISA HTML of an operation to find all the protos using that operation.

Floating-Point Conversion Operations

The Vision P6 supports conversion operations, including half precision to/from single precision conversions. Half-to-Single conversion operations are named CVTF32_F16. Single-to-Half conversion operations are named CVTF16_F32. All the conversions from a narrower data-type to a wider one are exact, requiring no rounding, and signaling no exceptions except Invalid exception when the input is sNaN. The conversions from a wider data-type to a narrower one may not be exact, may require rounding to fit into the narrower data-type, and may signal exceptions. Any required rounding is performed according to a currently-set rounding mode in the FCR. The conversions from a wider data-type to a narrower one signal Invalid exception when the input is sNaN; otherwise may signal Overflow, Underflow and/or Inexact flag(s) when such situation(s) occur.

Each of these operations has multiple protos, each of which supports a specific data-type related to the operation. Refer to the ISA HTML of an operation to find all the protos using that operation.

Integer to and from Half/Single Conversion Operations

The Vision P6 supports conversion operations, including integer to/from half precision conversions, unsigned integer to and from half precision conversions, integer to/from single precision conversions, unsigned integer to and from single precision conversions .

Integer-to-Half conversion operations are named FLOAT.H or FLOATF16. Unsigned-to-Half conversion operations are named UFLOAT.H or UFLOATF16. Integer-to-Single conversion operations are named FLOAT.S or FLOATF32. Unsigned-to-Single conversion operations are named UFLOAT.S or UFLOATF32. When the number of significant bits in the (unsigned)

integer is less than those available in the significand of the destination float format, the conversions are exact, requiring no rounding, and signal no exceptions. Otherwise, the conversions may not be exact, may require rounding to fit into the narrower significand, and may signal Inexact exception. Any required rounding is performed according to a currently-set rounding mode in the FCR, and Inexact exception is signaled when the conversion is not exact.

Half-to-Integer conversion operations are named TRUNC.H and TRUNCF16. Half-to-Unsigned conversion operations are named UTRUNC.H and TRUNCUF16. Single-to-Integer conversion operations are named TRUNC.S and TRUNCF32. Single-to-Unsigned conversion operations are named UTRUNC.S and TRUNCUF32. These conversions may, or may not, be exact. If rounding is required, these conversions always round to zero or truncate any fraction, regardless of the rounding mode. When the truncated value cannot be represented by the destination type, the operations deliver predetermined constants and signal Invalid exception only. The *Xtensa ISA Reference Manual* defines these constants. When the truncated value can be represented but is not exact, the operations deliver the rounded results and signal Inexact exception only.

Each of these operations has multiple protos, each of which supports a specific data-type related to the operation. Refer to the ISA HTML of an operation to find all the protos using that operation.

Half/Single to Integral Value Rounding Operations

The Vision P6 supports integral value rounding operations, including half to integral half rounding, single to integral single rounding. Most of these rounding operations do not depend on a currently-set rounding mode in the FCR.

Half-to-Integral rounding operations, Single-to-Integral rounding operations include FICEIL, FIFLOOR, FIRINT, FIROUND, and FITRUNC. FICEIL operations round the half, single value toward positive infinity, but maintain the original format. FIFLOOR operations round the half, single precision value toward negative infinity, but maintain the original format. FIRINT operations round the half, single precision value according to a currently-set rounding mode in the FCR, but maintain the original format. FIROUND operations round the half, single precision value to the nearest integral value, but round halfway cases away from zero (instead of to the nearest even integral value), and maintain the original format. FITRUNC operations round the half, single precision value toward zero, but maintain the original format. All these rounding operations signal Invalid exception for sNaN input. No operations except FIRINT additionally signal Inexact exception for not exact, conforming to the IEEE 754 standards.

Each of these operations has multiple protos, each of which supports a specific data-type related to the operation. Refer to the ISA HTML of an operation to find all the protos using that operation.

Classification, Comparison and Min/Max Operations

The Vision P6 supports classification, comparison as well as minimum and maximum operations for data-types. The classification operation returns a 8-bit value, zero-extended to the lane width. The comparison operations include 2 types of comparisons, namely Ordered Comparisons and Unordered Comparisons. Ordered Comparisons return true when neither operand is a NaN and comparison is valid; otherwise return false. Unordered Comparisons return true when either operand is a NaN or comparison is valid; otherwise return false.

Ordered Comparison operations include OEQ, OLE, and OLT. Unordered Comparison operations include ULEQ, ULTQ, UNEQ and UN. Here, “EQ” is equal. “LE” is less or equal. “LT” is less than. “NEQ” is not equal. “UN” is unordered. Positive zero is compared equal to negative zero, conforming to the IEEE 754 standards.

All comparison operations signal Invalid exception when either operand is a sNaN. OLE, OLT, ULEQ and ULTQ additionally signal the Invalid exception when either operand is a qNaN.

Additionally, MIN and MAX operations are available for data. MIN implements “(a<b)?a:b”; while MAX implements “(a>b)?a:b”. Here, both “<” and “>” are considered as ordered comparisons, just like OLT(a,b) and OLT(b,a) respectively. Both MIN and MAX signal Invalid exception when either operand is a sNaN or qNaN.

Furthermore, MINNUM and MAXNUM operations are available for data. MINNUM implements “fmin” in C; while MAX implements “fmax” in C. Both MINNUM and MAXNUM returns a number whenever at least one operand is a number. Both MINNUM and MAXNUM signal Invalid exception when either operand is a sNaN.

Each of these operations has multiple protos, each of which supports a specific data-type related to the operation. Refer to the ISA HTML of an operation to find all the protos using that operation.

The Vision P6 supports non-computational classification operations: CLSFY. CLSFY never raises any exceptions, even for sNaN. CLSFY conforms to IEEE 754-2008 “class ()” operation.

CLSFY tells a class of the input operand, by returning an 8-bit enum representation. In binary format, the 8-bit enum is

{isFinite, isSignaling, isNaN, isInfinite, isNormal, isSubnormal, isZero, isSignMinus}

, where isFinite is the MSB. isFinite is 1 if and only if the operand is normal, subnormal, or zero. isSignaling is 1 if and only if the operand is a sNaN. isNaN is 1 if and only if the operand is either a sNaN or a qNaN. isInfinite is 1 if and only if the operand is either +inf or – inf. isNormal is 1 if and only if the operand is \pm normal. isSubnormal is 1 if and only if the operand is \pm subnormal. isZero is 1 if and only if the operand is \pm 0. isSignMinus is 1 if and only if the operand has negative sign. isSignMinus applies to zeros and NaNs as well. isNaN, isInfinite, isNormal, isSubnormal, and isZero are exactly one-hot.

Each of these operations has multiple protos, each of which supports a specific data-type related to the operation. Refer to the ISA HTML of an operation to find all the protos using that operation.

Half/Single Division and Square Root Operations

The Vision P6 supports division and square root operations for half and single data-types, with instruction sequences. These division and square root instruction sequences emulate infinitely precise results, round the infinitely precise results according to a currently-set rounding mode in the FCR, generate correctly rounded results, and always signal any exception(s) according to the IEEE 754 standards. These instruction sequences utilize patented method and devices, as well as fused-multiplication-addition operations to conform to accuracy obligations specified by the IEEE 754 standards. The -mfused-madd or -mno-fused-madd compilation switches do not affect these instruction sequences.

Programmers may instantiate division and square root instruction sequences by using intrinsics, or may compile “/” and “sqrtf()” with -fno-reciprocal-math. The division intrinsics include DIV.H, DIV.S. The square root intrinsics include SQRT.H, SQRT.S. Here “.H” indicates half precision, “.S” indicates single precision. . All these operations normalize an infinitely precise result, round the normalized result according to a currently-set rounding mode in the FCR, and signal Invalid, Division by Zero, Overflow, Underflow and/or Inexact exception(s), conforming to the IEEE 754 standards. All these instruction sequences use instructions in the original precision only.

Each of these operations has multiple protos, each of which supports a specific data-type related to the operation. Refer to the ISA HTML of an operation to find all the protos using that operation.

Reciprocal, Reciprocal Square Root, and Fast Square Root Operations

The Vision P6 supports reciprocal, reciprocal square root, and fast square root operations for various data-types. All these operations are implemented with instruction sequences in the original format only. For smaller code size and shorter latency, the reciprocal, reciprocal square root, and fast square root instruction sequences do not generate infinitely precise results, assume rounding to the nearest tied to even, may generate results with bigger errors, and may signal false Underflow and/or Inexact exception(s). These instruction sequences utilize fused-multiplication-addition operations. The -mfused-madd or -mno-fused-madd compilation switches do not affect these instruction sequences. These operations are not mandated by the IEEE 754 standards, and accept a subset of all possible floating-point values.

These operations include RECIP, RSQRT, and FSQRT. Here, “RECIP” is reciprocal. “RSQRT” is reciprocal square root. “FSQRT” is fast square root. To utilize these operations, programmers may instantiate intrinsics, or compile with -freciprocal-math. When compiled with -freciprocal-math, the compiler may turn “/” or “sqrtf()” into reciprocal and multiplication, reciprocal square root and multiplication, or fast square root. When compiled with -fno-

reciprocal-math, the compiler will honor any intrinsic instantiations but will not generate any additional reciprocal, reciprocal square root, or fast square root instruction sequences.

Each of these operations has multiple protos, each of which supports a specific data-type related to the operation. Refer to the ISA HTML of an operation to find all the protos using that operation.

Notes on Not a Number (NaN) Propagation

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

However, programmers are reminded not to depend on NaN propagation, payload, or the sign bit, since recompilation may cause the propagation to change or to cease. Additionally, IEEE 754-2008 does specify some floating-point operations, having floating-point data as both input and output operands, that may not propagate a NaN. These operations comprise MINNUM and MAXNUM. In addition, the MIN and MAX operations may or may not propagate a NaN, depending on whether the NaN is the first or the second operand.

6.3 Vision P6 Floating-Point Programming

General Guidelines for Floating-Point Programming

Vision P6 floating-point operations are accessible in C or C++ by applying standard C operators to standard float types, including float. Programmers may simply start with legacy C/C++ code or write the floating-point algorithm in a natural style.

All floating-point operations deliver normalized results with adjusted exponents to maximize the precision. All floating operations are fully pipelined. Some operations may create or propagate NaN to represent invalid or unavailable results. No operations create sNaN.

Some aspects of floating point operations are configurable and are controlled by writing appropriate fields of the floating-point control registers (FCR). These configurable options include the selection of a rounding mode, enabling and disabling Arithmetic Exceptions, and others. Note, that although Arithmetic Exceptions can be disabled or enabled by writing appropriate fields of the FRC, the Arithmetic Exceptions physical interface and signals are always present if floating point operations are present in the processor.

Programmers may leave floating-point control registers (FCR) alone for default rounding to the nearest tied to even and for default disabled Arithmetic Exception. The disabled

Arithmetic Exception does not prevent floating-point operations from raising exception flags once a corresponding exception occurs.

Invalid Floating-Point Operation Examples

Programmers may ignore floating-point status registers (FSR), and rely on any qNaN or predetermined integer values to indicate invalid operations. Invalid operations include ADD(+ ∞ , - ∞), SUB (+ ∞ , + ∞), MUL(± 0 , $\pm \infty$), MADD(x, ± 0 , $\pm \infty$), DIV(± 0 , ± 0), DIV($\pm \infty$, $\pm \infty$), SQRT(negative non-zero), OLT(x, NaN), and similar operations. Here, x is any floating-point representation. These invalid floating-point operations return qNaN. TRUNC(NaN), TRUNC($\pm \infty$), UTRUNC(-1.0), and invalid floating-to-integer conversion operations return predetermined integer numbers since qNaN is not supported in integer types. For all the cases above and for similar invalid operations Invalid exception is signaled and Invalid Flag is raised.

There are other 4 less important flags in FSR, namely: Division by Zero, Overflow, Underflow, and Inexact flags. Each of the 4 flags is always raised once a corresponding exception occurs. In all these 4 exceptions, floating operations continue to commit correctly rounded results. Division by Zero results are correctly signed infinity. Overflow results are correctly signed infinity or maximum normal number, depending on a currently-set rounding mode in the FCR. Underflow results are correctly rounded signed zero, subnormal, or minimum normal number. Inexact results are correctly rounded signed zero, subnormal or normal number, or infinity.

Arithmetic Exception and Exception Reduction

Programmers can write to floating-point control registers (FCR), with WUR.FCR, to enable Arithmetic Exception based on any floating-point exception(s). Any enabled exception does not stop floating-point operations and affects neither delivered results nor flag behaviors.

When Arithmetic Exception is used as an interrupt, programmers may minimize interrupt penalties by enabling only the critical floating point exception(s) such as Invalid, by using operations or macros which cause no or fewer exceptions, and/or by filtering out potential NaN inputs before intensive computations. For example, the normal relation operation "<" may be replaced with the macro "isless". Alternatively, programmers may handle unordered cases expressively with the source code. For example, "CLSFY" intrinsics, "isnan" or other macros may be used to filter out NaN or other undesirable inputs. This kind of optimizations is especially beneficial inside loop bodies.

Associative and Distributive Rules as well as Expression Transformations

The mathematical associative rules for addition and multiplication, as well as the distributive rule, are not generally valid because of roundoff error, even in the absence of overflow and underflow. Some expressions cannot be transformed into each other, since they are not equivalent. For example, "x/x" is not equivalent to "1.0", since x can be zero, infinite, or NaN.

When compiled with `-fno-associative-math`, the compiler neither reorders operations, nor transforms expressions. When the small roundoff error is acceptable and special cases are evaluated, programmers may choose to compile with `-fassociative-math`, or to optimize programs by manually applying associative, distributive, or transformation rules.

Half/Single Fused-Multiply-Add (FMA)

Fusing a multiplication and an addition/subtraction into a fused-multiplication-addition/subtraction is an optimization, which has potential to affect the floating-point behavior. Vision P6 provides half- and single- precision fused-multiply-add (FMA) hardware, which performs multiplication to infinite precision, adds/subtracts a product to/from an addend, and then finishes with one single rounding on an infinitely precise sum. There is no rounding on the product. The FMA operation is therefore faster and with less rounding error, compared to separate multiplication and addition. In almost all cases, results from FMA differ very slightly, if any, from those of separate multiplication and addition. However, a special case exists when the addend is infinity. For example, $1.0 * 2^{120}$ multiplied by $1.0 * 2^{97}$ plus $-\infty$, in single precision. For FMA, $1.0 * 2^{120}$ (0x7b800000) multiplied by $1.0 * 2^{97}$ (0x70000000) is $1.0 * 2^{217}$. Then, $-\infty + 1.0 * 2^{217}$ equals to $-\infty$, which is an accurate result from FMA. However, for separate multiplication and addition, $1.0 * 2^{217}$ overflows and become $+\infty$. Then, $-\infty + +\infty$ becomes qNaN, which is not an accurate result but matching that of popular FPU.

Most commercial floating-point units do not include FMA hardware. To match their behavior, programmers may compile with `-mno-fused-madd`. When compiled with `-mno-fused-madd`, the compiler may keep multiplication and addition/subtraction separate. To take advantage of FMA, in terms of execution speed, code size and accuracy, programmers may compile with `-mfused-madd`. When compiled with `-mfused-madd`, the compiler may fuse multiplication and addition/subtraction into fused-multiplication-addition/subtraction.

Division, Square Root, Reciprocal, and Reciprocal Square Root Intrinsic Functions

Division operators, `sqrtf()` and `sqrt()` are automatically compiled into division and square root function calls, with `-fno-reciprocal-math`. When compiled with `-freciprocal-math`, division operators, `sqrtf()` and `sqrt()` may automatically turn into reciprocal, reciprocal square root, and fast square root function calls and maybe with multiplication.

To eliminate the function call overheads, programmers may use the division, square root, reciprocal, reciprocal square root, or fast square root intrinsic explicitly in a few strategic places to replace division operators, `sqrtf()` or `sqrt()`. The strategic intrinsic replacements can improve performance very effectively, especially in a loop. The compiler can still be counted on for efficient scheduling, optimization, and automatic selection of load/store instructions.

Sources of NaN and Potential Changes of NaN Sign and Payload

No floating-point operations generate sNaN. Uninitialized variables should be the only possible source of sNaN. A well coded program should never read uninitialized variables, therefore could never encounter sNaN, and might consider qNaN handling only.

Invalid floating-point operations, as exemplified in [Invalid Operation Examples](#), can generate qNaN. Programmers might avoid qNaN by avoiding invalid operations when possible, might filter out qNaN by using UN and/or CLSFY intrinsics when practical, or consider qNaN as possible inputs inherited from invalid operations.

Programmers should not rely either on sign or payload of NaN. Though floating-point operations perform best efforts to generate a default qNaN or to propagate a NaN, the sign and/or the payload of NaN may be altered by compiler optimizations.

Auto-Vectorizing Scalar Floating-Point Programs

When scalar floating point types are used in loops, the compiler will often automatically vectorize and infer or overload. A loop using scalar floating point types may turn into a loop of vector operations that is as tightly packed and efficient as manual code using intrinsics.

The xt-ccc 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*.

Compiler Switches Related to Floating-point Programs

The following compiler switches are important for floating-point code:

-mfused-madd: allows compiler to fuse a multiplication and an addition/subtraction into a fused-multiplication-addition/subtraction (FMA). By eliminating a rounding on the intermediate product, the performance and accuracy are both improved. The reduced rounding error, however, may cause a slight result difference if compared to the result from machines without FMA hardware. This switch is off by default at -O2 or lower optimization levels. This switch is on by default at -O3 or higher optimization levels.

-mno-fused-madd: disallows compiler to fuse a multiplication and an addition/subtraction into a fused-multiplication-addition/subtraction (FMA). This switch is on by default at -O2 or lower optimization levels. This switch is off by default at -O3 or higher optimization levels.

-menable-non-exact-imaps: enables -mfused-madd and other inexact imaps. This switch is off by default at -O2 or lower optimization levels. This switch is on by default at -O3 or higher optimization levels. Please refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* for information related to imaps.

-mno-enable-non-exact-imaps: disables -mfused-madd and other inexact imaps. This switch is on by default at -O2 or lower optimization levels. This switch is off by default at -O3 or higher optimization levels.

-freciprocal-math: allows compiler to turn division and square root into reciprocal, reciprocal square root, and fast square root with multiplication. By using reciprocal, reciprocal square root, or fast square root, the performance is improved. However, the input range is narrowed and the output error is increased. This switch is off by default at -O2 or lower optimization levels. This switch is on by default at -O3 or higher optimization levels.

-fno-reciprocal-math: disallows compiler to turn division or square root into reciprocal, reciprocal square root, or fast square root. This switch is on by default at -O2 or lower optimization levels. This switch is off by default at -O3 or higher optimization levels.

-fassociative-math: allows compiler to re-associate operands in series of floating-point operations, and to possibly change computation and/or comparison results. This switch is off by default at -O2 or lower optimization levels. This switch is on by default at -O3 or higher optimization levels.

-fno-associative-math: disallows compiler to re-associate operands in series of floating-point operations. This switch is on by default at -O2 or lower optimization levels. This switch is off by default at -O3 or higher optimization levels.

-funsafe-math-optimizations: enables -freciprocal-math and -fassociative-math . This switch is off by default at -O2 or lower optimization levels. This switch is on by default at -O3 or higher optimization levels.

-fno-unsafe-math-optimizations: disables -freciprocal-math and -fassociative-math . This switch is on by default at -O2 or lower optimization levels. This switch is off by default at -O3 or higher optimization levels.

Programmers may refer to the *Xtensa C and C++ Compiler User's Guide* for additional information.

Domain Expertise, Vector Data-types, C Intrinsics, and Libraries

Programmers should exercise their domain expertise when deciding whether to apply any recommendations above, for their specific applications to achieve required performance. Programmers may analyze code performance, by carrying out profiling and performance analysis using the right ISS options. Xtensa Xplorer may assist to determine functions, loops, and loop nests which take most of the cycles. When additional performance is required, programmers may manually vectorize critical functions, loops, or loop nests by using vector data-types, may use C intrinsic functions along with vector data-types, and/or may use libraries. The use of vector data-types, C intrinsics, and libraries is similar to that of fixed-point. Programmers may refer to fixed-point chapters of this User's Guide for details.

6.4 Accuracy and Robustness Optimizations on Vision P6

Programmers may accomplish all their floating-point programming goals, by applying their domain expertise together with information in prior chapters alone. This section provides suggestions concerning highly accurate numeric computations. This section will not repeat

essential information in prior chapters, but instead it will use that information as a foundation for further discussion.

From a system perspective, there may be 3 main causes of accuracy issues: measurement accuracy, computation accuracy, and actuator accuracy. For example, a car collision avoidance system may detect another vehicle with a radar, compute a distance and/or a speed with floating-point calculations, and when necessary actuate a brake system in the car. The radar measurement, floating-point computation, and brake actuator may all have their accuracy and/or resolution limitations. Programmers should exercise their domain expertise to account for or otherwise compensate accuracy or resolution issues of measurement and/or actuator(s). This section presumes that all measurements are properly compensated to provide accurate inputs and that all actuators are properly compensated to actualize faithfully, and focuses on computation accuracy and robustness.

Programmers should investigate any algorithms, methods, techniques, and references in this section for suitability in their specific applications; Cadence assumes no responsibility in any forms for supporting any algorithms, methods, techniques, and references in this chapter.

Some Old Beliefs

Some old beliefs are no longer valid. Many early floating-point units did not have internal representation of infinitely precise results and predated the IEEE 754 standards. Some early studies on these outdated floating-point units reported big ranges of errors depending on input operands.

The Vision P6 Vector FPU emulate infinitely precise results internally, conforms to the IEEE 754 standards, and generates correctly rounded results which can be off by an error of [0.0, 0.5] ULP when compared to infinitely precise results given all legal input operands.

Comparisons and Measurements against Infinitely Precise Results

When using comparisons to measure errors, the comparisons should be against infinitely precise results. A first common mistake is to compare against rounded results. A second common mistake is to compare against a different architecture. A third common mistake is to compare using converted results. All these three common mistakes may mischaracterize any FPU, and mislead programmers.

Comparing against rounded results is the first common mistake. Comparing against rounded results may mischaracterize an FPU with smaller or larger errors. For example, an addition operation conforming to the IEEE 754 standards will be with 0 errors, instead of [0.0, 0.5] ULP. Programmers may be misled to believe an FPU with 0 errors is better than another FPU with 0.5 ULP of error, while these 2 FPU are actually identical.

The second common mistake is to compare against a different architecture. The Vision P6 Vector FPU (VFPU) differs from many commercial FPUs in three architectural characteristics: subnormal support in hardware, fused-multiplication-addition/subtraction (FMA) in hardware, and uniform float representation. The Vision P6 VFPU supports subnormal numbers in hardware to conform to the IEEE 754 standards without latency penalties, while some

commercial FPUs flush subnormal to zero and do not conform to the standards. The Vision P6 VFPU contains FMA hardware to conform to the IEEE 754 2008 standard with higher accuracy, while most commercial FPUs lack FMA hardware and only conform to the IEEE 754 1985 standard with lower accuracy. The Vision P6 VFPU offers uniform representations of float data-types in both memory and register formats and eliminates a rounding error when moving float data from registers to memory, while some commercial FPUs represent float data-types in memory and registers differently and create an extra rounding error when moving float data from registers to memory. Because of these three architectural advantages, the Vision P6 VFPU better conforms to the IEEE 754 standards. Comparing the Vision P6 VFPU against a different architecture with less accuracy may mislead programmers by raising false accuracy issues.

Comparisons in decimal numbers and/or in printed formats may also mislead programmers. The Vision P6 VFPU and most commercial FPUs are binary FPUs. Programmers should evaluate accuracy in binary, to eliminate binary-to-decimal conversion errors. Evaluations in binary enable ULP calculations, while other evaluations do not.

Effects of Rounding Mode and Standards Conforming

Programmers should distinguish operations depending on rounding mode from other operations independent of the rounding mode. Operations depending on rounding mode may generate two different results, 1 ULP away from each other, when rounding mode is changed. When intending to utilize rounding mode to provide different results, programmers may use operations which depend on rounding mode and inherit a currently-set rounding mode in the FCR. Otherwise, programmers may set an intended rounding mode before executing operations which depend on rounding mode.

Additionally, programmers should also distinguish operations conforming to the IEEE 754 standards, and other operations beyond the standards. The operations conforming to the IEEE 754 standards accept all classifications and all values of float data-types, and generate correctly rounded results. For application requiring highest possible accuracy, programmers should focus on the operations conforming to the IEEE 754 standards. The operations conforming to the IEEE 754 standards are described in sections of [Half/Single Arithmetic Operations](#), [Floating-point Conversion Operations](#), [Integer to/from Half/Single Conversion Operations](#), [Half/Single to Integral Value Rounding Operations](#), [Classification, Comparison and Min/Max Operations](#), and [Half/Single Division and Square Root Operations](#), in the Chapter of [Vision P6 Floating-Point Operations](#).

[Reciprocal, Reciprocal Square Root, and Fast Square Root Operations](#) are beyond the IEEE 754 standards. The Vision P6 VFPU optimizes these operations for execution cycle and code size, but not for accuracy. Programmers should use division operations instead of reciprocal operations for high accuracy applications.

Computer Algebra, Algorithm Selections, Default Substitutions, and Exception Flags

$$(a + bi) / (c + di) = (ac + bd) / (c^2 + d^2) + (bc - ad) / (c^2 + d^2) i$$

This formula is always correct when a, b, c, d are all real numbers. In float data-types, the formula will overflow when c or d is big enough, even though the final result may be still within range. Programmers may evaluate Smith's method [\[Baudin\]](#), or other improved algorithms, when concerned with robustness.

In general, formulas should be evaluated for any spurious behaviors, with signed zero, and signed infinities. Programmers may utilize a computer algebra system (CAS), and/or perform symbolic computation, to enhance mathematical formulas, expressions, or functions.

Programmers should also monitor the exception/status flags closely, as a tool to warn about questionable behaviors. Programmers may investigate any Invalid, Division by Zero, Overflow, and Underflow exceptions in this priority order. These exceptions may cause loss of accuracy. Programmers may solve these exceptions, by selecting algorithms, by rearranging formulas, by recoding functions, or by substituting a default value for what would have been the result of the exception-causing operations [\[Kahan\]](#).

Error Bounds, Interval Arithmetic, and Directed Rounding

Programmers may calculate error bounds of mathematical formulas, expressions, and functions. The error bounds may be used to verify numerical correctness. Maximum errors could be indicative to formulas' weak points around some input values. Average errors could be indicative to formulas' overall accuracy limitations.

Programmers may take advantage of the Vision P6 FPU rounding modes, which confirm to the IEEE 754 standards, for interval arithmetic [\[Hijazi\]](#). The interval arithmetic is a powerful approach to bound rounding errors in a mathematical formula, expression, and function. Applying the interval analysis, programmers may consider an output of a mathematical formula to be an interval. The interval is a pair of endpoints. A first endpoint may be generated by executing the formula by rounding toward $-\infty$, and a second endpoint may be generated by executing the same but rounding toward $+\infty$. The interval should contain the exact result of the formula. The span of the interval could be inversely proportional to the accuracy of the formula. If the span is too wide, programmers may optimize the formula, or increase the precisions of critical variables inside the formula.

FMA, Split Number, Double-Precision, and Multi-Precision

Programmers should deploy their domain expertise when applying any techniques suggested above for their specific applications. After applying all the suitable suggested techniques, programmers may promote critical variables to higher precision when necessary to achieve an accuracy goal. Programmers should consider precision upgrades in the following order: FMA, split numbers, double precision, and multi-precision.


Vision P6 provides fused-multiplication-addition/subtraction (FMA) hardware conforming to the IEEE 754 2008 standard. The FMA accept all float inputs including subnormal, compute an infinitely precise product, add/subtract the product to/from an addend, round a sum/difference according to a currently-set rounding mode in the FCR, and signal exception(s) if any. A single precision number comprises a 24-bit significand. Inside the FMA, the infinitely precise product is represented with a 48-bit significand, and the addition or subtraction is performed with the 48-bit significand. Programmers should take advantage of these extra significand bits inside the FMA, before considering the following more expensive techniques.

Another technique of precision upgrade is to split one value into two single-precision numbers. When it's necessary to represent π with more than 24-bit precision, for example, programmers may split π into Pi_hi and Pi_lo . Pi_hi represents a higher 24 bits; while Pi_lo represents a lower 24 bits. With Pi_hi and Pi_lo , programmers are able to represent π with 48-bit precision by replacing π with $(Pi_hi + Pi_lo)$.

6.5 Floating-Point Operations List

Vision P6 supports half and single precision vector (SIMD) and scalar floating point operations. These operations are listed in this section. They are not part of the standard Vision P6 DSP operations. Floating point operations are only supported if Vision Family Half or Single Precision Vector Floating Point Package is enabled. Detailed description of each floating operation is provided in the Vision P6 ISA HTML.

Operations are described with unevaluated N, which is equal to 32 for Vision P6. This is done on purpose to get used to the N-way programming model.

 **Important:** The following instructions must not be used outside of the divide and other predefined sequences: DIV0, SQRT0, RECIP0, RSQRT0, DIVN, ADDEXP, ADDEXPM, NEXP0, NEXP01, MKDADJ, MKSADJ. They are not defined for stand-alone operation.

Vector Floating-Point Operations List

Table 57: List of Vector Floating-Point Single Precision Operations

Operation Name	Synopsis
IVP_CLSFYN_2XF32[T]	N/2-way classification
IVP_ADDN_2XF32[T]	N/2-way addition
IVP_ADDEXPMN_2XF32	N/2-way Add exponent from mantissa operation
IVP_ADDEXPN_2XF32	N/2-way floating point Add exponent operation
IVP_CONSTN_2XF32	Put an immediate from a predefined look up table to vec

Operation Name	Synopsis
IVP_CVTF16N_2XF32[T]_0	N/2-way convert single precision elements to half precision elements, even elements
IVP_CVTF32NXF16[T]_0	N-way convert half precision even elements to single precision elements
IVP_CVTF32NXF16[T]_1	N-way convert half precision odd elements to single precision elements
IVP_DIV0N_2XF32	Beginning step operation for single precision N/2 –way divide
IVP_DIVNN_2XF32	Final step operation for single precision N/2 –way divide
IVP_FICEILN_2XF32[T]	N/2-way single precision round to integral ceiling
IVP_FIFLOORN_2XF32[T]	N/2-way single precision round to integral floor
IVP_FIRINTN_2XF32[T]	N/2-way single precision round with mode to integral
IVP_FIROUNDN_2XF32[T]	N/2-way single precision round towards nearest integral value. If fractional input is exactly $\frac{1}{2}$, rounds away from 0
IVP_FITRUNCN_2XF32[T]	N/2-way single precision round towards 0 integral value (truncation)
IVP_FLOATN_2X32[T]	N/2-way integer conversion to a single precision floating point value using currently set round mode.
IVP_MAXN_2XF32[T]	N/2-way maximum operation of corresponding lanes from two input vecs.
IVP_MINN_2XF32[T]	N/2-way minimum operation of corresponding lanes from two input vecs.
IVP_MAXNUMN_2XF32[T]	N/2-way numeric maximum operation of corresponding lanes from two input vecs.
IVP_MINNUMN_2XF32[T]	N/2-way numeric minimum operation of corresponding lanes from two input vecs.
IVP_MKDADJN_2XF32[T]	N/2-way floating point division adjustment operation, used in the floating point division operations sequence
IVP_MKSADJN_2XF32[T]	N/2-way floating point square root adjustment operation, used in the floating point square root operations sequence
IVP_MULAN_2XF32[T]	N/2-way floating point multiply and accumulate

Operation Name	Synopsis
IVP_MULANN_2XF32	N/2-way floating point multiply and accumulate, rounding to the nearest.
IVP_MULN_2XF32[T]	N/2-way single precision multiply
IVP_MULSN_2XF32[T]	N/2-way single precision multiply and subtract
IVP_MULSNN_2XF32	N/2-way single precision multiply and subtract, rounding to nearest
IVP_NEGN_2XF32	N/2-way single precision negate
IVP_NEXP01N_2XF32	N/2-way narrow exponent range. Divides inputs by multiples of 4.0: the result is greater than or equal to 1.0 and less than 4.0
IVP_OEQN_2XF32[T]	N/2-way single precision equal to comparison
IVP_OLEN_2XF32[T]	N/2-way single precision less than or equal to comparison
IVP_OLTN_2XF32[T]	N/2-way single precision less than comparison
IVP_ONEQN_2XF32[T]	N/2-way single precision not equal comparison
IVP_RECIP0N_2XF32[T]	N/2-way first step of reciprocal (Newton-Raphson) computation
IVP_RSQRT0N_2XF32[T]	N/2-way first step of reciprocal square root (Newton-Raphson) computation
IVP_SQRT0N_2XF32	N/2-way first step of square root (Newton-Raphson) computation
IVP_SUBN_2XF32[T]	N/2-way subtract
IVP_TRUNCN_2XF32[T]	N/2-way scale and round toward 0
IVP_UFLOATN_2XF32[T]	N/2-way conversion of 32-bit unsigned integer to floating point
IVP_ULEQN_2XF32[T]	N/2-way single precision unordered equal to comparison
IVP_ULTQN_2XF32[T]	N/2-way single precision unordered less than comparison
IVP_UNN_2XF32[T]	N/2-way single precision comparison unordered
IVP_UTRUNCN_2XF32[T]	N/2-way scale and round toward 0 unordered
IVP_UEQN_2XF32[T]	N/2-way single precision unordered equal to comparison

Operation Name	Synopsis
IVP_ULEN_2XF32[T]	N/2-way single precision unordered less than or equal to comparison
ULTN_2XF32[T]	N/2-way single precision unordered less than comparison

Table 58: List of Vector Floating-Point Half Precision Operations

Operation Name	Synopsis
IVP_CLSFYNXF16[T]	N-way classification
IVP_ADDNXF16[T]	N-way addition
IVP_ADDEXPMNXF16	N-way Add exponent from mantissa operation
IVP_ADDEXPNXF16	N-way floating point Add exponent operation
IVP_CONSTNXF16	Put an immediate from a predefined look up table to vec
IVP_DIV0NXF16	Beginning step operation for half precision N-way divide
IVP_DIVNNXF16	Final step operation for half precision N-way divide
IVP_FICEILNXF16[T]	N-way half precision round to integral ceiling
IVP_FIFLOORNXF16[T]	N-way half precision round to integral floor
IVP_FIRINTNXF16[T]	N-way half precision round with mode to integral
IVP_FIROUNDNXF16[T]	N-way half precision round towards nearest integral value. If fractional input is exactly $\frac{1}{2}$, rounds away from 0
IVP_FITRUNCNXF16[T]	N-way half precision round towards 0 integral value (truncation)
IVP_FLOATX32[T]	N-way integer conversion to a half precision floating point value using currently set round mode.
IVP_MAXNXF16[T]	N-way maximum operation of corresponding lanes from two input vecs.
IVP_MINNXF16[T]	N-way minimum operation of corresponding lanes from two input vecs.
IVP_MAXNUMNXF16[T]	N-way numeric maximum operation of corresponding lanes from two input vecs.
IVP_MINNUMNXF16[T]	N-way numeric minimum operation of corresponding lanes from two input vecs.
IVP_MKDADJNXF16[T]	N-way floating point division adjustment operation, used in the floating point division operations sequence
IVP_MKSADJNXF16[T]	N-way floating point square root adjustment operation, used in the floating point square root operations sequence

Operation Name	Synopsis
IVP_MULANXF16[T]	N-way floating point multiply and accumulate
IVP_MULANNXF16	N-way floating point multiply and accumulate, rounding to the nearest.
IVP_MULNXF16[T]	N-way half precision multiply
IVP_MULSNXF16[T]	N-way half precision multiply and subtract
IVP_MULSNNXF16	N-way half precision multiply and subtract, rounding to nearest
IVP_NEGNXF16	N-way half precision negate
IVP_NEXP0NXF16	N-way narrow exponent range. Divides inputs by multiples of 2.0: the result is greater than or equal to 1.0 and less than 2.0
IVP_NEXP01NXF16	N-way narrow exponent range. Divides inputs by multiples of 4.0: the result is greater than or equal to 1.0 and less than 4.0
IVP_OEQNXF16[T]	N-way half precision equal to comparison
IVP_OLENXF16[T]	N-way half precision less than or equal to comparison
IVP_OLTNXF16[T]	N-way half precision less than comparison
IVP_ONEQNXF16[T]	N-way half precision not equal comparison
IVP_RECIP0NXF16[T]	N-way first step of reciprocal (Newton-Raphson) computation
IVP_RSQRT0NXF16[T]	N-way first step of reciprocal square root (Newton-Raphson) computation
IVP_SQRT0NXF16	N-way first step of square root (Newton-Raphson) computation
IVP_SUBNXF16[T]	N-way subtract
IVP_TRUNCNXF16[T]	N-way scale and round toward 0
IVP_UFLOATX32[T]	N-way conversion of 32-bit unsigned integer to floating point
IVP_ULEQNXF16[T]	N-way half precision unordered equal to comparison
IVP_ULTQNXF16[T]	N-way half precision unordered less than comparison
IVP_UNNXF16[T]	N-way half precision comparison unordered
IVP_UTRUNCNXF16[T]	N-way scale and round toward 0 unordered
IVP_UEQNXF16[T]	N-way half precision unordered equal to comparison
IVP_ULENXF16[T]	N-way half precision unordered less than or equal to comparison
ULTNXF16[T]	N-way half precision unordered less than comparison

Scalar Floating-Point Operations List

Table 59: List of Scalar Floating-Point Single Precision Operations

Operation Name	Synopsis
CLSFY.S	scalar single precision floating point classification operation
ABS.S	scalar single precision floating point Absolute value operation
ADD.S	scalar single precision floating point Add operation
ADDEXP.S	scalar single precision floating point Add exponent operation
ADDEXPM.S	scalar single precision floating point Add exponent from mantissa operation
CONST.S	scalar single precision floating point Constant operation: moves a constant into a vector register
IVP_CVTF16F32	scalar single precision floating point Integer convert operation, converting to scalar half precision floating point
IVP_CVTF32F16	scalar half precision floating point Integer convert operation, converting to scalar single precision floating point
DIV0.S	Beginning step operation for single precision scalar–divide
DIVN.S	Final step operation for single precision scalar divide
FICEIL.S	scalar single precision round to integral ceiling
FIFLOOR.S	scalar single precision round to integral floor
FIRINT.S	Scalar single precision round with mode to integral
FIROUND.S	Scalar single precision round towards nearest integral value. If fractional input is exactly $\frac{1}{2}$, rounds away from 0
FITRUNC.S	Scalar single precision round towards 0 integral value (truncation)
FLOAT.S	Scalar integer conversion to a single precision floating point value using currently set round mode.
MADD.S	scalar single precision floating point Multiply and accumulate operation
MADDN.S	scalar single precision floating point Multiply and accumulate operation, Round to nearest
MAX.S	Scalar single precision floating point maximum operation
MIN.S	Scalar single precision floating point minimum operation.

Operation Name	Synopsis
MAXNUM.S	Scalar single precision floating point numeric maximum operation
MINNUM.S	Scalar single precision floating point numeric minimum operation.
MKDADJ.S	Scalar single precision floating point division adjustment operation, used in the floating point division operations sequence
MKSADJ.S	Scalar single precision floating point square root adjustment operation, used in the floating point square root operations sequence
MOV.S	Moves 32-bit lsbs of a source vector register to the 32-lsbs of a destination vector register
MOVEQZ.S	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if the second input operand is zero
MOV.F.S	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if predicated FALSE
MOVGEZ.S	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if the second input operand is greater than or equal to zero
MOVLTZ.S	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if the second input operand is less than zero
MOVNEZ.S	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if the second input operand is non-zero
MOVT.S	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if predicated TRUE
MSUB.S	scalar single precision floating point Multiply and subtract from accumulator operation
MSUBN.S	scalar single precision floating point Multiply and subtract from accumulator operation, Round to nearest
MUL.S	scalar single precision floating point Multiply operation
NEG.S	scalar single precision floating point Negate operation
NEXP01.S	scalar single precision floating point narrow exponent range. Divides inputs by multiples of 4.0: the result is greater than or equal to 1.0 and less than 4.0
OEQ.S	scalar single precision floating point Equal to comparison, with ordered NaNs
OLE.S	scalar single precision floating point Less than or equal to comparison, with ordered NaNs

Operation Name	Synopsis
OLT.S	scalar single precision floating point Less than comparison, with ordered NaNs
UNEQ.S	scalar single precision floating point Not equal to comparison, with unordered NaNs
RECIP0.S	first step of reciprocal (Newton-Raphson) computation (scalar)
RFR	read single floating point operation reads the least significant 32 bits of the vector register input operand into the output address register operand
RSQRT0.S	first step of reciprocal square root (Newton-Raphson) computation (scalar)
SQRT0.S	first step of square root (Newton-Raphson) computation (scalar)
SUB.S	Scalar single precision floating point subtract
TRUNC.S	Scalar single precision floating point scale and round toward 0
UFLOAT.S	Scalar single precision floating point conversion of 32-bit unsigned integer to floating point
ULEQ.S	Scalar floating point single precision unordered equal to comparison
ULTQ.S	Scalar floating point single precision unordered less than comparison
UN.S	Scalar single precision comparison unordered
UTRUNC.S	Scalar scale and round toward 0 unordered
WFR	moves the contents of input address register to the lsbs of output vector register
IVP_MOVSCFV	scalar in vector register 32-bit signed Move operation, converting to status and control flag state
IVP_MOVVSCF	status and control flag state signed Move operation, converting to 32-bit scalar in vector register
UEQ.S	scalar single precision floating point Equal to comparison, with unordered NaNs
ULE.S	scalar single precision floating point Less than or equal to comparison, with unordered NaNs
ULT.S	scalar single precision floating point Less than comparison, with unordered NaNs
IVP_EXT0IB	The extension from bit 0 operation replicates bit0 of the input element of the input data operand into the lsb number of output bits specified by the immediate. Only power-of-two immediate values from 1 to the length of the output bitwidth of the register can be specified

Table 60: List of Scalar Floating-Point Half Precision Operations

Operation Name	Synopsis
CLSFY.H	scalar half precision floating point classification operation
ABS.H	scalar half precision floating point Absolute value operation
ADD.H	scalar half precision floating point Add operation
ADDEXP.H	scalar half precision floating point Add exponent operation
ADDEXPM.H	scalar half precision floating point Add exponent from mantissa operation
CONST.H	scalar half precision floating point Constant operation: moves a constant into a vector register
DIV0.H	Beginning step operation for half precision scalar-divide
DIVN.H	Final step operation for half precision scalar divide
FICEIL.H	scalar half precision round to integral ceiling
FIFLOOR.H	scalar half precision round to integral floor
FIRINT.H	Scalar half precision round with mode to integral
FIROUND.H	Scalar half precision round towards nearest integral value. If fractional input is exactly $\frac{1}{2}$, rounds away from 0
FITRUNC.H	Scalar half precision round towards 0 integral value (truncation)
FLOAT.H	Scalar integer conversion to a half precision floating point value using currently set round mode.
MADD.H	scalar half precision floating point Multiply and accumulate operation
MADDN.H	scalar half precision floating point Multiply and accumulate operation, Round to nearest
MAX.H	Scalar half precision floating point maximum operation
MIN.H	Scalar half precision floating point minimum operation.
MAXNUM.H	Scalar half precision floating point numeric maximum operation
MINNUM.H	Scalar half precision floating point numeric minimum operation.
MKDADJ.H	Scalar half precision floating point division adjustment operation, used in the floating point division operations sequence

Operation Name	Synopsis
MKSADJ.H	Scalar half precision floating point square root adjustment operation, used in the floating point square root operations sequence
MOV.H	Moves 32-bit lsbs of a source vector register to the 32-lsbs of a destination vector register
MOVEQZ.H	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if the second input operand is zero
MOVF.H	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if predicated FALSE
MOVGEZ.H	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if the second input operand is greater than or equal to zero
MOVLTZ.H	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if the second input operand is less than zero
MOVNEZ.H	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if the second input operand is non-zero
MOVT.H	Moves 64-bit lsbs of a source vector register to the 64-lsbs of a destination vector register if predicated TRUE
MSUB.H	scalar half precision floating point Multiply and subtract from accumulator operation
MSUBN.H	scalar half precision floating point Multiply and subtract from accumulator operation, Round to nearest
MUL.H	scalar half precision floating point Multiply operation
NEG.H	scalar half precision floating point Negate operation
NEXP0.H	scalar half precision floating point narrow exponent range. Divides inputs by multiples of 2.0: the result is greater than or equal to 1.0 and less than 2.0
NEXP01.H	scalar half precision floating point narrow exponent range. Divides inputs by multiples of 4.0: the result is greater than or equal to 1.0 and less than 4.0
OEQ.H	scalar half precision floating point Equal to comparison, with ordered NaNs
OLE.H	scalar half precision floating point Less than or equal to comparison, with ordered NaNs
OLT.H	scalar half precision floating point Less than comparison, with ordered NaNs
UNEQ.H	scalar half precision floating point Not equal to comparison, with unordered NaNs

Operation Name	Synopsis
RECIP0.H	first step of reciprocal (Newton-Raphson) computation (scalar)
RSQRT0.H	first step of reciprocal square root (Newton-Raphson) computation (scalar)
SQRT0.H	first step of square root (Newton-Raphson) computation (scalar)
SUB.H	Scalar half precision floating point subtract
TRUNC.H	Scalar half precision floating point scale and round toward 0
UFLOAT.H	Scalar half precision floating point conversion of 32-bit unsigned integer to floating point
ULEQ.H	Scalar floating point half precision unordered equal to comparison
ULTQ.H	Scalar floating point half precision unordered less than comparison
UN.H	Scalar half precision comparison unordered
UTRUNC.H	Scalar scale and round toward 0 unordered
UEQ.H	scalar half precision floating point Equal to comparison, with unordered NaNs
ULE.H	scalar half precision floating point Less than or equal to comparison, with unordered NaNs
ULT.H	scalar half precision floating point Less than comparison, with unordered NaNs

6.6 Floating Point References

- [1] M. Baudin and R. L. Smith, "A robust complex division in Scilab," October 2012, available at <http://arxiv.org/abs/1210.4539>.
- [2] Prof. W. Kahan, Joseph D. Darcy, "How Java's Floating-Point Hurts Everyone Everywhere", Originally presented 1 March 1998 at the invitation of the ACM 1998 Workshop on Java for High-Performance Network Computing (July 2004), Stanford University, Palo Alto, CA. Available at <http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf>.
- [3] Younis Hijazi, Hans Hagen, Charles Hansen, and Kenneth I. Joy, "Why Interval Arithmetic Is So Useful".
- [4] Standards Committee of IEEE Society, "IEEE Standard for Binary Floating Point Arithmetic", March 1985, IEEE New York, NY, USA. Available at <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=2355>

[5] IEEE Computer Society, "IEEE Standard for Floating-Point Arithmetic", August 2008, IEEE New York, NY, USA. Available at <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4610933>

7. Histogram Calculation Acceleration Operations

Topics:

- [*Algorithms and Terminology*](#)
- [*Histogram Operations*](#)

The operations in the Histogram package compute a histogram of two $2N \times 8U$ input vectors, producing a histogram of $N \times 16U$ counts (bins) in the output vector. Different versions of histogram operations generate cumulative histogram counts where each input value is counted as its own and all higher values; this directly computes a cumulative histogram. Other versions of the count operations use two $vboolN$ inputs to control which input elements are counted; this allows arbitrary regions and boundaries to be handled directly.

7.1 Algorithms and Terminology

A histogram algorithm determines whether to associate the value of an input variable P_i to an output variable in an array of variables Bin_j and if so, $CBin_j$ the element-count, is increased. If multiple values $\{P_i, P_k, \dots\}$ are associated with a particular Bin_j , the value of count of elements in Bin_j is increased multiple times to reflect the number of inputs which are associated with Bin_j . An array of $CBin_j$ arranged in ascending order $[CBin_0, CBin_1, CBin_2, \dots, CBin_{N-1}]$ containing counts of elements is a histogram of size N .

For histogram distribution count of elements, $CBin_j$, in Bin_j is increased according to the following formula:

- (1) $if(Bound_j \leq P_i < Bound_{j+1}): CBin_j = CBin_j + Weight_i, else: CBin_j = CBin_j$
- (2) $Bound_{j+1} > Bound_j$

$Bound_{j+1} - Bound_j$ is the width of Bin_j (bin width)

$Weight_i$ are application defined weights associated with individual inputs

For cumulative histogram calculation $CBin_j$ calculated according to Eq. (1) and the final $CBin_m$ is calculated according to Eq. (3)

- (3) $CBin_m = \sum_{j=0}^{m-1} CBin_j$

or alternatively:

- (4) $if(P_i < Bound_{j+1}): CBin_j = CBin_j + Weight_j, else: CBin_j = CBin_j$

For the basic histogram operations we restrict:

- (5) $Bound_{j+1} - Bound_j \equiv 1$
- (6) $Weight_i \equiv 1$,

which simplifies Eq. (1) and (4) for histogram distribution and cumulative histograms respectively:

- (7) $if(Bound_j == P_i): CBin_j = CBin_j + 1, else: CBin_j = CBin_j$

- (8) $if(P_i < Bound_{j+1}): CBin_j = CBin_j + 1, else: CBin_j = CBin_j$

7.2 Histogram Operations

These operations are designed for fast computation of distribution and cumulative histograms for positive integers.

Each operation has one cycle throughput. During that cycle, an operation takes 128 bytes of data from two narrow vector registers and produces 32 16-bit bins filled with histogram counts.

Histogram operations are capable of sustaining histogramming of 128 8-bit data items into 32 16-bit bins per cycle.

The Histogram package uses existing instruction formats that support two load operations or one load and one store operation, a histogram (COUNT) operation and an add operation. The two load operations enable full throughput even for histograms with 32 or fewer bins, where the histogram is held in registers and the data to the histogram is streamed from memory. The load and store operations enable full throughput for larger histograms where the histogram itself must be streamed from and back to memory.

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

COUNT operations take a scaling input parameter which controls the amount of shift the input data is shifted to the right before the histogram is computed. For example, the amount of shift 0 means that the data is not shifted or with respect to the original input data the width of the bin is 1, while setting the shift amount to 3, will right shift input data by 3. The shift by 3 effectively "widens" the input data bin by a factor of 8. In order to efficiently compute histograms larger than 32 bins, the histogram operations increment the histogram bin range by 32 with each subsequent operation. If we histogram 256-value (8-bit) data with the bin width 1, we execute histogram operations 8 times on the same data to produce a 256-bin histogram. The first time the operation is executed it fills bins 0 to 31, second time 32 to 63, .., last time 224 to 255.

The histogram COUNT operations which contain suffix "Z" in their name reset the range to 0 to 31 before computing histogram and increment it by 32 (32 to 63) for the next operation.

COUNT operations without suffix "Z" in their name take the range from the input address register, compute histogram, and increment the range by 32 after computing histogram.

With respect to input data histogram operations with suffix "M" (Masked) are predicated TRUE operations, they avoid histogramming data items which are not predicated TRUE.

Histograms Distribution Calculation Operations

The calculations performed to calculate basic histogram distribution are equivalent to Eq. (7)

IVP_COUNTEQZ4NX8

Resets the bin range to 0 to 31, computes histogram distribution, increments the address register which holds bin range to next 32-bin range.

IVP_COUNTEQ4NX8

Takes the bin range from an address register, computes histogram distribution, increments the address register which holds bin range to next bin range.

IVP_COUNTEQMZ4NX8

Resets the bin range to 0 to 31, computes histogram distribution, increments the address register which holds bin range to next 32-bin range.

This is a predicated TRUE operation, only data items which are predicated TRUE are taken into account, other input data is ignored – not part of the computed histogram.

IVP_COUNTEQM4NX8

Takes the bin range from an address register, computes histogram distribution, increments the address register which holds bin range to next bin range.

This is a predicated TRUE operation, only data items which are predicated TRUE are taken into account, other input data is ignored – not part of the computed histogram.

Cumulative Histograms Calculation Operations

The calculations performed to calculate basic histogram distribution are equivalent to Eq. (8)

IVP_COUNTLEZ4NX8

Resets the bin range to 0 to 31, computes cumulative histogram, increments the address register which holds bin range to next 32-bin range.

IVP_COUNTLE4NX8

Takes the bin range from an address register, computes cumulative histogram, increments the address register which holds bin range to next bin range.

IVP_COUNTLEMZ4NX8

Resets the bin range to 0 to 31, computes cumulative histogram, increments the address register which holds bin range to next 32-bin range.

This is a predicated TRUE operation, only data items which are predicated TRUE are taken into account, other input data is ignored – not part of the computed histogram.

IVP_COUNTLEM4NX8

Takes the bin range from an address register, computes distribution histogram, increments the address register which holds bin range to next bin range.

This is a predicated TRUE operation, only data items which are predicated TRUE are taken into account, other input data is ignored – not part of the computed histogram.

8. On-Line Information

Xtensa Xplorer offers a number of on-line Instruction Set Architecture (ISA) documentation references in HTML format for Xtensa configurations, including the Vision P6. 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 topics describing the ISA, protos, and other documentation information.

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.

Vision P6 ISA Enhancements over Vision P5

Topics:

- [*New Operations*](#)
- [*New Operation Variants*](#)
- [*Enhanced Operations*](#)
- [*Half-Precision Vector Floating-Point Option*](#)
- [*Slotting Changes*](#)

Vision P6 provides a number of ISA enhancements relative to Vision P5, the previous generation Cadence Tensilica vision/imaging DSP. These enhancements include both new operations and slotting optimizations, which enable Vision P6 to achieve higher performance compared to Vision P5 on many targetted applications.

The Vision P6 ISA is a superset of the Vision P5 ISA, and Vision P6 is source code compatible with Vision P5. The machines are not binary compatible, however; Vision P5 code must be recompiled for Vision P6.

Vision P6 adds both algorithmically new operations as well as new operation variants (e.g. adding a 2NX8 variant to the existing N_2X32 variant on Vision P5). This section provides an overview of the enhancements in Vision P6 and is primarily of interest to those familiar with Vision P5. For more detailed descriptions of the new Vision P6 operations see the Chapter [*DSP Operations by Type*](#) on page 66.

New Operations

Paired multiply operations

Paired multiply operations perform two multiplications on two pairs of inputs and then add the two products before writing the result to the wide vector output. For multiply-accumulates the result is added to the corresponding SIMD element in the inout wide vector.

The operations support both vector by scalar and vector by vector multiplications. 16-bit by 16-bit, 8-bit by 16-bit, and 8-bit by 8-bit element widths operations are supported.

Twenty four paired multiply operations are added in Vision P6 which allow it to achieve 2X the peak MAC throughput of Vision P5. The new paired multiplication operations are:

```
IVP_MUL[|US|UU]P[A]NX16
IVP_MUL[US]P[A]N16XR16
IVP_MUL[US]P[A]2N8XR16
IVP_MUL[US]P[A]I2N8XR16
IVP_MUL[|US|UU]P[A]2NX8
```

Quad multiply operations

Quad multiply operations perform four multiplications on four pairs of inputs and then add the four products before writing the result to the wide vector output. For multiply-accumulates the result is added to the corresponding SIMD element in the inout wide vector.

The operations support 8-bit by 8-bit wide element vector by scalar multiplications.

There are eight quad multiply operations available in Vision P6 which allow to achieve 4X peak MAC throughput of Vision P5.

The new quad multiplication operations are:

```
IVP_MUL[US]Q[A]2N8XR8
IVP_MUL[US]4T[A]2N8XR8
```

Vector Boolean reduction operations

Vision P5 does not have any vector Boolean reduction operations. Vector Boolean reduction operations perform computations along the lanes of a vbool register (similar to ALU reduction operations performing computations along the lanes of a vec register). The output of a vector Boolean reduction is a 1-bit Boolean scalar (vbool1 C type). The 1-bit result can be used as a condition variable in a control statement, thus among other things, these operations accelerate Boolean vector to execution control flow calculations. There are two reduction logical functions introduced: logical AND (RANDB operations) and logical OR (RORB

operations), each having three variants (operating on vboolN, vbool2N, and vboolN_2 inputs). Therefore, in total, Vision P6 supports six vector Boolean reduction operations.

The new vector Boolean reduction operations are:

```
IVP_RANDB[2N|N|N_2]  
IVP_RORB[2N|N|N_2]
```

Extract with position from AR

In Vision P5 extract operations copy a lane of vec register to AR with the lane position encoded as an immediate. This puts a limitation that the position of the lane, which needs to be extracted, must be known at compile time. Vision P6 removes this limitation by introducing an extract operation which receives the lane position from AR. This extract operation, `IVP_EXTRVRN_2X32`, provides acceleration in the situations when the lane position to be copied to AR becomes known only at runtime.

New base Xtensa operation

The new, set AR if less than, `SALT[U]`, operation improves computation of compound conditions.

New Operation Variants

ALU operations

Vision P5 supports signed 8-bit operations only for some ALU operations. Vision P6 adds 8-bit signed support for nine operations making 8-bit sign support universal for ALU operations.

The new 8-bit signed ALU operations are:

```
IVP_[B]MAX2NX8  
IVP_[B]MIN2NX8  
IVP_ABS2NX8  
IVP_ABSSUB2NX8  
IVP_NEG2NX8  
IVP_EXTR2NX8  
IVP_MOVA8
```

Average with and without rounding

Average operations, both with and without rounding (AVG[R]), add support for the signed and unsigned NX16 types, as well as for the signed 2NX8 type.

```
IVP_AVG[R] [U] NX16  
IVP_AVG[R] 2NX8
```

Wide vector add and subtract

In addition to supporting 2Nx8 input in Vision P5 for ADDW, Vision P6 adds support for SUBW, and both ADDW and SUBW for Nx16 types.

```
IVP_SUBW[U] [A] 2NX8  
IVP_SUBW[U] [A] NX16  
IVP_ADDW[U] [ | A | S ] NX16
```

Enhanced Operations

Select immediate

Six new patterns have been added to select immediate operations, SELI. These are individual output patterns that were available in Vision P5 in DSELI operations but not in SELI as well as other missing patterns to support rotate-right/rotate-left 3,5,7 bytes.

Table 61: New SELI Operation Patternns

New Immediate value	New pattern
67	IVP_SELI_8B_ROTATE_RIGHT_3
68	IVP_SELI_8B_ROTATE_LEFT_3
69	IVP_SELI_8B_ROTATE_RIGHT_5
70	IVP_SELI_8B_ROTATE_LEFT_5
71	IVP_SELI_8B_ROTATE_RIGHT_7
72	IVP_SELI_8B_ROTATE_LEFT_7

Shuffle immediate

Two new patterns have been added to the SHFLI operation. These are swap halves and byte interleave half

Table 62: New SHFLI Operation Patternns

New Immediate value	New pattern	Pattern Illustration
30	IVP_8B_INTERLEAVE_1_HALVES	0,32,1,33,2,34,...,31,63
31	IVP_SHFLI_8B_SWAP_32	32,33,34,...,62,63,0,1,2,...,30,31

Specialized select and shuffle immediate

These specialized operations implement a subset of the general select and shuffle immediate operations' patterns. The idea behind specialized operations is to implement light weight versions of general operations and make them available in more slots due to their smaller hardware cost. Thus, only the most frequently encountered patterns are implemented by the specialized select and shuffle immediate operations. When the compiler encounters a select immediate or a shuffle immediate operation, it checks if the pattern is available in the specialized operation as well. If it is available in a specialized operation, the compiler has an additional choice of scheduling operations from slots where the “full” operations are not available.

There are six specialized SELI and SHFLI operations listed below:

```
IVP_SEL2NX8I_[S0|S2|S4]
IVP_SHFL2NX8I_[S0|S2|S4]
```

The specialized SELI operations implement a subset of 28 (out of 73) patterns.

Table 63: Specialized SELI Operation Patternns

Immediate Value	Pattern
65	IVP_SELI_8B_ROTATE_RIGHT_1
0	IVP_SELI_8B_ROTATE_RIGHT_2
67	IVP_SELI_8B_ROTATE_RIGHT_3*
1	IVP_SELI_8B_ROTATE_RIGHT_4
69	IVP_SELI_8B_ROTATE_RIGHT_5*
2	IVP_SELI_8B_ROTATE_RIGHT_6
71	IVP_SELI_8B_ROTATE_RIGHT_7*
3	IVP_SELI_8B_ROTATE_RIGHT_8
66	IVP_SELI_8B_ROTATE_LEFT_1
8	IVP_SELI_8B_ROTATE_LEFT_2

Immediate Value	Pattern
68	IVP_SEL1_8B_ROTATE_LEFT_3*
9	IVP_SEL1_8B_ROTATE_LEFT_4
70	IVP_SEL1_8B_ROTATE_LEFT_5*
10	IVP_SEL1_8B_ROTATE_LEFT_6
72	IVP_SEL1_8B_ROTATE_LEFT_7*
11	IVP_SEL1_8B_ROTATE_LEFT_8
61	IVP_SEL1_8B_INTERLEAVE_1_EVEN
62	IVP_SEL1_8B_INTERLEAVE_1_ODD
34	IVP_SEL1_16B_INTERLEAVE_1_EVEN
35	IVP_SEL1_16B_INTERLEAVE_1_ODD
59	IVP_SEL1_8B_INTERLEAVE_1_LO
60	IVP_SEL1_8B_INTERLEAVE_1_HI
32	IVP_SEL1_16B_INTERLEAVE_1_LO
33	IVP_SEL1_16B_INTERLEAVE_1_HI
63	IVP_SEL1_8B_EXTRACT_1_OF_2_OFF_0
64	IVP_SEL1_8B_EXTRACT_1_OF_2_OFF_1
16	IVP_SEL1_16B_EXTRACT_1_OF_2_OFF_0
17	IVP_SEL1_16B_EXTRACT_1_OF_2_OFF_1

Patterns marked with an * are new to Vision P6 and also added to IVP_SEL2NX8I (see above).

The specialized SHFLI operation implements a subset of 2 (out of 32) patterns

Table 64: Specialized SHFLI Operation Patterns

Immediate Value	Pattern
8	IVP_SHFLI_SWAP_1
30	IVP_8B_INTERLEAVE_1_HALVES*

Half-Precision Vector Floating-Point Option

Vision P6 makes available the half-precision vector floating-point option (package). It consists of 141 operations, which are the same as single precision operations but perform N 16-bit half precision floating operations instead of N/2 32-bit single precision floating operations. This package uses the same control and status registers as its single precision counterpart. All half precision operations have the same latency as single precision operations. The half-precision package configuration can be selected independent of selecting the single precision package.

Slotting Changes

Numerous slotting additions (a specific operation is in a Slot-Format in Vision P6, but it was not in this Slot-Format in Vision P5) have been made in Vision P6 slotting resulting in a greater ability to bundle various operations together in a single FLIX instruction and improve the overall performance of the machine. A new wide four-slot format, F4, was added. This format serves to bundle new four narrow vector register input multiplication operations.

Slotting for two groups of operations has been reduced:

1. The LTR operations (`IVP_LTR2N[I]`, `IVP_LTRN[I]`, `IVP_LTRN_2[I]`, `IVP_LTRS2N`, `IVP_LTRSN`, `IVP_LTRSN_2`) have been removed from Slot 0 (Formats N1 and N2)
2. The non-immediate variants of the 16-bit right shifts (`IVP_SRLNX16` and `IVP_SRSNX16`) have been removed from Slot 2 (Format F2)

These removals should not have a noticeable impact on performance.

Configuration Options

Vision P6 is configured using Xtensa Xplorer. A number of customer-configurable options are available, including, for example:

- Size of instruction memory
- Size and organization of instruction cache
- Size of data memory and number of sub-banks
- ECC on local memories
- Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package, or Xtensa Single or Single +Double Precision FP coprocessor
- Vision P6 Histogram option package

The customer-specified configuration is checked by Xplorer during configuration entry for validity of the options, and warning and error messages highlight settings to review and invalid settings or combinations. Users should refer to Xplorer for further details on the configuration options available and determining valid Vision P6 configurations.

User TIE Extension

Topics:

- [User Formats](#)
- [Naming Restrictions](#)
- [User Registers](#)

Users can write their own operations with user TIE to extend an Xtensa base machine like a Vision P6. These user-defined operations can operate on Vision P6 registers and states, and be assigned to Vision P6 and/or user-defined instruction formats.

This chapter describes the Vision P6 requirements and limitations on user TIE extensions. The *Tensilica Instruction Extension (TIE) Language User's Guide* is a good reference to learn the fundamentals of TIE and writing TIE code to extend a core's ISA.

User Formats

Vision P6 can be extended with user TIE defining new operations. These operations can be assigned to the 24-bit regular instruction format, or you may wish to use the 64-/128-bit Vision P6 FLIX instruction formats which are available for user instruction additions. User-defined formats can also be used, and a portion of the Vision P6 FLIX instruction encoding space has been reserved for this purpose; placing user-defined formats in this space ensures the user-defined formats won't overlap or conflict with any of the standard Vision P6 instruction formats. Note that, due to the encoding structure, there are fewer bits available in the user-defined instruction format than the overall length of the instruction.

We illustrate the process of defining the user FLIX instruction format and adding instructions to it using two simple TIE examples, as follows. The first TIE example defines a new FLIX format, consisting of five FLIX slots. The following TIE template can be used to define a new 128-bit wide FLIX format:

```
// Define a new 128-bit wide FLIX format for user instructions in Vision P6
// Use this length declaration:
length lu128 128 { InstBuf[3:0] == 4'he }
// Use this format declaration:
format user128 lu128 { user_wide_slot0, user_wide_slot1, user_wide_slot2, user_wide_slot3,
user_wide_slot4 } { (((InstBuf[55:54] == 2'h0) && (InstBuf[102:101] == 2'h3)) && (InstBuf[107]
== 1'h1)) && (InstBuf[114:113] == 2'h2)) && (InstBuf[121] == 1'h1)}
// For now, only assign NOPs to the user slots
slot_opcodes user_wide_slot0 {NOP}
slot_opcodes user_wide_slot1 {NOP}
slot_opcodes user_wide_slot2 {NOP}
slot_opcodes user_wide_slot3 {NOP}
slot_opcodes user_wide_slot4 {NOP}
```

Based on the scheduling requirements in the target application, users may now add one or more operations - either from the Vision P6 ISA or user defined TIE operations - to the slot_opcodes defined. Based on the amount of encoding space available, the TIE Compiler will attempt to encode the format and opcodes assigned to its one or more slots.

Similarly, the following TIE template can be used to define a new 64-bit narrow FLIX format:

```
// Define a new 64-bit (narrow) FLIX format for user instructions in Vision P6
// Use this length declaration:
length lu64 64 { InstBuf[4:0] == 5'hf }
// Use this format declaration:
format user64 lu64 { user_narrow_slot0, user_narrow_slot1 } { InstBuf[55:54] == 2'h3 }
// For now, only assign NOPs to the user slots
slot_opcodes user_narrow_slot0 {NOP}
slot_opcodes user_narrow_slot1 {NOP}
```

Note TIE compiler places restrictions on the total number of FLIX formats and slots. Vision P6 defines 10 FLIX formats containing 39 total slots which count towards these totals. See the *Tensilica Instruction Extension (TIE) Language Reference Manual* for further details.

Naming Restrictions

There are restrictions on the names that can be used in a user TIE file when extending the Vision P6. The following TIE elements are reserved and cannot be used in TIE added to a configuration that includes the Vision P6. In general, names prefixed by "IVP_", "bbe_", and "xb_" are reserved for the Vision P6 coprocessor. More specifically, the names of the different TIE features used by the coprocessor are discussed below. The following table shows the state and register file names used by the Vision P6:

Table 65: Register File and State Names

Name	Type	Comment
vec	regfile	
wvec	regfile	
valign	regfile	
vbool	regfile	
gvr	regfile	
RoundMode	state	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package
InvalidFlag	state	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package
DivZeroFlag	state	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package
OverflowFlag	state	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package
UnderflowFlag	state	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package

Name	Type	Comment
InexactFlag	state	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package
InvalidEnable	state	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package
DivZeroEnable	state	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package
OverflowEnable	state	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package
UnderflowEnable	state	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package
InexactEnable	state	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package

Naming restrictions for other TIE constructs are as follows:

- Coprocessor Number: 1
- ctype Names: All Vision P6 ctype names are reserved names. They are all prefixed with "xb_". In addition, the ctypes starting with "vbool", and "valign" are also reserved.
- Operation Names: All Vision P6 operation names are reserved names, which are either: Prefixed with IVP_, named RUR.<User register name as defined above> or named WUR.<User register name as defined above>.
- Proto Names: All Vision P6 intrinsic (proto) names are reserved intrinsic names. These all have the prefix "IVP_", or the prefix that begins with the ctype names "xb _", "vbool", "valign" (for data type conversions protos).
- TIE Function Names: The Vision P6 uses a number of TIE functions, which you should not use for your own TIE functions. All functions have the prefix "xdref_", which should not be used by user TIE.

- Semantic Names: The Vision P6 uses a number of semantic names, which you should not use within your own TIE semantics. All semantics have the prefix “ivp_”, which should not be used by user TIE.

User Registers

The following table shows the user registers defined by Vision P6.

Table 66: User Register Entries

User Register Name	User Register Number	Comment
FCR	232	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package
FSR	233	Vision P6 Single Precision Vector Floating Point option package, Vision P6 Half Precision Vector Floating Point option package



Attention: User TIE should not use User Registers above 223.

