CEVA-XM4™

Architecture Specification Overview

Rev. 1.1.3.F

June 2016

DSP Division

Documentation Control

History Table

Version	Date	Description	Remarks
1.1.0	19 March 2015	Release	
1.1.2.F	31 March 2016	Release	
1.1.3.F	7 June 2016	Release	

Disclaimer and Proprietary Information Notice

Support

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 Antipolis France **Tel**: +33 4 83 76 06 00 **Fax**: +33 4 83 76 06 01

Table of Contents

List of Examples

List of Figures

List of Tables

1. Introduction

1.1 Scope

This document describes the architecture framework of the CEVA-XM4™ DSP core.

1.2 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 superscalar 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.

1.2.1 Vector Processor Highlights

1.2.1.1 Instruction-level Parallelism

The CEVA-XM4 architecture has a unique mix of Very Long Instruction Word (VLIW) superscalar 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.

1.2.1.2 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.

1.2.1.3 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.

1.2.1.4 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.

1.3 CEVA-XM4 Vector Processor

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

1.3.1 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 eight-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 8x16 bits, 32 multipliers of 16x16 bits and eight multipliers of 32x32 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 eight-bit, 16-bit and 32-bit operations
  - 16x16 bits, 32x16 bits and 32x32 bits multipliers
  - 16x16 bits, 32x16 bits and 32x32 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 address
- Byte-addressable data space
- 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
  - DMA data buffer manager
  - Dedicated AXI master and slave ports for hardware accelerators' connectivity
  - Separate I/O space for peripherals' connectivity

- Fully registered memory interface
- On-chip Emulation (OCEM) support via a JTAG port
- Standard system interface for easy integration with an existing SoC

1.4 Development Tools and Deliverables

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

1.4.1 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.

1.4.2 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.

1.4.3 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 superscalar 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 VP. The VP 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.

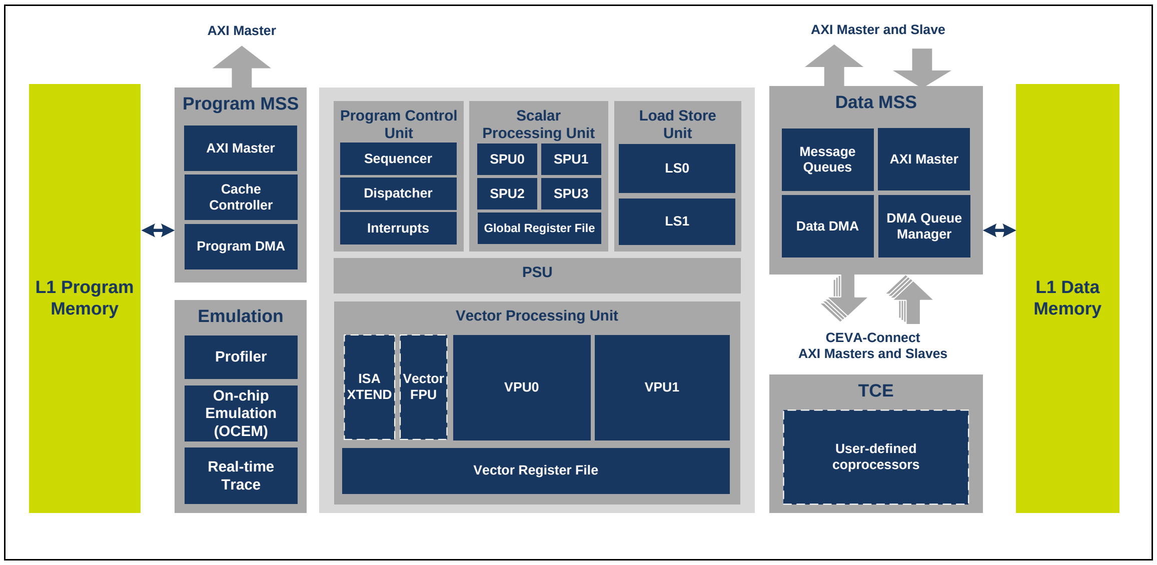


Figure 2-1: Vector Processor Block Diagram

The following sections describe each VP block in detail.

2.2.1 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	0/16/32 operations	16 operations per VPU
Scalar Floating Point	1/4	One floating point per SPU
Vector Floating Point	0/8/16	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	
Program Error Correction Code	Enabled/Disabled	For Program TCM and cache only
Internal Data Memory Size	128/256/512/1024KB	
AXI Slave Ports	1/2/3/4	Independent width configuration for each port can be 128 bits and 256 bits
AXI Data Master Ports	1/2/3	All ports are 128-, 256-bits wide (single configuration for all)
Queue Managers	0/4/8	
Data MSS Error Correction Code	Enabled/Disabled	Data Memory Sub-system Error correction
AXI Ports Error Correction Code	Enabled/Disabled	
On-chip Emulation (OCEM) Enhanced Mode	Enabled/Disabled	
Real-time Trace	Enabled/Disabled	

3. Register File

The VP 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 3.3
- **System Register File (SRF),** Section 3.3

The following sections describe these register files.

3.1 General Register File (GRF)

The VP 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

Int																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 3-1: GRF – 32-bit Integer

- 16-bit − Short Integer

Sign/Zero																Short															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 3-2: GRF – 16-bit Short Integer

- 8-bit − Character

Sign/Zero																								Char							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 3-3: GRF – 8-bit Character

- 32-bit − Float

Float																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 3-4: GRF – 32-bit Float

3.2 Vector Register File (VRF)

The VP 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 VLSU 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

vX.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

Figure 3-5: Vector Registers – i0 Through i7

- 16 short integers: s0 through s15

vX.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

Figure 3-6: Vector Registers – s0 Through s15

- 32 characters: c0 through c31

vX.c32															
c15	c14	c13	c12	c11	c10	c9	c8	c7	c6	c5	c4	c3	c2	c1	c0
127 120	119 112	111 104	103 96	95 88	87 80	79 72	71 64	63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0
c31	c30	c29	c28	c27	c26	c25	c24	c23	c22	c21	c20	c19	c18	c17	c16
255 248	247 240	239 232	231 224	223 216	215 208	207 200	199 192	191 184	183 176	175 168	167 160	159 152	151 144	143 136	135 128

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 VP 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 VP 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. Vector step registers are used for post-modification by the VLSU.

3.5 System Register File (SRF)

The System Register File (SRF) consists of the Program Counter (PC), return registers from subroutines and interrupts and loop counter registers that manage loop iterations. These registers are primarily used by the PCU.

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 seven predicate registers.

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. 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:
  - 32x32 multiplication and multiply-accumulate is supported only in SPU0.
- 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.

4.1.6 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.

4.2 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.

5. Vector Processing Units

5.1 Overview

The Vector Processor (VP) 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 a single result 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.

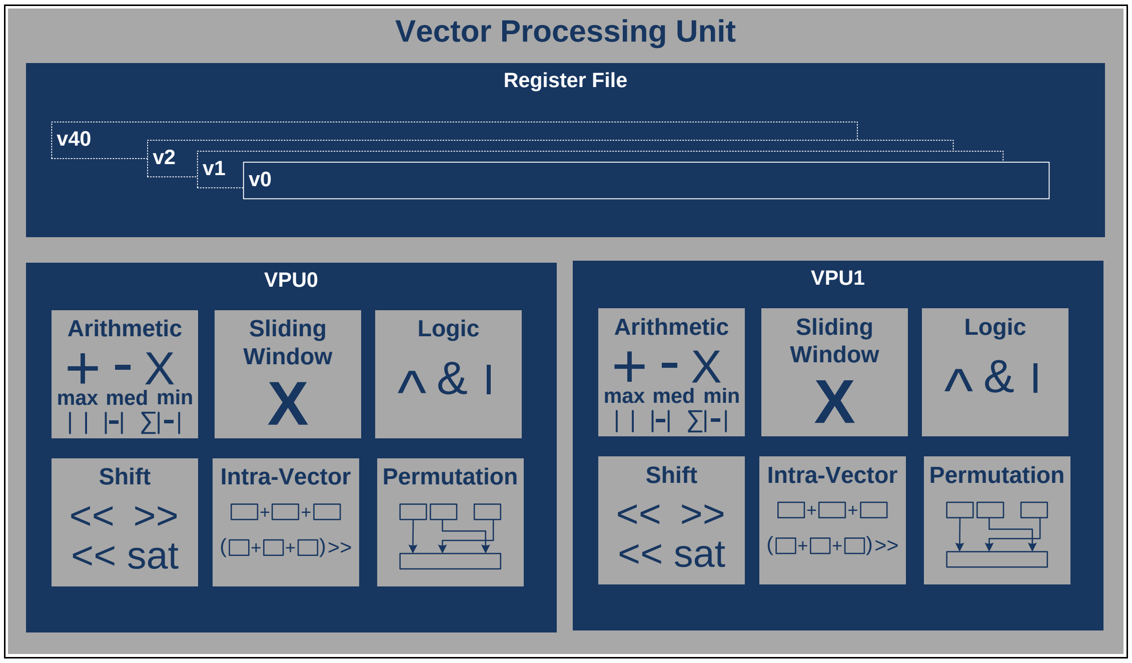


Figure 5-1: VPU Block Diagram

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, #uimmD4,
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 16 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.

5.3 Vector Operations

5.3.1 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*) and optionally comparing to a third value (*vabssubcmp*). This is frequently used in de-blocking and image processing algorithms.

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.

5.3.2 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.

5.3.3 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. This instruction is commonly used in the de-blocking algorithm.

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. The *vperm* instruction also includes fixed permutations that, among others, allow swapping adjacent bytes, reversing the byte order, diagonally spreading bytes and interleaving bytes. These operations are frequently used in intra-prediction and intra-estimation algorithms. For more details about the *vperm* instruction, refer to the *CEVA-XM4 Volume II Instruction Set* document.

5.3.4 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 5-1 describes the number of operations performed by each multiply operation.

Table 5-1: 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.

5.3.5 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*

5.4 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:

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

5.5 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:

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

5.6 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:

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

5.7 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.

5.7.1 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*

5.7.1.1 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, #uimmC4, vD.[u]s16,
#uimmE4, 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 vD contains the filter coefficients (16 short values).
Immediate C indicates the starting offset for {vA,vB}.
Immediate E indicates the starting offset for vD.
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.

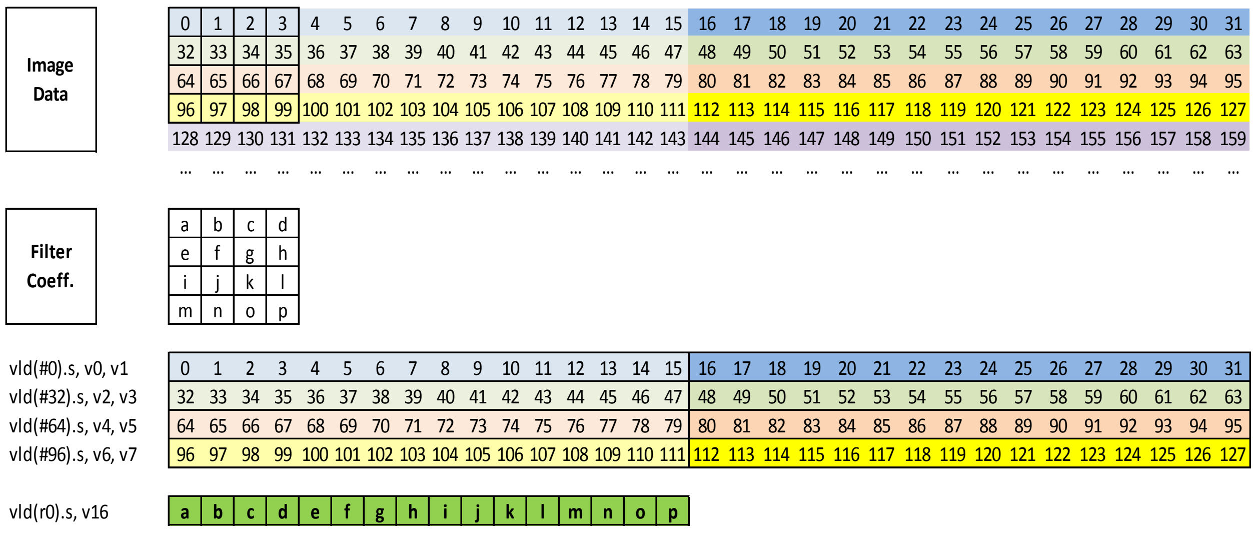


Image Data	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…

Filter Coeff.	a	b	c	d
	e	f	g	h
	i	j	k	l
	m	n	o	p

vld(#0).s, v0, v1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
vld(#32).s, v2, v3	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
vld(#64).s, v4, v5	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
vld(#96).s, v6, v7	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

vld(r0).s, v16	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p

Figure 5-2: Filter Coefficients

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

```
vswmac5  v0.s16, v1.s16, #0, v16.s16, #0, vacc0.i8
vswmac5  v2.s16, v3.s16, #0, v16.s16, #4, vacc0.i8
vswmac5  v4.s16, v5.s16, #0, v16.s16, #8, vacc0.i8
vswmac5  v6.s16, v7.s16, #0, v16.s16, #12, vacc0.i8
```

1

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

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**							

2

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

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**							

3

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

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**							

4

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

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**							

Figure 5-3: 4X4 Filter (Example 1)

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.

```
vswmac5  v0.s16, v1.s16, #8, v16.s16, #0, vacc0.i8
vswmac5  v2.s16, v3.s16, #8, v16.s16, #4, vacc0.i8
vswmac5  v4.s16, v5.s16, #8, v16.s16, #8, vacc0.i8
vswmac5  v6.s16, v7.s16, #8, v16.s16, #12, vacc0.i8
```

5

vswmac5 v0.s16, v1.s16, #8, v16.s16, #0, vacc1.i8

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		vacc1							

6

vswmac5 v2.s16, v3.s16, #8, v16.s16, #4, vacc1.i8

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		vacc1							

7

vswmac5 v4.s16, v5.s16, #8, v16.s16, #8, vacc1.i8

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		vacc1							

8

vswmac5 v6.s16, v7.s16, #8, v16.s16, #12, vacc1.i8

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		vacc1							

Figure 5-4: 4X4 Filter (Example 2)

5.7.2 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 4X2 grid.

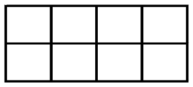


Figure 5-5: Sliding Pattern Grid

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

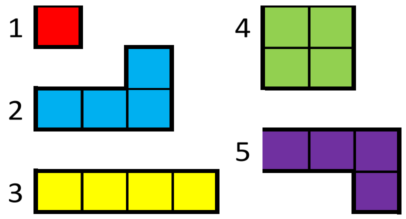


Figure 5-6: Pattern Examples

These patterns can be used for example to efficiently define coefficients for a 5X5 filter. The example below defines a 5X5 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.

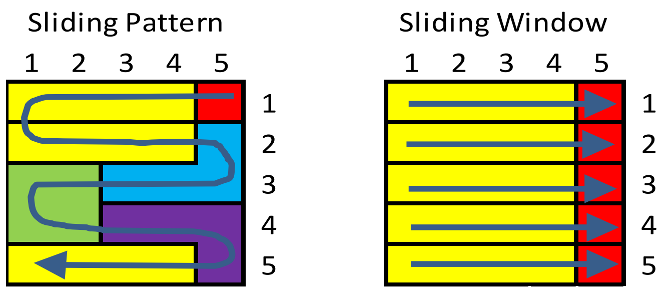


Figure 5-7: Sliding Pattern Efficiency

5.7.3 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 5x5 to 15x15 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.

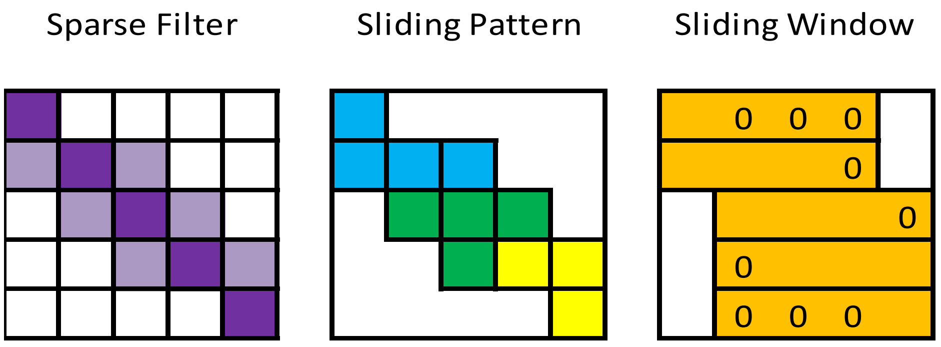


Figure 5-8: Sparse Filter Implementation

5.7.4 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:

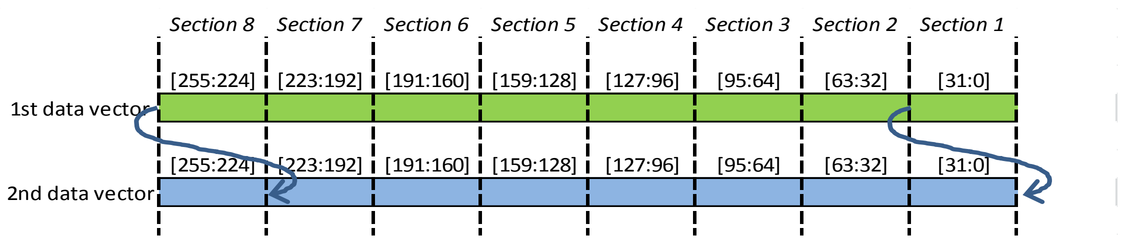


Figure 5-9: Multi-scalar Sliding Window

Each section has dedicated values in the coefficient vector.

5.7.4.1 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: $C = A \cdot (1 - \alpha) + B \cdot \alpha$

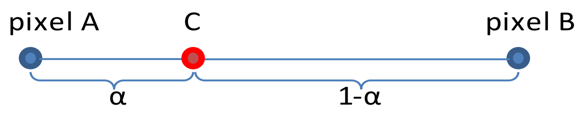


Figure 5-10: Linear Approximation

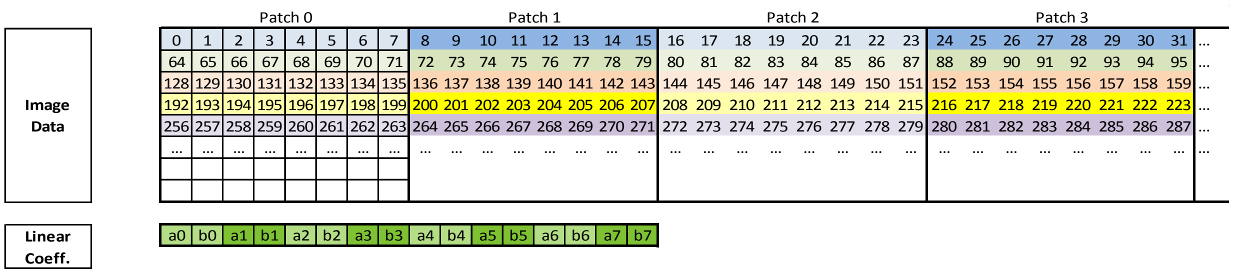
Both data and coefficients throughout the example are chars (eight-bits wide). The destination accumulates 32 short results (16-bits wide). Image dimensions are eight patches of 8X8 pixels.

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)

5.7.4.2 Multi-scalar Sliding-window Implementation

A linesr interpolation implementation memory map shows eight 8X8 image patches:

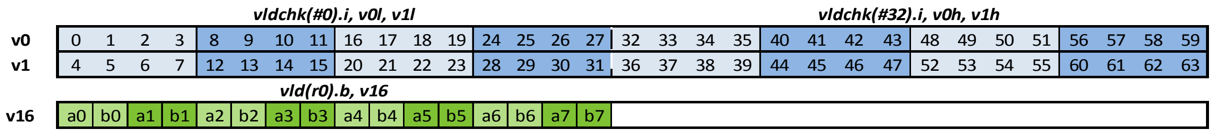


	Patch 0								Patch 1								Patch 2								Patch 3								
Image Data	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	...
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	...
	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	...
	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	...
	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	...
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Linear Coeff.	a0	b0	a1	b1	a2	b2	a3	b3	a4	b4	a5	b5	a6	b6	a7	b7																	

Figure 5-11: Linear Interpolation Implementation Memory Map

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).

Filter coefficients are loaded into v16.

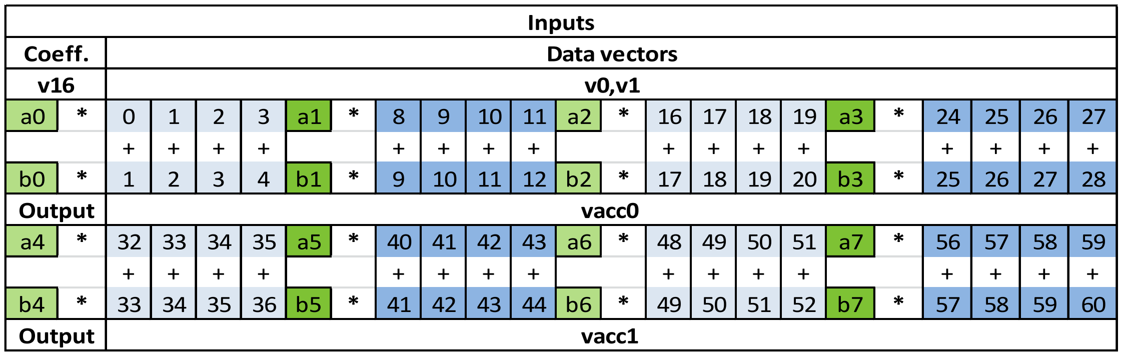


	vldchk(#0).i, v0l, v1l																vldchk(#32).i, v0h, v1h															
v0	0	1	2	3	8	9	10	11	16	17	18	19	24	25	26	27	32	33	34	35	40	41	42	43	48	49	50	51	56	57	58	59
v1	4	5	6	7	12	13	14	15	20	21	22	23	28	29	30	31	36	37	38	39	44	45	46	47	52	53	54	55	60	61	62	63
	vld(r0).b, v16																															
v16	a0	b0	a1	b1	a2	b2	a3	b3	a4	b4	a5	b5	a6	b6	a7	b7																

Figure 5-12: Filter Coefficients

Each *vmswmac3* instruction processes four offsets; therefore, two instructions are required to complete an 8X1 linear interpolation.

```
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																					

Figure 5-13: 8X1 Linear Interpolation (Instruction 1)

```
vmswmac3  v0.c32, v1.c32, #4, v16.c32, #0, 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																							

Figure 5-14: 8X1 Linear Interpolation (Instruction 2)

5.7.5 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 - vswsad instruction.

5.7.6 Vector Sliding Window Subtraction Operations

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

vswsubcmp and *vswsubsat*

5.7.6.1 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.

5.7.6.2 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.

5.7.7 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.

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.

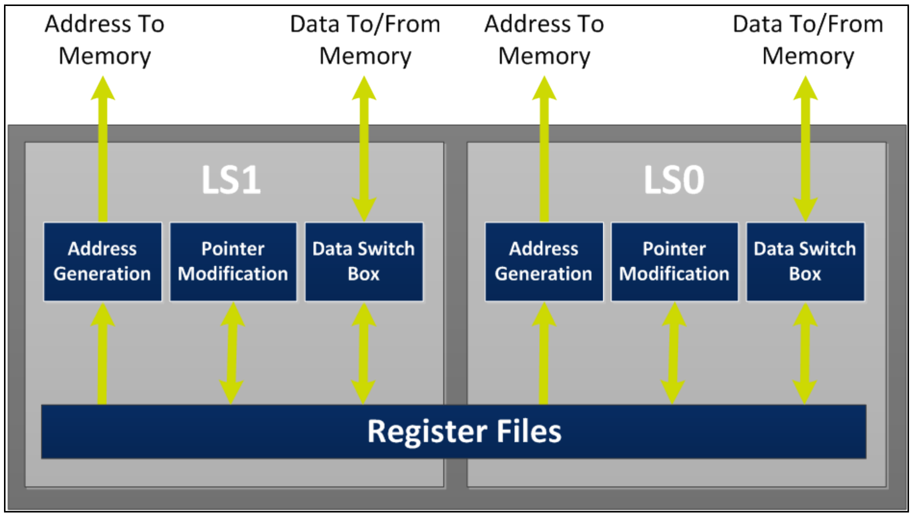


Figure 6-1: LSU Block Diagram

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.

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 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.

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 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.

6.4.3 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.

6.4.4 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.

6.4.5 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.

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. The pointer is modified after the address has been generated. Thus, this modification is referred to as post-modification.

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

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.

8.1 Integration with the Scalar Processing Unit

The SPU can be integrated with the CEVA-Xtend unit. The CEVA-Xtend (XH) unit has two source registers and one destination register. The sources are read from the General Register File (GRF) and the results are written to the GRF, making them available to any other unit within the core.

The interface enables the user to activate the CEVA-Xtend Scalar unit simultaneously and in parallel with other units in the core. When a CEVA-Xtend instruction is issued, it is done on the expense of SPU0.

8.2 Integration with the Vector Processing Unit

The VPU can be integrated with the CEVA-Xtend unit. CEVA-Xtend (VXH) unit has two source input vectors and one destination vector. The destination is output directly to the Vector Register File (VRF).

The interface enables the user to activate the CEVA-Xtend simultaneously and in parallel with other core units. CEVA-Xtend instruction is issued on the expense of VPU0.

8.3 Operation Modes

The CEVA-Xtend can operate in two modes: Synchronous mode and Trigger (Asynchronous) mode.

8.3.1 Synchronous Mode

In Synchronous mode, the instruction executions are coupled with the core pipeline stages, and the results can be sampled at each of the pipeline stages. The XH is coupled to SPU0 and the results are sampled at the V1 pipeline stage. The VXH is coupled to VPU0 and the result is sampled at the V5 pipeline stage. The sampling stage is encoded within the instruction opcode. This means that the result must be ready and valid on the output bus for proper functionality.

The sampling stage is determined automatically by the assembler tool from the instruction opcode, which specifies the destination too. The destination serves as a general register when coupled to XH or a vector when coupled to VXH. At the specified pipeline stage, the core samples the input bus from the CEVA-Xtend and writes the contents to the specified register or vector.

In Synchronous mode, the tools can embed the instruction behavior and information within the cycle-accurate simulator and debugger.

Example 8-1: CEVA-Xtend Synchronous Mode

```
VXH.vflip vA.[u]c32, vB.[u]c32, vZ.c32
```

The dispatch mechanism identifies the instruction as a VXH instruction (coupled to the VPU0) and the VXH unit decodes the instruction as a *vflip* instruction. Two sources are read from the vectors and the result of the calculation is written to the destination vector at the pipeline stage, which is specified in the VXH opcode.

Note that *vflip* is just example of possible syntax, the instruction syntax and operation is defined by the user.

8.3.2 Trigger (Asynchronous) Mode

In Trigger (Asynchronous) mode, the instructions are coupled with the core pipeline clock. However, the execution results are not sampled back to the core, but rather saved internally within the CEVA-Xtend registers or vectors. Trigger mode is intended for executions that require more stages than Synchronous mode's defined pipe stages, in order to complete the operations. While in Trigger mode, the execution of such instructions takes more cycles than the pipe stages and they are not sampled back to the core. Therefore, such information cannot be represented within the software IP tools. In Trigger mode, the user can use the user-define bits presented in the opcode to encode the Xtend's internal register or vector destinations.

Eventually, in order to move the results of the trigger operation from the CEVA-Xtend back to the core's internal register or vector, an additional independent Synchronous move-type instruction must be issued. The *move* instruction can be issued once CEVA-Xtend operation is completed. An indication for completing the operation can be handled in the following ways:

- Insertion of an interrupt by the CEVA-Xtend unit once the operation is completed.
- Polling by the core on the user input pins. The indication is raised by the CEVA-Xtend and can be cleared by one of the output user-indication pins. It is recommended to latch the input pin by an external device.
- If the exact number of cycles required to complete the operation is known, no polling is needed. A Synchronous move-type instruction can be issued when the operation is assumed to be completed.

Example 8-2: CEVA-Xtend Trigger Mode

1. Using an Interrupt:

```
VXH.vmpg4 vA.c32, vB.c32, vxZ.c32 ; assuming veZ is an VXH
internal vector (user-defined encoding)
 …                                                           ; as
soon as the operation is done, CXV_INT is asserted for one
cycle.
 …
```

Int_VXH_ service:

```
VXH.mov vxA.c32, vZ.c32                  ; the result is moved
from vx to VRF using a Synchronous mov-type

; instruction.
reti
```

2. Using Polling:

```
VXH.vmpg4 vA.c32, vB.c32, vxZ.c32 ; assuming veZ is an VXH
internal vector (user-defined encoding)
 …                                    ; as soon as the
operation is done, CXV_UI is asserted for one cycle
                      ; latched inside the GPIN MSS register.
_poll_loop:
  in {cmp} #gpin_reg, r0.ui                    ; read the GPIN
MSS register.
  nop #2
  tsts {bit} r0.ui, #bit, pr0                       ; update
pr0 if GPIN MSS[#bit] is set.
  br _poll_loop , ?pr0                             ; branch
back to polling loop.
VXH.mov vxA.c32, vZ.c32                  ; the result is moved
from vx to VRF using a Synchronous mov-type

; instruction.
```

3. Exact Operation Cycle Count is Known:

```
VXH.vmpg4 vA.c32, vB.c32, vxZ.c32 ; assuming veZ is an VXH
internal vector (user-defined encoding)
 inst1                                 ; operation is assumed
to be done in five cycles.
 inst2
 inst3
 inst4
VXH.mov vxA.c32, vZ.c32                  ; the result is moved
from vx to VRF using a Synchronous mov-type

; instruction.
```

9. 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. The following describes 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

9.1 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.

9.2 Dispatch / Decode Stages

- **D1:** The core selects a packet of 32-bit double-words 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.

9.3 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.

9.4 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, SPUs operations and VPU decoding are executed in this stage.
- **E2:** SPUs operations are executed. VPUs operations' register files are read.

9.5 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.

9.6 Pipeline Examples

9.6.1 Scalar Unit Instruction Flow

Figure 9-1 describes the pipeline flow of an instruction executed in the SPU.

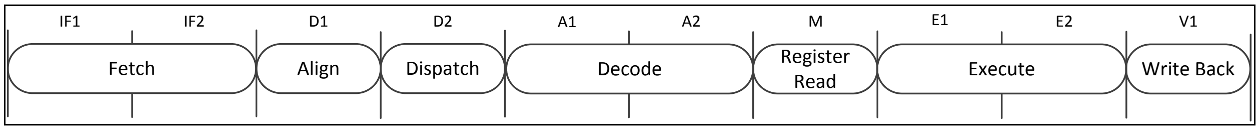


Figure 9-1: SPU Pipeline Flow

9.6.2 Data Load Instruction Flow

Figure 9-2 describes the pipeline flow of a load instruction.

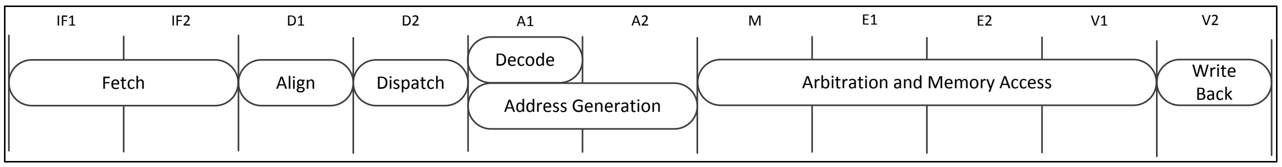


Figure 9-2: Load Instruction Pipeline Flow

9.6.3 VPU Instruction Flow

Figure 9-3 describes the pipeline flow of a three-cycle VPU instruction.

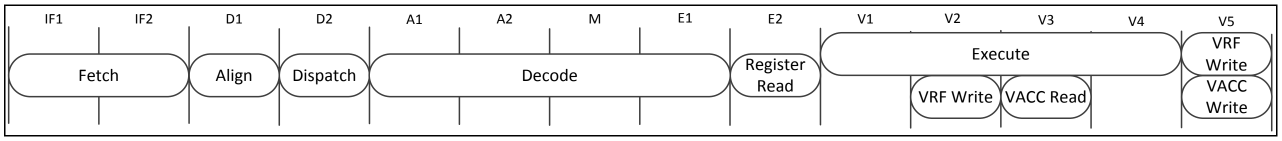


Figure 9-3: VPU Pipeline Flow

10. MSS Introduction

The CEVA-XM4™ MSS (Memory Subsystem) consists of a Program Memory Subsystem (PMSS) and Data Memory Subsystem (DMSS). Both the PMSS and DMSS have local memories and external interfaces. In addition, the PMSS has a four way set associative cache. Core accesses to the internal (local) memory have no wait-states, and external memory accesses may require several wait-states. The core supports up to 4 GB of unified memory space, and has up to four separate physical interfaces: up to three for data memory and one for program memory. This enables the core to access simultaneously both the program and data memories. The MSS provides the core with simultaneous accesses to one instruction fetch stream and two data fetch streams. The Data Address and Arithmetic Unit (DAAU) is responsible for accessing the data memory, and the Program Control Unit (PCU) is responsible for accessing the program memory.

The MSS contains standard interfaces for connecting the core to external devices and/or peripherals. These interfaces include up to four AXI master ports (program and data), up to five AXI slave ports and an APB3 port. All ports are fully compliant with Advanced Microcontroller Bus Architecture (AMBA). Additionally, the MSS contains JTAG and APB3 interfaces for debug support. A block diagram of the MSS structure is illustrated in .

Advanced power management is a key feature in the CEVA-XM4 architecture. The Power Scaling Unit (PSU) controls all clock signals in the system and facilitates power shutdown modes. PSU features enable the user to obtain the required application performance while minimizing power consumption.

10.1 Overview

The core can access two different types of memories: internal memories, also referred to as local memories (on-chip) and external memories (which may be located on-chip and/or off-chip).

While the core accesses the internal memories with no wait-states, accessing the external memories may require several wait-states.

The MSS contains the following elements:

- Internal Data Memory (IDM)
- Data memory arbiters and controllers
- Data write buffer
- Data DMA (DDMA)
- DDMA Manager containing a Queue Manager
- Internal Program Memory (IPM)
- Program memory arbiters and controllers
- Instruction cache (tag and set array) memory
- Program DMA (PDMA)
- Up to 10 AMBA external ports:
  - Configurable AMBA AXI4 program space (master)
  - Up to three configurable AMBA AXI4 data space (master)
  - Up to five configurable AMBA AXI4 external data (slave)
  - APB3 I/O space
- Multi-core support mechanism
- On-chip Emulation Module (OCEM)
- Power Scaling Unit (PSU)
- Optional Error Correction (ECC) on all PMSS RAM blocks
- Optional Error Correction on all DMSS RAM blocks
- Optional Error Correction on all AMBA AXI4 external ports

The MSS can be connected to the system via standard bridges using a bridge layer between the system and the MSS.

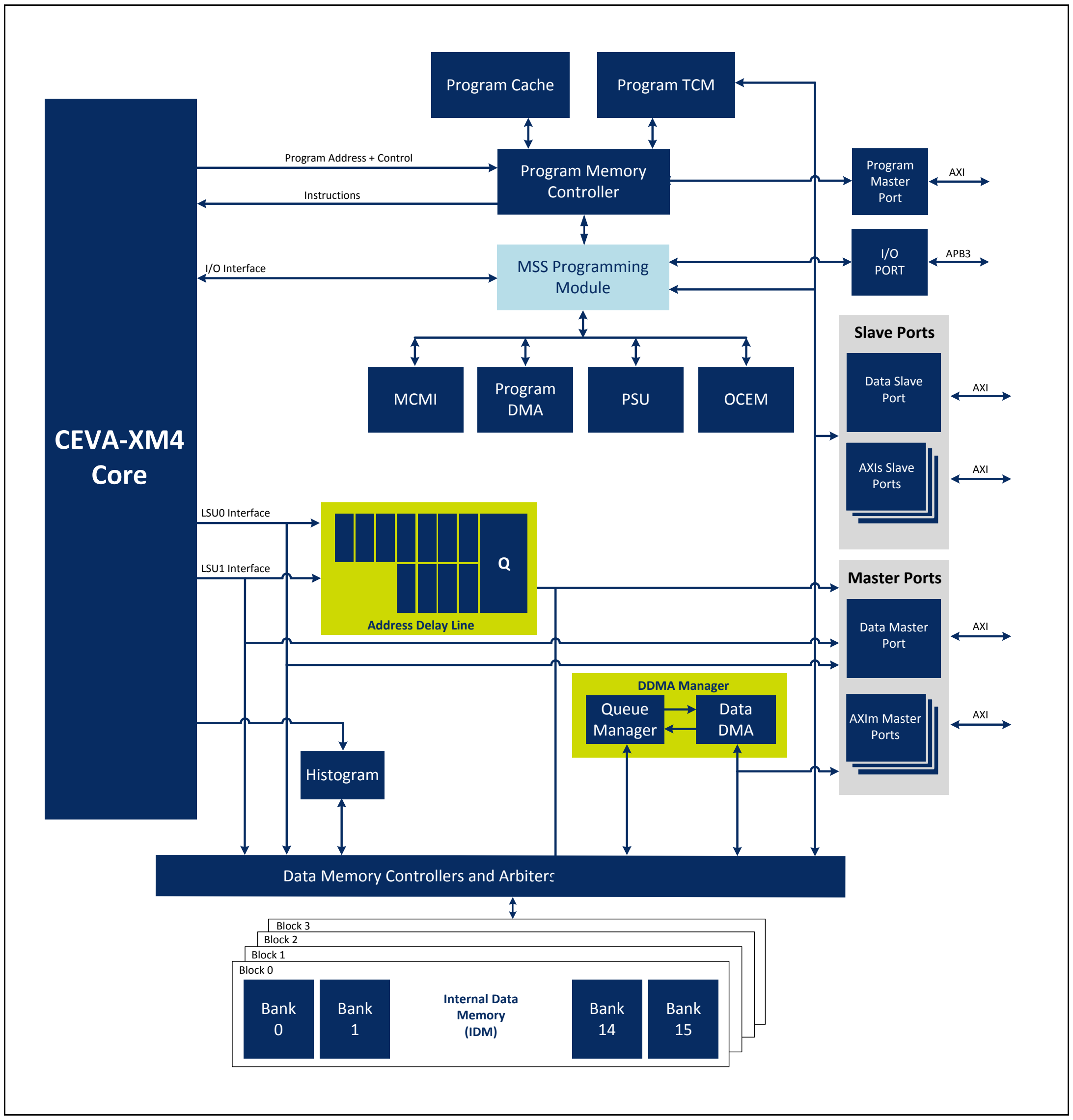


Figure 10-1: CEVA-XM4 MSS Structure

10.2 Program Memory Subsystem

10.2.1 Introduction

The Program Memory Subsystem (PMSS) comprises local memory and optional four-way set-associative cache memory. The PMSS provides the core with instructions from either local memory, the cache or external memory. The PMSS supports Error Correction and Detection (ECC) on local RAMs, thereby providing extra system robustness.

The PMSS is both hardware- and software-configurable, thus providing both hardware and software designers complete control over its structure, functionality and performance. End users have control over the availability of different components (such as local memory, cache and ECC), as well as their sizes.

The PMSS supports both highly predictable execution times, by placing instructions in local memory or in a locked cache line, and less predictable execution times, by placing instructions in the cache or from external memory directly. Software designers have complete control over the location of instructions within the PMSS. For example, time-critical code can run from local memory or from a locked cache line. In contrast, less-time-critical code can run from the cache or from external memory directly. In this way, software designers can benefit from both the highly predictable execution provided by local memory and the flexible software development capabilities provided by a cache.

The PMSS is designed to achieve a balance between high performance, small area and low power consumption, thus allowing it to be used in a wide variety of systems and applications. The PMSS provides tradeoffs between performance, area and power via hardware configurability. The size of the local memory and inclusion of the cache are configurable, based on performance requirements and area constraints.

The PMSS is connected to an external memory controller via an AMBA4 interface. The cache consists of a memory set that holds the actual cached line data, and a tag memory, which is used to determine whether an address is cached and whether the instruction is valid. Tightly Coupled Memory (TCM) is also supported and is accessed with zero wait-states.

The PMSS contains the following components:

- **Internal Program Memory:** Comprised of fast memories operating at the core frequency. The IPM consists of a local memory and a four-way set-associative cache.
- **Program Memory Controller:** Responsible for the correct access of the various sources to the internal TCM and cache.
- **Program DMA:** Enables downloading of code to the TCM.

- **External Program Port:** Enables access to the external program memory using a 128-bit-wide AXI master port. Any attempt by PDMA requests and cache misses to access the program memory outside the internal memory boundaries causes an external transaction request.
- **On-chip Emulation:** The OCEM module is a debug interface with direct access to PMSS EPP reads and writes, cache sets, tags, the TCM memory and the core queue. This performed in Debug mode.
- **Instruction Memory Access Protection:** The access protection models protect read accesses from accessing a non-permitted address in memory, according to the core access type (supervisor, user0 or user1). An access protection violation is generated if the requested access is illegal.
- **Software Operation Unit (SOU):** Performs software operations on the instruction cache. The SOU can perform software pre-fetch, invalidate, lock and unlock operations on the instruction cache.
- **Instruction Access Control Unit (IACU):** The IACU determines the behavior of the core fetch lines based on their addresses.
- **Undefined Opcodes:** Handles undefined opcodes (that is, opcodes that are not mapped to any of the core's instructions).

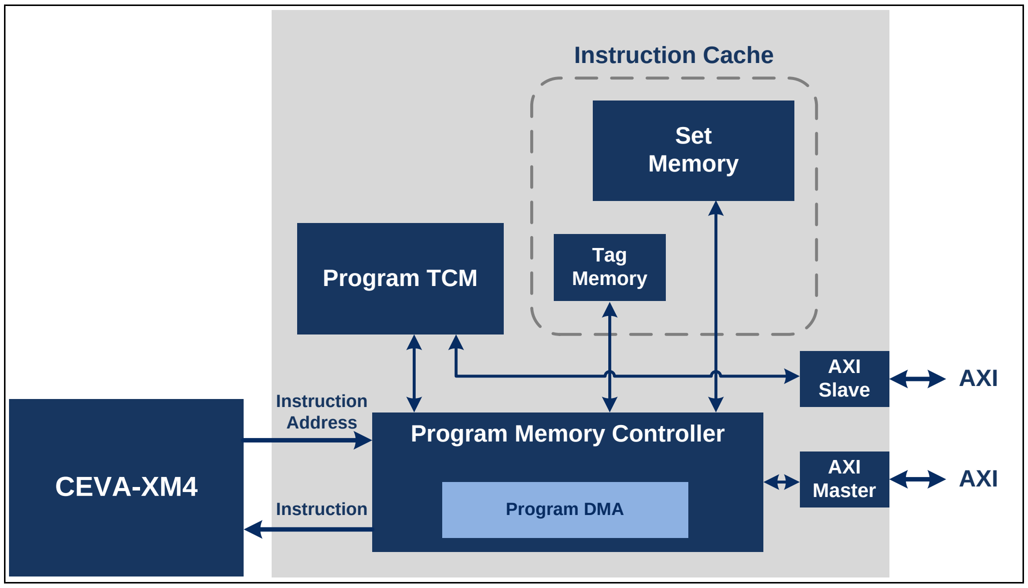


Figure 10-2: Memory Subsystem and Core System Example

10.2.2 Feature List

The PMSS supports the following main features:

- 4GB program memory linear space
- Configurable TCM memory size
- Software and hardware pre-fetch capability
- Program DMA (PDMA)
- Optional four-way set-associative instruction cache
- Cache software operations (invalidation, pre-fetch, lock and unlock)
- 128-bit AXI master port
- Optional error correction (ECC) one error correction and two error detection on all RAM and cache blocks
- Non-blocking cache and fetch-line access
- Handling of undefined opcodes

10.3 Data Memory Subsystem

10.3.1 Introduction

The Data Memory Subsystem (DMSS) provides data from either local memory or external memory to the CEVA-XM4 core. The DMSS supports error correction and detection on local data RAMs and on AMBA buses thereby providing extra system robustness.

The DMSS is both hardware- and software-configurable, thus providing both hardware and software designers complete control over its structure, functionality and performance. End users have control over the availability of different components (including number of AMBA ports, size of local RAMs and ECC).

The DMSS supports highly predictable execution times by placing data in local memory. Software designers have complete control over the location of data within the DMSS. For example, time-critical data can run from local memory while less time-critical data can run from external memory.

The DMSS is designed to achieve a balance between high performance, small area and low power consumption, thus allowing its use in a wide variety of systems and applications.

The core can issue up to two load or store operations simultaneously. The Scalar Processing Unit (SPU) and the Vector Processing Unit (VPU) share the same Load and Store Unit (LSU) resources.

The LS0 unit connects to the DMSS using 256-bit read and write buses and is used by the SPU and VPU for load and store operations.

The LS1 unit connects to the DMSS using a 256-bit read bus and a 64-bit write bus. LS1 is used by the SPU for load and store operations. LS1 is used by the VPU for load operations only.

The SPU accesses the data memory with byte, two bytes, four bytes or eight bytes without any alignment restriction. The VPU accesses the data memory from bytes up to 16 words without alignment restrictions for standard (non-parallel) accesses. Parallel vector accesses have some alignment restrictions. See the ISA in Volume II for details about any instruction-specific alignment restrictions.

Table 10-1: VCU Accesses Alignment Requirement

Operation	CEVA-XM4	Alignment
Load	When 512 bits (full bandwidth)	DW alignment

The data memory space is subdivided into IDM and external memory.

The IDM, which is part of the MSS, contains a relatively small amount of memory, and is also referred to as local memory.

The external memory can be located on-chip and/or off-chip, and is accessed via one of the AXIm0/AXIm1 master ports or the external data port (EDP).

The data-related part of the MSS (DMSS) contains the following sections:

- **Unaligned Access:** CEVA-XM4 is designed to support unaligned access.
- **Memory Ordering Models (MOM):** Two memory-ordering models are supported: Total Store Ordering (TSO) and Strong Ordering (SO).
- **Memory Access Protection:** The core access is compared with the operation mode and the access type permitted for this section of memory. An access protection violation is generated if the requested access is illegal.
- **Internal Data Memory:** This is composed of fast SRAM memories operating at the core frequency.
- **Histogram:** The Histogram block provides hardware acceleration for the histogram function.
- **Data DMA:** Supports the exchange of data between the external memory (via the EDP or AXIm0/1 master ports) and the IDM. The DMA can be configured for downloading or uploading data with up to three pending transactions. The transfer width is up to 256 bits (according to the master port used).
- **Queue Manager:** Offloads DDMA-related real-time tasks and performs them in the hardware.
- **Data Access Control Unit (DACU):** A set of registers that determines the behavior of data elements according to their addresses.

- **External Device Access Port (AXI Slave):** A single slave port with separate read and write interfaces that enables external devices (for example, the external DMA) to access the IDM.
- **Data Write Buffer:** Administrates the core write transactions in order to reduce memory conflicts and assure the correct write order.
- **Memory Arbiters and Controllers:** Responsible for the correct access of the different sources to the internal memory.
- **AXIsX Slave Ports (AXI Slave):** Up to three single slave ports with separate read and write interfaces that enable external devices (for example, the external DMA) to access the IDM.
- **AXIm0/1 Master Ports (AXI Master):** Two single master ports with separate read and write interfaces (with the same bandwidth as the data DMA) that are used to access the external data memory.
- **External Data Port (EDP - AXI Master):** A single master port with a separate read and write interface (with the same bandwidth as the data DMA), which is used to access the external data memory. The DMSS accesses the external memory via a dedicated AXI interface port.

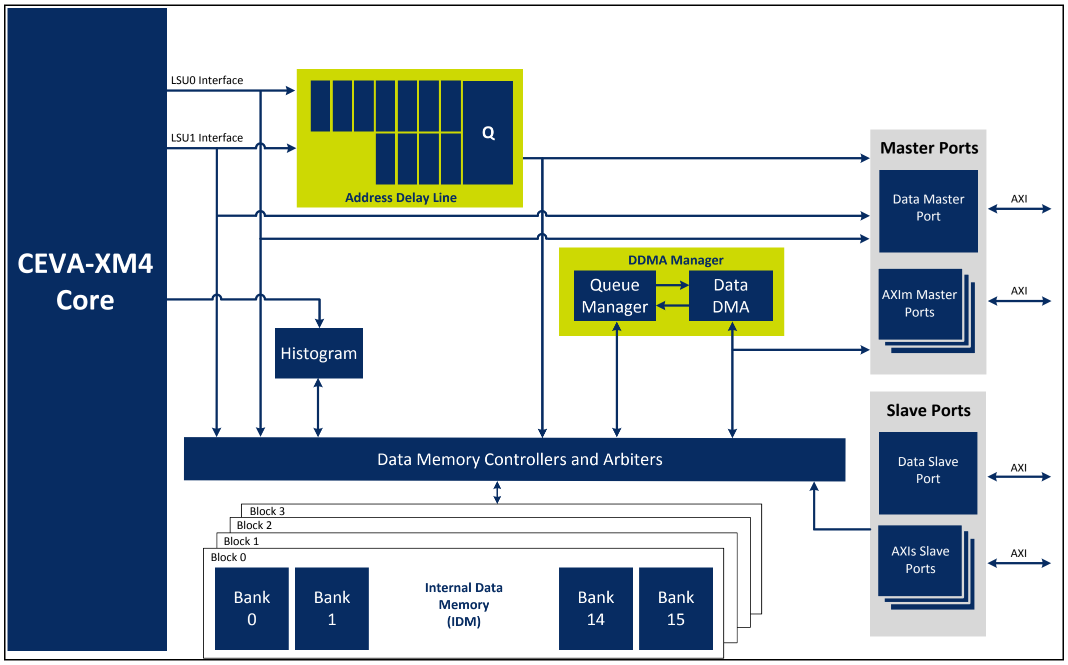


Figure 10-3: CEVA-XM4 Data MSS (DMSS)

The following sections describe these modules in more detail.

10.3.2 Feature List

10.3.2.1 Data Address Space

The DMSS supports a unified 4GB data address space. The address space is unrestricted, which means that there are no predefined dedicated address ranges and the entire address space can be used for any purpose. This feature is very important in multiprocessor systems that consist of other processors with address-space restrictions.

10.3.2.2 Parallel Memory Access

In addition to the standard dual-memory access operation, the DMSS supports parallel memory access allowing up to 32 different memory address accesses in a single access. The DMSS supports the following operations:

- Up to 16 aligned load/store operations per LSU (total of 32 load and 16 store operations)
- Up to eight unaligned load/store operations per LSU (total of 16 load and eight store operations)

The following restrictions apply for the aligned load and store operations:

- Bank contention is not allowed when using both LSUs. Each LSU must access different memory banks.
- Address must be aligned to the access width.
- Each address accesses a separate memory bank in the memory block. When using absolute addressing, it is the user’s responsibility to ensure that no bank conflicts occur.

The following restrictions apply for the unaligned load and store operations:

- Block contention is not allowed when using both LSUs. Each LSU must access a different memory block.
- Each address accesses a pair of memory banks in the memory block (0/1, 2/3 … 14/15). When using absolute addressing, it is the user’s responsibility to ensure that no bank conflict occurs.

10.3.2.3 Data Access Control Unit

The CEVA-XM4 MSS architecture allows complete control over the location and characteristics of data within the data address space.

10.3.2.4 Memory Access Protection

The MSS provides memory access protection for data accesses (external or internal). The type of access allowed is compared with the core operation mode and the access type (read/write for data accesses), and an access protection violation is generated if the requested access is illegal.

10.3.2.5 L2 Cache Control

L2 caches are external components that can connect between the MSS AXI port and the external system. Users have full control over this component using AXI L2 cache signals.

10.3.2.6 Power Scaling Unit

The Power Scaling Unit (PSU) provides the ability to reduce the dynamic power by controlling clock gating as well as reducing the leakage power by allowing power gating.

10.3.2.7 Unaligned Access

The CEVA-XM4 is designed to support unaligned access for load and store operations. The Unaligned mechanism does not require any penalty or performance reduction. This ability may reduce the code size and facilitate compiler code generation.

The CEVA-XM4 supports unaligned access in byte resolution for the SPU and unaligned access for most of VPU operations. Figure 10-4 shows two accesses.

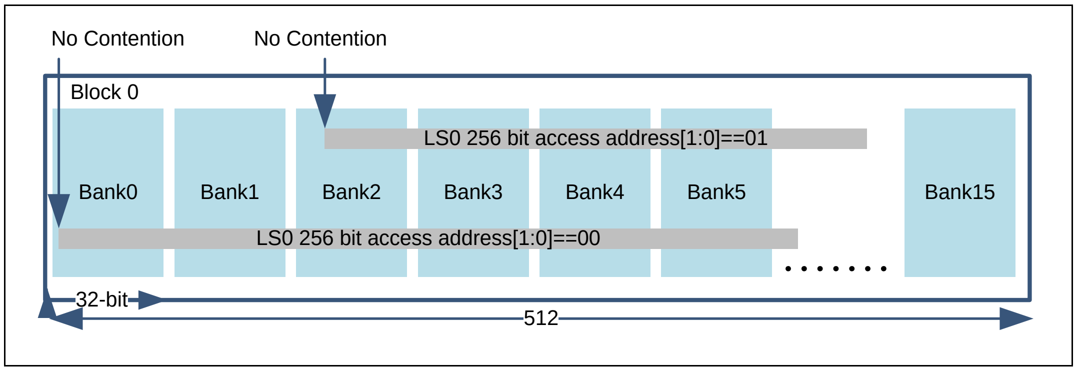


Figure 10-4: CEVA-XM4 IDM Contentions

Note: *It is restricted to have an unaligned address points to memory locations that are accessing both internal and external memory.*

10.3.2.8 Write Order Protection

When writing data to external and internal memory, older internal memory can bypass earlier external writes and vice versa. The user can protect the order by using an internal barrier.

10.3.2.9 ECC

The CEVA-XM4 supports configurable ECC schemes to protect content of internal data RAMS and AMBA bus signals. Internal data RAMs use a Single Error Correct Double Error Detect (SEC-DED) ECC scheme to protect memory content. AMBA bus ECC uses both parity and SEC-DED to protect AMBA signals.

10.3.3 Data Access Control Unit

The Data Access Control Unit (DACU) determines the location and access port of the requested address within the 4GB address space. In addition, it provides:

- DMSS attributes (memory ordering model and memory access protection to the specified memory areas)
- L2 cache attributes (see the *ARM AXI Specification* for details)

The 4GB address space is divided into up to 32 DACU pages, whose attributes define the location, size and characteristics of the data element within the page. The DACU checks the request addresses from the core (both ports are checked simultaneously), and determines the page where the data element is located.

10.3.4 Internal Data Memory

Because multiple sources can access the IDM simultaneously, it is divided into four blocks in order to reduce the possibility of conflicts between the various accesses. If a block and/or bank conflict occurs, a wait cycle is inserted to the non-granted source(s).

The IDM can be configured, as follows:

- Either 1024KB, 512KB, 256KB or 128KB memory size configuration.
- Four block memory structure.
- Total block memory width is the same as the 2xLS unit width.
- Each block is composed of 16 memory banks. The addresses are interleaved over the banks within each block, in the same row boundaries.
- Each bank is 32-bits wide (4-byte write enabled).
- **Block Bandwidth:** 2xLS bandwidth, which is the bandwidth of each of the LSUs.
- **Number of Blocks:** The various units can transfer data located in various blocks without conflicts.

Figure 10-5 illustrates the IDM organization using four blocks.

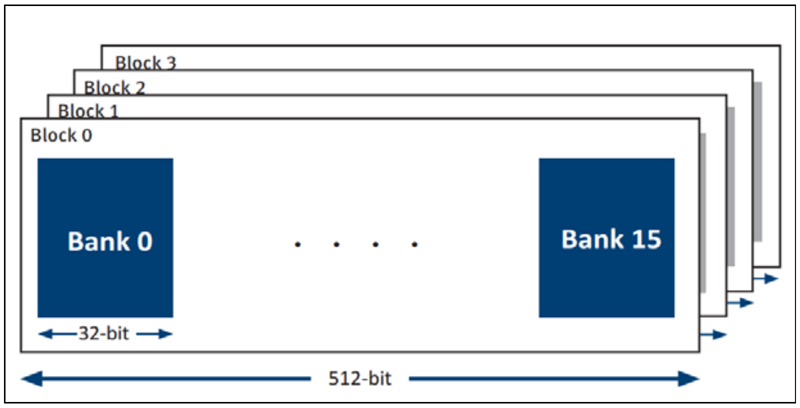


Figure 10-5: Four-block IDM Organization

10.3.5 Data Write Buffer

The Load Store Units (LSUs) issue the address and controls (read/write and size) at the end of the A2 stage of the pipeline. The data, in write transactions, is available at the end of the V1 stage, when the source is the SPU unit, and at the end of the V5 stage when the sources are the VPU unit, four or seven cycles later, respectively. An address delay mechanism is implemented in order to ensure the proper functionality of the memory during write transactions.

The address delay is used to synchronize the data with its respective address. When ready, the address along with the data and transaction attributes are pushed into one of the queue buffers that administer the pending write transactions. The Write Buffer (WB) queue reduces core wait cycles that may occur as a result of memory bank conflicts.

The WB unit is divided into the following functional blocks:

- LS0 and LS1 address delay line
- One queue
- LS0 and LS1 read-after-write and strong ordering (RAW/SO) match blocks
- External Output Stage (EOS), which buffers between the queues and the external ports (EDP/AXI)
- Internal Output Stage (IOS), which buffers between the L1DM queue and the internal data TCM

Figure 10-6 illustrates the write buffer structure.

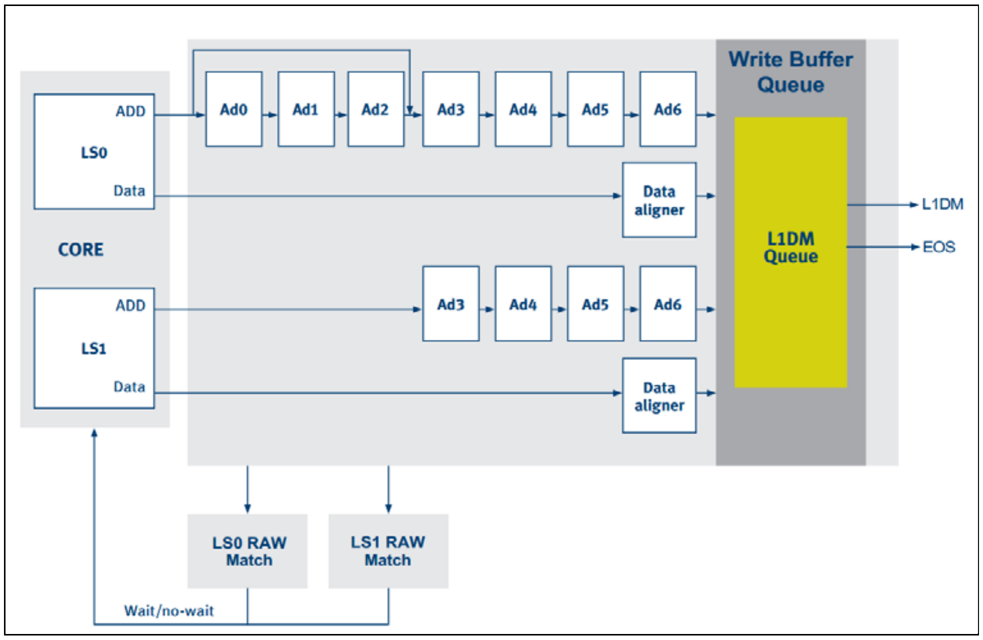


Figure 10-6: Write Buffer Block Diagram

The WB queues reduce the number of core stall cycles caused by the write transactions (but do not eliminate all of them), by delaying the actual write transactions until the target can be written to. The delayed transactions are stored in the queue, and can be delayed as long as the queue is not full and there are no more writes at the top of the ADLs. If the queue becomes full and must release a transaction but is not granted write access to the L1DM then the transaction in the queue is written to the IOS and the core is halted until the IOS is granted write access to the TCM. If the transaction does not leave the queue, for example, for EDP transactions where the EOS is full, the core is halted until the queue frees the necessary space.

The RAW/SO match mechanism ensures data coherence. This means that reads do not bypass writes targeting the same address or the same memory region for SO regions.

Note: *The WB manages all non-IO core write transactions, regardless of whether their destination is the internal memory or external data memory.*

10.3.6 Histogram Acceleration Customer

A histogram (HIST) is used to plot the frequency of score occurrences in continuous data that has been divided into classes/bins.

The XM4 core histogram uses:

- A histogram of the image: Counts the number of the pixels of each color.
- A weight histogram: Counts the weight of the pixels, where each pixel has a weight according to the distance from a reference pixel.

10.3.7 Data DMA

The DMA transfers data between the external memory and the IDM or within the IDM. It accesses external memory via the EDP, AXIm0 and AXIm1 Master ports.

The DDMA has the following capabilities:

- Transfer Directions:
  - Moves data from the IDM to external memory.
  - Moves data from external memory to the IDM.
  - Moves data within the IDM.

- Transfer Access Types:
  - **Message:** Destination is written with a double-word configured at the DDMA task descriptor.
  - **Linear:** Source or destination data is accessed in sequential order.
  - **Two-dimensional:** Source or destination data is accessed as an arbitrary two-dimensional tile.
  - **One-bank Tile:** Source or destination data is accessed as a two-dimensional tile with a double-word or two double-words width. The source or destination must reside within the IDM and the tile is allocated to a single or two adjacent IDM banks.
  - **Duplicate Bank Transfer:** Destination data is duplicated to 16 or eight bank tiles (double-word or two double-words wide).The destination must reside within the IDM.
- **Contention-less IDM Access:** The data DMA, EDAP, AXI slave port histograms and the core DSP can simultaneously access different blocks without memory conflicts.
- **DMA Queue:** The DDMA can accept three pending requests using a dedicated queue.
- **Automatic External Port Selection:** The external port used (EDP or AXIm0/1) is selected by matching the external memory address with the address regions configured in the DACU.
- **Configurable Burst Type:** Supports INCR or FIXED addressing per DDMA task.
- **Configurable Maximum External Master Port Burst Size:** Maximum burst size of up to 256 transfers is supported.
- **Byte-aligned:** Supports byte-aligned addressing and a byte-resolution transfer size.
- Configurable Maximum External Master Port Outstanding Bursts:
  - Supports up to 16 outstanding read bursts.
  - Supports four, eight, 16 and 32 outstanding write bursts.

10.3.8 Queue Manager

The CEVA-XM4 data memory subsystem (DMSS) implements a special Queue Manager (QMAN) that enables the user to activate the DDMA without the use of real- time software.

The CEVA-XM4 QMAN is used for managing up to 12 queues containing DDMA task descriptors. Each queue holds DDMA tasks meant to be executed in a strict sequential order (task B always follows task A). The queues are located at the IDM or at an external memory address that is configurable per queue.

The CEVA-XM4 QMAN reads head task descriptors from all queues, fragments the tasks into smaller chunks (configurable chunk size per queue) and activates the DDMA to transfer the chunks after arbitrating between the queues. A special arbitration scheme enables the user to order the tasks sent to the DDMA according to the order of data consumption at the destination, as opposed to the order of data availability at the source.

DDMA tasks pushed into a queue can be delayed from being passed to the DDMA until the appropriate system conditions are met (data to be transferred is available at the source and destination memory is free). Two enable-task counters are implemented at each queue. One enable-task counter delays head tasks until data is available at the source, and the other delays head tasks until destination memory is free. The CEVA-XM4 QMAN waits until both counters are larger than zero before it sends the head task to the DDMA. To indicate that the next DDMA task in a queue can be executed, the source and the destination counters can be incremented, where up to 63 DDMA tasks can be enabled using a single CPM access.

In addition, enable-task counter 0 can be incremented using a dedicated input signal. Enable-task counter 0 is incremented with a configured value for every cycle where enable is set.

After sending the task to the DDMA, both counters are decremented.

The user can disable each of the counters when they are not required; meaning, when the data is known to be available at the source and/or the destination memory is known to be free at the time the DDMA task is pushed to the queue).

The CEVA-XM4 QMAN can directly access queues located in the CEVA-XM4 IDM or it can access queues located in external memory using one of the CEVA-XM4 master ports. The QMAN uses a single memory port shared by all queues for accessing the IDM or external memory. Other than that, each queue is autonomous, thereby enabling maximum utilization of the DDMA.

10.3.8.1 QMAN Feature List

The Queue Manager implements the following features:

- Configurable number of queues (0, 4, 8, 12).
- Programmable queue location and size (direct access to IDM or access to external memory via one of the CEVA-XM4 master ports).
- Simplified interface for pushing new descriptors to the queue with minimal software intervention.
- Separate descriptor push and activation stages (tasks pushed to the queue are later enabled according to the system state).

- Descriptor activation controlled either by dedicated hardware trigger or software activation.
- Pulls task descriptors from queues.
- Splits tasks into programmable chunk sizes per queue.
- Priority of queues arbitrating for DDMA access is based either on preconfigured queue priority or on priority in task descriptor.
- Queues with the same priority arbitrate according to data packet order of execution.
- Queues with the same priority and same execution order arbitrate using a round-robin policy.
- Queue semaphore enables sharing of a queue by several queue managers.
- Enables bundling of several tasks into a task frame.
- Dynamic scheduling enables several QMANs to pull and execute task frames from a single queue.

10.4 Multi-core Support

10.4.1 Overview

A multi-core processor is a processing system composed of two or more CEVA-XM4 homogenous and independent cores. In order to support shared memory coherence and smooth communication between the cores, the CEVA-XM4 provides dedicated external and internal synchronization mechanisms for that purpose.

10.4.2 Feature List

The CEVA-XM4 supports multi-core synchronization with the following features:

- Exclusive external memory access synchronization either by using the AXI interface or by using dedicated interface signals.
- Supports a multi-core messaging interface between cores.
- Supports IDM snooping mechanism

10.4.2.1 External Memory Exclusive Access Synchronization

The CEVA-XM4 core can perform an atomic read-modify-write operation using the AXI exclusive-access mechanism. This enables memory coherence and data sharing between processors and/or between multiple tasks.

To use AXI exclusive transactions, an external slave must implement the AXI monitor logic either on the bus fabric (for a single-port slave) or internally (for a multi-port slave). Also, it is recommended to implement a monitor unit per each exclusive-capable master ID that can access the slave.

AXI exclusive access enables a user to read a value from an external address, change the value and attempt to write it back. In cases where another master is accessing the same address (for a read or write) after the read operation, monitor hardware at the slave cancels the write operation and returns a fail indication to the master.

For more details about AXI exclusive access, see the AMBA® AXI protocol.

10.4.2.2 Multi-core Communication Interface

The Multi-core Communication Interface (MCCI) mechanism enables communication of commands and messaging between the cores. Communication between the cores is achieved by using the AXI slave ports to directly access dedicated command registers. The core has dedicated controls and indications to follow the status of the communication via the MCCI.

10.4.2.3 IDM Snooping Mechanism

By using the IDM snooping mechanism, the core can get an indication that an external device accessed its IDM in a predefined address range.

10.4.3 AXI Master Interface

The CEVA-XM4 MSS has four AXI master ports (EPP, EDP, AXIm0 and AXIm1), each with separate read and write interfaces. The External Program Port (EPP) connects the core with external program memory, and an External Data Port (EDP), AXI master Port 0 (AXIm0) and AXI master Port 1 (AXIm1) connect the core with external data memory or peripherals.

AXI master ports are fully compatible with AMBA AXI4 protocols and are implemented as an AXI master interface.

The data width of each EPP slave port can be 128 bits.

The data width of each EDP slave port can be 128 bits.

The data width of each AXIm0 and AXIm1 slave port can be 128 bits or 256 bits.

The following master configuration options exist:

- No AXIm0 and AXIm1 master port
- AXIm0 and AXIm1 master ports using a data width of 128 bits
- AXIm0 and AXIm1 master ports using a data width of 256 bits

10.5 Power Scaling Unit

10.5.1 Introduction

Sophisticated power management is a key feature in the CEVA-XM4 architecture. The Power Scaling Unit (PSU) provides the ability to reduce the dynamic power by controlling clock gating, as well as to reduce the leakage power by allowing voltage shutdown modes. The PSU controls the DSP and the memory subsystem clocks and provides a handshake with an external power management unit in order to facilitate power shutdown modes. This unit enables the user to both obtain the required application horsepower while minimizing power consumption.

10.5.2 Power Scaling Features

The PSU features are described below:

- **Free Run Mode:** All domains are active; clocks are gated according to activity.
- Dynamic power control modes:
  - **Dynamic Power Save (DPS):** Each unit receives clock only when it is active.
  - **Light Sleep Mode:** Core clock is in shut down while memories and ports are enabled to provide upload or download.
  - **Standby Mode:** Clock root is shut down externally.
- Leakage Power-save
- **Operational:** All is powered up.
- **Deep Sleep Mode:** Core and MSS power is shut down while memories power is retained.
- **Shutdown Mode:** Enables complete power shutdown without retention.
- **Operational-Production:** Debug block (OCEM and Real-time Trace [RTT]) power down.
- AXI low-power interface for enabling or disabling MSS clocks.

10.5.3 PSU States

The PSU generates the gated clocks to the core and to the MSS units according to activity. The operating mode of the PSU is configured by an *in/out* instruction from the core and through an external interface. Figure 10-7 illustrates the PSU modes. The interface and functionality are described in the following sections.

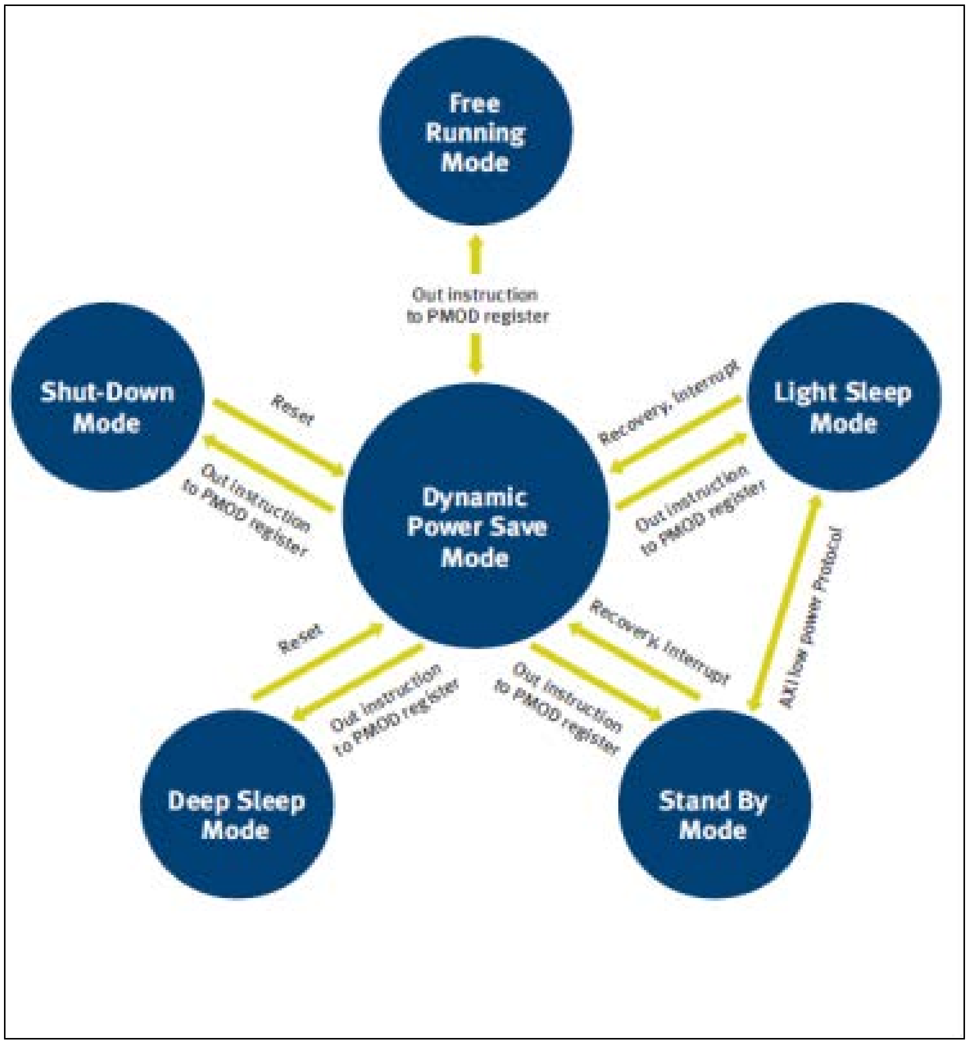


Figure 10-7: PSU State Diagram

11. 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.

12. 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.

12.1 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.

12.1.1 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.

12.1.2 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.

12.1.3 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.