CEVA-XM4™ Architecture Specification Volume I

Rev 1.1.3.F

June 2016

Documentation Control

History Table

Version	Date	Description	Remarks
1.0.0.F	January 2015	Beta Release	
1.1.0.F	August 2015	Official Release	
1.1.1.0	October 2015	Beta Release	
1.1.1.1	October 2015	Beta Release	No change
1.1.1.2	October 2015	Beta Release	No change
1.1.1.3	November 2015	Beta Release	No change
1.1.1.6	December 2015	Beta Release	No change
1.1.1.A	January 2016	Official Release	
1.1.1.F	February 2016	Official Release	
1.1.2.F	March 2016	Official Release	
1.1.3.F	June 2016	Official Release	

Disclaimer and Proprietary Information Notice

The information contained in this document is subject to change without notice and does not represent a commitment on any part of CEVA®, Inc. CEVA®, Inc. and its subsidiaries make no warranty of any kind with regard to this material, including, but not limited to implied warranties of merchantability and fitness for a particular purpose whether arising out of law, custom, conduct or otherwise.

While the information contained herein is assumed to be accurate, CEVA®, Inc. assumes no responsibility for any errors or omissions contained herein, and assumes no liability for special, direct, indirect or consequential damage, losses, costs, charges, claims, demands, fees or expenses, of any nature or kind, which are incurred in connection with the furnishing, performance or use of this material.

CEVA®, CEVA-XC™, CEVA-XC5™, CEVA-XC321™, CEVA-XC323™, CEVA-XC8™, CEVA-Xtend™, CEVA-XC4000™, CEVA-XC4100™, CEVA-XC4200™, CEVA-XC4210™, CEVA-XC4400™, CEVA-XC4410™, CEVA-XC4500™, CEVA-XC4600™, CEVA-TeakLite™, CEVA-TeakLite-II™, CEVA-TeakLite-III™, CEVA-TL3210™, CEVA-TL3211™, CEVA-TeakLite-4™, CEVA-TL410™, CEVA-TL411™, CEVA-TL420™, CEVA-TL421™, CEVA-Quark™, CEVA-Teak™, CEVA-X™, CEVA-X1620™, CEVA-X1622™, CEVA-X1641™, CEVA-X1643™, Xpert-TeakLite-II™, Xpert-Teak™, CEVA-XS1100A™, CEVA-XS1200™, CEVA-XS1200A™, CEVA-TLS100™, Mobile-Media™, CEVA-MM1000™, CEVA-MM2000™, CEVA-SP™, CEVA-VP™, CEVA-MM3000™, CEVA-MM3100™, CEVA-MM3101™, CEVA-XM™, CEVA-XM4™, CEVA-X2™ CEVA-Audio™, CEVA-HD-Audio™, CEVA-VoP™, CEVA-Bluetooth™, CEVA-SATA™, CEVA-SAS™, CEVA-Toolbox™, SmartNcode™ are trademarks of CEVA, Inc.

All other product names are trademarks or registered trademarks of their respective owners.

CEVA® makes great efforts to provide a user-friendly software and hardware development environment. Along with this, CEVA provides comprehensive documentation, enabling users to learn and develop applications on their own. Due to the complexities involved in the development of DSP applications that might be beyond the scope of the documentation, an online Technical Support Service has been established. This service includes useful tips and provides fast and efficient help, assisting users to quickly resolve development problems.

How to Get Technical Support:

- **FAQs**: Visit our website http://www.ceva-dsp.com or your company's protected page on the CEVA website for the latest answers to frequently asked questions.
- **Application Notes**: Visit our website http://www.ceva-dsp.com or your company's protected page on the CEVA website for the latest application notes.
- **Email**: Use the CEVA central support email address ceva-support@ceva-dsp.com. Your email will be forwarded automatically to the relevant support engineers and tools developers who will provide you with the most professional support to help you resolve any problem.
- **License Keys**: Refer any license key requests or problems to sdtkeys@ceva-dsp.com. For SDT license keys installation information, see the *SDT Installation and Licensing Scheme Guide*.

Email: ceva-support@ceva-dsp.com
Visit us at: www.ceva-dsp.com

List of Sales and Support Centers

Israel	USA	Ireland	Sweden
2 Maskit Street P.O. Box 2068 Herzelia 46120 Israel **Tel:** +972 9 961 3700 **Fax:** +972 9 961 3800	1174 Castro Street Suite 210 Mountain View, CA 94040 USA **Tel**: +1-650-417-7923 **Fax**: +1-650-417-7924	Segrave House 19/20 Earlsfort Terrace 3rd Floor Dublin 2 Ireland **Tel**: +353 1 237 3900 **Fax**: +353 1 237 3923	Klarabergsviadukten 70 Box 70396 107 24 Stockholm, Sweden **Tel**: +46(0)8 506 362 24 **Fax**: +46(0)8 506 362 20
China (Shanghai)	**China (Beijing)**	**China (Shenzhen)**	**Hong Kong**
Unit 1203, Building E Chamtime Plaza Office Lane 2889, Jinke Road Pudong New District Shanghai, 201203 China **Tel**: +86-21-20577000 **Fax**: +86-21-20577111	Rm 503, Tower C Raycom InfoTech Park No.2, Kexueyuan South Road Haidian District Beijing 100190 China **Tel**: +86-10 5982 2285 **Fax**: +86-10 5982 2284	Rm 709, Tower A SCC Financial Centre No. 88 First Haide Avenue Nanshan District Shenzhen 518064 China **Tel**: +86-755-8435 6038 **Fax**: +86-755-8435 6077	Level 43, AIA Tower 183 Electric Road North Point Hong Kong **Tel**: +852-39751264
South Korea	**Taiwan**	**Japan**	**France**
#478, Hyundai Arion 147, Gumgok-Dong Bundang-Gu Sungnam-Si Kyunggi-Do, 463-853 South Korea **Tel**: +82-31-704-4471 **Fax**:+82-31-704-4479	Room 621 No.1, Industry E, 2nd Rd Hsinchu, Science Park Hsinchu 300 Taiwan R.O.C **Tel**: +886 3 5798750 **Fax**: +886 3 5798750	1-6-5 Shibuya SK Aoyama Bldg. 3F Shibuya-ku, Tokyo 150-0002 Japan **Tel**: +81 3 5774 8250	RivieraWaves S.A.S 400, avenue Roumanille Les Bureaux Green Side 5, Bât 6 06410 Biot - Sophia Antipo-lis France **Tel**: +33 4 83 76 06 00 **Fax**: +33 4 83 76 06 01

Table of Contents

List of Figures

List of Tables

List of Examples

1 Introduction

1.1 Scope

This document provides the specification for the CEVA-XM4™ architecture framework DSP core.

1.2 Applicable Documents

Applicable documents are:

- CEVA-XM4 Instruction Set (Architecture Specification Volume II)
- CEVA-XM4 Memory Subsystem (MSS)

1.3 CEVA-XM4 Architecture Framework

The CEVA-XM4 is a licensable DSP and memory subsystem platform targeted for high-performance computer vision and image processing applications that provide very high processing power while maintaining a small footprint and low power consumption.

The CEVA-XM4 consists of a Vector Processor (VP), which is the main DSP core responsible for the platform's data processing, a Program Memory Subsystem (PMSS) and a Data Memory Subsystem (DMSS).

The CEVA-XM4 is an extremely powerful, low-power DSP processor designed and optimized for computer vision and image processing. This fully programmable architecture supports the high computational pixel and features processing requirements of the most advanced computer vision and image processing in software applications today.

The extreme performance demands are coupled with the necessity for a low-power solution. This stems from the need for highly integrated systems on a chip (SoCs) with inexpensive packaging and longer battery life in mobile devices. The VP is specifically designed to address the stringent power consumption, time-to-market and cost constraints associated with developing high-performance video and image processing for the mobile market.

The CEVA-XM4 support automotive application requires medium level of Safety Integrity Level (SIL) in the area of image processing. The CEVA-XM4 targets ISO-26262 SIL-B integrity level. For further details, refer to *CEVA-XM4 Safety Manual*

1.3.1 Vector Processor Highlights

1.3.1.1 Instruction-level Parallelism

The CEVA-XM4 architecture has a unique mix of Very Long Instruction Word (VLIW) and Single Instruction Multiple Data (SIMD) architectures. The VLIW architecture enables a high level of concurrent instructions processing, thus providing extended parallelism, as well as low power consumption. The SIMD architecture enables single instructions to operate on multiple data elements, resulting in code size reduction and increased performance. Low power consumption is also achieved in the CEVA-XM4 by its instructions and dedicated mechanisms.

1.3.1.2 High-level Programming

The CEVA-XM4 architecture enables efficient programming in high-level languages that significantly reduces development cost and time-to-market. The CEVA-XM4 architecture is designed in conjunction with the CEVA-XM4 C compiler. CEVA provides a very efficient, optimized C-driven architecture compiler. The optimized C compiler, together with a single-core design, facilitates easier development, integration and debug efforts in target SoCs.

1.3.1.3 Soft Core

CEVA-XM4 design implementations are soft-core based, enabling the customer to select the optimal operating point in terms of die size, power consumption and performance. In addition, the customer has complete flexibility in selecting the foundry, process and complementary IPs. The CEVA-XM4 IP incorporates a fully automated design flow that supports mainstream Electronic Design Automation (EDA) tools, which significantly shortens time-to-market. The CEVA-XM4 design can be ported to a Field-programmable Gate Array (FPGA) that can be used for product prototyping, system integration, design acceleration and clarification.

1.3.1.4 Development Tools, Software and Platforms

The CEVA-XM4 is supported by a complete set of Hardware and Software and Development Tools (SDTs). The software tools include a C Compiler, Macro Assembler, Linker, Debugger, Simulator and Profiler, as well as utilities and DSP libraries working under an Integrated Development Environment (IDE). The Hardware tools contain various modular development system boards with associated accessories. A DSP hardware platform that contains the CEVA-XM4, DMA controller, Power Scaling Unit (PSU), CPU interfaces and a large offering of peripherals and interfaces is also offered. Additional software and algorithms are provided by CEVA through its third-party network.

1.4 CEVA-XM4 Vector Processor

The CEVA-XM4 feature set is described in the following section.

1.4.1 CEVA-XM4 Feature Set

CEVA-XM4 architecture includes the following features:

- Soft IP, fully synthesizable, single-edge clock design, process- and library-independent
- High code compactness due to:
  - Variable instruction width (16-bit, 32-bit, 48-bit and 64-bit)
  - Variable-size instruction packets
  - Instruction replication method
  - Powerful instruction set capabilities
- All instructions support predication:
  - Conditional execution
  - Reduces cycle count and code size on control and overhead code
- Enhanced register file that also includes:
  - 32 32-bit general registers used for scalar operations and address generation
  - 40 256-bit vector registers used for all vector-related operations
- Two Vector Processing Units (VPUs):
  - Parallel processing of up to 256-bits on each operation
  - Supports 32 8-bit, 16 16-bit or eight 32-bit operations in each unit
  - All operations are signed or unsigned
  - Supports both inter-vector and intra-vector operations
  - Up to 64 multipliers of 8×16 bits, 32 multipliers of 16×16 bits and eight multipliers of 32×32 bits on each VPU
  - Advanced filter operations for two-dimensional frames
  - Up to 64 Sum of Absolute Differences (SADs) per cycle with an option to accumulate partial results
  - Ability to sort vectors according to minimum, median and maximum
  - Bit-manipulation operations including vector permutations

- Logical operations
- Non-linear operation support, such as $\frac{1}{x}$, $\sqrt{x}$, $\frac{1}{\sqrt{x}}$
- Supports up to 16 single-precision floating-point operations

- Four Scalar Processing Units (SPUs):
  - Supports 16-bit and 32-bit operations
  - 16×16 bits, 32×16 bits and 32×32 bits multipliers
  - 16×16 bits, 32×16 bits and 32×32 MAC operations
  - Full support of bit-manipulation and logical unit
  - Supports single-precision floating-point operations
- Two Load/Store Units (LSUs) for two independent accesses to the data memories:
  - Maximum bandwidth of 512-bits when using both load units
  - Maximum bandwidth of 256-bits for store operations
  - Ability to access up to 32 different memory addresses in one memory access
  - Multiple data addressing modes, including:
    - Indirect addressing
    - Modulo addressing
    - Direct addressing
    - Indexed addressing
    - Stack addressing
    - Parallel addressing
  - Four gigabyte program and data address
  - Byte-addressable data address
  - Unaligned data memory access
- Program memory subsystem that includes:
  - L1 program memory
  - L1 four-way program cache
  - Dedicated AXI master bus for SoC connectivity
  - Program Direct Memory Access (DMA) available for background transfers and cache preloading

- Data memory subsystem that includes:
  - L1 data memory
  - Data DMA available for background data transfers
  - DMA task queue manager
  - Dedicated AXI master and slave ports for hardware accelerators' connectivity
  - Separate I/O space for peripherals' connectivity
  - Safety Support
    - Single error correction (SEC) and double Error Detection (DED) on all IDM, IPM, Program Tag and Set RAM blocks
    - Parity, Single error correction (SEC) and double Error Detection (DED) on EPP program master port, EDP, AXIm0/1 data master port, EDAP and AXIs0-2 data slave ports.
    - Parity and Single error correction (SEC) on APB3 master
- Fully registered memory interface
- On-chip Emulation (OCEM) support via a JTAG port
- Standard system interface for easy integration with an existing SoC

1.5 Development Tools and Deliverables

A complete set of powerful software and hardware development tools are provided with the CEVA-XM4.

1.5.1 Software Development Tools

Software development tools (SDTs) are available for software application development on the CEVA-XM4. The package includes an advanced IDE-based tool chain. The CEVA-XM4's SDTs are available on all PC/Windows platforms as well as Linux platforms.

1.5.2 XM4-Complete Soft IP Package

The CEVA-XM4 is a soft Silicon Intellectual Property (SIP) that reduces time-to-market for DSP subsystem development. The SIP source code is written in HDL, which is process-independent and may be easily embedded into an SoC using various processes. A key element in soft SIP is the method in which it is packaged and delivered to customers. The package includes the HDL code and a broad set of deliverables that enable customers to process their own design flow in the most simple and straightforward manner. Simulation, verification, synthesis and layout environments, as well as high-level documentation, are included in the delivery as references for customers to achieve quick and smooth integration of the core.

1.5.3 RTL to GDSII

The CEVA-XM4's IP package comprises a complete flow from RTL to GDSII, based on advanced physical synthesis, including synthesis scripts, adjustable constraints, detailed descriptions and synthesis-guiding steps. The scripts are robust and include all the knowledge of the synthesis, dedicated to the specific RTL code.

2 Architecture Overview

2.1 Introduction

The CEVA-XM4 is a DSP based on a VLIW model combined with an SIMD concept. This approach enables the processor to achieve a high level of parallelism, low power consumption and high code density.

CEVA-XM4 architecture is based on a load/store computer architecture utilizing RISC operations and instructions only. The architecture has dedicated load/store and load units responsible for loading/storing data from/to the data memory directly to/from the registers. All other computational instructions always utilize these registers as sources and destinations.

The CEVA-XM4 instruction set can be 16-bits, 32-bits, 48-bits or 64-bits wide. Up to eight such instructions can be grouped to form an instruction packet, which is executed in a single cycle. Each instruction within an instruction packet is associated with a different functional unit in the core.

2.2 CEVA-XM4 Block Diagram

Figure 2-1 presents a block diagram of the XM4. The XM4 consists of four Scalar Processing Units (SPUs), two Load/Store Units (LSUs), a Program Control Unit (PCU), two Vector Processing Units (VPUs), a Power Scaling Unit (PSU), Memory Subsystem (MSS) and Emulation interface.

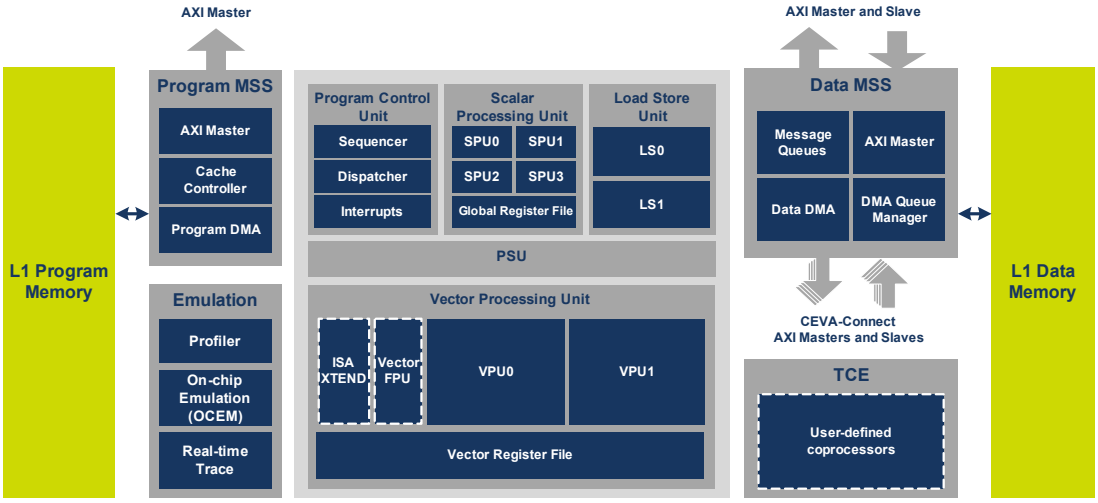


Figure 2-1. Vector Processor Block Diagram

The following sections describe each XM4 block in detail.

2.2.1 Program Control Unit

The Program Control Unit (PCU) is divided into three sub-units: the Dispatcher, Interrupts and the Program Sequencer.

The Dispatcher aligns the instructions from the program memory and dispatches them to the different units. It includes an instruction queue and manages 16-bit and 32-bit instruction alignment.

The Program Sequencer ensures proper program flow, whether sequential or non-sequential. It manages the Program Counter (PC) and has various mechanisms for different types of noncontinuous instructions:

- BKREP Mechanism: Handles loops of a block of instructions, with up to four hardware nesting levels and infinite software nesting levels
- Branch Mechanism: Manages all branch, subroutine calls and subroutine return operations
- Exception Handling: Manages all software/hardware interrupts and resets acceptance

2.2.2 Scalar Processing Unit

The scalar processing unit (SPU) handles all scalar computations and bit-manipulation operations that are non-vectorized DSP operations. The SPUs provide efficient OOB C compiler support for both control and DSP-oriented operations. The SPU consists of four separate computational sub-units named SPU0, SPU1, SPU2 and SPU3. The sub-units are independent and can execute instructions in parallel to other sub-units.

2.2.3 Load and Store Unit

The LSU is responsible for all data memory accesses. The unit is divided into two sub-units, the LS0 and LS1 capable of loading and storing from/to the data memory using various addressing modes.

The LS0 and LS1 units support a load bandwidth of up to 512 bits per cycle, grouped into two individual memory accesses of 256 bits each.

The units also support store bandwidth of up to 256 bits in LS0, or two 64-bit stores in LS0 and LS1.

The LS0 and LS1 units generate two independent 32-bit addresses in each cycle, according to the following addressing modes:

- Indirect, using one of eight pointers, including post-modification
- Indexed, using a base register and an immediate value or a pointer, including post-modification
- Direct, fully embedded in the instruction (long direct)
- Stack, using a Stack Pointer (SP) register for pushing and popping to/from the software stack

In addition to the addressing modes mentioned above, each LSU can also generate a vector of up to 16 addresses that enables scatter-gather memory access in one operation. Using this capability, the XM4 can load data from up to 32 different memory locations in one cycle and gather this data into one, two or four vector registers. The LSUs can also perform memory store operations of up to 16 addresses in a single operation.

The LS0/LS1 units support two types of address register modifications (also referred to as pointers):

- Linear modification
- Modulo modification

All LSU instructions can be conditional using scalar predicates and/or vector predicates.

The source/destination of the data being stored/loaded to/from the data memory can be any one of the core registers, including registers from the GRF, the LSU register file (ARF), the Sequencer Register File (SRF) in the PCU or the VPU Register File (VRF) in the VPU.

2.2.4 Vector Processing Unit

The XM4 includes two independent vector processing units (VPU). The VPUs are responsible for all vector computations. VPUs handle both inter-vector operations (using single-instruction multiple data) and intra-vector operations. Inter-vector instructions can operate on 32 eight-bit (Char, C32) elements, 16 16-bit (Short, S16) elements or eight 32-bit (Int, I8) elements.

Both VPU0 and VPU1 can perform arithmetic operations, logical operations and bit-manipulation operations. Arithmetic operations include addition, subtraction, average, compare, minimum, maximum and more. Logical operations include bitwise and, nand, nor, not, or, exclusive-or, exclusive-nor and more. Bit-manipulation operations include permute, shift and more.

In addition to the above, VPU0 and VPU1 can also perform multiplication operations and specialized operations. Multiplication operations include multiply, multiply-accumulate and 2D filter operations. Specialized operations include sort and sum of absolute-difference operations.

The VPUs are fully pipelined, each with a throughput of up to 32 results per cycle.

2.2.5 Memory Subsystem

The CEVA-XM4 Memory Subsystem (MSS) is an extended system that can be easily adapted for full SoC integration. The MSS consists of a Program Memory Subsystem (PMSS) and Data Memory Subsystem (DMSS). The PMSS comprises an optional L1 program memory and four-way cache. The DMSS contains L1 memory only. While the core accesses the L1 memories and cache with no wait-states, accessing the external memories may require several wait-states.

The CEVA-XM4 supports up to four giga-bytes of memory space, and has up to nine separate physical interfaces – with up to seven for data memory and one for program memory. This enables the core to simultaneously access both the program and data memories in parallel to Tightly Coupled Extensions (TCE). The MSS provides the core with simultaneous accesses to one instruction fetch stream and two data fetch streams. The PCU accesses the program memory and the LSUs access the data memory.

The MSS contains standard interfaces for connecting the core to external devices and/or peripherals. These interfaces include three AXI master ports for data and one AXI master port for program, four optional AXI slave ports for external masters and hardware accelerator connection to the data memory and an APB3 port. All ports are fully compliant with Advanced Microcontroller Bus Architecture (AMBA).

The CEVA-XM4 supports four giga-bytes of data memory space and four giga-bytes of program memory space. Up to 32 32-bit data addresses can be issued in every cycle to the data memory, as well as a single 32-bit program address to the program memory.

Program memory access is always aligned as a single 256-bit line (fetch-line). The fetch-line is fetched only when it is required by the PCU. Instructions within the program memory are not aligned, since the Dispatcher in the PCU is responsible for aligning them.

The CEVA-XM4 DMSS uses Data DMA (DDMA) to transfer data between the local memory and an external memory, without interfering with core execution. In addition, the DMSS implements a special DMA Queue manager (QMAN) mechanism that enables the user to activate the DDMA without core intervention and without having to use real-time software.

The CEVA-XM4 architecture supports sophisticated power management. The CEVA-XM4 MSS incorporates a Power Scaling Unit (PSU), which controls all clock signals in the system and facilitates power shutdown modes. PSU features enable the user to obtain the required application horsepower, while minimizing the power consumption.

AXI ports are fully compliant with Advanced Microcontroller Bus Architecture (AMBA) versions 3 and 4. The MSS also includes an I/O space that uses dedicated instructions and dedicated space configurations and ports. This space is useful for connecting peripherals to the core.

The CEVA-XM4 MSS supports multi-core communication and shared-memory coherence using the following dedicated external and internal synchronization mechanisms:

- Exclusive external memory access synchronization using either an AXI interface or signals from a dedicated interface
- Supports a multi-core messaging interface between cores using a dedicated command and messaging interface
- Internal Data Memory (IDM) access synchronization that uses a snooping mechanism in order to notify the core about any preferred address space accessed by an external master device

2.2.5.1 Memory Subsystem

In silicon devices, stray radiation and other effects can cause the data stored in RAM to be corrupted. The CEVA-XM4 can be configured to detect and correct errors that can occur in program RAM. Redundant parity bits are computed by the Error-correcting Code (ECC) encoding unit and stored in the RAM. When the processor reads data from the RAM, it checks to ensure that the parity data is consistent with the original data. If any data is corrupted, the ECC decoding unit may correct the error.

This mechanism is extensively used in the CEVA-XM4 various RAMs and interfaces to support CEVA-XM4 as an item in safety system, for further information, refer to *CEVA-XM4 safety manual* and to the *CEVA-XM4 MSS* Document.

2.2.6 CEVA-XM4 Hardware Configurations

The CEVA-XM4 supports several hardware configuration options. These configurations enable users to select the hardware configuration most suitable to their system requirements. Table 2-1 describes the supported configurations in the CEVA-XM4.

Table 2-1. Hardware Configuration Options

Hardware Configuration Name	Configuration Options	Remarks
Non-linear Functions Support	0/32 operations	16 operations per VPU
Scalar Floating Point	1/4	One floating point per SPU
Vector Floating Point	0/16	Eight floating points per VPU
MM3101 Compatibility Instructions	Yes/No	Added support for MM3101 Vec-C compatibility
Program TCM Size	None/32/64/128/256KB	
Program Cache	32/64/128KB	
Error Correction Code	Enabled/Disabled	For Program TCM and cache only
Internal Data Memory Size	128/256/512KB	
AXI Slave Ports	1/2/4	
AXIS0/1/2/3	128/ 256 bits	Independent width configuration for each port can be 128 bits and 256 bits
AXI Data Master Ports	1/3	
AXIM WIDTH	128/ 256 bits	All ports are 128-, 256-bits wide (single configuration for all)
Queue Managers	0/8	
Real-time Trace (ETM)	Enabled/Disabled	
ECC BUS	Enabled/Disabled	AMBA bus parity and SECC protection
SPU XTEND	Enabled/Disabled	SPU XTEND included/not-included
VPU XTEND	Enabled/Disabled	VPU XTEND included/not-included
Memory Power Grating	Enabled/Disabled	Use memories with power gating

2.2.7 Emulation

2.2.7.1 On-Chip Emulation

The OCEM supports core emulation through a standard JTAG interface. It provides various emulation features, such as:

- Counted program address breakpoints
- Data address breakpoints
- Data value match breakpoints, combined data value breakpoints and data address breakpoints
- Single step
- Real-time tracing

2.2.7.2 Real-Time Trace

The CEVA-XM4 supports optional Real-time Trace (RTT) capabilities. The RTT hardware is optional and may be extracted during CEVA-XM4 installation. The purpose of the CEVA-XM4 RTT is to facilitate program instruction and data tracing with minimal impact on core performance. A program instruction trace can show the list of all instructions executed by the CEVA-XM4, including whether or not each instruction passed its predicate code. A data trace can show the load and store accesses performed by the core. A data trace can be enabled for addresses, data values or both.

3 Register File

The CEVA-XM4 contains five register files:

- General Register File (GRF), Section 3.1
- Vector Register File (VRF), Section 3.2
- Predicate Register File (PRF), Section 3.3
- Address Register File (ARF), Section 3.4
- System Register File (SRF), Section 3.5

The following sections describe these register files.

3.1 General Register File (GRF)

The CEVA-XM4 contains a GRF consisting of 32 32-bit registers. The registers are referred to as r0 through r31.

The GRF registers can be organized and referred to in the following four ways:

- 32-bit − Integer

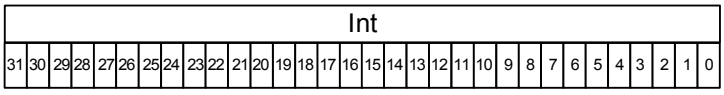


Figure 3-1. GRF – 32-bit Integer

- 16-bit − Short Integer

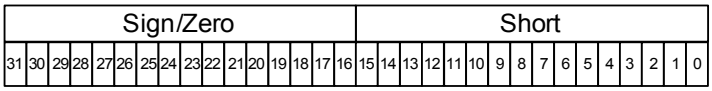


Figure 3-2. GRF – 6-bit Short Integer

- 8-bit − Character

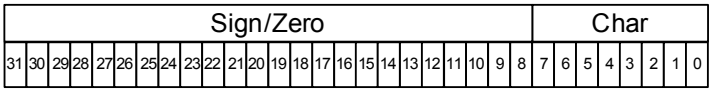


Figure 3-3. GRF – 8-bit Character

- 32-bit − Float

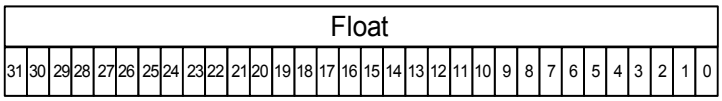


Figure 3-4. GRF – 32-bit Float

3.2 Vector Register File (VRF)

The CEVA-XM4 contains a Vector Register File (VRF) and consisting of 40 256-bit vector registers. The vector registers are used by the VPUs for vector operations and by the LSU for address pointers, loading/storing data from/to data memory and as source and destination registers for the VPUs. Pairs of vector registers can be coupled to form 512-bit accumulators.

Some vector registers are used as general-purpose registers and some as accumulators. There are 32 general-purpose vector registers, marked as v0−v31, and 16 vector registers that can be used for accumulators, marked as v24−v39. Note that v24 to v31 can be used as general-purpose vector registers and as accumulators.

3.2.1 Vector Registers

The CEVA-XM4 includes 32 vector registers marked as vX that are common, enabling any vector register to be used as source or destination operands. Each vector register has a width of 256 bits, and can be accessed as eight integers (32 bits each), 16 short integers (16 bits each), 32 characters (8 bits each) or 8 single-precision floating point numbers (32-bit each).

The vector registers can be organized and referred to in the following ways:

- 8 integers: i0 through i7

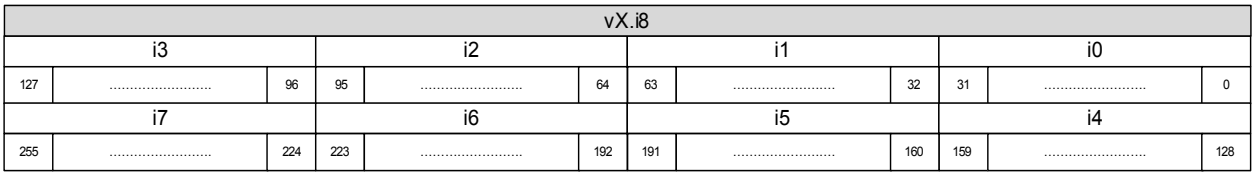


Figure 3-5. Vector Registers – i0 Through i7

- 16 short integers: s0 through s15

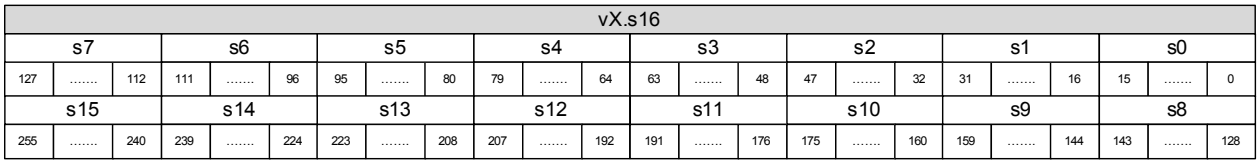


Figure 3-6. Vector Registers – s0 Through s15

- 32 characters: c0 through c31

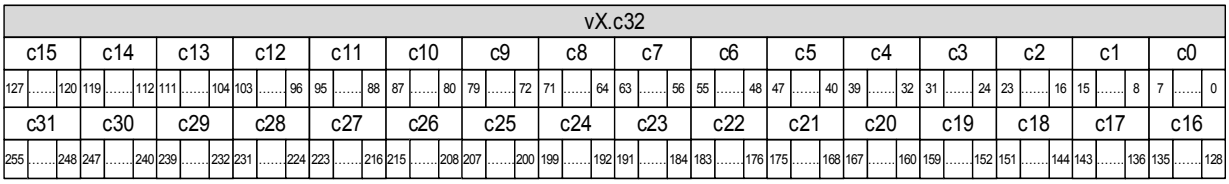


Figure 3-7. Vector Registers – c0 Through c31

- 8 floats: f0 through f7

vX.f8											
f3			f2			f1			f0		
127	.....	96	95	.....	64	63	.....	32	31	.....	0
f7			f6			f5			f4		
255	.....	224	223	.....	192	191	.....	160	159	.....	128

Figure 3-8. Vector Registers – f0 Through f7

3.2.2 Vector Accumulators

A pair of vector registers can be coupled to form a 512-bit accumulator. The pair of registers can be any of the upper 16 vector registers v24−v39.

The vector accumulators can be organized and referred to in the following ways:

- 16 integers: i0 through i15

vW0.i8											
i3			i2			i1			i0		
127	.....	96	95	.....	64	63	.....	32	31	.....	0
i7			i6			i5			i4		
255	.....	224	223	.....	192	191	.....	160	159	.....	128
vW1.i8											
i11			i10			i9			i8		
127	.....	96	95	.....	64	63	.....	32	31	.....	0
i15			i14			i13			i12		
255	.....	224	223	.....	192	191	.....	160	159	.....	128

Figure 3-9. Vector Accumulators – i0 Through i7

- 32 short integers: s0 through s31

vW0.s16																								
s7			s6			s5			s4			s3			s2			s1			s0			
127	.....	112	111	.....	96	95	.....	80	79	.....	64	63	.....	48	47	.....	32	31	.....	16	15	.....	0
s15			s14			s13			s12			s11			s10			s9			s8		
255	.....	240	239	.....	224	223	.....	208	207	.....	192	191	.....	176	175	.....	160	159	.....	144	143	.....	128
vW1.s16																								
s23			s22			s21			s20			s19			s18			s17			s16		
127	.....	112	111	.....	96	95	.....	80	79	.....	64	63	.....	48	47	.....	32	31	.....	16	15	.....	0
s31			s30			s29			s28			s27			s26			s25			s24		
255	.....	240	239	.....	224	223	.....	208	207	.....	192	191	.....	176	175	.....	160	159	.....	144	143	.....		

Figure 3-10. Vector Accumulators – s0 Through s31

3.3 Predicate Register File (PRF)

The Predicate Register File (PRF) consists of predicate registers and vector predicate registers. Predicate registers are used for scalar operations and load/store operations, while vector predicate registers are used for vector load/store and vector operations to conditionally mask individual SIMD operations.

3.3.1 Predicate Registers

The CEVA-XM4 includes 15 predicate registers marked as prX. Each predicate register is a single bit. Predicate registers can be used for conditional execution of instructions. For more information regarding predicate registers and conditional execution, refer to the CEVA-XM4 Volume II Instruction Set document.

3.3.2 Vector Predicate Registers

The CEVA-XM4 includes seven vector predicate registers marked as vprX. Each vector predicate register has a width of 32 bits. Vector predicate registers can be used for conditional execution of vector instructions by masking individual operations in the vector.

The vector predicate can be one of the following types:

- Eight-bits type (b8): Used on vector types with eight elements (i8, s8)
- 16-bits type (b16): Used on vector types with 16 elements (i16, s16, c16)
- 32-bits type (b32): Used on vector types with 32 elements (s32, c32)

When using the b16 type, the upper 16 bits of the predicate are ignored. When using the b8 type, the upper 24 bits are ignored. Casting between different types of vector predicates is possible using dedicated casting instructions.

3.4 Address Register File (ARF)

The Address Register File (ARF) consists of a stack pointer, modulo registers and step registers. For more information, refer to Section 6.5, Pointer-modification Mechanism. Vector step registers are used for post-modification by the LSU. For more information, refer to the CEVA-XM4 Volume II Instruction Set document.

3.5 System Register File (SRF)

The System Register File (SRF) consists of the Program Counter (PC), return registers from subroutines and interrupts (retreg, retregi and retregn) and loop counter registers (lc0 to lc3) that manage loop iterations. These registers are primarily used by the PCU. For more information, refer to the CEVA-XM4 Volume II Instruction Set document.

4 Scalar Processing Unit

The Scalar Processing Unit (SPU) consists of four independent units named SPU0, SPU1, SPU2 and SPU3. The functional units perform independent operations in parallel, each according to the specific instruction within the instruction packet.

Each SPU instruction within the instruction packet can be independently conditional, based on one of 15 predicate registers (refer to Section 4.4, Conditional Execution and Predication for more details).

The SPUs incorporate two execution stages. As a result, various instructions take a different number of cycles to execute. The number of stages an instruction requires for execution determines the cycle penalty before the result can be used by another instruction. The stage count for each instruction is detailed in the CEVA-XM4 Volume II Instruction Set document. Note that the throughput of such operations is still a single result per cycle.

4.1 SPU Instructions

All instructions are identical for all units except for 32-bit multiply and multiply-add and the in and out instructions that are unique to SPU0. The SPUs support the following types of operations:

- Arithmetic operations
- Logic operations
- Bit-manipulation operations
- Miscellaneous operations

The following sections describe these operations.

4.1.1 Arithmetic Operations

Arithmetic operations are supported in all units. The supported operations are:

- Add and subtract operations.
- Multiply and multiply-accumulate operations:
- Minimum and maximum operations.
- Compare operation.
- Absolute and negate operations.

4.1.2 Logic Operations

Standard bit-wise logical 32-bit operations (AND, ANDNOT, NAND, NOR, NOT, OR, XNOR and XOR) are supported.

Logical operations can operate between two registers or between a register and an immediate value. For an immediate operand shorter than 32 bits, the immediate operand is zero-padded.

Test-type instructions are also supported using the tst instruction that enables the setting and clearing of predicate registers according to a specific predicate register.

4.1.3 Bit-manipulation Operations

The supported bit-manipulation operations are:

- Shift of registers up to 32 bits left or right
- Shift operation up to seven bits left, followed by an addition (shiftadd)
- Insert and extract operations
- Count the number of set bits in a register (cntbits)
- Count the number of consecutive bits either from the MSB or from the LSB (ffb)
- Duplicate bits operation (bitdup)

Support for shift operations is provided via a barrel shifter. The barrel shifter can perform both arithmetic and logical shifts, either in full mode or in two-way split mode. It is also used for insert and extract operations. The number of shifted bits is limited to 32 bits in either direction.

The insert operation replaces (inserts) a specified bit/field in a destination operand with a new bit/field placed in the LSB of a source operand. The width and offset of the bit field can be specified by two five-bit immediate values or can be packed in a register with the width at the high part and the offset in the low part.

The extract operation entails extracting a signed or unsigned (based on the sign switch in the instruction) bit/field from a register. The destination operand is one of the registers. The width and the offset of the extracted field can be specified by two five-bit immediate values or packed in a register, with the width at the high part and the offset in the low part.

The bit duplication operation enables duplicating each single bit in a short-type operand into two consecutive bits and writing the result into an integer-type operand.

4.1.4 Miscellaneous Operations

The supported miscellaneous operations are:

- Move Operations: These operations enable clearing a register, copying one register to another, copying registers to another type of register and vice-versa, including copying registers to vector registers and vice-versa.
- Predicate Registers Manipulation: These operations enable copying to/from predicate registers, and performing logical relations between predicate and vector predicate registers.
- Load Bit Field (LBF): This operation assigns values to the various fields in the mode registers.
- Pointers Initialization: This operation initializes all the vector pointers simultaneously.

4.1.5 Add/Sub with Previous Carry

The add and sub instructions enable the inclusion of the carry flag in the operation. This feature is used to support addition and subtraction operations with a precision greater than 32 bits.

In order to use this feature, the operands must be separated into 32-bit quantities.

4.2 Supported Data Types

The Scalar Processing Units (SPUs) support three data types – 8-bit character, 16-bit short integer and 32-bit integer. The data type's size, syntax and range are described in Table 4-1. The supported types of the sources and destinations are specifically described in each instruction at XM4 Arch Spec Volume II document.

Table 4-1. SPU Data Types

Register Type	Size (Bits)	Value Range	Data Bits	Sign Bits	Syntax
Signed Char	8	$-2^7 – (2^7-1)$	0–7	8–31	rX.c
Unsigned Char	8	$0 – (2^8-1)$	0–7	8–31	rX.uc
Signed Short	16	$-2^{15} – (2^{15}-1)$	0–15	16–31	rX.s
Unsigned Short	16	$0 – (2^{16}-1)$	0–15	16–31	rX.us
Signed Integer	32	$-2^{31} – (2^{31}-1)$	0–31	---	rX.i
Unsigned Integer	32	$0 – (2^{32}-1)$	0–31	---	rX.ui
Float	32	$-2^{-149}–2^{128}$	0–30*	31	rX.f

(*) bits [22:0] for mantissa and [30:23] for exponent

Because the size of GRF registers is 32-bits, SPU results are always written as 32 bits. This means that signed char and signed short data types are sign-extended to 32 bits and unsigned char and unsigned short data types are zero-extended to 32 bits.

4.2.1 Source Operands

All SPU operations assume that the source is sign- or zero-extended to 32 bits, according to the source type. Thus, the SPU performs the same operation, regardless of the source types. Because of this, the user must ensure that the input type's value does not exceed the value bits and that the sign bits are correct, based on the input type. Using a type that does not comply with the data type specified by the specific ISA syntax in use and as described in Table 4-1 results in an undefined value in the destination.

4.2.2 Destination Operands

The SPU destination is automatically sign- or zero-extended according to the destination type. This ensures that the register value complies with data-type definitions and that it can be used as a source for the following instruction, without having to make any casting or modification.

4.2.3 Scalar Floating-point Mechanism

The SPUs support a floating-point mechanism, in accordance with the IEEE-754 standard for 32-bit single-precision normal floating-point numbers. This mechanism supports a significantly greater dynamic range than is available with the fixed-point format. Real arithmetic can be coded directly into hardware operations with the floating-point format, which facilitates ease of use and reduced development time. A wide dynamic range is especially important when dealing with extremely large data sets and with data sets for which the range cannot be easily predicted.

The SPU can optionally perform single-precision floating operations on each cycle. The supported operations are:

- Addition of a Floating-point − fpadd
- Subtraction of a Floating-point − fpsub
- Multiplication of a Floating-point − fpmpy
- Comparison of two floating-point numbers − fpcmp
- Conversion of a floating-point number to an integer – fp2int
- Conversion of an integer to a floating-point number – int2fp
- Extraction of a floating-point number to its components – fpextract
- Combination of integer components into a floating-point number – fpcombine

Extracting a floating-point number into its components produces three parts for each floating-point number: the floating-point sign, the floating-point exponent and the floating-point mantissa, where each of these components is a fixed-point number. By extracting the FP number components, the user can perform fixed-point operations such as square-root and division and then combine the components together into a floating-point number.

The floating-point mechanism (FLP) supports the following rounding modes, as described in the IEEE-754 standard:

- Round Towards Even: Returns the floating-point number nearest to the infinitely precise result. If the two nearest floating-point numbers are equally near to the infinitely precise result, it returns the one with an even least significant bit.
- Round Towards Away: Returns the floating-point number nearest to the infinitely precise result. If the two nearest floating-point numbers are equally near to the infinitely precise result, it returns the one with the larger magnitude.
- Round Towards Zero: Returns the floating-point number closest to and not greater than the infinitely precise result.
- Round Towards Positive: Returns the floating-point number closest to and not less than the infinitely precise result.
- Round Towards Negative: Returns the floating-point number closest to and not greater than the infinitely precise result.

Rounding mode usage for scaling floating-point operations is determined by the FLPRND field in the MODG register.

The floating-point ISAs and mechanism are supported only when the appropriate hardware configuration is used.

4.2.4 Type Conversion

Conversion between data types can be performed using the type casting instruction named cast. This instruction should be used when the destination data type has fewer data bits then the source or when converting from signed to unsigned (and vice versa):

- Int or Unsigned Int → Short or Unsigned Short
- Int or Unsigned Int → Char or Unsigned Char
- Short or Unsigned Short → Char or Unsigned Char
- Short → Unsigned Short
- Unsigned Short → Short
- Char → Unsigned Char
- Unsigned Char → Char

If the conversion result cannot be represented by the destination, it is either truncated, taking only the relevant LSB, or saturated to the maximum or minimum possible value.

Saturation is supported only when source and destination are signed.

Example 4-1. Using Cast Instruction to Decrease Type Size

```
mov #0x456789AB, r0.ui // 0x456789AB → r0
…
// cast from unsigned integer at r0 to unsigned short at r1
cast r0.ui, r1.us      // 0x000089AB → r1

// cast from unsigned integer at r0 to signed short at r1
cast r0.ui, r1.s       //  0xFFFF89AB → r1

//cast from unsigned integer at r0 to unsigned short at r1
cast r0.i, r1.us       //  0x000089AB → r1

// cast from unsigned integer at r0 to signed short at r1
cast r0.i, r1.s       //   0xFFFF89AB → r1
```

Example 4-2. Using Cast Instruction to Change Signed or Unsigned Type

```
mov #0xFFFF89AB, r0.s  // 0xFFFF89AB → r0
…
// cast from short at r0 to unsigned short at r1
cast r0.s, r1.us       // 0x000089AB → r1

mov #0x000089AB, r0.us
…
// cast from short at r0 to unsigned short at r1
cast r0.us, r1.s        // 0xFFFF89AB → r1
```

Example 4-3. Using Cast Instruction to Decrease Type Size and Saturate

```
mov #0x456789AB, r0.i  // 0x456789AB → r0
…
// cast from unsigned integer at r0 to signed short at r1
cast {sat} r0.i, r1.s  // 0x00007FFF → r1

mov #0xC56789AB, r0.i  // 0xC56789AB → r0
…
// cast from unsigned integer at r0 to signed short at r1
cast {sat} r0.i, r1.s  // 0xFFFF8000 → r1
```

4.3 Flags and Saturation

4.3.1 Result Saturation

The Scalar Processing Units (SPUs) support automatic saturation of the calculation results in some instructions. Saturation is supported in instructions related to addition and subtraction, as well as shifting and multiplication. The saturation operation enables the SPU to limit the value of the calculated result to the minimum or maximum value that the destination register can represent. The saturation is applied for signed results when the value of the result cannot be represented by the type of the register. Table 4-2 describes saturation values for the supported register types.

Table 4-2. Saturation Value

Register Type	Minimum Value	Maximum Value
Signed Char – rZ.c	-128	127
Signed Short – rZ.s	-32,768	32,767
Signed Short – rZ.i	- 2,147,483,648	2,147,483,647

Saturation can optionally be applied through a dedicated switch in the instruction. For more details, refer to the CEVA-XM4 Volume II Instruction Set document. When saturation is applied, the processor automatically asserts the limit flag (LSPUx and SLF).

4.3.2 Scalar Processing Unit Flags

Each SPU (SPU0−SPU3) has three types of flags: a carry flag, overflow flag and limit flag. The flags for each SPU are set and cleared based on the result of the unit's calculations.

Table 4-3. SPU Flags

Register Type	Carry Flag	Overflow Flag	Limit Flag
SPU0	CSPU0	OSPU0	LSPU0
SPU1	CSPU1	OSPU1	LSPU1
SPU2	CSPU2	OSPU2	LSPU2
SPU3	CSPU3	OSPU3	LSPU3

In addition to the flags for each SPU, there are three flags that are common to all SPUs. These flags are sticky flags, which are set when one of the corresponding SPU flags is set and cleared by writing to the flag or to the MODD register. The sticky flags are:

- Sticky Carry Flag – SCF
- Sticky Overflow Flag – SOF
- Sticky Limit Flag - SLF

For example, when an instruction executed in unit SPU0 causes an overflow, the following occurs:

- The overflow flag (OSPU0) in SPU0 is set.
- The SPU's Sticky Overflow Flag (SOF) is set.

Both flags are calculated regardless of the inputs or result type.

- For unsigned addition and subtraction operations the carry flag is used to detect errors and for signed addition and subtraction the carry flag should be ignored since it doesn't give useful information.
- For signed addition and subtraction operations the overflow flag is used to detect errors and for unsigned addition and subtraction the overflow flag should be ignored since it doesn't give useful information.

4.3.2.1 Carry Flag

The carry flag may be asserted due to addition (add, inc, mac and shiftadd), subtraction (abs, dec, msu, neg, shiftsub and sub) and shift operation (shiftl, shiftr).

4.3.2.2 Addition and Subtraction Operations

In an addition operations, the carry flag is set if the operation of two numbers causes a carry-out of the most significant (leftmost) bits that were added.

Subtraction operations are treated as addition with negative number and the carry flag is calculated in the same method as in addition.

The carry flag can be used for 64-bit operations as an input to further addition and subtraction instructions. For unsigned addition and subtraction operations, the carry flag can be used to detect overflow errors.

4.3.2.2.1 Shift Operation

In a shift operation the carry flag is set with the last bit that was shifted out by the shift operation.

- In shift right operation: CSPUx = A(shift-1)
- In shift left operation: CSPUx = A(msb-shift+1)

The value of msb is defined as 7 in char, 15 in short and 31 in int. Shift is defined as the shift value of the operation. If shift equals 0 carry will be 0.

4.3.2.3 Overflow Flag

The overflow flag may be asserted due to addition (add, inc and mac), subtraction (sub, dec and msu), multiplication (mpy), shift (shiftl, shiftr) and negation operations (neg).

The overflow flag will not be affected by instructions that use automatic saturation switch.

4.3.2.4 Shift Operations

In shift operations the overflow flag is set if the result of the shift cannot be represented by the destination due to the casting of the result to the output type.

4.3.2.5 Multiplication Operations

In multiplication operations the overflow flag is set if the result of the multiplication cannot be represented by the destination due to the operation or due to the casting of the result to the output type.

4.3.2.6 Addition and Subtraction Operations

The Overflow flag is set if the result of the addition or subtraction cannot be represented by the destination due to the operation or due to the casting of the result to the output type.

4.3.2.7 Limit Flag

The limit flag may be asserted due to addition (add and inc), subtraction (sub and dec), multiplication (mpy), shift left (shiftl) and negation operations (neg).

The limit flag is set when the resulting number is replaced with the maximum or minimum possible numbers due to saturation.

The limit flag is never set for instructions that do not use an automatic saturation switch.

The limit flag is used to detect the replacement of results with saturated values.

4.4 Conditional Execution and Predication

The SPU supports a predication mechanism that enables the conditional execution of all instructions. Furthermore, 15 one-bit predicate registers also indicate whether an instruction is executed (predicate register is set) or not (predicate register is cleared). The predication mechanism significantly reduces the number of conditional branch instructions, and therefore shortens execution time.

Dedicated compare and test instructions affect the predicate registers according to various conditions.

Example 4-4. Assigning Predicate Registers and Conditional Execution

Using the compare instruction to assign a predicate register:

```
cmp{lt} r2.i, #3, pr1
```

Assuming r2 = 3, *pr1* is set.

Conditional instruction:

```
add r3.i, r4.i, r5.i, ?pr1
```

Since *pr1* is set, the addition takes place as follows: r3 + r4 → r5.

5 Vector Processing Unit

5.1 Overview

The CEVA-XM4 includes two independent Vector Processing Units (VPUs). The VPUs (VPU0 and VPU1) are responsible for all vector computations. These computations consist of both inter-vector operations (using single-instruction multiple data) and intra-vector operations. Inter-vector instructions can operate on 32 8-bit (char) elements, 16 16-bit (short) or eight 32-bit (integer) elements. Both VPU0 and VPU1 can perform arithmetic operations, logical operations and bit-manipulation operations.

In addition to the above, VPU0 and VPU1 can also perform multiplication operations and specialized operations. Multiplication operations include multiply, multiply-accumulate and filter operations. Specialized operations include sort and sum of absolute differences operations.

The VPUs are fully pipelined, each with a throughput of one instruction per cycle. Both VPU0 and VPU1 operate in the same pipeline stages. However, each instruction, depending on its complexity, takes two or five cycles to execute. The number of stages an instruction requires for execution determines the cycle penalty before the result can be used by another instruction. Due to pipelining, the throughput of all operations is a single cycle. The following figure shows a block diagram of the VPU.

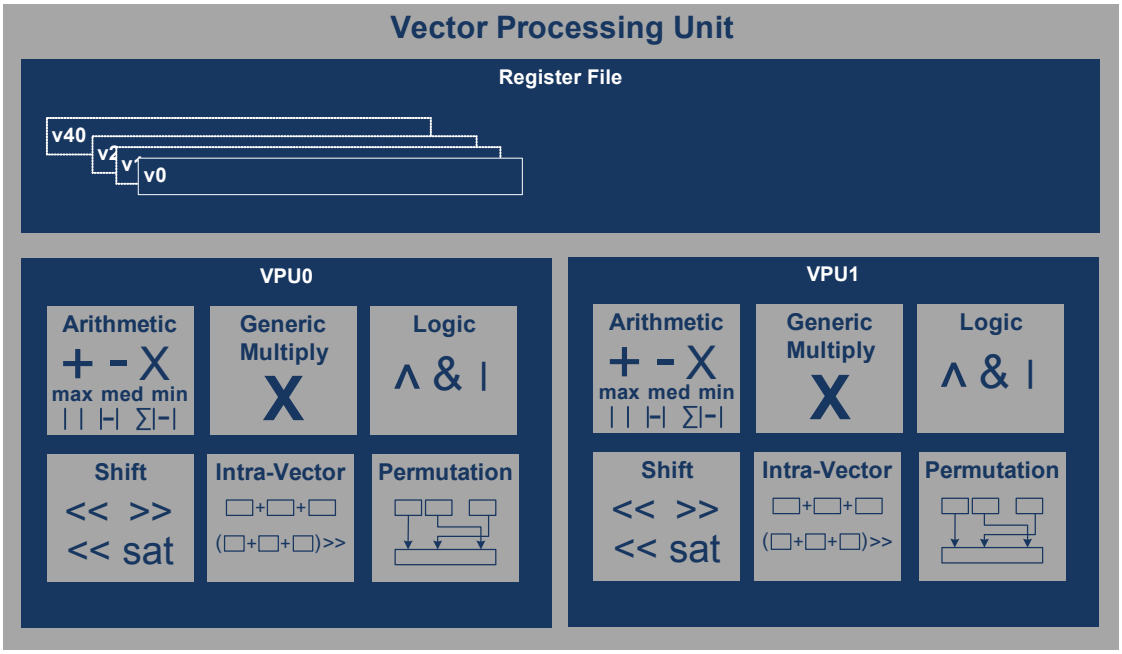


Figure 5-1. VPU Block Diagram

5.2 Registers

5.2.1 Vector Registers

The vector registers are accessed by the VPUs, both as source and destination registers. Each VPU can independently access up to five source vector registers. VPU0 and VPU1 can independently output to two destination vector registers.

The following examples show the capabilities of VPU0 to access vector registers.

Example 5-1. Vector Registers and Accumulator Accesses

VPU0 instruction accessing three source vector registers and one destination vector register:

```
vpu0. vmpyadd vA.c32,  vB.c32,  vC.c32, rD.uc, vZ.c32
```

VPU0 instruction accessing three 256-bit source vector registers and writing the result to a 512-bit accumulator:

```
vpu0. vmpyadd  vA.c32,  vB.c32,  vC.c32,  vaccZ0.s16, vaccZ1.s16
```

5.2.2 Vector Predicate Registers

The programming model utilizes conditional execution of every VPU instruction. Executing the operation on the relative element depends on the corresponding VPR bit. There are seven VPR registers of 32 bits each, where each bit holds a value of true or false. The VPR registers are named vpr0 through vpr6. If no VPR is specified, then the execution is unconditional.

Each element operation within the instruction can be conditional, based on the VPR register bits. If bit N within the VPR is set, the operation of element N is executed. Otherwise, the operation is not executed.

When the number of elements in the instruction is 32, all 32 bits of the VPR are in use and each bit enables the operation of one element. When the number of elements in the instruction is 16, the 16 LSB bits of the VPR are used. When the number of elements in the instruction is eight, the eight LSBs of the VPR are used.

When a general-purpose register is used as a destination in a VPU instruction, then the destination is written in all cases to the VPR, and the operation is performed only on the corresponding elements.

The following example describes how to set a VPR bit and to perform a conditional instruction using it.

Example 5-2. Using Vector Predicate Registers

```
Setting VPR:
vcmp { eq } vA.c32, vB.c32, vprZ.b32

vprZ[0] is set if char 0 of vA equals char 0 of vB. Otherwise, it is cleared.
vprZ[1] is set if char 1 of vA equals char 1of vB. Otherwise, it is cleared
…
vprZ[31] is set if char 31 of vA equals char 31 of vB. Otherwise, it is cleared

Using a VPR:
vadd vA.c32, vB.c32, vZ.c32 ,?vprX.b32

vZ char 0 is the result of vA char 0 + vB char 0 if vprX[0] is set; otherwise, it retains its previous value.
vZ char 1 is the result of vA char 1 + vB char 1 if vprX[1] is set; otherwise, it retains its previous value.
…
vZ char 31 is the result of vA char 31 + vB char 31 if vprX[31] is set; otherwise, it retains its previous value.
```

VPRs are affected by multiple instructions. Some such instructions enable VPR registers to be logically combined in order to create more complex predicate conditions. For more information, refer to the CEVA-XM4 Volume II Instruction Set.

5.2.3 Source and Destination Registers

The VPUs have flexible decoding which enables source registers that are a mix of vector registers, general-purpose registers and immediate values. This flexibility removes the need for move instructions and eliminates registers being wasted on transfers between register files.

The following example shows VPU capabilities for operating on a mix of source registers.

Example 5-3. Mix of Source Registers

```
A multiply of vector and registers:
vmpy {[rnd]}      vA.c32,  rB.c,    rD.uc, vZ.c32
```

The following examples show VPU capabilities to write to different destination registers.

Example 5-4. Write to Different Destination Registers

```
Intra-summation of vector register writing to vector register:
vintrasum vA.s16, vZ.i8

Intra-summation of vector register writing to general-purpose register:
vintrasum vA.i8,  rZ.i
```

5.3 Single Instruction Multiple Data

The basic mechanism of the VPUs is to use a single instruction with multiple data. In all instructions, the data type determines the number of data elements and the size of each element. The vector register contains several data values, where each operation uses different data values according to the data type. The following figure shows the basic usage of multiple operations.

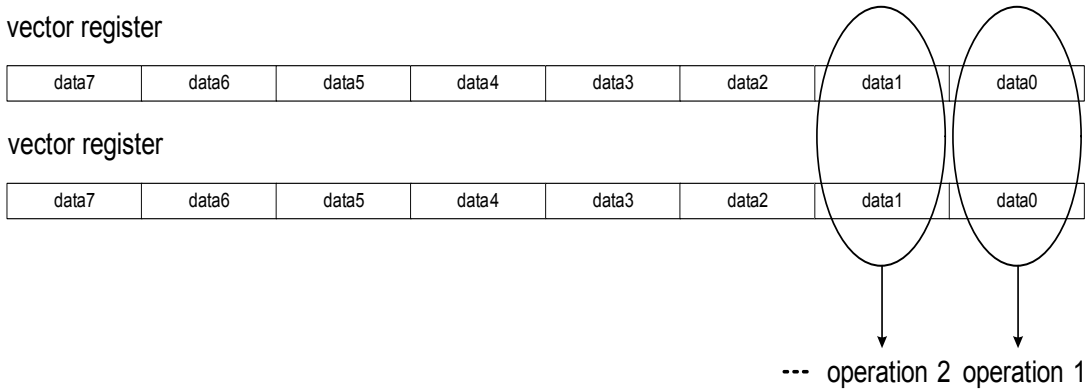


Figure 5-2. Single Instruction Multiple Data

Table 5-1 describes the supported vector data types. The data types determine the number of data elements and the size of each element.

Table 5-1. Vector Data Types

Vector Type	Number of Elements	Size (Bits)	Value Range	Data Bits	Sign Bits	Syntax
Signed Char	32	8	$-2^7 – (2^7-1)$	0-7	8-31	vX.c32
Unsigned Char	32	8	$0 – (2^8-1)$	0-7	8-31	vX.uc32
Signed Short	16	16	$-2^{15} – (2^{15}-1)$	0-15	16-31	vX.s16
Unsigned Short	16	16	$0 – (2^{16}-1)$	0-15	16-31	vX.us16
Signed Integer	8	32	$-2^{31} – (2^{31}-1)$	0-31	---	vX.i8
Unsigned Integer	8	32	$0 – (2^{32}-1)$	0-31	---	vX.ui8
Float	8	32	---	---	---	vX.f8

5.4 Vector Operations

5.4.1 Vector Arithmetic Operations

Both VPU0 and VPU1 support vector arithmetic operations. These include both inter-vector and intra-vector operations. Inter-vector operations consist of basic arithmetic operations, compare operations and complex arithmetic operations. The basic arithmetic operations include addition, subtraction, negation and average. The compare operations include minimum, maximum and compare, where multiple vector predicate registers can be logically combined. The complex arithmetic operations include taking the absolute value of subtraction (vabssub).

Intra-vector operations include summing (vintrasum), where elements can be selectively excluded and/or negated. These instruction enable fast computation of Hadamard transforms to set the DC, while simultaneously detecting blocks without AC values.

5.4.2 Vector Logical Operations

Both VPU0 and VPU1 support vector logical operations. Logical operations include bitwise AND, NAND, NOR, NOT, OR, EXCLUSIVE-OR and EXCLUSIVE-NOR operations.

5.4.3 Vector Bit-manipulation Operations

Both VPU0 and VPU1 support vector bit-manipulation operations. Bit-manipulation operations include vclip, vperm ,vshiftr and vshiftl. The vclip instruction is used to symmetrically clip vector A values between B and C, where the B and C value can be different for each vector element or can be a scalar value.

The vperm instruction allows flexible permutations using a control vector to select char short or integer elements from multiple source vector registers for writing to the destination vector register. For more details about the vperm instruction, refer to the CEVA-XM4 Volume II Instruction Set document.

5.4.4 Vector Multiplication Operations

Both VPU0 and VPU1 support vector multiplication operations. Multiplication operations include multiply and multiply-accumulate operations. The multiply and multiply-accumulate operations also include multiply-add, multiply-subtract and instructions that contain two multiplications, where the products are added together and can be written out or accumulated. Table 5-2 describes the number of operations performed by each multiply operation.

Table 5-2. Vector Multiply Types

Vector Instructions	Source 1 Size	Source 2 Size	Number of Results	Result Size
vmpy, vmac	8 bits	8 bits	32	16 bits
	8 bits	16 bits	16	32 bits
	16 bits	16 bits	16	32 bits
	16 bits	32 bits	8	32 bits
	32 bits	32 bits	8	32 LSBs or 32 MSBs
vmad, vmac3	8 bits	8 bits	32	16 bits
	8 bits	16 bits	16	32 bits
	16 bits	16 bits	16	32 bits
	32 bits	16 bits	8	32 LSBs

Each multiplication operation can either write full results to the vacc accumulator, or can write partial results to a vector register by taking the LSBs of the result after post-shifting.

5.4.5 Vector Floating-point Mechanism

The VPUs supports a vector floating-point mechanism (FLP), according to the IEEE-754 standard for 32-bit single-precision normal floating-point numbers. The FLP mechanism supports a significantly greater dynamic range than is available with the fixed-point format. Real arithmetic can be coded directly into hardware operations with the floating-point format, which is easier to use and reduces development time. A wide dynamic range is especially important when dealing with extremely large data sets and with data sets whose range cannot be easily predicted.

Each VPU can optionally perform eight single-precision floating operations. The supported operations are:

- Addition of a Floating-point − vfpadd
- Subtraction of a Floating-point − vfpsub
- Multiplication of a Floating-point − vfpmpy
- Comparison of two floating-point numbers − vfpcmp
- Conversion of a floating-point number to an integer – vfp2int
- Conversion of an integer to a floating-point number – vint2fp
- Extraction of a floating-point number to its components – vfpextract
- Combination of integer components into a floating-point number – vfpcombine

Extracting vector floating-point numbers into components produces three parts for each floating-point number: the floating-point sign, the floating-point exponent and the floating-point mantissa, where each of these components is a fixed-point number. By extracting the floating-point number components, the user can perform fixed-point operations, such as square-root and division and then combine the components together into a floating-point number.

FLP supports the following rounding modes, as described in the IEEE-754 standard:

- Round Towards Even: Returns the floating-point number nearest to the infinitely precise result. If the two nearest floating-point numbers are equally near to the infinitely precise result, it returns the one with an even least significant bit.
- Round Towards Zero: Returns the floating-point number closest to and not greater than the infinitely precise result.
- Round Towards Positive: Returns the floating-point number closest to and not less than the infinitely precise result.
- Round Towards Negative: Returns the floating-point number closest to and not greater than the infinitely precise result.

Rounding modes usage of vector floating-point operations is determined by the VFLPRND field in the modvFP register.

The floating-point ISAs and mechanism are supported only when the appropriate hardware configuration is used.

5.5 Vector Inversion

The vector inversion operation is performed on an unsigned 16-bit or 32-bit source. The inversion instruction produces two output vectors. The 16-bit mantissa of the inversion operation is written to the first destination vector register (z0). The 16-bit exponent of the result is written to the second destination vector (z1). The vector inversion operation is performed using the vinv instruction. The vector inversion instruction can perform up to 16 inversion operations per VPU.

The vector inversion result is calculated as follows:

$$Inv_{result} = \left(\frac{2^{16-\,exp_{out}}}{X}\right) = mantissa \cdot\ 2^{16-\,exp_{out}}$$

Where X represents an unsigned 16-bit source number U(16,0) or an unsigned 32-bit source number U(32,0).

When using vinv, the resulting mantissa is represented as U(16,16 - expout).

expout is calculated as: -15 - expin and is written into z1 destination vector.

expin (exponent input) is defined as the power of two required, in order to represent X as a normalized number.

For example:

X(16-bit) = 0x00AB.

Normalizing X will result in 0xAB00.2-8.

Therefore expin = -8.

The mantissa is represented as an unsigned number.

The accuracy level achieved is an average precision of -104dB and worst case of -95dB

Note: If the input source is x=0, then the output result is 0xFFFF and the overflow bit is set.

The accuracy is calculated as follows:

$$SQNR_{dB} = 20log_{10}\left(\left|\frac{Result_{float} - Result_{fixed}}{Result_{float}}\right|\right)$$

5.6 Vector Inverse Square Root

The vector inverse square-root operation is performed on an unsigned 16-bit or 32-bit source. The operation produces two output vectors. The 16-bit mantissa result of the inverse square-root operation is written to the first destination vector register (z0). The 16-bit exponent of the result is written to the second destination vector (z1). The vector inverse square-root operation is performed using the vsqrti instruction. The vector inverse square-root unit can perform up to 16 operations per VPU.

The vector inverse square-root operation is performed in fixed point, using the formula:

$$sqrti_{result} = \frac{2^{16-exp_{out}}}{\sqrt{X}} = mantissa \cdot 2^{16-exp_{out}}$$

Where X represents an unsigned 16-bit source number U(16,0) or 32-bit source number U(32,0).

When using vsqrti, the mantissa result is represented as U(16,16 - expout).

expout is calculated as:

For even expin : $-\left\lfloor \frac{exp_{in}}{2} \right\rfloor - 7$

For odd expin : $-\left\lfloor \frac{exp_{in}}{2} \right\rfloor - 8$

expin (exponent input) is defined as the power of two required, in order to represent X as a normalized number. (refer to example in 5.5)

Vector Inversion

The accuracy level achieved is an average precision of -104dB and worst case of -92dB

Note: If the input source is x=0, then the output result is 0xFFFF and, the overflow bit is set.

The accuracy is calculated as follows:

$$SQNR_{dB} = 20log_{10}\left(\left|\frac{Result_{float} - Result_{fixed}}{Result_{float}}\right|\right)$$

5.7 Vector Square-root

The vector square-root operation is performed on an unsigned 16-bit or 32-bit source. The operation produces two output vectors. The 16-bit mantissa result of the square-root operation is written to the first destination vector register (z0). The 16-bit exponent of the result is written to a second destination vector (z1). The vector square-root operation is performed using the vsqrt instruction. The vector square-root unit can perform up to 16 operations per VPU.

The vector square-root operation is performed in fixed point using the following formula:

$$Sqrt_{result} = \sqrt{X} \cdot 2^{exp_{out}-8} = mantissa \cdot 2^{exp_{out}-8}$$

Where X represents an unsigned 16-bit source number U(16,0) or 32-bit source number U(32,0).

When using vsqrt, the mantissa result is represented as U(16,expout -8).

It is calculated as:

For even expin : $\left\lfloor \frac{exp_{in}}{2} \right\rfloor$

For odd expin : $\left\lfloor \frac{exp_{in}}{2} \right\rfloor + 1$

expin (exponent input) is defined as the power of two required, in order to represent X as a normalized number (refer to example in Vector Inversion).

The accuracy level achieved is an average precision of -105dB and worst case of -92dB

The accuracy is calculated as follows:

$$SQNR_{dB} = 20log_{10}\left(\left|\frac{Result_{float} - Result_{fixed}}{Result_{float}}\right|\right)$$

5.8 Flags and Saturation

5.8.1 Result Saturation

The VPUs support automatic saturation or the calculation results in some instructions. Saturation is supported in instructions related to type casting. Type casting enables the VPU to limit the value of the destination result to the minimum or maximum value. The saturation is applied when the value of the result cannot be represented by the type of the register. Table 5-3 describes the saturation values for the supported register types.

Table 5-3. VPU Saturation Values

Register Type	Minimum Value	Maximum Value
Signed Char – vZ.c32	-128	127
Unsigned Char – vZ.uc32	0	255
Signed Short – vZ.s16	-32,768	32,767
Unsigned Short – vZ.us16	0	65,535

Saturation can optionally be applied using a dedicated switch in the instruction. For more details, refer to the CEVA-XM4 Volume II Instruction Set document. When saturation is applied, the processor automatically asserts the limit flag (VLFx and SVLF).

5.8.2 Vector Processing Unit Flags

Each VPU (VPU0, VPU1) has a vector limit flags (VLFx). This flag is set and cleared based on the result of the unit's calculations.

The vector limit flags VLF0 and VLF1 consist of 32 bits each. Each corresponding bit represents the result of the corresponding operation in the result vector. For example, a saturation in vector operation i in VPU0 asserts bit i in VLF0.

In addition to VLF0 and VLF1, there is additional sticky limit flag (VSLF) common to both VPUs. This flag is sticky flags, which set when one of the corresponding VPU flags is set and cleared only by writing to the flag or to the modD register.

5.9 Vector Sliding-window Operations

Sliding-window operations enable efficient data reuse. This leads to reduced memory accesses and lower power consumption.

The same vectors are accessed multiple times, thereby reducing resource usage.

Kernels that contain overlapping input sources can utilize the sliding-window instruction to increase performance. Sliding-window instruction flexibility enables the user to implement any required 2D filter dimension.

Some relevant algorithms that make use of the sliding window include Harris Corner Detector, Bi-lateral filter, 2D correlation, 2D convolution, Gaussian Filter, KLT feature tracker, Nagao Matsuyama filter, algorithms that require the sum of absolute differences and Sobel Filter.

Sliding-window sources include two data vectors containing the elements to be processed and a third input vector containing filter coefficients.

The sliding-window operation is performed in three stages, as follows:

1. Up to four data elements are selected from two data input vectors. These are commonly selected in consecutive order (an advanced user may want to define elements out of order).
2. An arithmetic operation is performed on the data elements. This may be multiplication, absolute difference or subtraction between the selected data elements and a third coefficient vector source.
3. The resulting products/differences are added together (up to four results) and accumulated with the destination vector.

The atomic operation, as described above, is performed multiple times in parallel (producing eight, 16 or 32 results). The next sliding-window operation is offset by one data element from the previous operation.

5.9.1 Sliding-window Operation

By default, the sliding-window operation considers the two data input vectors as one concatenated source, thus enabling the instruction to slide from one vector to the next seamlessly.

Sliding-window multiplication operations are possible in the following instructions:

vswmac3/vswmad and vswmac5/vswmpy5

Example 5-5. Sliding-window Implementation

The following example describes the use of the sliding window to implement a two-dimension filter of 4X4 coefficients that is applied on an image.

Both data and coefficients throughout the example are shorts (16-bits wide).
The destination accumulates eight integer results (32-bits wide).
Image dimensions are 32X32 pixels.

This filter is implemented using the *vswmac5* instruction.

VSWMAC5: Vector Sliding Window MAC (multiply accumulate) 5 (four products and the destination are added together)

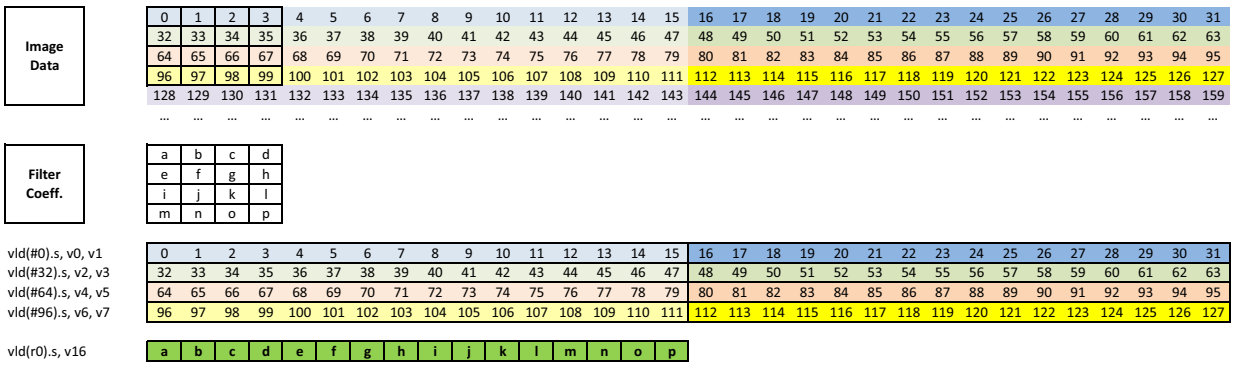
Assembly syntax:

```
vswmac5   vA.[u]s16, vB.[u]s16, vC.[u]s16, rD.ui, vaccW.[u]i8
```

Input vectors vA and vB contain the pixel data (16 short values each) and are regarded as one concatenated source.
Input vector vC contains the filter coefficients (16 short values).
rD[13:8] indicates the starting offset for {vA,vB}.
rD[20:16] indicates the starting offset for vC.
Output accumulator vaccW contains the MAC results (eight integer values).

Filter implementation:

The first four rows of the image are loaded into vectors v0-v7.
Filter coefficients are loaded into v16.



Example 5-5. Sliding-window Implementation (Continued)

Each *vswmac5* instruction processes four coefficients; therefore, four instructions are required to complete a 4X4 filter.

```
1. vswmac5  v0.s16, v1.s16, v16.s16, r0.ui, vacc0.i8   ;r0[20:16] =0
2. vswmac5  v2.s16, v3.s16, v16.s16, r1.ui, vacc0.i8   ;r1[20:16] =4
3. vswmac5  v4.s16, v5.s16, v16.s16, r2.ui, vacc0.i8   ;r2[20:16] =8
4. vswmac5  v6.s16, v7.s16, v16.s16, r3.ui, vacc0.i8   ;r3[20:16] =12
```

1

Inputs									
Coeff.		Data vectors							
v16		v0,v1							
a	*	0	1	2	3	4	5	6	7
		+	+	+	+	+	+	+	+
b	*	1	2	3	4	5	6	7	8
		+	+	+	+	+	+	+	+
c	*	2	3	4	5	6	7	8	9
		+	+	+	+	+	+	+	+
d	*	3	4	5	6	7	8	9	10
Output		vacc0							

vswmac5 v0.s16, v1.s16, #0, v16.s16, #0, vacc0.i8

2

Inputs									
Coeff.		Data vectors							
v16		v2,v3							
e	*	32	33	34	35	36	37	38	39
		+	+	+	+	+	+	+	+
f	*	33	34	35	36	37	38	39	40
		+	+	+	+	+	+	+	+
g	*	34	35	36	37	38	39	40	41
		+	+	+	+	+	+	+	+
h	*	35	36	37	38	39	40	41	42
Output		vacc0							

vswmac5 v2.s16, v3.s16, #0, v16.s16, #4, vacc0.i8

3

Inputs									
Coeff.		Data vectors							
v16		v4,v5							
i	*	64	65	66	67	68	69	70	71
		+	+	+	+	+	+	+	+
j	*	65	66	67	68	69	70	71	72
		+	+	+	+	+	+	+	+
k	*	66	67	68	69	70	71	72	73
		+	+	+	+	+	+	+	+
l	*	67	68	69	70	71	72	73	74
Output		vacc0							

vswmac5 v4.s16, v5.s16, #0, v16.s16, #8, vacc0.i8

4

Inputs									
Coeff.		Data vectors							
v16		v6,v7							
m	*	96	97	98	99	100	101	102	103
		+	+	+	+	+	+	+	+
n	*	97	98	99	100	101	102	103	104
		+	+	+	+	+	+	+	+
o	*	98	99	100	101	102	103	104	105
		+	+	+	+	+	+	+	+
p	*	99	100	101	102	103	104	105	106
Output		vacc0							

vswmac5 v6.s16, v7.s16, #0, v16.s16, #12, vacc0.i8

Example 5-5. Sliding-window Implementation (Continued)

The next eight filter results are written to a different destination: *vacc1*.

Note that since the vA offset is now eight, data is read crossing between source vectors.

```
5. vswmac5  v0.s16, v1.s16, v16.s16, r0.ui, vacc0.i8 ;r0[20:16]=0 r0[13:8]=8
6. vswmac5  v2.s16, v3.s16, v16.s16, r1.ui, vacc0.i8 ;r1[20:16]=4 r1[13:8]=8
7. vswmac5  v4.s16, v5.s16, v16.s16, r2.ui, vacc0.i8 ;r2[20:16]=8 r2[13:8]=8
8. vswmac5  v6.s16, v7.s16, v16.s16, r3.ui, vacc0.i8 ;r3[20:16]=12 r3[13:8]=8
```

5

Inputs									
Coeff.		Data vectors							
v16		v0,v1							
a	*	8	9	10	11	12	13	14	15
		+	+	+	+	+	+	+	+
b	*	9	10	11	12	13	14	15	16
		+	+	+	+	+	+	+	+
c	*	10	11	12	13	14	15	16	17
		+	+	+	+	+	+	+	+
d	*	11	12	13	14	15	16	17	18
Output		vacc0							

vswmac5 v0.s16, v1.s16, #8, v16.s16, #0, vacc0.i8

6

Inputs									
Coeff.		Data vectors							
v16		v2,v3							
e	*	40	41	42	43	44	45	46	47
		+	+	+	+	+	+	+	+
f	*	41	42	43	44	45	46	47	48
		+	+	+	+	+	+	+	+
g	*	42	43	44	45	46	47	48	49
		+	+	+	+	+	+	+	+
h	*	43	44	45	46	47	48	49	50
Output		vacc0							

vswmac5 v2.s16, v3.s16, #8, v16.s16, #4, vacc0.i8

7

Inputs									
Coeff.		Data vectors							
v16		v4,v5							
i	*	72	73	74	75	76	77	78	79
		+	+	+	+	+	+	+	+
j	*	73	74	75	76	77	78	79	80
		+	+	+	+	+	+	+	+
k	*	74	75	76	77	78	79	80	81
		+	+	+	+	+	+	+	+
l	*	75	76	77	78	79	80	81	82
Output		vacc0							

vswmac5 v4.s16, v5.s16, #8, v16.s16, #8, vacc0.i8

8

Inputs									
Coeff.		Data vectors							
v16		v6,v7							
m	*	104	105	106	107	108	109	110	111
		+	+	+	+	+	+	+	+
n	*	105	106	107	108	109	110	111	112
		+	+	+	+	+	+	+	+
o	*	106	107	108	109	110	111	112	113
		+	+	+	+	+	+	+	+
p	*	107	108	109	110	111	112	113	114
Output		vacc0							

vswmac5 v6.s16, v7.s16, #8, v16.s16, #12, vacc0.i8

5.9.2 Vector Sliding-Pattern Operations

Sliding-pattern operations are performed using the vspmac instruction. Using sliding pattern enables maximum processor efficiency. Sliding pattern enables the user to define a four pixel pattern that will be applied to the image. This is opposed to regular sliding window instructions that process four consecutive pixels.

The four pixel pattern can be chosen out of a 4×2 grid.

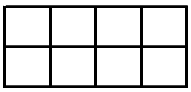


Figure 5-3. Sliding Pattern Grid

Any pattern of up to four pixels fitting into this grid may be configured by the user. Example patterns may be:

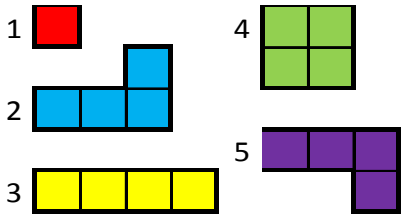


Figure 5-4. Pattern Examples

These patterns can be used for example to efficiently define coefficients for a 5×5 filter. The example below defines a 5×5 filter using 7 instructions (patterns). The sliding pattern scans the filter shape in a snake like fashion therefore utilizes almost all available processor resources (multipliers).

The same filter would take 10 instructions using vswmac5 instructions (~43% more instructions per filter). The conventional approach involves scanning from left to right. Each time the filter edge is reached the remaining coefficients are discarded, wasting processor resources.

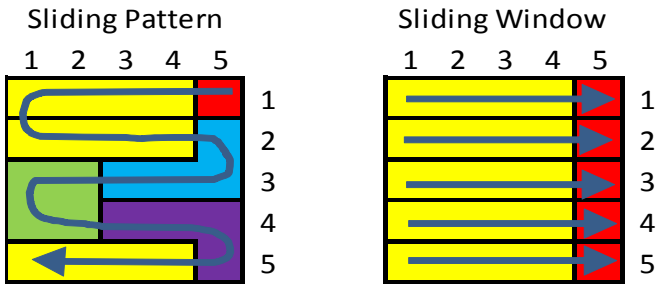


Figure 5-5. Sliding Pattern Efficiency

vspmac vA.[u]s16, vB.[u]s16, vC.[u]s16, rD.ui, vaccW.[u]i8

5.9.2.1 Sliding Pattern Syntax

The sliding pattern instruction contains a configuration register (rD) that describes the pattern shape and offset.

Bits rD[23:16] define the pattern shape. The first four bits [19:16] refer to input vector vA, the next four bits [23:20] refer to vector vB.

For example: In order to configure pattern below, bits 16, 17, 18 and 22 need to be set (rD[23:16] = 0x27).

16	17	18	19
20	21	22	23

Figure 5-6. Pattern Configuration

Bits rD[7:0] define the start offset of the pattern.

5.9.3 Sparse Filters

Sparse filters are defined as filters that contain some coefficients that are equal to zero. In this case instead of multiplying the input data by zero the sliding pattern instruction may skip the zero coefficients and improve the processor efficiency even further.

CNN (Convolutional Neural Networks) algorithms and their derivatives make frequent use of sparse filters. 2D Sparse convolution in different sizes from 5×5 to 15×15 constitutes more than 90% of the cycles for these algorithms.

The example describes a sparse filter with ~45% non-zero coefficients. In this case the vspmac will achieve 92% multiplier utilization completing the filter in 3 instructions, as opposed to vswmac5 implementation that will achieve 55% multiplier utilization and complete the filter in 5 instructions due to multiplications by zero.

For this filter vspmac is 60% faster than the conventional sliding window instruction.

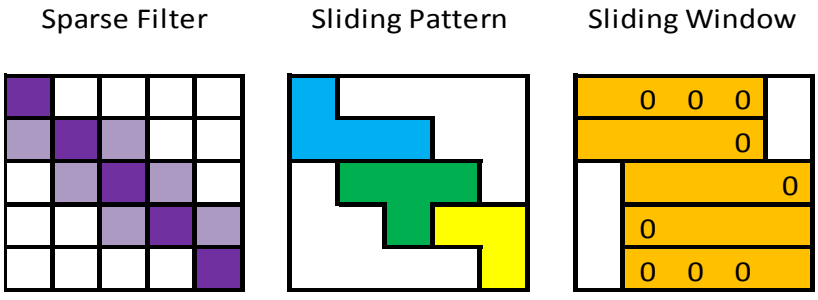


Figure 5-7. Sparse Filter Implementation

5.9.4 Vector Multi-scalar Sliding-window Operations

Multi-scalar sliding-window operations are performed using the following instructions:

vmswmac3 and *vmswmad*

Multi-scalar instructions can work on eight separate image patches in parallel. This is useful, for example, when implementing the KLT feature tracker, where features are extracted from several different locations in the image.

Each input data vector is divided into eight 32-bit sections. Section 1 in the second data vector extends section 1 in the first data vector.

The sliding operation is done across matching sections of the first and second data vectors, as illustrated below:

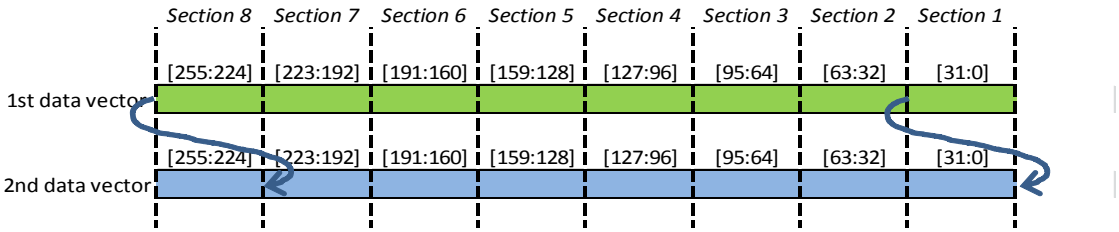


Figure 5-8. Multi-scalar Sliding Window

Each section has dedicated values in the coefficient vector.

5.9.4.1 Linear Approximation Using Vector Multi-scalar Sliding Windows

The following example demonstrates the use of the multi-scalar sliding window to implement a single-dimension linear interpolation of eight image patches.

Linear interpolation is used to approximate a sub-pixel value (C) between two existing known pixels (A and B). Each pixel pair is multiplied by a pair of coefficients (α and 1-α) that describe the distance of the sub-pixel point from each original pixel.

The value of C can be described by the following equation:

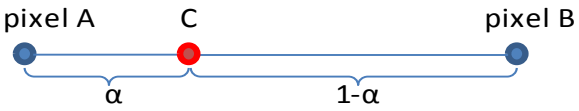


Figure 5-9. Linear Approximation

Both data and coefficients throughout the example are chars (eight-bits wide).
The destination accumulates 32 short results (16-bits wide).
Image dimensions are eight patches of 8×8 pixels.

The linear interpolation is implemented using the vmswmac3 instruction.

VMSWMAC3: Vector Multi-scalar Sliding Window MAC (multiply accumulate) 3 (two products and the destination are added together)

Assembly syntax:

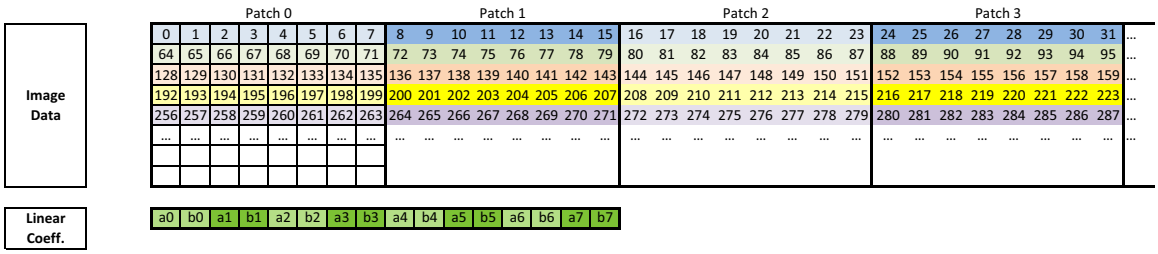
vmswmac3 vA.[u]c32, vB.[u]c32, vC.[u]c32, #rD.ui, vaccW0.[u]s16, vaccW1.[u]s16

Input vectors vA and vB contain the pixel data (32 char values each) and are regarded as eight concatenated sources, as described in the following example.
Input vector vD contains the linear interpolation coefficients (32 char values).
rD[13:8] indicates the starting offset for {vA,vB}.

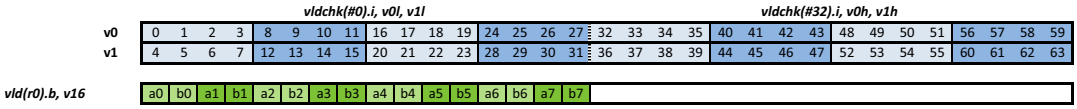
rD[20:16] indicates the starting offset for vC.
Output accumulators vaccW0 and vaccW1 contain the MAC results (32 short values).

Example 5-6. Multi-scalar Sliding-window Implementation

1. A liner interpolation implementation memory map shows eight 8×8 image patches:



The first row of the image patches is loaded into vectors v0 and v1 using load checkered. (load checkered loads even integers into one destination vector and odd integers into a second destination vector. For more information regarding this instruction, refer to XM4 in the *CEVA-XM4 Volume II Instruction Set* document.
Filter coefficients are loaded into v16.



Each vmswmac3 instruction processes four offsets; therefore, two instructions are required to complete an 8×1 linear interpolation.

```
vmswmac3  v0.c32, v1.c32, v16.c32, r0.ui, vacc0.s16, vacc1.s16
;r0[20:16]=0 r0[13:8]=0
```

vmswmac3 v0.c32, v1.c32, #0, v16.c32, #0, vacc0.s16, vacc1.s16

Inputs																						
Coeff.	**Data vectors**																					
v16	**v0,v1**																					
a0	*	0	1	2	3	a1	*	8	9	10	11	a2	*	16	17	18	19	a3	*	24	25	26	27
		+	+	+	+			+	+	+	+			+	+	+	+			+	+	+	+
b0	*	1	2	3	4	b1	*	9	10	11	12	b2	*	17	18	19	20	b3	*	25	26	27	28
Output	**vacc0**																					
a4	*	32	33	34	35	a5	*	40	41	42	43	a6	*	48	49	50	51	a7	*	56	57	58	59
		+	+	+	+			+	+	+	+			+	+	+	+			+	+	+	+
b4	*	33	34	35	36	b5	*	41	42	43	44	b6	*	49	50	51	52	b7	*	57	58	59	60
Output	**vacc1**																					

```
vmswmac3  v0.c32, v1.c32, v16.c32, r0.ui, vacc2.s16, vacc3.s16
;r0[20:16]=4 r0[13:8]=0
```

vmswmac3 v0.c32, v1.c32, #4, v16.c32, #4, vacc2.s16, vacc3.s16

Inputs																						
Coeff.	**Data vectors**																					
v16	**v0,v1**																					
a0	*	4	5	6	7	a1	*	12	13	14	15	a2	*	20	21	22	23	a3	*	28	29	30	31
		+	+	+	+			+	+	+	+			+	+	+	+			+	+	+	+
b0	*	5	6	7	X	b1	*	13	14	15	X	b2	*	21	22	23	X	b3	*	29	30	31	X
Output	**vacc2**																					
a4	*	36	37	38	39	a5	*	44	45	46	47	a6	*	52	53	54	55	a7	*	60	61	62	63
		+	+	+	+			+	+	+	+			+	+	+	+			+	+	+	+
b4	*	37	38	39	X	b5	*	45	46	47	X	b6	*	53	54	55	X	b7	*	61	62	63	X
Output	**vacc3**																					

5.9.5 Vector Sliding Window Sum of Absolute Differences Operations

The Sum of Absolute Differences (SAD) operation is used to find the correlation between an image and a section of a different reference image.

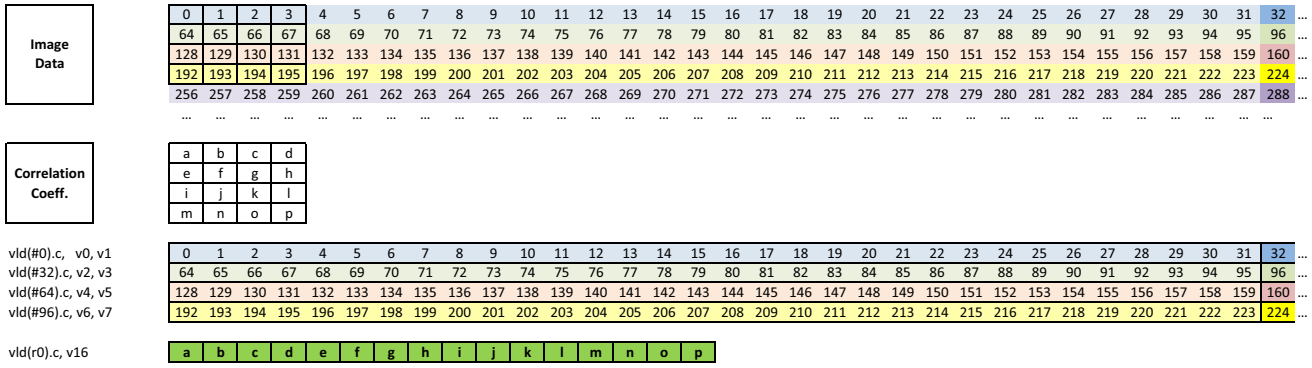
Example 5-7. Sliding Window Correlation

The following example describes the use of the sliding window to implement a two-dimension correlation of 4×4 coefficients that is applied on an image.
Both data and coefficients throughout the example are chars (8-bits wide).
The destination accumulates 16 integer results (32-bits wide).
Image dimensions are 64×64 pixels.
This filter is implemented using the vswsad instruction.
VSWSAD: Vector Sliding Window Sum of Absolute Differences (four absolute differences and the destination are added together)

Assembly syntax:

```
vswsad vA.[u]c32, vB.[u]c32, vC.[u]c32, rD.ui vaccW0.[u]i8, vaccW1.[u]i8
```

Input vectors vA and vB contain the pixel data (32 char values each) and are regarded as a single concatenated source.
Input vector vD contains the correlation coefficients (32 char values).
rD[13:8] indicates the starting offset for {vA,vB}.
rD[20:16] indicates the starting offset for vD.
rD[7:6] indicates how many absolute differences to summarize (one, two, three or four).
Output accumulators vaccW0 and vaccW1 contain the SAD results (16 integer values).
Correlation implementation:
The first four rows of the image are loaded into vectors v0−v7.
Filter coefficients are loaded into v16.



Example 5-7. Sliding Window Correlation (Continued)

Each vswsad instruction processes four coefficients; therefore, four instructions are required to complete a 4×4 filter for 16 locations.

```
vswsad  v0.c32, v1.c32, v16.c32, r0.ui, vacc0.i8 , vacc1.i8
;r0[20:16]=0 r0[13:8]=0 r0[7:6]=3
vswsad  v2.c32, v3.c32, v16.c32, r1.ui, vacc0.i8 , vacc1.i8
;r1[20:16]=0 r0[13:8]=4 r0[7:6]=3
vswsad  v4.c32, v5.c32, v16.c32, r2.ui, vacc0.i8 , vacc1.i8
;r2[20:16]=0 r0[13:8]=8 r0[7:6]=3
vswsad  v6.c32, v7.c32, v16.c32, r3.ui, vacc0.i8 , vacc1.i8 ;r2[20:16]=0 r0[13:8]=12 r0[7:6]=3
```

1

vswsad v0.c32, v1.c32, #0, v16.c32, #0, #0xf, vacc0.i8 , vacc1.i8

Inputs																
Coeff.	Data vectors															
v16	v0,v1															
a |sub|	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
b |sub|	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
c |sub|	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
d |sub|	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Output	vacc0								vacc1							

2

vswsad v2.c32, v3.c32, #0, v16.c32, #4, #0xf, vacc0.i8 , vacc1.i8

Inputs																
Coeff.	Data vectors															
v16	v2,v3															
e |sub|	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
f |sub|	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
g |sub|	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
h |sub|	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82
Output	vacc0								vacc1							

3

vswsad v4.c32, v5.c32, #0, v16.c32, #8, #0xf, vacc0.i8 , vacc1.i8

Inputs																
Coeff.	Data vectors															
v16	v4,v5															
i |sub|	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
j |sub|	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
k |sub|	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
l |sub|	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146
Output	vacc0								vacc1							

4

vswsad v6.c32, v7.c32, #0, v16.c32, #12, #0xf, vacc0.i8 , vacc1.i8

Inputs																
Coeff.	Data vectors															
v16	v6,v7															
m |sub|	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
n |sub|	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
o |sub|	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
p |sub|	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210
Output	vacc0								vacc1							

5.9.6 Vector Sliding Window Subtraction Operations

Sliding-window subtraction operations are possible in the following instructions:

vswsubcmp and *vswsubsat*

5.9.6.1 VSWSUBCMP

Like the vswsad instruction, the vswsubcmp instruction performs a sum of absolute differences.

The difference is that in this case, the sum is conditional. vswsubcmp uses a threshold parameter. Each difference is compared to this threshold. If the difference exceeds the threshold, it is discarded from the sum operation.

Another output from this instruction is a counter value that indicates the number of differences that were summed.

VSWSUBCMP: Vector Sliding Window Subtract and Compare (four absolute differences and the destination are added together conditionally).

Assembly syntax:

vswsubcmp vA.t, vB.t, vC.t, rD.ui, vaccW0.t, vaccW1.t

Input vectors vA and vB contain the pixel data and are regarded as one concatenated source.
Input vector vD contains the correlation coefficients.
rD[13:8] indicates the starting offset for {vA,vB}.
rD[7:6] indicates how many absolute differences to summarize (one, two, three or four).
rD[31:24] indicates the threshold for conditional summation. (Threshold = 1 << F).
Output accumulator vaccW0 contains the sum results.
Output accumulator vaccW1 contains a counter recording the number of differences that were summed

5.9.6.2 VSWSUBSAT

The sliding-window subtract and saturate instruction generates differences between a data vector pair and a reference coefficient vector.

The data vector is read in an overlapping fashion, while the coefficient vector remains stationary.

Each difference value is limited according to a threshold defined in the instruction (a power of 2).

The limited differences are not summed together. Instead they are written to the output vector.

VSWSUBSAT: Vector Sliding Window Subtract and Saturate (limited differences are written to the destination).

Assembly syntax:

vswsub {sat}vA.[u]s16, vB.[u]s16, vC.[u]s16, rD.ui, vaccZ1.s16

Input vectors vA and vB contain the pixel data and are regarded as one concatenated source.
Input vector vD contains the reference coefficients.
rD[12:8] indicates the starting offset for {vA,vB}.

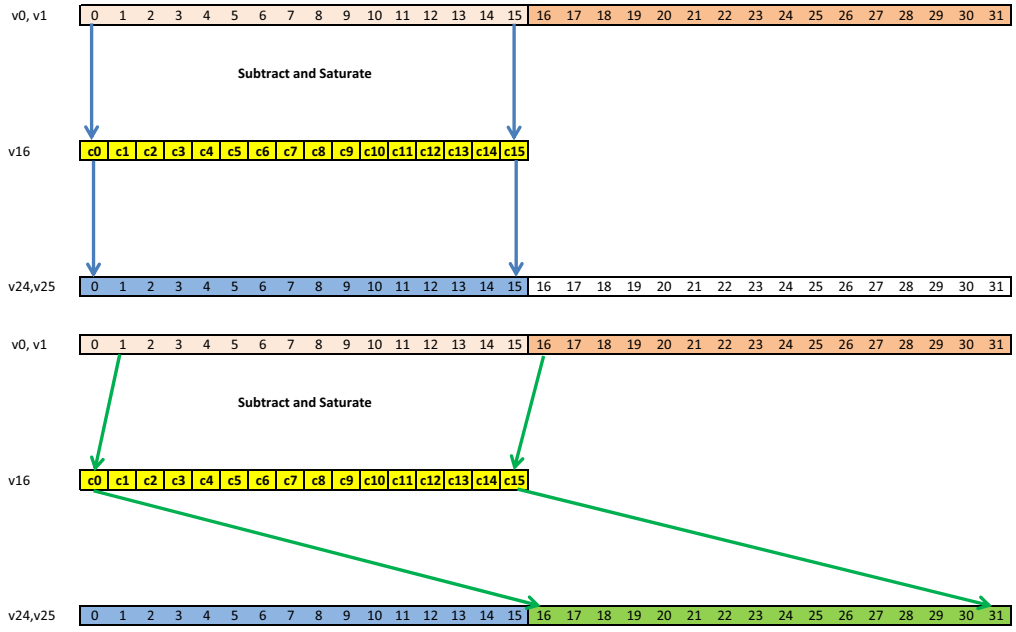
rD[27:24] indicates the threshold for limiting.
Output accumulators vaccZ0 and vaccZ1 contain the limited difference results.

Example 5-8. VSWSUBSAT

The following figure describes the results of one instruction .

Each source is 16-bits wide. Results are also 16-bits wide.

```
Vswsub {sat} v0.s16,  v1.s16, v16.s16,  r0.ui, v24.s16, v25.s16  ;r0[12:8]=0
r0[27:24]=12
```



In this example, a total of 32 differences are written.
Differences are limited to 212 -1 ≤ difference ≤ 212.

5.10 Histogram Operations

The histogram instruction is used to count the number of times each pixel appears in an image. The CEVA-XM4 enhances this capability by enabling a weight to be assigned to each pixel.

VPU0 have the ability to send a base address, a vector of pixel values (bin) and an optional corresponding weight vector to a memory block, where a dedicated mechanism reads the previous value and adds the weight according to the bin per memory bank. This operation can be performed every two cycles per memory block. This means that when using it in consecutive cycles, writing is required to a different memory block every two cycles.

It is the user's responsibility to reset the bin values in the memory former the beginning of the operation.

6 Load and Store Unit

6.1 Overview

The main purpose of the Load and Store Unit (LSU) is to generate the address for the data, the data memory load and store operations, to unpack and pack the loaded and stored data and to write/read it to the internal registers of the VP. For this purpose, the LSU includes two load and store units named LS0 and LS1.

The LS0 and LS1 units support the following main mechanisms:

- Linear address generation of a 32-bit address according to one of the addressing modes. The address points to a byte location in the data memory space.
- Parallel address generation of a 32-bit address according to one of the parallel addressing modes. The address points to a byte location in the data memory space.
- For the SPU, LSU and PCU units, read and write bandwidth is 64 bits, which enables both the LS0 and LS1 to read and write their maximum bandwidth with byte alignment.
- For the VPU unit, read data bandwidth is 256 bits per LSU. This enables both the LS0 and LS1 to read their maximum bandwidth of 512 bits with byte- alignment address generation.
- For the VPU unit, write data bandwidth is limited to 256 bits using only LS0, with byte-alignment address generation.
- Modification of address registers. Two modification types are available: linear modification and modulo modification. Each LSU can modify general registers, the stack pointer or a vector register.

The LS0 and LS1 units are connected to a common register file, referred to as the General Register File (GRF). The GRF contains 32 32-bit registers, general registers, step registers and stack-pointer register.

The LSU includes three mode registers that control the behavior of the post-modification and address-generation mechanisms.

All LSU instructions can be conditional using one of the predicate registers. The SPU contains dedicated instructions that can affect the predicate registers. In addition, vector load and store instructions can be predicated using the vector predicates in order to mask the load and store operations. This enables the LSU to load and store only part of the vector according to the vector predicates.

Figure 6-1 shows a schematic block diagram of the LSU.

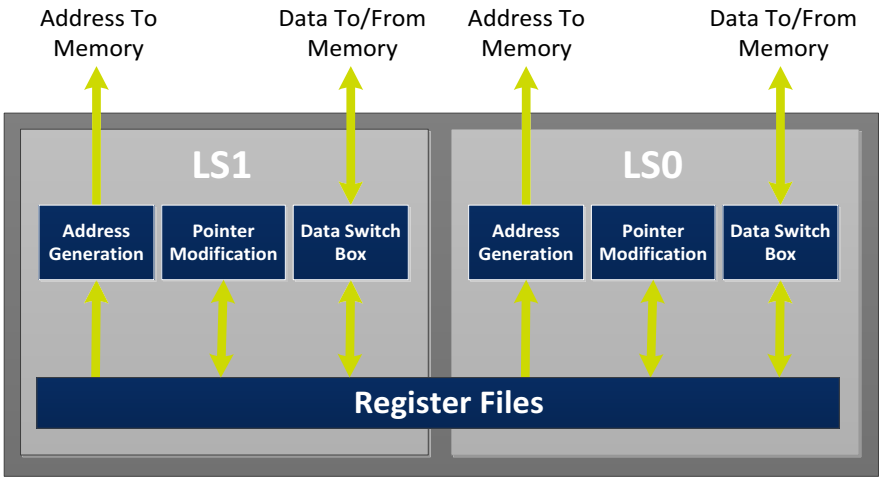


Figure 6-1. LSU Block Diagram

The following sections describe the LSUs and their supported addressing mechanisms.

6.2 Load and Store Units (LS0 and LS1)

The LSU contains two independent load and store units: LS0 and LS1. Each unit supports two main mechanisms:

- Address-generation mechanism
- Pointer-modification mechanism

The address-generation mechanism computes a single 32-bit address according to the addressing mode specified in the instruction or vector (up to 16 addresses when using parallel memory access). These addresses are then supplied to the data memory along with the read/write strobe and the access width. Note that the address points to a byte location. Since each unit can generate a single address or a vector of addresses, a total of two addresses or two address vectors can be simultaneously generated.

The LS0 and LS1 units support a set of instructions for accessing the data memory. Table 6-1 describes these instructions. The third column indicates the address alignment of each instruction. Non-aligned memory access is described in Section 6.4.1, Unaligned Memory Access.

Table 6-1. Memory Access Instructions

Instruction	Description
ld	Loads memory operands from the data memory into the SPU, LSU or PCU registers
st	Stores SPU, LSU or PCU registers or parts of them into the data memory
pop (LS0 only)	Pops an operand from the software stack into the SPU, LSU or PCU registers
push (LS0 only)	Pushes SPU, LSU or PCU registers into the software stack
vld, vldov	Loads memory operands from the data memory into the VPU registers or parts of them
vst (LS0 only)	Stores VPU vectors or parts of them into data memory
vpld	In parallel, loads memory operands from the data memory into the VPU registers or parts of them
vpst	Stores VPU vectors or parts of them into data memory
vpop (LS0 only)	Pops two vectors from the software stack into vectors
vpush (LS0 only)	Pushes vector into the software stack

The LS0 and LS1 units access the memory using a Little Endian method, which means the least significant byte is stored in the lowest address.

The following sections describe the different addressing modes supported in the LS0 and LS1 units, along with details about the supported access types and the pointer-modification mechanism. These sections describe the specifications for both the LS0 and LS1 units.

6.3 Addressing Modes

Each LSU supports the following addressing modes:

- Indirect addressing mode
- Indexed addressing mode
- Direct addressing mode
- Stack addressing mode
- Parallel addressing mode

Because the LS0 and LS1 are independent units, the addressing mode used in each of them can be different. The addressing mode is specified in the instruction using a dedicated syntax. This syntax is described in the relevant section of each addressing mode.

6.3.1 Indirect Addressing Mode

In indirect addressing mode, a GRF register contains the 32-bit address. The indirect addressing mode works in linear mode method, meaning that the address is the content of the pointer. The pointer register can be post-modified.

The example below describes the indirect addressing mode.

Example 6-1. Indirect in Linear Mode

```
mov #0x1004,r4.ui      ; r4 contains the address 0x1004.
nop
nop
ld (r4.ui).s+, r0.i    ; Loads short from address 0x1004 into r0.
```

6.3.2 Indexed Addressing Mode

In indexed addressing mode, the address is the sum of a base register and an offset. The base register can be any of the 32 GRF registers and the stack pointer. The offset can be one of the following:

- A GRF register
- A signed immediate value up to 16 bits

Although the offset is added to the base register, the base register remains unchanged.

The offset contains a number of bytes and is not automatically scaled according to the access type.

When a register specifies the offset, it can be post-modified. For more information about post-modification options, refer to Section 6.5, Pointer-modification Mechanism.

The syntax of the indexed addressing mode is described in Table 6-2.

Table 6-2. Indexed Addressing Mode Syntax

Addressing Mode		Syntax
Indexed	Offset is a register.	(rM + rN)
	Offset is a signed 16-bit immediate value.	(rM + #immN16)

Below are three examples showing the usage of the indexed addressing mode.

Example 6-2. Indexed with Pointer as Offset

```
mov #0x004,r0.ui       ; Initializes the base register r0.
mov #0x100,r3.i        ; Initializes the offset pointer r3.
nop
nop
st r7.i,(r0.i+r3.i).i+ ; An integer is stored at address 0x104.
                       ; r0 is kept unchanged.
                       ; r3 is post-modified and its value after
                       ; the store instruction execution is 0x104.
```

Example 6-3. Indexed with Pointer as Offset and vst Post-modification

```
mov #0x100,r0.i              ; Initializes the base register r0.
mov #0x20,r3.i               ; Initializes the offset register r3.
nop
nop
vst v0.c32,(r0.i+r3.i).c32+  ; 32 characters are stored at address 0x120.
                             ; r0 is kept unchanged.
                             ; The pointer r3 is post-modified and
                             ; its value after the store instruction
                             ; execution is 0x40.
```

Example 6-4. Indexed with Immediate Value as Offset

```
mov #0x010,r1.i;             Initializes the base register r1.
nop
nop
ld (r1.i+#0x300).s, r7       ; Loads a short integer from address 0x310.
                             ; r1 is kept unchanged.
```

6.3.3 Direct Addressing Mode

In direct addressing mode, the entire 32-bit address is explicitly specified in the instruction. Direct addressing mode is specified in the instruction by the syntax [#address] when the address is a 32-bit unsigned immediate value.

The following example illustrates the usage of this addressing mode.

Example 6-5. Direct Addressing

```
st r0.s,r4.s,(#0x1000A234).i2      ; Two words are stored at address
                                   ; 0x1000_A234.
```

6.3.4 Parallel Addressing Mode

The LSU enables parallel access to eight or 16 L1 data memory addresses in the LS0 and LS1. In parallel addressing mode, the address is the sum of a base register and a vector offset. The base register can be any of the 32 GRF registers and the offset can be any of the vector registers. The base register type is 32-bit integer and the offset vector is eight integers (i8), 16 short integers (s16) or 16 chars (c16).

When a register specifies the offset, it can be post-modified. For more information about post-modification options, refer to Section 6.5, Pointer-modification Mechanism.

The syntax of the parallel addressing mode is shown in Table 6-3.

Table 6-3. Parallel Addressing Mode Syntax

Addressing Mode		Syntax
Parallel	Offset is a vector. 16 parallel addresses.	(rM.ui + vN.s16)
	Offset is a vector. 16 parallel addresses.	(rM.ui + vNp.c32)
	Offset is a vector. Eight parallel addresses.	(rM.ui + vN.i8)
	Offset is a register. 16 parallel addresses.	(vM.s16 + rN.ui)

When calculating the parallel address, the value of the register (rN/rM) is treated as a byte address and must be aligned, while the value in the vector (vN/vM) is treated as an element address in relative mode and as bytes in absolute mode. For example, in relative mode, if the element of the memory access is a character, then the value of the vector is taken as is.

Table 6-4. Relative Parallel Address Calculation

Addressing Mode	Syntax
(rM + vN).c	Address[i] = rM + vN[i]
(rM + vN).uc	Address[i] = rM + vN[i]
(rM + vN).c2	Address[i] = rM + vN[i]
(rM + vN).c4	Address[i] = rM + vN[i]
(rM + vN).s	Address[i] = rM + 2 x vN[i]
(rM + vN).us	Address[i] = rM + 2 x vN[i]
(rM + vN).s2	Address[i] = rM + 2 x vN[i]
(rM + vN).i	Address[i] = rM + 4 x vN[i]
(rM + vN).ui	Address[i] = rM + 4 x vN[i]

6.3.4.1 Absolute Parallel Addressing Mode

In absolute parallel addressing mode, a GRF register contains a 32-bit address and the VRF vector register contains eight 32-bit, 16 16-bit or 16 eight-bit addresses. The indexed addressing mode is calculated according to Table 6-5 and then the memory is accessed.

It is the user's responsibility to ensure that there are no memory bank conflicts in the generated addresses.

Table 6-5. Absolute Parallel Address Calculation

Addressing Mode	Syntax
(rM + vN).c	Address[i] = rM + vN[i]
(rM + vN).uc	Address[i] = rM + vN[i]
(rM + vN).c2	Address[i] = rM + vN[i]
(rM + vN).c4	Address[i] = rM + vN[i]
(rM + vN).s	Address[i] = rM + vN[i]
(rM + vN).us	Address[i] = rM + vN[i]
(rM + vN).s2	Address[i] = rM + vN[i]
(rM + vN).i	Address[i] = rM + vN[i]
(rM + vN).ui	Address[i] = rM + vN[i]

When using 16 addresses, the generated addresses must be aligned to the loaded data type. When using eight addresses, no alignment is required.

The example below illustrates the absolute parallel addressing mode.

Example 6-6. Absolute Parallel Addressing Mode

```
Assume r0 = 20                                  ; Initializes the offset pointers r0 and v0.

Assume v0 = {270,208,350,56,118,254,346,292,170,8,70,96,306,46,150,2}

vpld (r0.ui+v0.s16).s, v10.s16                  ; 16 parallel memory reads of short integer.
                                                ; Memory addresses are calculated by adding the
                                                ; base register r0 with the offset vector v0.
                                                ; r0 is treated as bytes and v0 is treated as shorts.
```

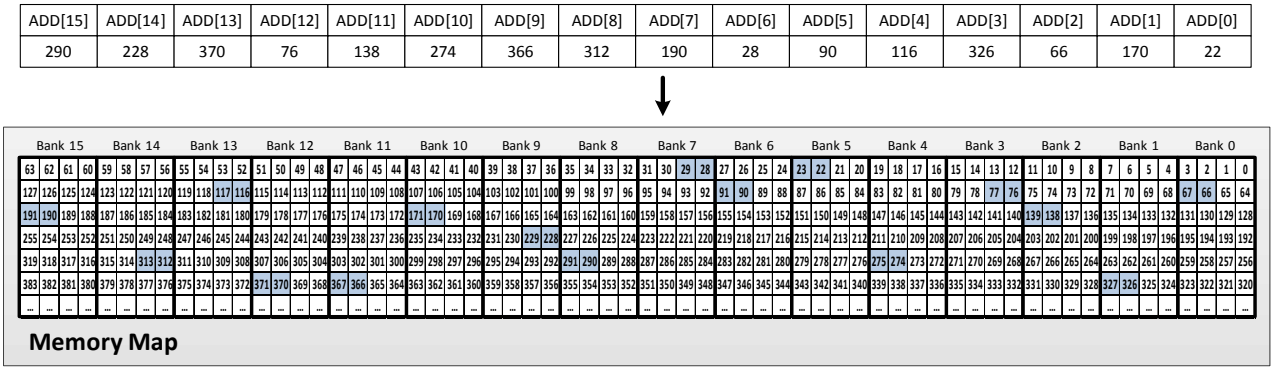


Figure 6-2. Absolute Parallel Addressing Mode

6.3.4.2 Relative Parallel Addressing Mode

In relative parallel addressing mode, a GRF register contains a 32-bit address and the VRF vector register contains 16 16-bit addresses. The indexed addressing mode is calculated according to Table 6-7 and then the memory is accessed. When using this addressing mode, each memory bank is treated as a separate memory space. Therefore, the generated address is relative to the memory bank and not to the entire memory space as in other addressing modes.

The generated addresses must be aligned to the loaded data type. The base address must be aligned to 64 bytes.

The example below illustrates the relative parallel addressing mode.

Example 6-7. Relative Parallel Addressing Mode

```
Assume r4 = 32                                  ; Initializes the offset r4 absolute
                                                ; address in bytes aligned to 64 bytes.
                                                ; Initializes v7 with offsets.

Assume v7 =
{30,24,34,28,32,26,38,20,34,24,36,28,32,30,26,22}
vpld {rel} (r4.ui+v7.s16).uc2, v10h.uc32, v12h.uc32   ; 16 parallel memory reads of
                                                ; two chars.
                                                ; Memory addresses are calculated by
                                                ; adding the base register r4 with
                                                ; the offset vector v7.
                                                ; r4 and v7 are treated as bytes.
                                                ; 16 chars are loaded
                                                ; into the high part of v10.
```

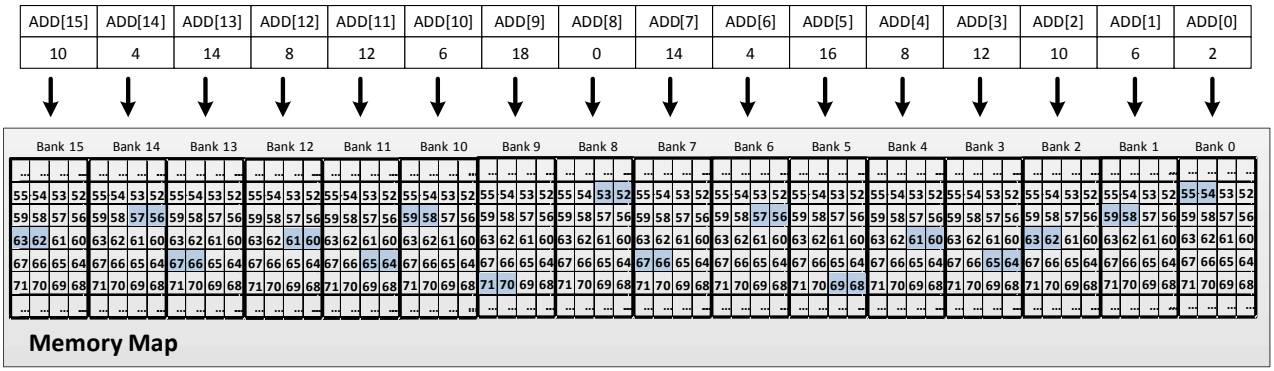


Figure 6-3. Relative Parallel Addressing Mode

6.3.5 Stack Addressing Mode

The GRF contains a 32-bit stack pointer register, referred to as sp. The stack pointer points to the last value pushed into the software stack. This software stack can reside anywhere in the data space. The stack is filled from the high memory address to the low memory address. Therefore, when a new operand is pushed onto the stack, the stack pointer is decremented before the memory write address is generated. In the same manner, when an operand is popped from the stack, the stack pointer is incremented after the memory read address is generated.

The stack pointer is used in pop, push, vpop and vpush instructions. Each of these instructions performs a dedicated operation on the software stack and the stack pointer. In general, the stack pointer is automatically modified based on the access width defined in the instruction. Table 6-6 describes the stack-pointer modification according to the various instructions.

Table 6-6. Stack-pointer Modification

Instruction	Stack-pointer Modification
pop {ch1, u}	sp+1
pop {sh1, u }	sp + 2
pop {in1, u}	sp + 4
pop {ch2, u}	sp+2
pop {sh2, u}	sp + 4
pop {in2, u}	sp + 8
pop {ch4, u}	sp + 4
pop {sh4, u}	sp + 8
pop {in4, u}	sp + 16
pop #uimmA5	sp + #uimmA5*4
pop auxregZ	sp + 32
push {ch1}	sp - 1
push {sh1}	sp - 2
push {in1}	sp - 4
push {ch2}	sp - 2
push {sh2}	sp - 4
push {in2}	sp - 8
push {ch4}	sp - 4
push {sh4}	sp - 8
push {in4}	sp - 16
push #uimmA5	sp - #uimmA5*4
push auxregZ	sp - 32
vpop vectors	sp + 64
vpush vectors	sp - 32

The push/vpush instructions can store several operands in the stack. The first (leftmost) operand in the instruction is the first to be pushed in the stack.

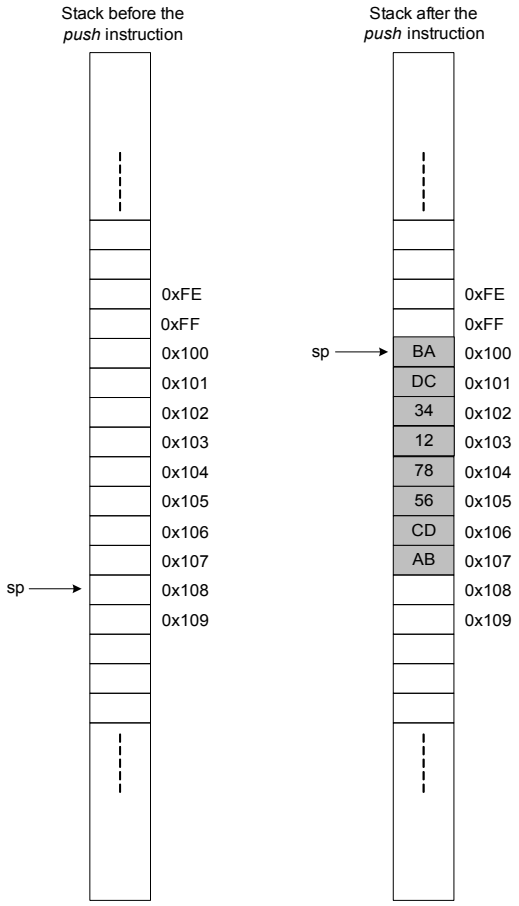
The following examples illustrate this point.

Example 6-8. push Instruction

Assuming that the following instruction is executed:

```
push{u,s4} r7.i, r13.i, r4.i, r30.i
```

Assuming:
r7=0x0000ABCD, r13= 0x00005678,
r4= 0x00001234, r30 = 0x0000DCBA
The next figure shows the stack before and after the push instructions.



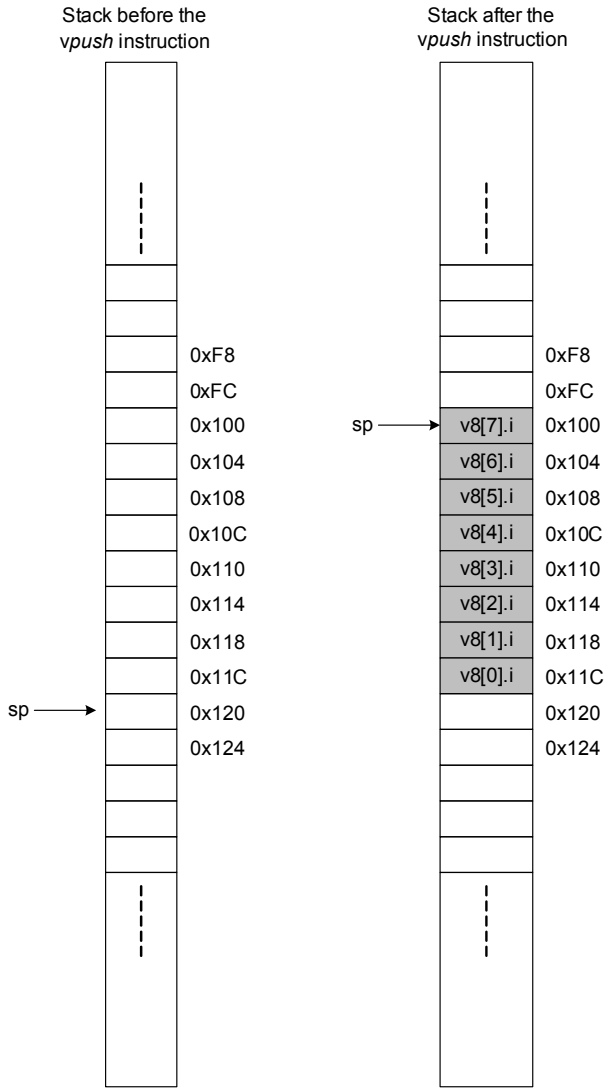
Note: that the operands are stored in the stack in the order they appear in the push instructions: from left to right. Therefore, r30 is pushed last in the stack. The stack pointer is decremented by eight bytes. The data is stored as Little Endian, which means that the least significant byte is stored in the lowest address.

Example 6-9. vpush Instruction

Assuming that the following instruction is executed:

```
vpush v8.i8
```

The figure below shows the stack before and after the vpush instruction.



Note that the operands are stored in the stack in reverse order, meaning first v8[7].i and then v8[6].i and so on. The stack pointer is decremented by 32 bytes. The data is stored as Little Endian, which means that the least significant byte is stored in the lowest address.

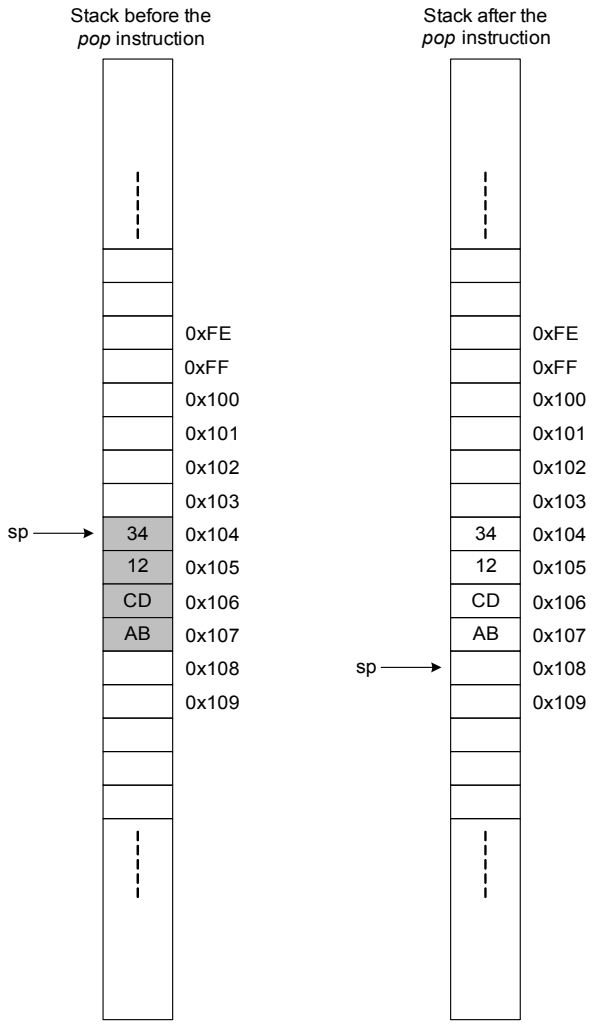
In the same manner, the pop/vpop instructions can load several operands. The first (leftmost) operand in the instruction is the first to be loaded. This rule also applies for parallel pop instructions. The following examples illustrate this point.

Example 6-10. pop Instruction

Assuming the following instruction is being executed:

```
pop{i1} r17.i
```

The figure below shows the stack before and after the pop{i1} instruction.



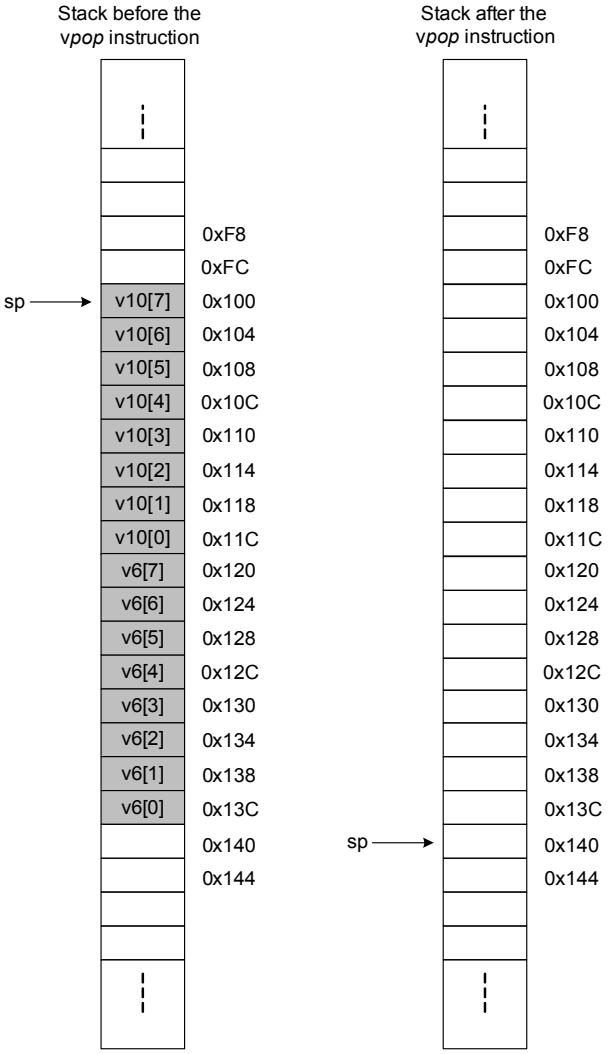
The memory operands are loaded into the r17 register. The stack pointer is incremented by four bytes. The data is popped from the memory as Little Endian, which means that the least significant byte is popped into the lowest byte of r17. After the pop{i1} instruction, the content of r17 is 0xABCD1234.

Example 6-11. vpop Instruction

Assuming that the following instruction is executed:

```
vpop v10.i8, v6.i8
```

The figure below shows the stack before and after the vpop instruction



.

The memory operands are loaded into the registers in reverse order, where the register in the instruction is the last to be loaded, meaning, first v10[7] and then v10[6] and so on. The stack pointer is incremented by 64 bytes. The data is loaded as Little Endian, which means that the lowest address is loaded to the least significant byte.

For more details regarding push/vpush and pop/vpopd instructions, refer to the CEVA-XM4 Volume II Instruction Set document.

6.4 Memory Accesses

Both the LS0 and LS1 units support various types of memory accesses to the data memory. The following sections describe the parameters that characterize each access, explain the requirements for aligned memory access and describe the vector load and store mechanism.

6.4.1 Unaligned Memory Access

Unaligned data memory access is fully supported in hardware without any core stall cycles. The ability to support aligned and unaligned accesses without penalty can have the advantage of fewer instructions leading to increased core performance and reduced code size. A few instructions, such as vpld and vpst, have alignment restrictions that are specified in the relevant ISA.

6.4.2 Access Parameters

Each access is characterized by the following parameters:

- Access Type: The basic types are eight-bit character (c), 16-bit short integer (s) and 32-bit integer (i). An access can also be a multiple of a basic type. For example, two characters (c2).
- The memory access width varies from one character (8-bit) to 64 characters/32 short-integers/16 integers (512-bit).
- Data Type: Signed operand or unsigned operand. This information is described in the syntaxes of the instruction. By default, the memory operands are sign-extended. If the parameter [,u] is specified in the instruction, the memory operands are treated as unsigned operands and they are zero-extended.

6.4.3 LSU Operations

The LSU has the ability to manipulate the loaded and stored data types. In load operations, the LSU can read one data element from memory, change the element type and then write it to the processor register. In store operations, the LSU can read the data element from the processor register, change the element type and then store it to memory.

6.4.4 Scalar Load and Store Operations

Scalar load and store operations are used for loading and storing the GRF, ARF and SRF registers. The scalar load and store operations are supported in both the LS0 and LS1. Each can read or write up to 64 bits from the data memory. In some cases, the two units can be used by a single instruction to access 128 bits using one memory pointer.

The example below illustrates a scalar memory load operation.

Example 6-12. Scalar Load Operation

Moving an immediate value to an address pointer, which is used as a base:

```
mov #0x1000, r7
```

Scalar loading using indexed addressing mode:

```
ld (0x41+r7.ui).us2,r10,r5
```

Assuming memory contents at location 0x1040 are:

Two shorts are loaded from memory, zero-extended and written to registers r10 and r5.

Address	0x1047	0x1046	0x1045	0x1044	0x1043	0x1042	0x1041	0x1040
Value	0x76	0x1f	0x01	0x1b	0xcd	0x12	0x34	0x55

After execution:
r10 = 0x00001234
r5 = 0x00001bcd

6.4.5 Vector Load and Store Operations

Vector load and store operations are used for loading and storing the VRF registers. The vector load operations are supported in both the LS0 and LS1. Each can read up to 256 bits from the data memory. The store operation is supported only in LS0 and can write up to 256 bits from the data memory. When two vector load operations are targeted to the same memory block, the processor has a wait-state due to memory contention unless the two load operations are consecutive and aligned to 32 bits.

The example below illustrates a vector memory load operation.

Example 6-13. Vector Load Operation

Moving an immediate value to an address pointer, which is used as a base:

```
mov #0x100, r7
```

Vector loading using indexed addressing mode:

```
vld (r7.ui+0x40).s16, v10.i8, v20.i8
```

Assuming memory contents at location 0x140 are:

Address	0x15E	0x15C	0x15A	0x158	0x156	0x154	0x152	0x150	0x14E	0x14C	0x14A	0x148	0x146	0x144	0x142	0x140
Value	0x1048	0x4FC	0x64E5	0x45CF	0x1E16	0xE18B	0x2C07	0x7743	0x69CE	0x4AB6	0xF99A	0xDD4C	0xC0D7	0x3C56	0xF3B3	0xCBDA

16 shorts are loaded from memory, sign-extended to int and written to vectors v10 and v20.

After execution:
v10.i8 = {0x000069CE, 0x00004AB6, 0xFFFFF99A, 0xFFFFDD4C,
0xFFFFC0D7, 0x00003C56, 0xFFFFF3B3, 0xFFFFCBDA}

v20.i8 = {0x00001048,0x000004FC,0x000064E5,0x000045CF,
0x00001E16,0xFFFFE18B, 0x00002C07,0x00007743}

6.4.6 Overlapped Load Operation

Overlapped load operations are used for loading vector registers with overlapped data. The overlap load operations are supported only in LS0. Another load operation cannot be issued in parallel to an overlap load operation.

The overlap load operation loads four vector registers by shifting one element between them. Table 6-7 illustrates the vector register values after an overlapped load operation of integers, starting at M0.

Table 6-7. Overlapped Load

vZ0.i8[7]	vZ0.i8[6]	vZ0.i8[5]	vZ0.i8[4]	vZ0.i8[3]	vZ0.i8[2]	vZ0.i8[1]	vZ0.i8[0]
M7	M6	M5	M4	M3	M2	M1	M0
vZ1.i8[7]	vZ1.i8[6]	vZ1.i8[5]	vZ1.i8[4]	vZ1.i8[3]	vZ1.i8[2]	vZ1.i8[1]	vZ1.i8[0]
M8	M7	M6	M5	M4	M3	M2	M1
vZ2.i8[7]	vZ2.i8[6]	vZ2.i8[5]	vZ2.i8[4]	vZ2.i8[3]	vZ2.i8[2]	vZ2.i8[1]	vZ2.i8[0]
M9	M6	M7	M6	M5	M4	M3	M2
vZ3.i8[7]	vZ3.i8[6]	vZ3.i8[5]	vZ3.i8[4]	vZ3.i8[3]	vZ3.i8[2]	vZ3.i8[1]	vZ3.i8[0]
M10	M9	M8	M7	M6	M5	M4	M3

The example below illustrates an overlapped memory load operation.

Example 6-14. Vector Load Overlap Operation

Moving an immediate value to an address pointer, which is used as a base:

```
mov #0x100, r7
```

Vector loading using indexed addressing mode:

```
vldov (r7.ui+0x40).s16, v10.s16, v4.s16, v30.s16, v12.s16
```

Assuming memory contents at location 0x140 are:

Address	0x15E	0x15C	0x15A	0x158	0x156	0x154	0x152	0x150	0x14E	0x14C	0x14A	0x148	0x146	0x144	0x142	0x140
Value	0x1048	0x4FC	0x64E5	0x45CF	0x1E16	0xE18B	0x2C07	0x7743	0x69CE	0x4AB6	0xF99A	0xDD4C	0xC0D7	0x3C56	0xF3B3	0xCBDA
Address	0x17E	0x17C	0x17A	0x178	0x176	0x174	0x172	0x170	0x16E	0x16C	0x16A	0x168	0x166	0x164	0x162	0x160
Value	0x220E	0x42AB	0xB20C	0x3906	0x7302	0x1720	0x57B3	0xDA56	0xE5F0	0xEB42	0xE623	0xFC70	0x4F62	0xCDFC	0x3FED	0x2546

19 shorts are loaded from memory and written to vectors v10, v4, v30 and v12 with element overlapping between the vectors.
After execution:

v10.s16 = {0x1048, 0x04FC, 0x64E5, 0x45CF, 0x1E16,0xE18B, 0x2C07, 0x7743,
0x69CE, 0x4AB6, 0xF99A, 0xDD4C,0xC0D7, 0x3C56, 0xF3B3, 0xCBDA}

v4.s16 = {0x2546, 0x1048, 0x04FC, 0x64E5, 0x45CF, 0x1E16,0xE18B, 0x2C07,
0x7743, 0x69CE, 0x4AB6, 0xF99A, 0xDD4C,0xC0D7, 0x3C56, 0xF3B3}

v30.s16 = {0x3FED, 0x2546, 0x1048, 0x04FC, 0x64E5, 0x45CF, 0x1E16, 0xE18B,
0x2C07, 0x7743, 0x69CE, 0x4AB6, 0xF99A, 0xDD4C,0xC0D7, 0x3C56}

v12.s16 = {0xCDFC, 3FED, 0x2546, 0x1048, 0x04FC, 0x64E5, 0x45CF, 0x1E16,
0xE18B, 0x2C07, 0x7743, 0x69CE, 0x4AB6, 0xF99A, 0xDD4C,0xC0D7}

6.4.7 Checkered Load Operation

Checkered load operations are used for loading vector registers with even and/or odd memory elements. The checkered load operations are supported in both the LS0 and LS1.

The checkered load operations read 256 bits from the data memory and write the vector registers with either the even elements or the odd elements of the memory. Optionally, it can write both even and odd elements. Table 6-8 illustrates the vector register values after a checkered load operation of shorts, starting at M0.

Table 6-8. Checkered Load

vZ0l.16[7]	vZ0l.16[6]	vZ0l.16[5]	vZ0l.16[4]	vZ0l.16[3]	vZ0l.16[2]	vZ0l.16[1]	vZ0l.16[0]
M14	M12	M10	M8	M6	M4	M2	M0
vZ1l.16[7]	vZ1l.16[6]	vZ1l.16[5]	vZ1l.16[4]	vZ1l.16[3]	vZ1l.16[2]	vZ1l.16[1]	vZ1l.16[0]
M15	M13	M11	M9	M7	M5	M3	M1

Example 6-15. Checkered Load Operation

Moving an immediate value to an address pointer, which is used as a base:

```
mov #0x100, r7
```

Vector loading using indexed addressing mode:

```
vldchk(r7.ui+0x44).c32, v16h.c32, v22h.c32
```

Assuming memory contents at location 0x140 are:

Address	0x15E	0x15C	0x15A	0x158	0x156	0x154	0x152	0x150	0x14E	0x14C	0x14A	0x148	0x146	0x144	0x142	0x140
Value	0x1048	0x4FC	0x64E5	0x45CF	0x1E16	0xE18B	0x2C07	0x7743	0x69CE	0x4AB6	0xF99A	0xDD4C	0xC0D7	0x3C56	0xF3B3	0xCBDA
Address	0x17E	0x17C	0x17A	0x178	0x176	0x174	0x172	0x170	0x16E	0x16C	0x16A	0x168	0x166	0x164	0x162	0x160
Value	0x220E	0x42AB	0xB20C	0x3906	0x7302	0x1720	0x57B3	0xDA56	0xE5F0	0xEB42	0xE623	0xFC70	0x4F62	0xCDFC	0x3FED	0x2546

19 shorts are loaded from memory. The even elements of the memory access are written to v16h registers and the odd elements are written to v22h.

After execution:

v16h.c32 = {0xED, 0x46, 0x48, 0xFC, 0xE5, 0xCF, 0x16, 0x8B,
0x07, 0x43, 0xCE, 0xB6, 0x9A, 0x4C,0xD7, 0x56}

v22h.c32 = {0x3F, 0x25, 0x10, 0x04, 0x64, 0x45, 0x1E, 0xE1
0x2C, 0x77, 0x69, 0x4A, 0xF9, 0xDD, 0xC0, 0x3C}

6.5 Pointer-modification Mechanism

Both the LS0 and LS1 include a pointer-modification mechanism. The modification of a pointer can be performed in parallel with address generation when using indirect, indexed or parallel addressing mode. In indirect addressing mode, the pointer contains the address while in indexed and parallel addressing modes, the pointer contains the offset (for more information about these addressing modes, refer to Section 6.3.1, Indirect Addressing Mode and Section 6.3.2, Indexed Addressing Mode). The pointer is modified after the address has been generated. Thus, this modification is referred to as post-modification.

The following sections explain the methods for modifying a pointer using post-modification, when using LSU post-modification (non-vector memory access) or when using a vector post-modification method.

The following options are available for a pointer post-modification:

- Post-increment: The pointer is incremented by the access width specified in the instruction. The access width is determined according to the memory access operation and can be one, two or four. This option is specified in the instruction by using the symbol +.
- Post-decrement: The pointer is decremented by the access width specified in the instruction. The access width is determined according to the memory access operation and can be one, two or four. This option is specified in the instruction by using the symbol -.
- Step modification: One of the eight step registers (s0 – s7) is added to the pointer. The step register contains a signed value. This value represents a number of bytes. In contrast to the previous options, the step modification is not scaled according to the access width. This option is specified in the instruction by using the symbols +s0, +s1 …+s7.
- Offset modification: A signed immediate value is added to the pointer. The width of this value is up to 32 bits. The immediate value contains a number of bytes. Similar to the step modification, the offset modification is not scaled according to the access width. This option is specified in the instruction by using the symbol +#imm, where #imm represents the signed immediate value.

Below are several examples that illustrate the options described above.

Example 6-16. Post-increment in Indirect Addressing Mode

```
mov #0x100,r5.ui ; r5 is initialized with the value 0x100
nop
ld (r5.ui).us2+, r0.ui, r2.ui .
```

This ld instruction applies the indirect addressing mode and uses the pointer r5. Since two words are fetched, the access width is 32 bits or four bytes. The post-increment option is used and it is specified by the symbol +. The address generated to the memory is 0x100. The value of the pointer after the ld instruction is 0x104.

Example 6-17. Post-decrement in Indexed Addressing Mode

```
mov #0x1004,r0.ui   ; g0 is the base register.
mov #0x10,r1.ui     ; r1 contains the offset.
nop
nop
st r2.ui, (r0.ui+r1.i).ui-
```

The st instruction uses the indexed addressing mode with r1 as the offset. A integer stored into the memory. Therefore, the access width is four bytes (32 bits) and the pointer is post-decremented by four and its value after the st instruction is 0x0C. The post-decrement option is specified by the symbol -. The content of r0 is not changed.

Example 6-18. Step Modification

```
mov #0x10004,s3.i  ; Step register initialization.
mov #0x200,r3.ui   ; Offset initialization.
nop
nop
ld (r0.ui+r3.i).ui+s3, r2.ui
```

The offset r3 is post-modified using step register s3. The value of r3 after the ld instruction is 0x10204. Note that s3 is not scaled according to the access width. The content of r0 is not changed.

Example 6-19. Offset Modification

```
mov #0x100,r5.ui  ; r5 is initialized with the value 0x100.
nop
ld (r5.ui).us2+#0x50,r0.ui,r1.ui
```

The pointer r5 is incremented by the immediate value #0x50. Therefore, the value of r5 after the ld instruction is 0x150.

All post-modification options are summarized in Table 6-9.

Table 6-9. Post-modification Summary

Instructions	Syntax	Access Type	Operation
ld, st	+ / -	c, c2	pN = pN + number of chars
		s, s2, s4	pN = pN + (2 x number of shorts)
		i, i2, i4	pN = pN + (4 X number of ints)
vld, vst, vldchk	+ / -	c32, s16, i8	pN = pN + 32
vldov	+ / -	c32, s16, i8	pN = pN + (4 X element size)
All load/store instructions	+s0 … +s7	All	pN = pN + step_register step_register = s0/s1/s2/s3/s4/s5/s6/s7 Step register includes a signed value
All load/store instructions	+#imm	All	pN = pN + #imm

6.5.1 modr Instruction

The modr instruction performs a modification of the pN pointer without accessing the data memory. This instruction specifies the pointer and the value to be added to it. This value can be one of the four step registers or a signed immediate value. The width of the immediate value is up to 32 bits. For more information, refer to the CEVA-XM4 Volume II Instruction Set document.

Example 6-20. modr Instruction

```
mov #0x50,r4.ui
nop
modr (r4.ui).us+#0x200
```

The pointer r4 is post-modified using an immediate value. The value of r4 after the modr instruction is 0x250.

6.5.2 Modulo Mode

The aim of modulo mode is to create cyclic buffers. The size of a buffer is determined within a dedicated 32-bit modulo register (modu0, modu1, modu2 and modu3). The size of the buffer represents a number of bytes, where the maximum buffer size is 64K-1 bytes. The modulo register is initialized with the value of the preferred buffer size M. Activating modulo mode is done by using the corresponding muX switch in one of the supporting ld, st, vld and vst instructions.

Modulo arithmetic implies that the post-modified value on the indirect rN+pm_value remains within the data buffer limits (pm_value represents scaling or step register post-modification).

Therefore, whenever the post-modified value (the rN+pm_value) is out of the buffer boundaries, two situations are possible:

- If pm_value > 0, then the new value is rN + pm_value – M.
- If pm_value < 0, then the new value is rN + pm_value + M.

When using modulo mode, three rules must be observed:

- M ³ the absolute value of the step, pm_value.
- The pointer should point to an address inside the buffer.
- M should be a multiple of the largest access width used in the load or store instructions that access the buffer.

The start address of the buffer must contain zeros in its k LSBs, where k is the minimum positive integer that satisfies 2k ≥ M. The upper boundary is *start address + M -1*. Therefore, the data space is automatically divided into multiple cyclic buffers so that the lower boundaries of each buffer satisfy the previous conditions. The start address cannot point anywhere in the data space except to the specific locations described above.

Note: When using a step register to specify the pm_value, only the lower 16 bits of it are used. The sign of the post-modification is determined according to bit number 16 of the step register.

Example 6-21. Modulo - Multiple Wrap-around Buffers

```
mov #0x0c,modu2.ui  ; The buffer size is 12.
mov #0x4,s2.i ; s2 ≤ 12.
mov #0x308,r4.ui
bkrep #5 ; Execute six iterations.
{
   modr (r4.ui).i+s2.i
   add r4.ui,r2.ui,r3.ui
}
```

In this example, the buffer size is 12, and therefore every address with four LSBs equal to zero is qualified as a start address. In the example, the pointer is initialized with the value 0x308. This pointer is located in a buffer whose start address is 0x300 (four LSBs are zeros) and end address is 0x30B (size of the buffer is 12).
The value of r4 during the six iterations is 0x308, 0x300, 0x304, 0x308, 0x300 and 0x304.

6.5.3 Load and Store Instruction Predication

The LSU supports a predication mechanism that enables the conditional execution of all load and store instructions. Scalar load and store instructions support the scalar predicate (prX) while the vector load and store instructions support vector predicates (vprX).

When the scalar predicate is set, the load and store instructions are accessing the memory according to the instruction. When the scalar predicate is cleared, no load or store operation occurs. Note that the internal memory can be accessed in the load operation even if the predicate is cleared, but the destination registers are not written and no access protection valuation asserts because of this instruction. The external memory is not accessed when the scalar predicate is cleared.

Using a vector predicate does not affect the memory access − only the read and write to/from the vectors.

6.6 Read-after-write Sequence

The write buffer adds latency to the write transactions. However, it must be transparent to the core read transactions. An address-match mechanism enables stalling the core reading data that is not yet written to memory.

Every read transaction that accesses the memory generates an additional access to check if the required data is in the write buffer. A set of comparators checks whether the requested address, or part of it, is in the address delay-lines or in the write buffer waiting to be written to the memory. The read-match mechanism then generates a stall for each byte of the read transaction.

The above read-after-write protection is valid as long as one of the reading and/or writing instructions is a parallel load/store instruction (vpld/vpst). If one of the accesses is a parallel load or store, the core checks for a read-after-write match only in the write buffer and not in the address delay lines. This means that unless the user can guarantee that there is no read-after-write sequence between a parallel memory access and another access, the user must wait for seven cycles after a parallel store and another load and between a store and a parallel load. For more information, refer to the CEVA-XM4 Memory Subsystem (MSS) document.

7 Program Control Unit

7.1 Introduction

The PCU handles the program memory interface, the alignment of instruction packets from the program memory, the dispatching of instructions to the various functional units and the execution flow of instruction packets in the core.

Dedicated mechanisms in the PCU support both sequential and non-sequential instruction flow. The latter occurs due to branches, block-repeat loops and interrupts.

The PCU fetches the instruction packet opcodes from program memory, aligns the instruction packets and identifies to which functional unit each instruction is associated. It then dispatches the instructions to the various functional units for further decoding and execution. PCU instructions go through the same process before they are fully decoded and executed by the PCU.

The PCU consists of two main components:

- The instruction dispatcher that contains the following functional elements:
  - Alignment unit
  - Dispatch unit
- The program sequencer that contains the following functional elements:
  - Program address generation and Instruction Fetch Unit (IFU)
  - Instruction decoder (for PCU instructions)
  - Branch mechanism
  - Block-repeat mechanism
  - Interrupt handler

Table 7-1 describes the direct accessible registers of the program sequencer.

Table 7-1. Directly Accessible Program Sequencer Registers

Name	Description	Size (Bits)
lc	Block-repeat loop counter register.	32
lciX	Block-repeat loop counter initialization registers (X = 0 to 3).	32
lcstep0, lcstep1	Step registers for post-modification of the lciX registers.	32
Bknest0, bknest1, bknest2	Block-repeat nesting-level store/restore registers.	32
retreg	Return register for subroutine calls.	32
retregi	Return register for maskable interrupts and for the software interrupt (trap).	32

Table 7-1. Directly Accessible Program Sequencer Registers(Continued)

Name	Description	Size (Bits)
retregn	Return register for non-maskable interrupts.	32
retregb	Return register for the emulation software interrupt (trape instruction) and the hardware breakpoint interrupt (BI).	32
modA, modAb, modC, modH	Status and mode registers.	32
version	Hardware version identification (read-only). The 16 MSBs indicate the core type version and the 16 LSBs indicate the RTL hardware version. This register can be read via the OCEM chain. For more information, refer to the On-Chip Emulation Reference Guide.	32

The set of program sequencer registers that are directly accessible is referred to as the Sequencer Register File (SRF). The general notation for these registers is srX.

The program counter (pc) is a 32-bit register that cannot be directly accessed. To use the pc value, a dedicated mov pc, arX instruction can be used.

7.1.1 Block Diagram

Figure 7-1 shows a block diagram of the PCU.

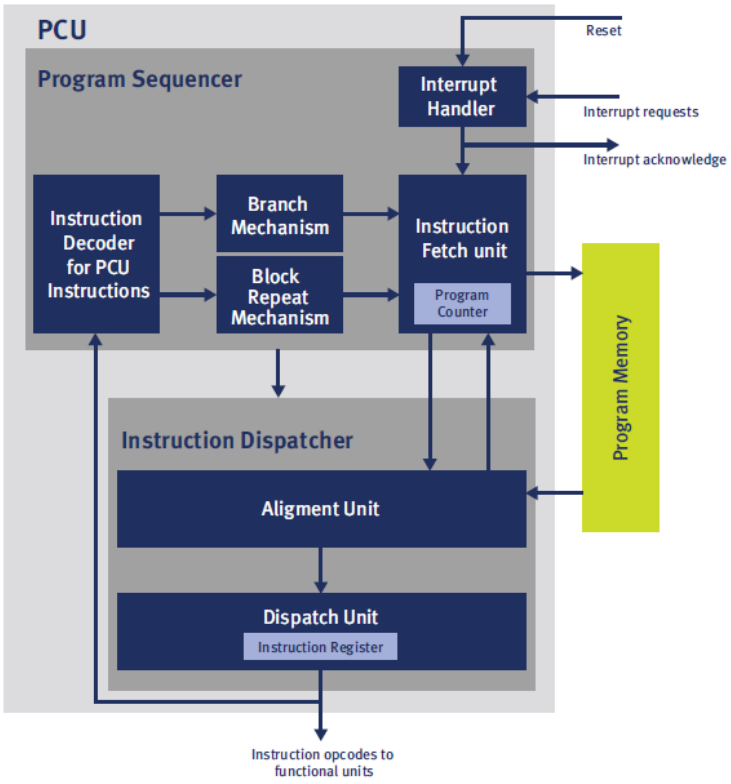


Figure 7-1. PCU – Simplified Block Diagram

The IFU generates the program address bus driven to the program memory. In order to generate the address, it receives indications from the instruction dispatcher, interrupt handler and the block-repeat and branch mechanisms.

The instruction packet opcodes that are read from program memory are loaded to the instruction dispatcher, which aligns them and issues them to the various functional units (each unit includes its own instruction decoder).

The PCU's instruction decoder generates control signals to all of its internal units.

Each module shown in Figure 7-1 is described in detail in the following sections.

7.1.2 Terminology

This section defines the terms that are used in this chapter. A description of one term may include some of the other terms defined in this section:

- *Integer*: A field of 32 bits.
- *Fetch-line*: A set of words that the program sequencer reads from program memory during the instruction fetch stage.
- *Instruction*: An atomic operation that is performed by one of the core's functional units. Instruction sizes are 16 or 32 bits.
- *Control Word*: A pseudo-instruction that enables one of the various features that are supported by the Dispatch unit. Control word sizes are 16 bits or 32 bits.
- *Component*: A general name for instructions and control words; denotes either an instruction or a control word.
- *Instruction Packet*: A group of instructions that are executed in parallel. An instruction packet can also contain control words.
- *Header*: A 16-bit control word that supports parallel execution of 16-bit instructions. If a header is used, it is the first (leftmost) component of the instruction packet. The header does not occupy a slot in the instruction packet.
- *Slot*: A place within the instruction packet in which a component can be placed. Each component within the instruction packet occupies a single slot of the instruction packet (excluding the header, if used).
- *Slot Limit*: The maximum number of components (excluding the header, if used) that can form an instruction packet.

7.2 Instruction Dispatcher

The encoding scheme of the core is VLIW. This implies that an instruction packet comprises one or several instructions, where each instruction is associated with one of the core's functional units. Each instruction occupies one slot in the instruction packet.

The objective of the instruction dispatcher is to identify the instructions that form the instruction packet, and issue each instruction to the functional unit(s) with which it is associated.

The instruction dispatcher contains the following functional elements:

- Alignment unit
- Dispatch unit

The alignment unit identifies the instruction packet, and loads it to the instruction register. When 16-bit instructions are included in the instruction packet, this unit also aligns the 16-bit instructions to 32-bit boundaries before loading them to the instruction register.

The Dispatch unit performs the actual issuing of the instructions to the proper functional units.

These units are discussed in detail in Section 7.2.2,Alignment Unit and Section 7.2.3, Dispatch Unit.

7.2.1 Instruction Dispatcher Features

The main features of the instruction dispatcher are:

- *Variable Instruction Size Support*: Basic instruction sizes are 16 bits and 32 bits. 16-bit instructions provide good code compactness. Both instruction types can be used as single instructions, as well as parallel instructions (both types can also be combined in the same instruction packet).
- *Instruction-level Parallelism Support*: Any number of instructions can form an instruction packet. All possible combinations of 16-bit and 32-bit instructions are supported (bounded only by the slot limit).
- *Instruction Extensions*: Immediate operands and direct address (program and data) operands in 32-bit instructions can be extended by an additional field. The extension size can be either 10 bits or 26 bits.

7.2.1.1 Instruction-level Parallelism Support

Each instruction packet can include any combination of the following components:

- 32-bit instructions
- 16-bit instructions
- 32-bit control words
- 16-bit control words

An instruction packet can be either a single instruction or a group of components. The alignment unit determines how many components are grouped together, based on information encoded within the components. There are two ways to indicate that components are grouped together:

- By using a dedicated bit within each component (a sequence bit)
- By using a dedicated 16-bit header that precedes the components

7.2.1.2 Instruction Extensions

Some 32-bit instructions can be extended with additional bits. Such extensions are needed for instructions that include immediate operands or address operands (program or data address), based on the size of these operands. A dedicated 32-bit control word is added to the instruction packet:

- *Immediate Operands*: When the value of the immediate operand cannot be represented by the immediate field within the 32-bit instruction, an extension is required. The size of the immediate field within the different 32-bit instructions can vary, depending on the instruction itself.
- *Program Address Operands (in Branch-type Instructions)*: When the value of the address (or relative address) cannot be represented by the address field within the 32-bit instruction, an extension is required.
- *Data Address Operands*: When using long-direct or indexed addressing modes, the addressing operands can be extended.

Refer to the CEVA-XM4 Volume II Instruction Set document for the sizes of these operands within the various instructions.

There are two types of extensions: 10 bits or 26 bits. The Assembler tool determines if an extension is needed, and selects which one to use based on the size of the immediate or address operand.

Extensions are contained in the instruction packet as control words (called extension control words). Each control word occupies a single slot in the instruction packet. In some cases, a single control word can extend more than one instruction, as described in the following table.

Table 7-2. Number of Instructions Extended by Each Extension Control Word Type

Extension Type	Number of Instructions That Can Be Extended by a Single Control Word	Control Word Size (Bits)
10-bit Extension	1	16
Dual 10-bit Extension	2	32
26-bit Extension	1	32

All three types of extension control words can be simultaneously used within an instruction packet to extend several instructions. The only limitation of their usage is the slot limit.

7.2.2 Alignment Unit

The alignment unit identifies the instruction packet, and loads it to the instruction register.

The instruction packet opcodes that are fetched from program memory are stored in a temporary buffer, called the instruction queue (implemented in the program sequencer; refer to Section 7.3, Program Sequencer, for details). During sequential operations, the alignment unit selects a group of 32-bits from the queue. The number of DWs selected is determined by the fetch-line width, which is eight DWs.

The selected group contains either a single instruction or a set of parallel instructions that should be dispatched to the functional units (an instruction packet). In cases where the maximal available parallelism is not utilized or the instruction packet includes 16-bit instructions, the selected group includes additional words of the instruction packet(s) that follows.

At the end of the alignment stage, the instruction register is loaded with the selected group. The current instruction packet is dispatched from this instruction register to the functional units, during the dispatch stages.

If 16-bit components (instructions or control words) are included in the instruction packet, the alignment unit aligns them to 32-bit boundaries before they are loaded to the instruction register. This is done by padding each 16-bit component with 16 zeroes. This enables 16-bit components to be handled by the Dispatch unit in the same manner as 32-bit components. Hence, they do not need any dedicated support in the Dispatch unit.

The alignment unit also calculates the instruction packet length and the address of the next sequential instruction packet.

7.2.3 Dispatch Unit

The Dispatch unit issues all the instructions that have to be executed in the current cycle (the instruction packet) to the appropriate functional units. Immediate extension indications are also issued to the functional units by the Dispatch unit.

The input to the Dispatch unit is the instruction register, which is loaded by the alignment unit.

Each instruction has an identification field that indicates the functional unit with which it is associated. The Dispatch unit issues each instruction to the appropriate functional unit according to this field.

Control words include a field that indicates their type. The Dispatch unit uses this field to identify each control word.

When extension control words are included in the instruction packet, the Dispatch unit determines to which functional unit each extension field is associated. It then uses this information to issue all the extension fields to the various functional units accordingly.

7.2.4 Undefined Opcodes Types

The CEVA-XM4 DSP core implements a dedicated mechanism for handling undefined opcodes (that is, opcodes that are not mapped to any CEVA-XM4 core instructions). When the core detects an undefined opcode, it translates it as an nop instruction, stores the instruction packet address and issues an interrupt signal at the undefined opcode's V1 pipe stage.

There are several types of undefined opcodes:

- 16- or 32-bit opcodes
- Opcodes that cannot be mapped to any of the core units (system-level undefined opcode)
- Core unit opcodes that cannot be translated as an instruction

For any undefined opcode the corresponding unit bit is set in the UOC_STS register. For example, if the error is in the LS1 unit, the LS1UOP bit is set.

For more information about the status registers' space, refer to the CEVA-XM4 Architecture Specification MSS document.

7.3 Program Sequencer

The Program Sequencer controls the fetching of instruction packets from the program memory, and all the aspects of sequential and non-sequential instruction flow in the core.

The following elements form the program sequencer:

- IFU
- Instruction decoder (for PCU instructions)
- Branch mechanism
- Block-repeat mechanism
- Interrupt handler

The sections that follow describe each of these elements.

Note: In the text that follows, the term word refers to a 16-bit entity, and the term double-word (DW) refers to a 32-bit entity.

7.3.1 Instruction Fetch Unit

The IFU generates the address of the instruction packet that should be fetched by the core from program memory, and coordinates the read accesses that the core performs. The PCU calculates a 32-bit address, which enables a 4GB address space. For more information about program memory organization, refer to the CEVA-XM4 MSS document.

During each instruction fetch, eight consecutive DWs are fetched from program memory: the DW at the current address and seven additional DWs from the addresses that immediately follow. The set of fetched DWs is called a fetch-line and its width is 256 bits.

The first address of the fetch-line must be a multiple of the fetch width. Instruction packet addresses can point to any word address within the fetch-line. Byte-address resolution is not needed for program memory addressing, because the shortest instruction packet can be a word (two bytes). Therefore, the LSB of the instruction packet address is always cleared.

The encoding scheme of the core is VLIW, with dedicated support for SIMD operations. Each VLIW contains up to eight instruction slots. The number of available slots is called the slot limit, which determines the size of the instruction dispatcher.

The generated program address can either be a sequential address or a non-sequential address, as described in the following sections.

7.3.1.1 Sequential Fetch Mechanism

Sequential fetching of instruction packets is performed when the core executes a sequential block of instruction packets.

During sequential operation, successive fetch-lines are read from memory. Each fetch-line contains one or more instruction packets. Since the number of instructions that are executed in parallel constantly varies, instruction packets are often fetched several cycles in advance (meaning, ahead of the cycle during which they are needed).

A set of four registers, called the queue, is implemented in the program sequencer. Each fetched line is loaded to one of the queue's registers, which provides temporary storage for instruction packets that are fetched in advance of their execution. The width of the queue registers is identical to the width of the fetch-line.

Program memory fetches are not initiated every cycle. The fetch mechanism determines when to issue a program memory fetch, depending on the state of the queue.

The sequential fetch mechanism ensures correct and continuous program flow, and prevents overflow in the queue. In every cycle, this mechanism decides whether to issue a new fetch. The decision is based on the following information:

- The status of the queue (the number of valid words in the queue).
- The number of words that the current instruction packet consumes.
- The currently active fetches: As a fetch operation takes two cycles, the mechanism keeps track of fetches that were initiated in the two preceding cycles.

7.3.1.2 Non-sequential Fetch Mechanism

Non-sequential instruction flow is an event in which the normal sequential flow of instruction packets is suspended, and an instruction packet from a different address is fetched. The non-sequential fetch mechanism generates all the non-sequential addresses.

Table 7-3 describes the events that cause non-sequential address generation.

Table 7-3. Non-sequential Address Generation Scenarios

Event	New Program Address
Reset	0x0000_0000.
Boot	Defined by the value on the vectored interrupt input bus during reset. The OCEM can mask the boot input pin.
Interrupt (non-vectored)	Predefined according to the type of the accepted interrupt.
Vectored interrupt	Specified by an interrupt vector.
Branch- or call-type instructions (relative or absolute)	Specified within the instruction opcode, or taken from an operand that is specified in the instruction.
Return from a subroutine or from an interrupt routine	Taken from the appropriate return register.
Block-repeat loop folding	The first instruction packet of the loop.

The above non-sequential events cause all four registers of the queue to be invalidated (flushed), and the PCU begins to refill the queue with opcodes from the new program address.

7.3.2 Program Sequencer Registers Access

The set of directly accessible registers of the program sequencer is referred to as the Sequencer Register File (SRF). The general notation for these registers is srX. These registers can be accessed by move-type instructions, as well as push and pop instructions. The following transactions are supported:

- Move an immediate value to an SRF register
- Move from an SRF register to another SRF register
- Move from a GRF register (addressing register file of the LSU) to an SRF register
- Move from an SRF register to a GRF register
- Push an SRF register to the stack
- Pop an SRF register from the stack

Refer to the CEVA-XM4 Architecture Specification Volume II document for the complete list of SRF registers.

All the above-mentioned instructions are conditional (can use a predicate register), and cannot write to the program counter (pc). Refer to the CEVA-XM4 Volume II Instruction Set document for more details about the specific instructions.

7.3.3 Branch Mechanism

The program sequencer supports the following branch-type instructions: br, brr, brar, call, callr, callar, ret, retb, reti, retn and breako.

The target address of a branch-type instruction can be specified in the following ways:

- An absolute address encoded within the instruction opcode (br and call instructions)
- A relative offset (relative to the address of the branch instruction), encoded within the instruction opcode (brr, callr and breako instructions)
- An absolute address specified within a GRF register (brar and callar instructions)
- An absolute address specified within one of the return registers (ret, retb, reti, and retn instructions)

All these instructions are multi-cycle and conditional (except retb, which is unconditional). These instructions take several cycles to execute, due to two main reasons:

- Fetching the opcode of the instruction packet at the target address takes two cycles.
- The status of the condition (predicate register) is known only at a late pipeline stage.

In order to minimize the cycle penalty of these instructions, dedicated mechanisms are implemented in the PCU. The following sections provide a description of these mechanisms.

Note:

- While the core is in transition mode (that is, a situation where the instruction queue is not updated due to change of flow; for example, returning from an interrupt)) and an instruction packet with a false-condition branch-type instruction with 0 or 1 delay slots is located at the end of the address fetch-line and the next instruction packet size is 256-bit, then the core behavior is unexpected. Such behavior can be automatically resolved by the Assembler tool when placing a .NEWLINE assembler directive before every conditional branch type instruction. The .NEWLINE directive forces the Assembler to place the next instruction (in this case, the branch instruction) at the start of the fetch-line.
- In the following sections, the term branch is used for any branch-type instruction.

For details about the cycle count of each branch instruction, refer to the CEVA-XM4 Volume II Instruction Set document.

7.3.3.1 Delay Slots in Branch Instructions

The programmer can utilize up to four instruction packets in delay slots in most branch instructions. The instruction packets in the delay slots are performed before the actual execution of the branch itself. In conditional branches, the delay slots are executed regardless of the condition of the branch (unless they explicitly use the same predicate register as the branch instruction), and they do not affect the decision about whether or not to perform the branch. Each instruction in a delay slot can be conditional. Each delay slot utilized reduces the cycle count of the branch instruction by one cycle.

Delay slots are optional. The number of delay slots used must be specified within the branch instruction (refer to the CEVA-XM4 Volume II Instruction Set document for details).

The instruction packets in delay slots must be composed of single-cycle instructions that do not break the pipeline. In addition, instruction packets in delay slots are non-interruptible.

7.3.3.2 Branch Prediction Support

When a conditional branch is executed, the core should continue execution from the target address if the condition is met or from the next sequential instruction packet if the condition is not met. For a branch without delay slots, the next sequential instruction packet is the instruction packet that immediately follows the branch. For a branch with delay slots, the next sequential instruction packet is the instruction packet that immediately follows the last delay slot.

Branch instructions include a prediction switch that enables selection between taken and nottaken prediction. When this switch is used to indicate not-taken prediction, the core performs speculative execution of the next sequential instruction packets. When the status of the condition is known, the core either continues sequential execution (if the condition is not met), or annuls the sequential instruction packets that it began executing (if the condition is met, meaning, the prediction was incorrect). In the latter case, the core also begins to fetch and execute the instruction packets at the target address.

When the prediction switch is used to indicate taken prediction, the core begins speculative execution of the target instruction packets as soon as the branch is identified. When the status of the condition is known, the core either continues execution of the target instruction packets (if the condition is met), or annuls the target instruction packets that it began executing (if the condition is not met, meaning, the prediction was incorrect). In the latter case, the core also resumes execution of the next sequential instruction packets.

If the taken/not-taken switch in the syntax is omitted, the default behavior for br, brr, call, callr, ret, reti and retn instructions is:

- Not-taken when the instruction is predicated.
- Taken when no predicate is used.

The default breako instruction behavior is always not-taken when the instruction is predicated, and taken when no predicate is used.

Example 7-1. Branch Prediction

As a straightforward example, consider the following code that sums an array of numbers in memory, and replaces all zeros in the array to a value of –1. In this example, it is known that most of the values in the specified memory range are not equal to zero:

```
mov 0x0, r0.ui
mov #0xffff, r2.us
mov #0x1000, r1.ui    ; In the range 0x1000 - 0x10FF, most words
                      ; are not equal to zero.
bkrep #0xff
{
   ld(r1.ui).us, r0.ui
   nop
   cmp{neq} r0.ui, #0x0, pr0
   nop
   br{t} cont1,?pr0; Branch to cont1 if not equal,
                   ; prediction = taken.
   st r0.us, (r1.ui).us; Executed only if the condition is not met.
cont1: modr (r1.ui).us+
   add r0.ui, r2.ui, r0.ui
}
```

Since it is known that most of the values in the specified memory range are not equal to zero, it is obvious that the branch is taken most times. Therefore, selecting a taken prediction for this branch reduces the cycle count of this loop.

Determining whether a certain branch is mostly taken or not-taken is generally not as straightforward as in the previous example. To efficiently determine the best prediction switch for every branch, a profiler utility can be run on the code, and the selection of the prediction switches can be based on these statistics.

Correct prediction can reduce the cycle count of the branch by two cycles, and incorrect prediction can increase it by two cycles. Refer to the CEVA-XM4 Volume II Instruction Set document for details about the effect of prediction on the cycle count of each branch instruction.

7.3.3.3 Call and Return Instructions

During execution of a call instruction, the return address is written to an internal register, which is called retreg (return register). When a return instruction is executed, the return address is taken from the retreg register.

The return registers can be accessed by move-type instructions so that the return address can be modified before the return instruction is performed. This enables returning to a different location, if required.

In order to enable nested subroutine calls, the return register of subroutine calls (retreg) must be pushed to the software stack before or in parallel to performing a call instruction, if this call instruction is located inside a higher-level subroutine. The retreg register must be popped from the software stack before performing the ret instruction of the higher-level subroutine. Depending on the save/restore convention, this can be done within the lower-level subroutine (in parallel to the ret instruction), or after returning from it (inside the calling subroutine). See the following example for reference.

Example 7-2. Nested Subroutine Calls

```
main:   call sub1
        :
sub1:   inc a0
        :
        push retreg || call sub2  ; Return address of sub1 is saved.
        :                         ; retreg contains sub2 return address.
        :
        pop retreg                ; Restores return address of sub1.
        :                         ; Restore can be alternatively performed in sub2.
        :
        ret
```

7.3.3.4 Target Instruction Packet Alignment

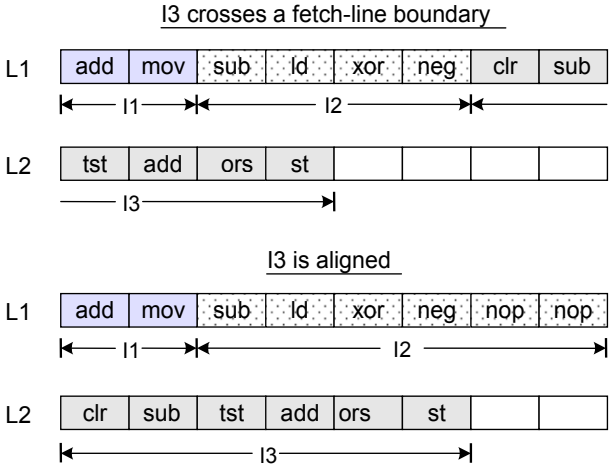
The target of a branch instruction can be located anywhere within a fetch-line, and instruction packets can cross fetch-line boundaries. If the instruction packet at the target of the branch crosses a fetch-line boundary, two lines must be fetched before execution of the instruction packet can begin. In such cases, an additional penalty of one cycle is added to the cycle count of the branch instruction, due to the fetch of the second line.

The additional single-cycle penalty described above can be avoided by aligning instruction packets to the fetch-line boundaries that are targets of branch instructions and originally cross a fetch-line boundary. This can be done by padding the instruction packet preceding the target instruction packet with parallel nop instructions, as shown in the example below.

Such alignment can be automatically performed by the Assembler tool.

Example 7-3. Alignment by Adding Parallel nop Instructions

Consider the following instruction packet sequence:
add || mov
sub || ld || xor || neg
clr || prm || tst || and || ors || st
For simplicity, operands are not detailed.
The figure below illustrates the two fetch-lines that these instruction packets occupy. As shown in the figure, the third instruction packet (labeled I3) crosses a fetch-line boundary.
Adding two nop instructions to the second instruction packet moves the third instruction packet two slots forward, and causes it to be aligned. The modified code is:
add || mov
sub || ld || xor || neg || nop || nop
clr || sub || tst || add || ors || st
Below is a graphical description of the alignment.



7.3.4 Block-repeat Mechanism

The block-repeat mechanism in the program sequencer supports zero-overhead looping of a block of instruction packets. It is activated by the bkrep instruction.

The bkrep instruction causes a block of instruction packets to be repeated. The number of repetitions is up to 216 (65,536) and is either specified by one of the four lciX registers, or embedded in an immediate operand within the bkrep instruction.

7.3.4.1 Block-repeat Mechanism Registers

The following table details the registers associated with the block-repeat mechanism. The register size for the lciX, lcstep0 and lcstep1 registers is 32 bits. These registers can be used as general-purpose registers.

Table 7-4. Block-repeat Mechanism Registers

Name	Description	Size (Bits)
lc	Loop Counter register.	32
lciX	Loop Counter Initialization registers. These registers hold an initial value of the number of repetitions. (x = 0 to 3)	32
lcstep0, lcstep1	Step registers. These registers can be used to post-modify the lciX registers.	32

Table 7-4. Block-repeat Mechanism Registers(Continued)

Name	Description	Size (Bits)
bknest0, bknest1, bknest2	Nesting-level store/restore registers. For more information, refer to Section 7.3.4, Block-repeat Mechanism.	32
fia	First Instruction Packet Address register (internal register).	32
ea_off	Offset of the end-address relative to the address of the first instruction packet of the loop (internal register).	13

The term internal register means that this register is not directly accessible by move-type instructions. The functionality of internal registers is explained in the section that follows.

Each of the lc, fia and ea_off registers represents an internal stack of four such registers that support four nesting levels of block-repeat loops (refer to Section 7.3.4, Block-repeat Mechanism for a further description of the nesting mechanism). When an access to the lc register is performed, the lc of the active loop level is accessed.

The lc register can be used as an index inside the block-repeat loop.

7.3.4.2 Block-repeat Mechanism Operation

Upon execution of the block-repeat instruction (bkrep), the number of repetitions is loaded to the lc register. The value is specified by either a 16-bit immediate value or by one of the lciX registers. The actual number of repetitions to be performed is the value written to the lc register plus one.

The relative end address of the block (the distance between the address of the instruction packet containing the bkrep instruction and the address of the third instruction packet from the end of the loop) is encoded within the bkrep instruction. The ea_off register is loaded with the distance between the first instruction packet address and the address of the third instruction packet from the end of the loop for loop-nesting support (refer to Section 7.3.4, Block-repeat Mechanism). For loops of two, three or four instruction packets, the relative end address is not encoded within the instruction. The sequencer internally determines the end address in these cases, and loads the result to the ea_off register.

When the end of the loop is reached, an address match is detected. Then, execution folds back to the beginning of the loop, if the loop counter is greater than zero.

Folding to the beginning of the loop means aborting the sequential instruction fetch, and fetching the instruction packets at the beginning of the loop instead.

The end of the loop is identified by detecting the third instruction packet from the end of the loop (that is, two packets before the last packet) – applicable to loops of three instruction packets or more. The relative address encoded in the bkrep instruction points to this instruction packet. For a loop of two instruction packets, the end of the loop is reached every second cycle, as there are only two packets.

The early identification of the end of the loop provides a zero-cycle penalty when folding to the start of the loop. There is a one-cycle penalty only if the first instruction packet of the loop (which is fetched during the folding) crosses a fetch-line boundary. Therefore, it is highly recommended to make sure that the first instruction packet in the loop does not cross a fetchline boundary.

If the first instruction packet crosses a fetch-line boundary, it can be aligned to the fetch-line boundary in order to avoid the cycle penalty during the folding of the loop. Aligning the first instruction packet is performed by padding the instruction packet that precedes it (the packet that includes the bkrep instruction) with parallel nop instructions. Such alignment ensures zerooverhead looping, and can be performed automatically by the Assembler tool.

When using a loop of two instruction packets, the first instruction packet must not cross a fetchline boundary.

When a block-repeat loop of two instruction packets is performed, fetches from program memory are halted after the packets of the loop body are fetched, until the last repetition of the loop. Since there are only two instruction packets in the loop body, there is no need to re-fetch them from memory, as both packets can be stored in the queue. Note that a zero-cycle penalty is achieved also in loops of two instruction packets.

A single delay slot is available when using a block-repeat loop of two or three instruction packets. Delay slot usage is mandatory in loops of two instruction packets..

7.3.4.3 Post-modification of lciX Registers

When using the bkrep instruction with an lciX register, the register can be post-modified so that the next time a bkrep instruction with the same lciX register is executed, a modified value of lciX is used.

The post-modification options are:

- None
- +1
- -1
- Shift right by one
- Shift left by one, plus one (the LSB is written with 1); see Example 7-4
- Shift right by one, minus one
- +lcstep0
- +lcstep1

Example 7-4. lci Post-modified by Shift Left Add One

When the lci register is post-modified using shift left by one, plus one, the LSB is written with 1, meaning, lci=2*lci+1. This enables repetition values that are multiples of two every time lci is post-modified, as illustrated in the table below. Recall that when using bkrep lci, the number of repetitions is lci+1.

value of lci (lci = 2 * lci + 1)	4	9	19	39	...
number of repetitions	5	10	20	40	...

7.3.4.4 Block-repeat with a Negative Count Value

The value of the lciX register is treated as a signed 32-bit number. When the value is negative, the loop is not performed, and execution continues at the first instruction packet outside the blockrepeat loop (the instruction packet that immediately follows the last instruction packet of the loop). In addition, the lc register of the skipped bkrep level is loaded with zero (in effect, lc itself only becomes zero if the outermost level is skipped). Post-modification of lciX is always performed, regardless of its sign.

There is cycle penalty when a loop is skipped. The sequencer performs a branch over the loop for a long loop or annuls the instruction packets inside it for a short loop. Refer to the CEVA-XM4 Volume II Instruction Set document for the cycle count in this case.

7.3.4.5 Block-repeat Nesting

The program sequencer supports nested loops, which means that a bkrep instruction can be used inside a block-repeat loop.

When a block-repeat loop is performed, only the loop counter (lc) of the current active loop can be accessed by move-type instructions.

7.3.4.6 Hardware Support

Hardware nesting provides up to four nesting levels. Execution of a bkrep instruction within an outer block-repeat loop automatically causes the lc, fia and ea_off registers to be pushed into a four-stage internal last in first out (LIFO) stack, allowing a new set of parameters to be used. When the inner loop is completed, the previous lc, fia and ea_off parameters are popped out of the internal stack, and execution of the outer block-repeat loop resumes.

An indication of the nesting level is specified in a read-only three-bit field, named BCx, located within the modA mode register. The MSB of these bits can be used to indicate that all four loop levels are in use. When this bit is set, the core is inside the fourth block-repeat loop and cannot use additional nested bkrep instructions. In this case, nesting levels should be freed (using dedicated instructions) if additional loop levels are needed.

In addition to the BCx bits, the In-loop (LP) bit inside the modA mode register can be accessed. This bit is set when a block-repeat is executed and it is reset upon normal completion of the outermost active loop level.

7.3.4.7 Higher Nesting Levels by Software

As described in the previous section, the core supports four levels of nested loops in hardware. Dedicated instructions enable unlimited levels of nesting.

The core supports two dedicated instructions, bkst and bkrest, which enable higher block-repeat nesting levels.

The bkst instruction stores a single level of the block-repeat parameters (of the active loop level) in dedicated registers (bknest0, bknest1 and bknest2) of the block-repeat mechanism. In order to store more than one bkrep level, the bknest0, bknest1 and bknest2 registers should be pushed to the software stack prior to further bkst instructions.

The bkrest instruction restores a single level of the block-repeat parameters from the bknest0, bknest1 and bknest2 registers. The block-repeat parameters of the lowest available loop level are written by this instruction. In order to restore more than one bkrep level, the bknest0, bknest1 and bknest2 registers should be popped from the software stack prior to further bkrest instructions.

The bkst and bkrest instructions store and restore the loop parameters described in Table 7-5.

Table 7-5. Stored/Restored Loop Parameters

Parameter Description	Parameter Name	Number of Bits	Location
Address of the first instruction packet of the loop	fia	32	bknest1[12:0]
Offset of the end address of the loop from the first instruction packet address	ea_off	13	bknest1[15]
LP bit	LP	1	bknest1[31]
Loop counter	lc	32	bknest2[31:0]

Refer to the CEVA-XM4 Volume II Instruction Set document for details about the order of these parameters as they are stored/restored to/from memory.

When a bkst instruction is executed, the active hardware nesting level is released so that another block-repeat loop can be used. The bkst instruction releases a single nesting level every time it is used, since it stores only the parameters of the current active nesting level.

The bkrest instruction reconstructs a loop that was previously stored. It causes the mentioned parameters to be restored from the bknest0, bknest1 and bknest2 registers. Upon execution of a bkrest instruction, the program sequencer calculates the value of the end address of the loop by adding the offset (ea_off register, restored from memory) to the restored fia register.

Example 7-5. bkst and bkrest

```
Assuming an initial state of four active hardware nesting levels:
push{bknest} || bkst; move current nesting level to bknest0,1,2 register
                ; while storing the old bknest0,1,2 to stack
                ; if bknest0,1,2 are empty then the push instruction
                ; is redundant
push{bknest} || bkst; move next nesting level to bknest0,1,2 register
                ; while storing current nesting level to stack
                ; now only two Hardwar nesting levels are active
nop
nop
bkrep #3                ; Three active hardware nesting levels,
 {
  :
  bkrep lci0            ; Four active hardware nesting levels,
   {                                    ; and two stored levels.
    :
    :
   }
  :                                     ; The inner loop is completed. Therefore,
  :                                     ; there are three active hardware levels.
 }
:                                       ; The outer loop is completed. Therefore,
:                                       ; there are two active hardware levels.
:
pop{bknest} || bkrest; Restores one level from bknest0,1,2 registers
                ; and pops another level from the stack.
nop
nop
nop
pop{bknest} || bkrest; Restores last level from popped bknest0,1,2 and pop
                ;  next level from the stack.
                ; If highest nesting level is in bknest0,1,2 registers
                ;  then the pop instruction is redundant.
```

Any number of bkst instructions can be performed. Executing the same number of bkrest instructions restores the state of the core to the state it was before the bkst instructions were executed. These instructions can be safely used even outside a block-repeat loop.

For example, bkst instructions are often used at the beginning of an interrupt service routine if this routine includes block-repeat loops. This is useful since the interrupt may have been accepted inside a block-repeat loop, and possibly at the fourth hardware nesting level. In such cases, loop levels must be released in order to enable usage of block-repeat loops in the service routine. However, the interrupt can also be accepted outside a block-repeat loop. It is not required to check if there are active loops (or how many) before the bkst instructions at the beginning of the service routine are executed. Using the same number of bkrest instructions at the end of the interrupt service routine guarantees that the original state of the core is restored.

For more information about bkst and bkrest, refer to the CEVA-XM4 Volume II Instruction Set document.

7.3.4.8 Breaking Out of Loops

Breaking out of a block-repeat loop is performed by executing one of the two dedicated instructions: break and breako.

The break instruction is used to break out only from the current block-repeat loop. The break forces the next iteration of the loop not to take place. Hence, the program continues sequentially. The current repetition of the loop continues normally, but when the end of the loop is reached, folding back to the beginning of the loop does not occur. When a break instruction is executed, the following actions are taken:

- The nesting level counter (BCx) is decremented by one.
- If the new value of the nesting level counter is zero, the LP bit is cleared.
- Execution flow continues from the instruction packet following the break instruction.

The breako instruction is used to exit from the active block-repeat loop(s) and continue the execution flow at a different location, specified by an offset (up to 32 bits wide). Up to four active hardware nesting levels can be aborted by this instruction. If more than one level is skipped, the number of loop levels that are aborted must be specified in the instruction (the default is one level). The breako instruction is conditional, meaning that breako causes the program to leave the blockrepeat loop once a condition is met. If the condition is met, the following actions are taken:

- The number of skipped hardware levels (specified as a parameter in the breako instruction) is subtracted from the nesting level counter (BCx).
- If the new value of the nesting level counter is zero, the LP bit is cleared.
- Execution branches to the instruction packet in the location specified by the offset (encoded in the instruction), relative to the location of the breako instruction.

7.3.5 Exception Handling

An exception is an external event that causes the normal instruction flow to stop and continue from another location. Exceptions are:

- Reset
- Maskable interrupt
- Non-maskable interrupt
- Breakpoint interrupt
- Software interrupt

Reset causes the core to unconditionally abort all operations and restart execution from address 0x0000_0000. It also causes all the registers to be loaded with their reset values, but does not affect the memory contents. When an interrupt request is active, the program sequencer checks if the core is in an interruptible state, meaning that the instruction flow can be suspended. If so, the interrupt is accepted.

When an interrupt is accepted, the program sequencer activates the proper interrupt acknowledge signal, and stores the current program counter in the appropriate return register. Then, the instruction flow branches to a predefined address location, according to the accepted interrupt, and the interrupt service routine located at that address is performed.

When the interrupt service routine is completed, the appropriate return-from-interrupt instruction (reti, retn, or retb) must be used to return from the service routine to the instruction packet that was interrupted. When a return instruction is performed, the return address is taken from the appropriate return register, which was loaded when the interrupt was accepted.

The core supports the following interrupts:

- Six hardware maskable interrupts (INT0, INT1, INT2, INT3, INT4 and VINT)
- Four software maskable interrupts (trap{t0}, trap{t1}, trap{t2} and trap{t3})
- One non-maskable interrupt (NMI)
- One non-maskable software interrupt (trap)
- One emulation software interrupt (trape)
- Two hardware breakpoint interrupts (BI and PABP)

In order to enable the acceptance of maskable interrupts (INT0, INT1, INT2, INT3, INT4, trap{t0}, trap{t1}, trap{t2}, trap{t3} and VINT), the interrupt enable bit must be set. Moreover, each maskable interrupt has a corresponding interrupt mask bit (IM0, IM1, IM2, IM3, IM4, IM5, IM6, IM7, IM8 and IMV) that corresponds to INT0, INT1, INT2, INT3, INT4, trap{t0}, trap{t1}, trap{t2}, trap{t3} and VINT, respectively. Setting the mask bit enables the interrupt associated with it. The IE and mask bits are located in the modA and modAb mode registers.

When a maskable interrupt is accepted, execution branches to the interrupt service routine and the IE bit is cleared, thereby disabling all other maskable interrupts. They are enabled as soon as the return-from-interrupt instruction (reti) at the end of the interrupt service routine is performed, as this instruction sets the IE bit only when returning from a maskable-interrupt service routine.

Each of the hardware maskable interrupts has an Interrupt Pending bit. These bits, IP0, IP1, IP2, IP3, IP4 and IPV (located in the modA mode register), reflect the status of the corresponding interrupt input pin, INT0, INT1, INT2, INT3, INT4 and VINT, respectively. These bits can be used in applications that use interrupt polling, while disabling the automatic response to the interrupts.

Each of the software maskable interrupts (trap{t0-t3}) has an Interrupt Pending bit. These bits, IP5, IP6, IP7 and IP8 (located in the modAb mode register), reflect the status of the corresponding software interrupt trap{t0}, trap{t1}, trap{t2}, trap{t3}, respectively. The corresponding pending bit is set after one of the software maskable interrupt instructions is identified. When the corresponding pending bit is set, additional software maskable interrupts at the same priority level (that is, the same priority level trapX instruction) are ignored and the interrupt is regarded as an nop instruction. After the software interrupt is accepted, the corresponding pending bit is cleared.

VINT is a vectored interrupt. When VINT is issued, the destination address of the interrupt is taken from a core input bus, which is driven by off-core logic. Thus, multiple interrupts can be serviced by the same interrupt pin, each with a different destination address (controlled by the off-core logic).

7.3.5.1 Interrupt Priorities and Nesting

Different interrupts have various priorities and cause the program sequencer to continue the instruction flow from different locations. Among the maskable interrupts, INT0 has the highest priority and VINT has the lowest.

The priority between INT0, INT1, INT2, INT3, INT4, trap{t0}, trap{t1}, trap{t2}, trap{t3} and VINT is significant only when several interrupts are received simultaneously. In this case, the interrupt with the highest priority is serviced. The following table describes the priorities and target addresses of the different interrupts.

Note: The Target addresses for interrupts are the sum of INTBASE address register and fixed offset (INTBASE is located in modH register) as described below

Table 7-6. Interrupt Priorities and Related Information

Name	Description	Priority	Target Address	Ack. Signal	Return Register	Return Instr.
reset	Reset	1 (highest)	0x0000_0000	---	---	---
	Boot		Programmable	---	---	---
trap	Software interrupt	2	INTBASE+0x0000_0040	---	retregi	reti
trape	Emulation software interrupt	3	INTBASE+0x0000_0020	piackbn	retregb	retb
BI	Breakpoint interrupt					
NMI	Non-maskable interrupt	4	INTBASE+0x0000_0060	piacknn	retregn	retn
INT0	Maskable interrupt 0	5	INTBASE+0x0000_0080	piack0n	retregi	reti
INT1	Maskable interrupt 1	6	INTBASE+0x0000_00C0	piack1n	retregi	reti
INT2	Maskable interrupt 2	7	INTBASE+0x0000_0100	piack2n	retregi	reti
INT3	Maskable internal interrupt 3	8	INTBASE+0x0000_0140	piack3n	retregi	reti
INT4	Maskable internal interrupt 4	9	INTBASE+0x0000_0180	piack4n	retregi	reti
VINT	Vectored interrupt	10	Programmable	piackvn	retregi	reti
Trap{t0}	Software interrupt	11	INTBASE+0x0000_0200	-	retregi	reti
Trap{t1}	Software interrupt	12	INTBASE+0x0000_0240	-	retregi	reti
Trap{t2}	Software interrupt	13	INTBASE+0x0000_0280	-	retregi	reti
Trap{t3}	Software interrupt	14	INTBASE+0x0000_02C0	-	retregi	reti
PABP	Program address breakpoint	15 (lowest)	INTBASE+0x0000_0020	piackbn	retregb	retb

For proper operation of a debug session, a branch-to-self must be located at address INTBASE+0x20 (target address of BI and trape interrupts). For more information, refer to the CEVA-XM4 MSS document.

Maskable interrupts can be enabled inside an interrupt service routine of another maskable interrupt. The following sequence should be performed in order to enable interrupt nesting:

- The retregi register must be pushed onto the software stack.
- In systems that allow operation modes other than Supervisor mode (refer to Section 7.4, Operation Modes, for details), it is necessary to push the modC register onto the software stack.
- The IE bit in the modA register should be set.
- The internal interrupts (INT3 and INT4) are reserved for use by internal devices (such as an internal DMA).

At the end of a service routine in which nested interrupts are enabled, interrupts must be disabled and the retregi register must be popped from the software stack. If the modC register is pushed onto the software stack, it must be popped.

Example 7-6. Nested Interrupts

```
int0rt:   inc a0
                :
                push retregi
                eint                    ; Interrupt nesting is now enabled.
                :
                :
                dint                    ; At the end of the service routine, interrupts are
                pop retregi; disabled and retregi is popped.
                :
                Reti
```

Note: The push retregi and eint instructions can be paralleled, and can appear as the first instruction packet in the interrupt service routine. The dint and pop retregi instructions can also be paralleled.

The following interrupts cannot be nested:

- A non-maskable interrupt (NMI) or any of the maskable interrupts cannot be accepted within the NMI service routine.
- During execution of the trape/BI routine, other interrupt requests (NMI, INT0−INT4, trap{t0}-trap{t3}, VINT and BI) are disabled. The usage of a trap or trape instruction in this routine is forbidden too.
- During execution of a trap routine (priority 2), other interrupt requests (NMI, INT0−INT4, trap{t0}−trap{t3} and VINT) are disabled. The usage of a BI or trape instruction in this routine is allowed. The usage of a trap instruction in this routine is forbidden.

- In systems that support NMIs and allow operation in Non-supervisor mode (refer to Section 7.4, Operation Modes, for details), it is necessary to push the modC register onto the software stack in the first instruction packet of all maskable interrupt service routines. modC must be popped inside one of the delay slots of the reti instruction. This is required in order to ensure that the modC register is not modified before it is saved, in case an NMI is accepted. The following example illustrates this requirement.

Example 7-7. ModC in Maskable Interrupt Service Routine (Non-supervisor Mode)

```
int0rt:   push modC
                :
                :
                :
                reti {ds1}
          pop modC
```

7.3.5.2 Interrupt Latency and Non-interruptible States

Latency of the NMI, INT0, INT1, INT2, INT3, INT4, VINT and BI interrupts is two cycles, assuming that the core is in an interruptible state. The interrupt latency is calculated as the number of cycles between the assertion of the interrupt request and activation of the appropriate interrupt acknowledge signal.

The penalty of the hardware interrupts (NMI, INT0, INT1, INT2, INT3, INT4, VINT and BI) is five cycles, assuming that the processor is in an interruptible state. The interrupt penalty is calculated as the number of cycles in which the core does not execute any instructions.

During non-interruptible states, the core continues to execute instructions and does not service the interrupt. The interrupt is accepted once the core exits these states, if it is still active.

Non-interruptible states are:

- During reset (reset signal is active).
- The first three instructions being fetched after de-activation of the reset input signal.
- Execution of a branch-type instruction, until the branch is over.
- Stop mode in which no clock is provided to the core.
- During execution of instruction packets in delay slots.
- Between the three last instruction packets of a block-repeat loop comprised of three or more instruction packets, excluding the last iteration of the loop.
- Between the first and second instruction packets of a block-repeat loop comprised of two instruction packets, excluding the last iteration of the loop. In this case, interrupts are enabled when folding from the second instruction packet to the first.
- Between a bkrep lciX instruction to the first instruction packet outside the loop, if the lciX register is negative (the loop body is skipped in this case).
- During the first repetition of a block-repeat loop comprised of two, three or four instruction packets.
- During the first iteration after returning to a block-repeat loop comprised of two instruction packets from an interrupt service routine.

- During a block-repeat loop of two instruction packets when the value of the loop counter is one.
- Between a conditional absolute/relative branch/call instruction or a return instruction to the next sequential instruction, if the prediction is not-taken, the branch is not-taken and zero or two delay slots are used.
- Between a conditional absolute/relative branch/call instruction or a return instruction to the next sequential instruction, if the prediction is taken, the branch is not-taken and no delay slots are used.
- During the first instruction packet in all interrupt service routines. However, software interrupts (trap and trape) are legal in this location and are accepted.
- After a PCU instruction, excluding the following instructions: break, lbf, mov, modlci, nop, verifend, verifclr, monitor and verifbegin. Software interrupts (trap and trape) are legal after any PCU instruction and are accepted.

7.3.5.3 Interrupts Inside Block-repeat Loops of Two Instruction Packets

When an interrupt is accepted inside a block-repeat loop of two instruction packets, dedicated support is provided (for bigger loops, no such dedicated support is needed).

When an interrupt is accepted, a dedicated indication bit (named bk2int) is saved in the return register together with the value of the program counter. This indication specifies if the interrupt is accepted during a block-repeat loop of two instruction packets. The bit is set when the interrupt is accepted during a block-repeat loop of two instruction packets, and cleared otherwise.

bk2int is written to the LSB of the return register, which is a redundant bit (for addressing purposes) in the return register. This bit is redundant because the memory is byte-addressable, but instruction packet addresses can only point to word addresses (a word is two bytes). Therefore, the LSB of the address generated by the program sequencer is always zero. Figure 7-2 illustrates this concept.

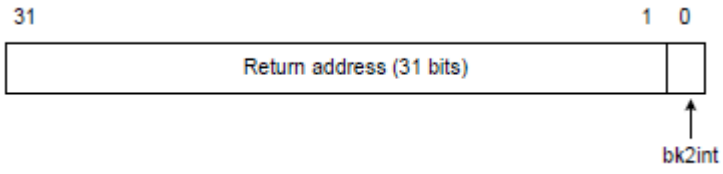


Figure 7-2. Return Register Structure

When returning from the interrupt service routine, the program sequencer checks the value of the restored bk2int bit. If it is cleared, sequential execution flow continues from the instruction packet that was interrupted. If it is set (meaning, a block-repeat loop of two instruction packets was interrupted), the program sequencer continues to perform repetitions of the loop that was interrupted.

7.4 Operation Modes

7.4.1 Overview

The core can operate in three operation modes:

- Supervisor mode
- User0 mode
- User1 mode

These operation modes determine the ability of the software to execute certain instructions and modify certain registers, thus providing a software protection mechanism. A permission violation indication is generated when the software attempts to execute an instruction or modify a register while running under a non-privileged operation mode.

7.4.2 Software Limitations

While in Supervisor mode, there are no limitations on software operations. When the core is in User0 or User1 modes, several limitations apply, which are described in Table 7-7.

Table 7-7. Operation Modes and Software Restrictions

Software Limitations	Supervisor	User0	User1
Restricted instructions	None	dint, eint, retb, reti, retn, psu, ld[a]	dint, eint, retb, reti, retn, in, out, psu, ld[a]
Restricted registers	None	modA (excludes modAb), modC, modH	modA (excludes modAb), modC, modH

a. Ld to the restricted registers (modA or modC)

When attempting to execute a restricted instruction or write to a restricted register while not in the appropriate operation mode, no operation is executed (except in ld or pop instructions in which only the sp or rN are incremented). In addition, a permission violation indication is issued.

Note: Execution of a trap or trap{t0-t3} or trape instruction updates the POM bit (OM bits are cleared) in the user-restricted modC register. This is not considered a permission violation, as these instructions are used to return from a User mode to supervisor mode.

Any external module, such as the MSS, can provide its own permission-level checking, which means that certain external registers or memory locations may not be accessible in a certain operation mode. In this case, the external module should generate a permission violation, and can update the modC register via a dedicated core input.

7.4.3 Switching Between Operation Modes

At reset, the core enters Supervisor mode. Only when operating in this mode, the software can change the operation mode to User0 or User1 by modifying the OM field in the modC register. Refer to the CEVA-XM4 Architecture Specification Volume II document for more details about the modC register.

When an interrupt is accepted or a trap/trape instruction is executed, the current operation mode is saved to the POM field in the modC register. The core then enters Supervisor mode (OM bits are cleared).

When a retb, reti or retn instruction is executed, the OM bits in the modC register are set to the value of the POM bits, restoring the operation mode in use prior to the interrupt acceptance.

7.4.4 Permission Violations

7.4.4.1 Permission Violation Indication

When a permission violation occurs, the bit in the modC register is set as an indication. In addition, the value of the OM field is copied to the VOM field, indicating the source of the violation. Permission violations can be detected by polling the PV bit.

In addition to the static permission violation indication, an active indication is also available. A dedicated output is set once a permission violation occurs. The external system can use this output to generate an interrupt (any core interrupt can be used). Inside the interrupt service routine, software can determine the operation mode at the time of the violation according to the VOM bits in the modC register. In order to enable further violation detections, the PV bit should be manually cleared at the end of the interrupt service routine.

The active indication is enabled by setting the PI bit in the modC register.

Note: The POM bits should not be used to determine the operation mode at the time of the violation. These bits indicate the operation mode at the time of the latest interrupt or trap/trape instruction.

7.4.4.2 Ignoring the Violating Operation

The software operation that caused the permission violation is ignored, which means that no operation is executed. As a result, the software may not function correctly in this case. This mechanism is required in order to ensure that unauthorized accesses do not cause damage to restricted system resources.

7.4.5 Debug Mode

The core enters Debug mode when either a BI or PABP hardware interrupt is accepted, or a trape instruction is executed. When entering Debug mode, the core automatically jumps to address 0x0000_0020 and starts to execute the instruction located there. Exiting Debug mode is achieved by using the retb instruction. The core then returns to the operation mode that was in use prior to Debug mode. During Debug mode, all interrupts are disabled, including NMIs. The permission level during Debug mode is equivalent to that of Supervisor mode.

8 CEVA-Xtend

The CEVA-XM4 DSP cores family enables end-users to customize the system according to a specific application(s). CEVA-Xtend™ units are additional user-defined instructions intended to expand the original core's instruction set for specific applications.

CEVA-Xtend units can be integrated along with SPUs and VPUs. End-users can create new instructions that activate the CEVA-Xtend hardware units. The syntax and the encoding of such instructions are customized and defined according to the application needs and architecture guidelines.

For more details, refer to the CEVA-XM4 Xtend Arch Spec.

9 On-Chip Emulation Module

The OCEM supports various debugging capabilities, and provides the interface between a host computer and the DSP. The OCEM offers a glue-less approach that uses a scan chain methodology, which eliminates the usage of a mailbox and a monitor program. All communications with the host are carried out via a standard JTAG port.

The OCEM has the following main tasks:

- Breakpoint generation
- Core control during debug
- Core internal registers and memories access support
- File I/O support
- Program counter profiling support

The DSP enters Debug mode either when a breakpoint occurs or when the debugger issues a stop request. A debug session is then initiated and the debugger gains access to the internal core registers and memories. During the debug session, the debugger controls the core and can have it execute any instruction, single step through the code and so on.

The OCEM consists of the following modules:

- JTAG module: Supports the interface between the OCEM and the host via JTAG protocol
- Break module: Generates the breakpoint request signals
- Control module: Controls the OCEM's operating mode

During a debug session, the debugger can make the core execute any instruction by feeding its opcode to the core via a dedicated scan chain, which is input to the core. The core's clock is controlled by the debugger and is stopped/started as needed. Any program memory location can be read from or written to using dedicated scan chains.

For more information, refer to the CEVA-XM4 Emulation Concept document.

10 Real-Time Trace

The CEVA-XM4 RTT solution combines the ARM ETM-R4 macrocell, the CEVA-XM4 core and a CEVA-XM4 Trace Wrapper interface block. The Trace Wrapper maps CEVA-XM4 core data to the ETM-R4 interface. The ETM-R4 macrocell complies with ETM V3.3-architecture. Program flow reconstruction is performed by decompressor software that resides on an external analyzer.

The CEVA-XM4 RTT solution supports the use of one or two ETM-R4 macrocells. Both solutions provide the same instruction trace capability, but the single solution can impact the CEVA-XM4 core performance more severely for a given code image and trace configuration, due to the need to stall the CEVA-XM4 core to transfer data to the ETM. In the dual ETM solution, the program flow is traced in both ETM macrocells, in order to simplify the synchronization of packets by the decompressor.

The use of the ARM ETM-R4 enables the CEVA-XM4 RTT to be incorporated in a CoreSight-compliant architecture for single-core and multiple-core debugging.

10.1 Real-Time Trace Highlights

CEVA-XM4 Real-Time Trace (RTT) supports various debugging capabilities, and provides the interface between the CEVA-XM4 core, the Trace Wrapper and the ETM-R4 macrocell host.

10.1.1 Program Instruction Flow Trace Abilities

- Cycle and non-cycle accurate program address trace:
  - A cycle-accurate trace indicates the number of cycles required to execute each instruction under most trace conditions.
  - A non-cycle-accurate trace indicates the execution of an instruction only, without regard for the number of cycles required.
- High compression is achievable by tracing change-of-flow only.
- Predicate trace to facilitate condition code reconstruction by the decompressor:
  - The CEVA-XM4 core supports instruction predication, whereby an instruction can be executed or not depending on the value of a specified predicate flag.
  - The predicate flags can be changed by dedicated instructions.
  - Knowing the predicate flags and the code image enables the decompressor to determine whether or not a predicated instruction was executed.
  - Predicate flags are sent to the ETM in normal data and ContextID packets.
  - The CEVA-XM4 core can be optionally stalled for up to two cycles when LSU accesses and predicate flag changes are concurrent.

10.1.2 Data Address and Data Value Trace

- The RTT logic enables the following data and address tracing capabilities
- Data Address trace only
- Data Value trace only
- Both Address and Data Value trace
- Two solutions are supported to enable tracing of two independent LS units:
  - A single ETM solution: Stalls the core for at least one cycle and at most two cycles when two LS accesses occur.
  - A second ETM: This solution also requires a trace funnel to interface to the trace buffer, plus that the decompressor synchronize two ETM packet streams.

10.1.3 Additional Features

The CEVA-XM4 RTT solution provides a comprehensive trace trigger facility that enables an instruction trace to be captured on an exact address match, address range match or complex sequential conditions. A cycle-accurate trace is supported at the maximum frequency of the core.

All trace compression is performed by the ETM cell. A program trace is highly compressed, particularly in non-cycle-accurate mode. Data trace compression is more limited. The ETM-R4 supports trace suppression when its internal trace FIFO is full.

Both the Trace Wrapper and the ETM can be configured through a non-intrusive JTAG or through CEVA-XM4 instructions. The ETM is programmed through the CoreSight debug APB bus. The wrapper has a dedicated programming interface to the CEVA-XM4 OCEM module.

A trace can be accessed through a real-time, high-speed trace port (suitable for an instruction trace only) or stored on-chip in an embedded trace buffer for subsequent access at a lower clock speed via JTAG or AHB. The trace port width and clock frequency are selected by the user.

The ETM trace logic provides user-programmable trace triggers and programmable trace filtering through the trace event resources, which include single address comparators, address range comparators, data value comparators, sequencers, counters, external inputs and outputs and complex trace events. For more information, refer to the CEVA-XM4 Real-Time Trace document.

11 Pipeline

The core has a dynamic pipeline, which is used by the Program Control Unit (PCU), Load Store Unit (LSU), Vector Processing Unit (VPU) and the Scalar Processing Unit (SPU). The PCU uses the first part of the pipeline, which contains four stages: IF1, IF2, D1, D2. Stages A1 and A2 are used for address generation by the LSU. Stages E1 and E2 are used by the SPU. Stages V1 to V5 are used by the VPU. Following is a description of the common stages:

- IF – Instruction Fetch, divided into two sub-stages: IF1 and IF2
- D – Dispatch/Decode, divided into two sub-stages: D1 and D2
- A – Address Generation, divided into two sub-stages: A1 and A2
- M – Memory Access, one stage: M
- E – Scalar Execution, divided into two sub-stages: E1 and E2
- V – Vector Execution, divided into five sub-stages: V1 to V5

11.1 Instruction Fetch Stages

- IF1: During this stage, the program address is issued by the PCU. The program address is decoded by the MSS and a memory read is initiated.
- IF2: This stage is dedicated for program memory access. The fetch-line is registered in the instruction queue at the end of this stage. The size of the fetch-line is 256 bits.

11.2 Dispatch/Decode Stages

- D1: The core selects a packet from the queue. This packet contains the instruction packet that is the next instruction or group of instructions that should be executed in parallel. At the end of this stage, the instruction packet register is loaded with the instruction packet.
- D2: In this stage, each instruction within the instruction packet register is dispatched to the appropriate unit in the core.

11.3 Address Generation Stages

- A1: Instructions are decoded within each unit. The data memory address generation is performed during this stage. This includes address and modulo calculation and read/write strobe generation.
- A2: The pointer’s post-modification value is calculated. SPU operations are executed, and the data address is decoded by the MSS.
- M: GRFs are read from the register file into the SPUs. and also data memory (TCM, cache, tag) setup time.

11.4 Scalar Execution Stages

The scalar execution is operating over two pipe stages, E1and E2.

- E1: Data memory is accessed. At the end of this stage, data memory read buffers are registered inside the core. In addition, SPU operations and VPU decoding are executed in this stage.
- E2: Scalar operations are executed. VPU operations' register files are read.
- V1: Scalar results are registered. The GRF register file is read for memory write. Single-stage VPU operations are executed. Four-stage VPU operations begin executing.

11.4.1 Vector Execution Stages

The vector execution is operating over five pipe stages, V1 to V5.

- V1: Scalar results are registered. The GRF register file is read for memory write. Single stage VPU operations are executed. Four-stage VPU operations begin executing.
- V2: Four-stage VPU operations execute. Single-stage VPU operation results are written to the VRF register file. At the end of this stage, the previous stage's read registers are written to the WB (Write Buffer).
- V3: Four-stage VPU operations execute. Vector accumulator register files are read.
- V4: Four-stage VPU operations execute. GRF, VRF and VPR register files are read for memory writes.
- V5: Four-stage VPU operations are registered. At the end of this stage, the previous stage's read registers are written to the WB.

11.5 Pipeline Examples

11.5.1 Scalar Unit Instruction Flow

Figure 11-1 describes the pipeline flow of an instruction executed in the SPU.

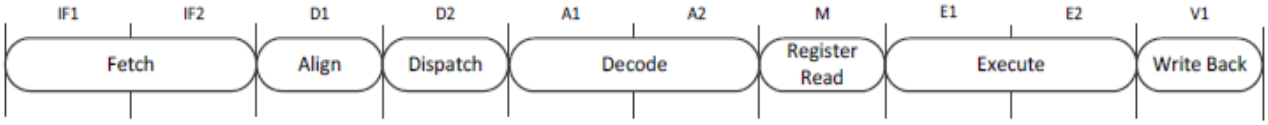


Figure 11-1. SPU Pipeline Flow

11.5.2 Data Load Instruction Flow

Figure 11-2 describes the pipeline flow of a load instruction.

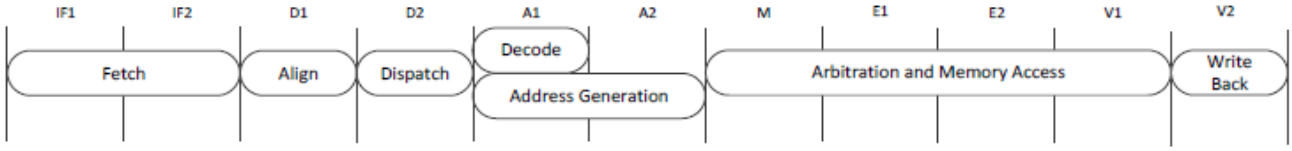


Figure 11-2. Load Instruction Pipeline Flow

11.5.3 VPU Instruction Flow

Figure 11-3 describes the pipeline flow of a three-cycle VPU instruction.

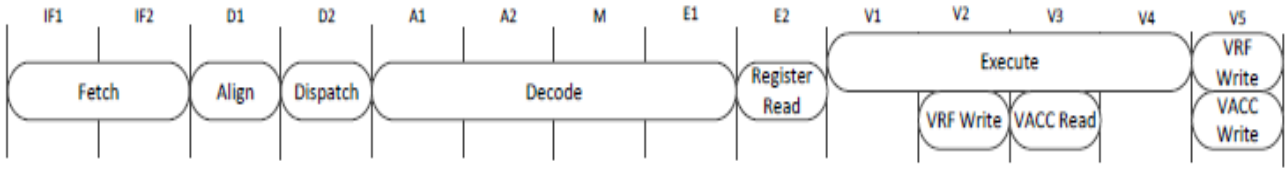


Figure 11-3. VPU Pipeline Flow