User's Guide

C71x DSP CPU, Instruction Set, and Matrix Multiply Accelerator

TEXAS INSTRUMENTS

Table of Contents

List of Figures

List of Tables

2 Preface

2.1 C71x DSP CPU, Instruction Set, and Matrix Multiply Accelerator

The C7x is the next-generation fixed and floating-point DSP platform. The C7x DSP is a new core in the Texas Instruments DSP family. The C7x DSP provides vector signal processing, providing significant lift in DSP processing power over a broad range of general signal processing tasks in comparison to the C6x family of DSP. In addition, the C7x provides several specialized functions which accelerate targeted functions by 30 to 128 times. Besides expanding vector processing capabilities, the new C7x core also incorporates advanced techniques to improve control code efficiency and ease of programming, such as branch prediction, protected pipeline, precise exception and virtual memory management. This document describes the CPU architecture, pipeline, instruction set, and interrupts of the C7x DSP.

3 C71x CPU and ISA

3.1 Introduction

The C7x is the next-generation fixed and floating-point DSP platform. The C7x DSP is a new core in the Texas Instruments DSP family. The C7x DSP provides vector signal processing, providing significant lift in DSP processing power over a broad range of general signal processing tasks in comparison to the C6x family of DSP. In addition, the C7x provides several specialized functions which accelerate targeted functions by 30 to 128 times. Besides expanding vector processing capabilities, the new C7x core also incorporates advanced techniques to improve control code efficiency and ease of programming such as branch prediction, protected pipeline, precise exception and virtual memory management.

This document describes the CPU architecture, pipeline, instruction set, and interrupts of the C7x DSP.

The C7x builds on the C6x's legacy but it is not binary compatible with any previous C6x DSPs. Many of the C7x architecture features are new and unique. The C7x is designed as a platform with these characteristics:

- True 64-bit machine with 64-bit memory addressing and single-cycle 64-bit base arithmetic operations
- Achieves 4 to 8x or more DSP processing capacity compared to C66x
- Illustrates scalability of Architecture, Design development (and Software)
- Enables Keystone Hardware and Software platform reuse
- Preserves out-of-the-box performance parity for C code which has been optimized for C66x with a few exceptions
- Provides a direct software migration path from C66x
- Provides direct architecture support for accelerating OpenCL

The C7x platform can be configured to fit particular market needs. Potential configurations are:

- Optimal PPA per end market requirements with maximal software/IP reuse
- CPU core configurations
  - Variable vector width – 64, 128, 256, 512 etc.
  - Flexible data path with removable or replaceable functional units
  - Flexible functional units with removable functions, i.e Fixed vs Float etc.
  - Minimal scalar processor configuration

The first implementation of the C7x platform is a full-feature with 512-bit vector width CPU core. The overall goal of this first core is to provide 4 to 8 times or more the DSP capacity compared to the C66x for most DSP intensive kernels. All descriptions below refers to this full-feature 512-bit vector width core.

3.2 DSP Overview

Figure 3-1 is the block diagram for the C7x DSP. All C7x devices include L1 program and data memories. The L1 data memory can be configured as cache and/or SRAM. The devices also have varying sizes of local and shared L2 cache/SRAM as well as external memory interface(s). The number of instances, memory sizes and locations, as well as a full list of peripherals and chip-level connectivity are fully documented in the device data sheet.

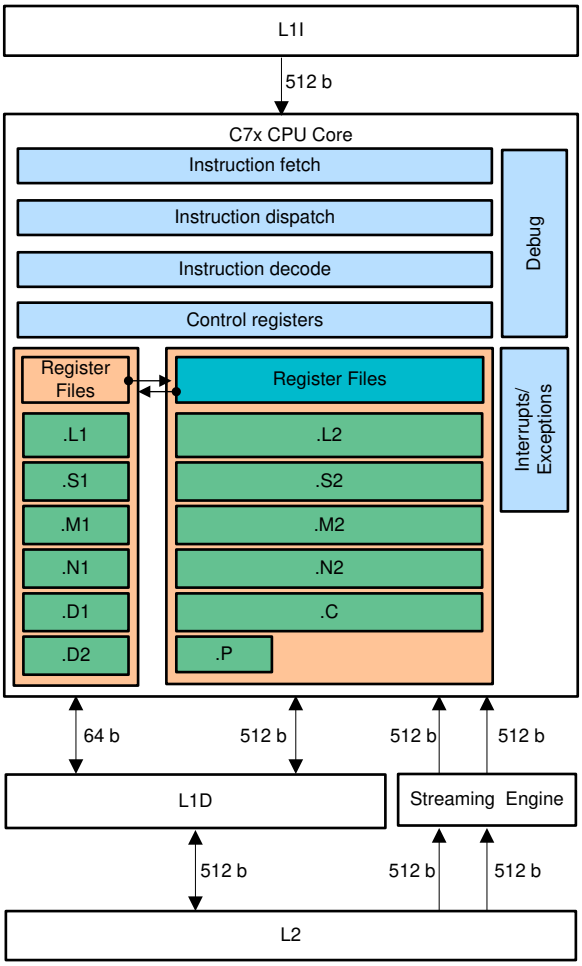


Figure 3-1. C7x DSP Block Diagram

3.2.1 Central Processing Unit (CPU)

The C7x CPU in Figure 3-1, contains:

- Program fetch unit
- Instruction dispatch unit, advanced instruction packing
- Instruction decode unit
- Dual data path with one 64-bit scalar side (Side A) and one 512-bit vector side (Side B)
  - Side A includes the following functional units › Four main scalar processing units (.L1, .S1, .M1, .N1) capable of operating up to 64-bit wide data
    - Two units (.D1, .D2) for address calculations, enabling parallel load/store operations
    - The following operations can be executed at the same time in a single clock cycle
      - 1 non-aligned 64-bit load or store operation
      - 2 64-bit arithmetic/logical operations (non-multiply arithmetic instructions)
      - 1 128-bit multiply operation
  - Side B includes the following functional units
    - Five main vector processing units (.L2, .S2, .M2, .N2, .C) capable of operating up to 512-bit wide vector data
    - Streaming Engine supplies up to two 512-bit data streams directly from L2 to the vector units
    - A Predication Processing Unit (.P) for vector predication
    - The following operations can be executed at the same time in a single clock cycle
      - 1 non-aligned 512-bit load or store operation
      - 2 512-bit arithmetic/logical operations (non-multiply arithmetic instructions)
      - 1 1024-bit multiply operation
      - 1 512-bit correlation operation or regular arithmetic operation

  - 1 vector predicate manipulation operation
- Large set of machine registers:
  - Up to 16x512-bit global vector registers
  - Up to 16x64-bit global scalar registers
  - Local registers
- Control registers
- Control logic
- Test, debug, and interrupt logic

The program fetch, instruction dispatch, and instruction decode units can deliver up to sixteen 32-bit instruction slots to the functional units every CPU clock cycle. The processing of instructions occurs in the data path. The data path contains 12 functional units (.L1, .S1, .M1, .N1,.D1, .L2, .S2, .M2, .N2, .D2, .C, .P) and the registers. The data path is described in more detail in Section 3.3.1. A control register file provides the means to configure and control various processor operations. To understand how instructions are fetched, dispatched, decoded, and executed in the data path, see Section 3.5.

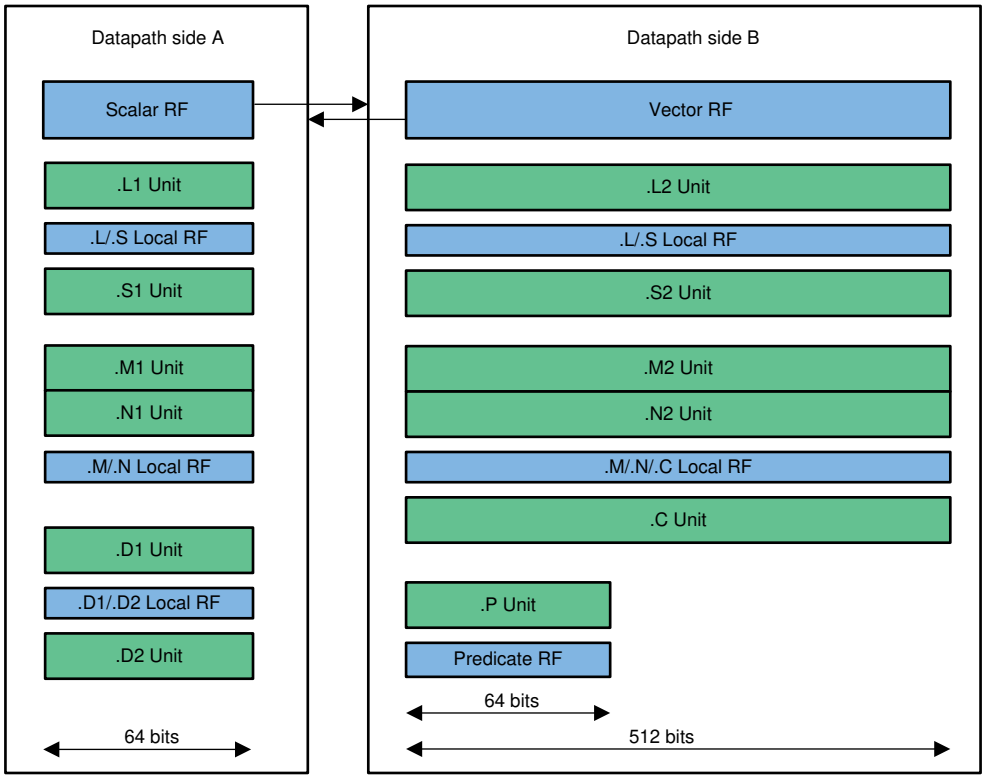


Figure 3-2. C7x Data Path Overview

3.3 C7x Native Features

Compared to C66x DSP, the C7x DSP major improvements are summarized below:

3.3.1 Data Path

- The C7x CPU is a true 64-bit machine, including 64-bit memory addressing
- The C7x vector side B can performs up to 128 16-bit fixed-point MAC, 4.0 times the MAC capacity compared to C66x
- The C7x vector side B can performs up to 80 single precision FLOPs/cycle or 32 double precision FLOPs/cycle, 5.0 times the floating point compared to C66x
- The C7x CPU can load or store in parallel 64 bits and 512 bits of data per clock cycle, more than four times the bandwidth compared to C66x. In addition, a novel streaming data interface can read an additional 1024 bits of data per clock cycle, providing a total of 12 times more bandwidth compared to the C66x
- Multi-dimensional streaming data management unit

  - Flexible, high bandwidth mechanism to read large quantities of data into the DSP CPU via two independent, multi-dimensional data streams
  - Supplies up to two 512-bit data streams directly from L2 to the vector units
  - Capable of performing matrix transpose while loading data from memory
  - Supports early exit
  - Supports 2D circular buffering for vision algorithms
- Independent streaming address generator(s)
  - Flexible multi-dimensional address calculator(s)
  - Provide offset addresses for load and store instructions
- New “Correlation unit” performs
  - General vector SIMD arithmetic functions
  - Flexible vector permutation functions
  - Horizontal add and horizontal min/max functions
  - Galois Field Multiply functions
  - WCDMA “Rake” and “Search” instructions. Performs up to 512 2-bit PN * 8-bit I/Q complex multiplies per clock cycle
  - 8-bit and 16-bit Sum-of-Absolute-Difference (SAD) calculations - up to 512 SADs per clock cycle (64x improvement over c66x)
- Increased register file storage:
  - Scalar register files for storing 64-bit scalar data
  - Vector register files for storing 512-bit vector results
  - Local register files dedicated to functional units for intermediate/short term results
- Parallel table lookup and histograms: up to 16 tables can be looked up in parallel with element sizes of byte, half word, word and double word.
- The C7x CPU can fetch 512-bit of instruction fetch packet per clock cycle, twice the fetch bandwidth of C66x

3.3.2 Control

- The C7x architecture provides two security states, each with associated memory attributes, aligned with K3 security architecture, full support of K3 firewall features
  - Secure state: in this state, the CPU can access both secure and non-secure space
  - Non-secure state: in this state, the CPU can only access non-secure space, and cannot access secure space
- The C7x architecture provides six privilege and execution levels for security and virtualization support with banked control registers for full isolation and protection
  - Secure Root Supervisor: provides secure kernel function and switching between secure and non-secure states
  - Secure Root User
  - Non-secure Root Supervisor: provides support for virtualization of non-secure operation
  - Non-secure Root User
  - Non-secure Guest Supervisor
  - Non-secure Guest User
- Changes between privilege levels are restricted:
  - A change from secure to non-secure state can only occur on the event return from secure supervisor mode
  - A change from root to non-root state can only occur on the event return from root supervisor mode
  - A change from non-secure to secure can only occur on a secure event handler entry or secure system call (SYSCALL)
  - A change from non-root to root can only occur on a root supervisor owned event handler entry or root system call (ROOTCALL)
- Supports for 64-bit virtual addressing fully compliant with ARM AArch64 architecture and ARM AArch32 LPAE architecture
  - Provides MMU which controls address translation, access permissions and memory attribute determination and checking for all accesses made by C7x
  - Translations are defined independently for different privilege levels and security states.
  - Table base registers are banked according to privilege levels and security state
- Pipeline can operate in both unprotected and protected modes

  - Unprotected mode: traditional C6x VLIW DSP operating mode with exposed instruction delay slots
  - Protected mode: control code execution mode where instructions delay slots are not exposed
- Scoreboarding mechanism to help hide memory system stalls and to allow the compiler to generate more efficient control codes
- Vector predicate register file to aid in vectorized control code/decision making
  - Vector predicate file can also be used to conditionally update memory locations
  - Vector predicate register file also assist in solving min, max, sorting and other search algorithms by using it to store index position information
- Zero-delay-slot Branches and Calls instructions. Branches, Calls and Returns now have no delay slots. Dynamic branch prediction is performed to remove pipeline delays on branches and calls. A return stack is provided for accelerating returns.
- Supports virtual memory, contains micro-TLB block to perform address translation from 64-bit virtual address to 40-bit physical address
- Supports recoverable interrupts and internal precise exception:
  - No need to disable interrupts during multiple assignment code. In-flight results in multiple-assignment code are recorded by new “pipeline capture queues.” These queues allow all programs to be interrupted at any arbitrary cycle, even during software pipelined loops, and then to be restarted correctly upon return from the interrupt or exception handler
  - Pipeline capture queues may be unloaded and reloaded, allowing the OS to swap out tasks regardless of whether the task was operating in protected or unprotected pipeline mode
  - Programmable event priority, hardware automatic event mask based on priority levels – Hardware performs automatic stacking at even handler entry and automatic unstacking at event exiting
- Extends constant range - allows “Branch relative” instruction to reach whole address space, as well as new move instruction which can move 32-bits constant to a register, i.e MVK cst32, reg
- Supports memory-mapped debug architecture instead of JTAG architecture

3.3.3 ISA

- Single-cycle 64-bit arithmetic, logical, and shift instructions
- Improvements to load/store instructions:
  - Endian aware vector load instructions
  - Unpacking load / packing store instructions
  - Vector load/store with element reversal
  - Vector load with data duplications
- Specialized instructions to speed up key benchmarks:
  - Horizontal MAX/MIN search acceleration instructions - find the max of 16 32-bit numbers, or 32 16-bit numbers, in one instruction
  - Horizontal ADD, SUB instructions
  - Dedicated FIR instructions
  - Sliding Window Sum of Absolute Differences
  - Maskable complex dot products
- Circular comparison instructions to accelerate Viterbi
- New and improved atomic instructions to replace LL/SC/CL combination
- Increases orthogonality between 32-bit, 64-bit, and vector operations compared to c66x
- Specialized instructions only allowed to execute at required privilege level and secure state
- Bidirectional Shift Instructions Without Saturation
- Right Shift and Round Instructions
- SORT16 instructions which can sort 16 elements in increasing order
- IEEE 754 Floating Point Standard Compliance
  - Fully compliance with IEEE 754 Floating Point Standard
  - Support subnormal number

3.4 Raw Performance Comparison

Table 3-1 provides a comparison of the raw performance between the C66x and the 512 bits vector width C7x CPU side B.

Table 3-1. Raw Performance Comparison Between the C66x and the C7x Vector Side B

	C66x	C711 Vector Side B
Fixed point 16x16 MACs per cycle	32	128
Fixed point 32x32 MACs per cycle	8	32
Floating point single precision MACs per cycle	8	32
Floating point single precision addition per cycle	8	48
Floating point double precision MAC per cycle	2	8
Floating point double precision additions per cycle	4	24
Arithmetic floating point operations per cycle	16(1)	80(2)
Load/store bandwidth	2 x 64-bits	2x512(3)-bits
Streaming Engine bandwidth	N/A	2x512-bits
Total bandwidth to memory	128-bits	2048-bits
Vector size (SIMD capability)	128-bits	512-bits

(1) 2-way SIMD SP addition on each .L and .S units (e.g. 8 SP operations for both side A and side B) and 4 SP multiply on each .M unit (e.g 8 SP multiply operations for both side A and side B).
(2) 16-way SIMD SP additions on .L2 , .S2 and .C2 units, 32-way SP multiply on .M2 and .N2 unit.
(3) 1 512-bit vector load in parallel with 1 512-bit vector store.

3.5 CPU Pipeline

This chapter starts with a description of the pipeline flow. Highlights are:

- The pipeline can dispatch up to 16 parallel 32-bits slots to provide instructions to the execution units (.L1, .S1, .M1, D1, .L2, .S2, .M2, .D2, .C, .P) every cycle
- Parallel instructions proceed simultaneously through each pipeline phase.
- Serial instructions proceed through the pipeline with a fixed relative phase difference between instructions.
- Load and store addresses appear on the CPU boundary during the same pipeline phase, eliminating read-after-write memory conflicts.

All instructions require the same number of pipeline phases for fetch and decode, but require a varying number of execute phases. This chapter contains a description of the number of execution phases for each type of instruction.

Finally, this chapter contains performance considerations for the pipeline. These considerations include the occurrence of fetch packets that contain multiple execute packets, execute packets that contain multi cycle NOPs, and memory considerations for the pipeline

3.5.1 Pipeline Overview

The C7x CPU core, shown in Figure 3-3, contains the following phases:

- Fetch phase (Phases PG, PA, PR)
- Dispatch and Decode phases (Phases DS, DC1, DC2)
- Execute phases (Phases E1, E2, E3, E4)

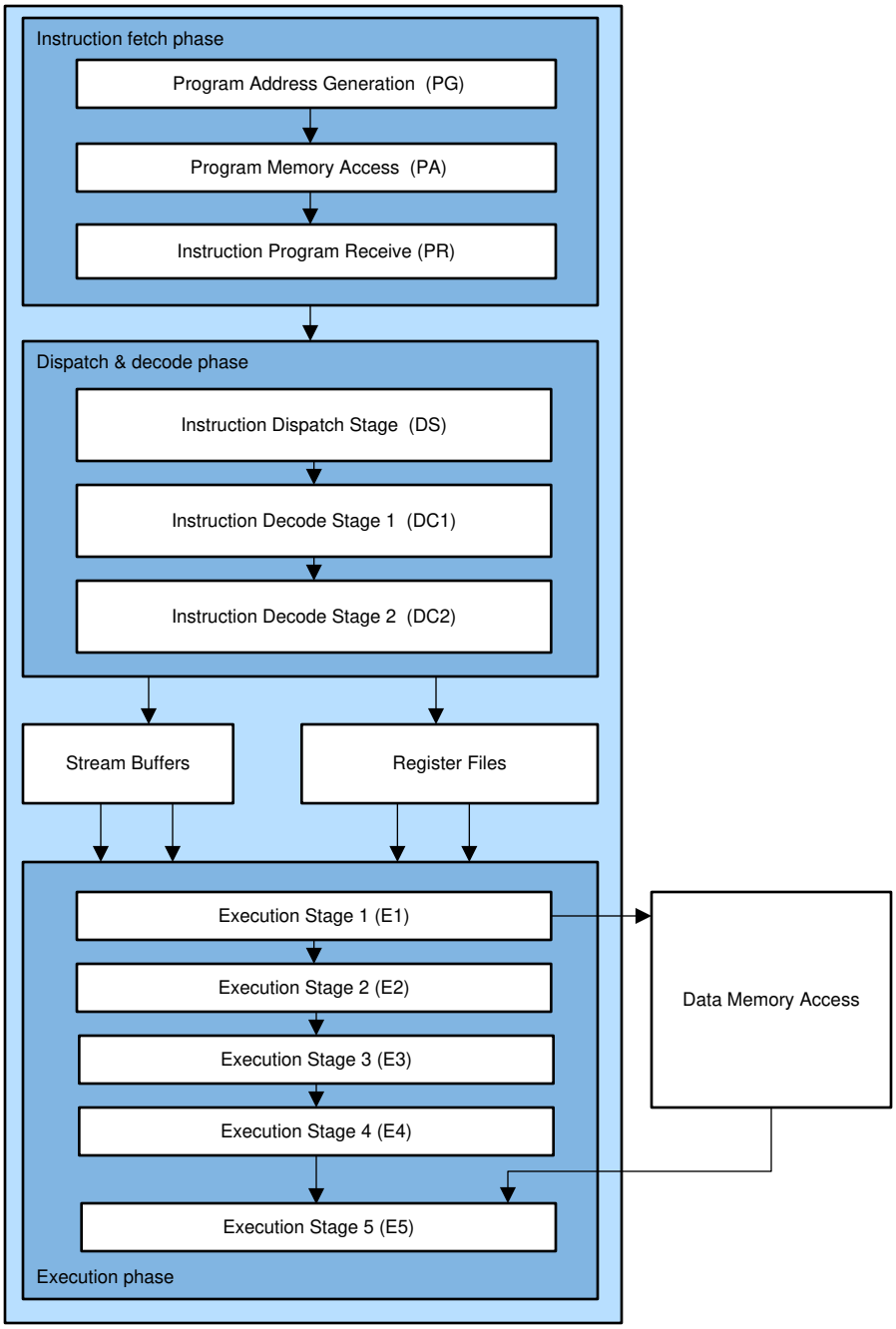


Figure 3-3. C7x Pipeline

3.5.1.1 Fetch

The fetch phase of the pipeline includes the Program Address Generation stage (PG), Program Access stage (PA) and Program Receive stage (PR)

- PG: During stage PG, the program address is generated in the CPU and the read request is sent to L1I memory controller.
- PA: During stage PA, L1I processes the request, access the data in its memory and then send the fetch packet to the CPU boundary during stage PR.
- PR: During stage PR, the CPU registered the fetch packet.

Note that the CPU and L1I pipelines are de-coupled from each other and the fetch packet returns by L1I can take different number of clock cycles, depending on external circumstances such as whether there is a hit in L1I or not. Therefore PA stage can take several clock cycles instead of 1 clock cycle as in the other stages.

3.5.1.2 Dispatch and Decode

The decode phases of the pipeline are:

- DS: Instruction dispatch to appropriate execution units
- D1: Instruction pre-decode
- D2: Instruction decode, operand reads

In the DS stage of the pipeline, the fetch packets are split into execute packets. Execute packets consist of one instruction or more parallel instructions. During the DS stage, the instructions in an execute packet are assigned to the appropriate functional units. In the D1 stage, the source registers, destination registers, and associated paths are decoded for the execution of the instructions in the functional units. In the D2 stage, more detail unit decodes are done, as well as reading operands from the register files

3.5.1.3 Execute

The execute portion of the pipeline is subdivided into execution stages (E1-E5). Different types of instructions require different numbers of these stages to complete their execution. These stages of the pipeline play an important role in your understanding the device state at CPU cycle boundaries.

3.5.2 Pipeline Phases Summary

Figure 3-4 shows an example of the pipeline flow of consecutive fetch packets that contain eight parallel instructions. In this case, where the pipeline is full, all instructions in a fetch packet are in parallel and split into one execute packet per fetch packet. The fetch packets flow in lockstep fashion through each phase of the pipeline.

For example, examine cycle 7 in Figure 3-4. When the instructions from FPn reach E1, the instructions in the execute packet from FP n +1 are being decoded in stage DC2. FP n + 2 is in DC1, while FPs n + 3 and n + 4 are in dispatch (DS) and program receive (PR) stage.

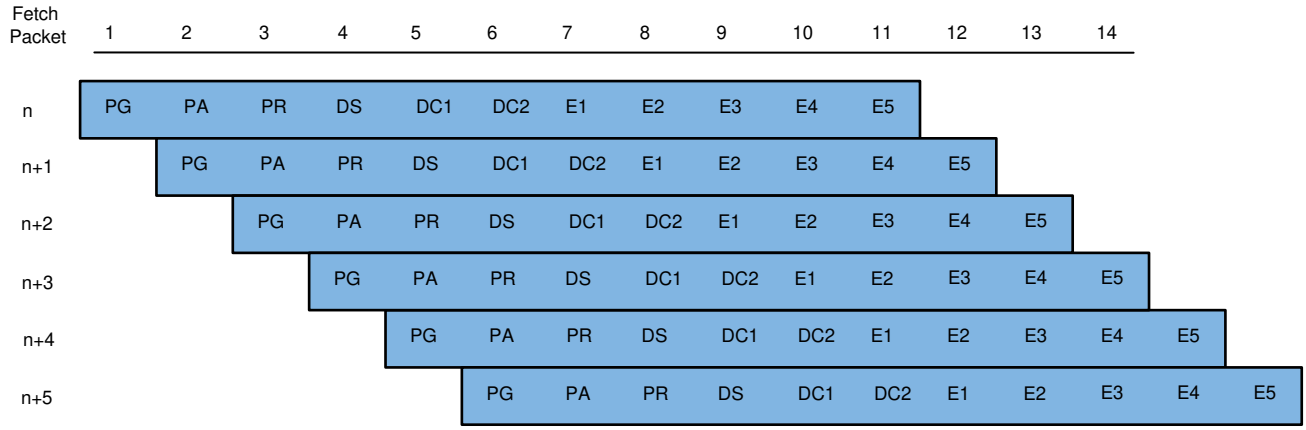


Figure 3-4. Pipeline Operation: One Execute Packet per Fetch Packet

Table 3-2 summarizes the pipeline phases and what happens in each phase.

Table 3-2. Operations Occur During Pipeline Phases

Phase	Symbol	During This Phase
Program Address Generate	PG	The address of the fetch packet is determined and send to L1I.
Program Memory Access	PA	L1I fetch request and access data RAM
Program Receive	PR	L1I sends instruction fetch packet to the CPU
Dispatch	DS	The fetch packet is received at the CPU boundary. The next execute packet in the fetch packet is determined and sent to the appropriate functional units to be decoded.
Pre-decode	DC1	Instructions are pre-decodes to determine usage requirements
Decode	DC2	Instructions are decoded in functional units and operands are read.

Table 3-2. Operations Occur During Pipeline Phases (continued)

Phase	Symbol	During This Phase
Execute 1	E1	For all instruction types, the conditions for the instructions are evaluated and operands are operated on. For load and store instructions, address generation is performed and address modifications are written to a register file. For branch instructions, branch fetch packet in PG phase is affected.(1) For single-cycle instructions, results are written to a register file.(1)
Execute 2	E2	For load instructions, the address is sent to memory. For store instructions, the address and data are sent to memory.(1) Single-cycle instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs.(1) For 2-cycle instructions, results are written to a register file.
Execute 3	E3	Data memory accesses are performed. Any multiply instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs.(1) For 3-cycle instructions, results are written to a register file.
Execute 4	E4	For load instructions, data is brought to the CPU boundary.(1) For 4-cycle instructions, results are written to a register file.(1)
Execute 5	E5	For load instructions, data is written into a register.(1)

(1) This assumes that the conditions for the instructions are evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.

3.5.3 Pipeline Execution of Instruction Types

The pipeline operation of the C7x DSP instructions can be categorized into eight instruction types. Seven of these (NOP is not included) are shown in Table 3-3 which is a mapping of operations occurring in each execution phase for the different instruction types. The delay slots and functional unit latency associated with each instruction type are listed in the bottom row.

The execution of instructions is defined in terms of delay slots. A delay slot is a CPU cycle that occurs after the first execution phase (E1) of an instruction. Results from instructions with delay slots are not available until the end of the last delay slot. For example, a multiply instruction has one delay slot, which means that one CPU cycle elapses before the results of the multiply are available for use by a subsequent instruction. However, results are available from other instructions finishing execution during the same CPU cycle in which the multiply is in a delay slot.

Table 3-3. Execution Stage Length Description for Each Instruction Type

	Instruction Type							
Execution Phase(1) (2)	**Single Cycle**	**2 cycles**	**3 cycles**	**4 cycles**	**Store**	**Scalar Load**		**Branch**
E1	Compute result and write to register	Read operands and start computations	Read operands and start computations	Read operands and start computations	Compute address	Compute address		Target code in PG(3)
E2		Compute result and write to register	Continue computation	Continue computation	Send address and data to memory	Send address to memory		
E3			Complete computation and write results to register	Continue computation	Access memory	Access memory		
E4				Complete computation and write results to register		Send data back to CPU	Send data back to CPU	
E5						Write data into register	Data formatting and distribution	

Table 3-3. Execution Stage Length Description for Each Instruction Type (continued)

	Instruction Type							
Execution Phase(1) (2)	**Single Cycle**	**2 cycles**	**3 cycles**	**4 cycles**	**Store**	**Scalar Load**		**Branch**
E6							Write data into register	
Delay slots	0	1	2	3	0NEED XREF	4NEED XREF		5NEED XREF
Functional unit latency	1	1	1	1	1	1		1

(1) This table assumes that the condition for each instruction is evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.

(2) NOP is not shown and has no operation in any of the execution phases.

(3) See **section 5.2.6** for more information on branches.

3.5.3.1 Single Cycle Instructions

Single-cycle instructions complete execution during the E1 phase of the pipeline (see Table 3-3). Figure 3-5 shows the fetch, decode, and execute phases of the pipeline that the single-cycle instructions use.

Figure 3-5. Single-Cycle Instruction Phases

Figure 3-6 shows the single-cycle execution diagram. The operands are read, the operation is performed, and the results are written to a register, all during E1. Single-cycle instructions have no delay slots.

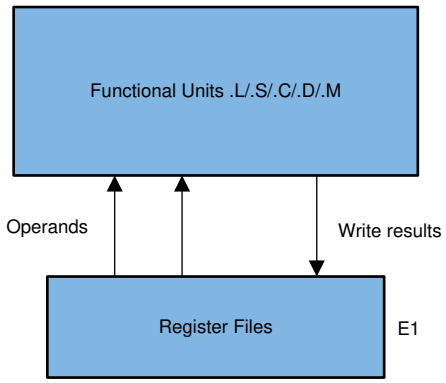


Figure 3-6. Single-Cycle Instruction Execution Block Diagram

Table 3-4. Single-Cycle Instruction Execution

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.L, .S, .M, .C, , or .D

3.5.3.2 Two Cycle Instructions

Two-cycle instructions use both the E1 and E2 phases of the pipeline to complete their operations (see Table 3-5). **Figure 5-10** shows the fetch, decode, and execute phases of the pipeline that the two-cycle instructions use.

Table 3-5 shows the operations occurring in the pipeline for a two-cycle instruction. In the E1 phase, the operands are read and the computation begins. In the E2 phase, the computation finishes, and the result is written to the destination register. Two-cycle instructions have one delay slot.

Table 3-5. Two Cycle Instruction Execution

Pipeline Stage	E1	E2
Read	src1, src2	
Written		dst
Unit in use	.M,.L,.S,.C	

3.5.3.3 Three Cycle Instructions

Three-cycle instructions use the E1, E2 and E3 phases of the pipeline to complete their operations (see Table 3-6). **Figure 5-10** shows the fetch, decode, and execute phases of the pipeline that the three-cycle instructions use.

Table 3-6 shows the operations occurring in the pipeline for a three-cycle instruction. In the E1 phase, the operands are read and the computation begins. In the E2 phase, the computation continues, and in E3, the result is written to the destination register. Three-cycle instructions have two delay slots.

Table 3-6. Three Cycle Instructions

Pipeline Stage	E1	E2	E3
Read	src1, src2		
Written			dst
Unit in use	.M,.L,.S,.C		

3.5.3.4 Four Cycle Instructions

Four Cycle Instructions Four-cycle instructions use the E1, E2, E3, and E4 phases of the pipeline to complete their operations (see Table 3-7). **Figure 5-10** shows the fetch, decode, and execute phases of the pipeline that the four-cycle instructions use.

Table 3-7 shows the operations occurring in the pipeline for a four-cycle instruction. In the E1 phase, the operands are read and the computation begins. In the E2 and E3 phases, the computation continues, and in E4, the result is written to the destination register. Four-cycle instructions have three delay slots.

Table 3-7. Four Cycle Instruction Execution

Pipeline Stage	E1	E2	E3	E3
Read	src1, src2			
Written				dst
Unit in use	.M,.L,.S,.C			

3.5.3.5 Load Instructions

Data loads require all five, E1 through E5, of the pipeline execute phases to complete their operations (see Table 3-8). Figure 3-7 shows the fetch, decode, and execute phases of the pipeline that the load instructions use.

Figure 3-8 shows the operations occurring in the pipeline phases for a load. In the E1 phase, the data address pointer is modified in its register. In the E2 phase, the data address is sent to data memory. In the E3 phase, a memory read at that address is performed.

Table 3-8. Load Instruction Execution

Pipeline Stage	E1	E2	E3	E4	E5
Read	baseR, offsetR, src				
Written	baseR				dst
Unit in use	.D				

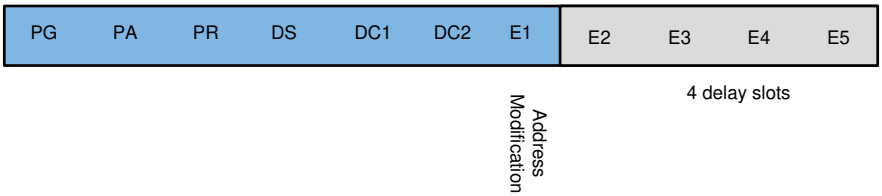


Figure 3-7. Load Instruction Phases

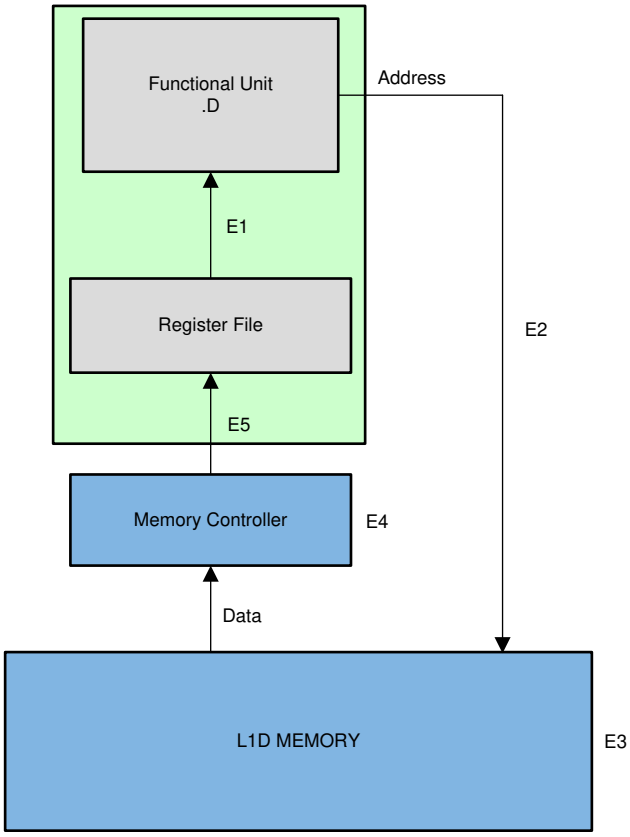


Figure 3-8. Load Instruction Execution Block Diagram

In the E4 stage of a load, the data is received at the CPU core boundary. Finally, in the E5 phase, the data is loaded into a register. Because data is not written to the register until E5, load instructions have four delay slots. Because pointer results are written to the register in E1, there are no delay slots associated with the address modification.

3.5.3.6 Store Instructions

Store instructions require phases E1 through E3 of the pipeline to complete their operations (Table 3-9). Figure 3-9 shows the fetch, decode, and execute phases of the pipeline that the store instructions use.

Figure 3-10 shows the operations occurring in the pipeline phases for a store instruction. In the E1 phase, the address of the data to be stored is computed. In the E2 phase, the data and destination addresses are sent to data memory. In the E3 phase, a memory write is performed. The address modification is performed in the E1 stage of the pipeline. Even though stores finish their execution in the E3 phase of the pipeline, they have no delay slots. There is additional explanation of why stores have zero delay slots in **Section 5.2.5**.

Table 3-9. Store Instruction Execution

Pipeline Stage	E1	E2	E3
Read	baseR, offsetR, src		
Written	baseR		

Table 3-9. Store Instruction Execution (continued)

Pipeline Stage	E1	E2	E3
Unit in use	.D2		

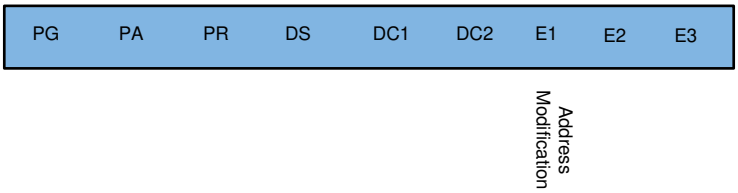


Figure 3-9. Store Instruction Phases

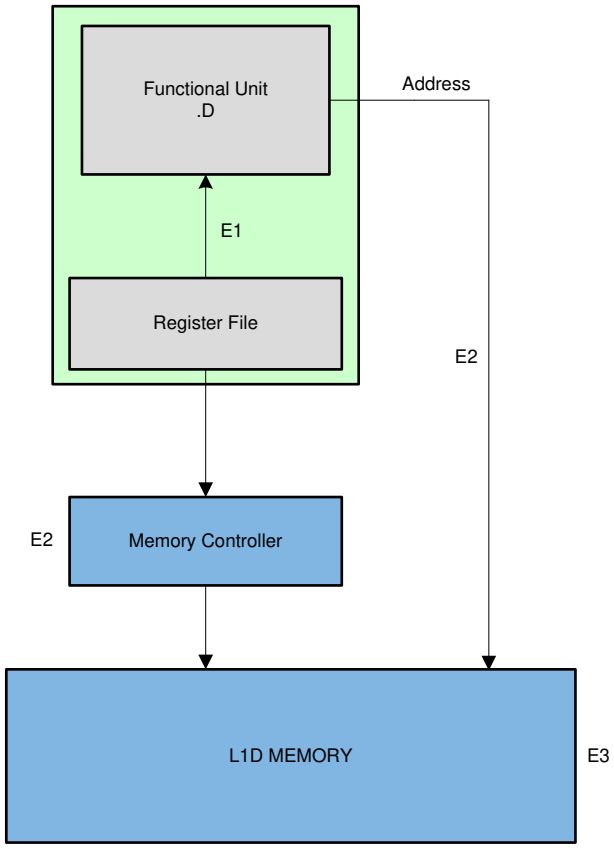


Figure 3-10. Store Instruction Execution Block Diagram

3.5.3.7 Branch Instructions

Although branch instructions take one execute phase, there are some cycles between the execution of the branch and execution of the target code. Table 3-10 shows the pipeline phases used by the branch instruction and branch target code. If a branch is in the E1 phase of the pipeline (in the .S unit in Table 3-10), its branch target is in the fetch packet that is in PG during that same cycle. Because the branch target has to wait until it reaches the E1 phase to begin execution, the branch takes some delay slots before the branch target code executes.

In C7x, all branches are treated as zero-delay slot branches. That means the delay slots of all branches are not exposed to the compiler and scheduler. Instead, the C7x will rely on a sophisticated branch prediction unit to perform branch prediction. If a branch is predicted correctly, then there is no lost cycles incurred as shown in Figure 3-11. If the branch is predicted incorrectly, then the processor will take some cycles hit to recover from the mispredicted branch, as shown in Figure 3-12.

On the DSP, a stall is inserted if a branch is taken to an execute packet that spans fetch packets to give time to fetch the second packet. Normally the assembler compensates for this by preventing branch targets from

spanning fetch packets. The one case in which this cannot be done is in the case that an interrupt or exception occurred and the return target is a fetch packet spanning execute packet.

Table 3-10. Branch Instruction Execution

	Target Instruction						
Pipeline Stage	**E1**	**PA**	**PR**	**DS**	**DC1**	**DC2**	**E1**
Read	src2						
Written							
Branch taken							
Unit in use	.S						

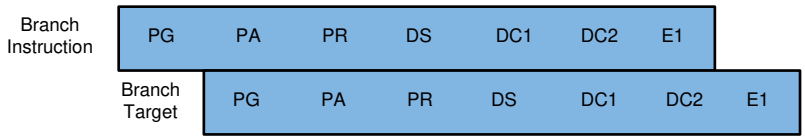


Figure 3-11. Correctly Predicted Branch Instruction Phases

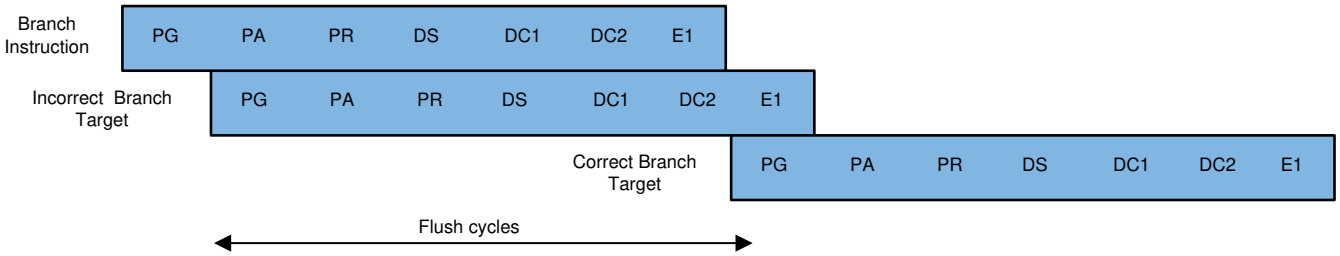


Figure 3-12. Incorrectly Predicted Branch Instruction Phases

Note that branch instructions are always at the beginning of the execution packet, i.e its P-bit is always 0. Also, only one branch instruction can be issued per execute packet, regardless of whether it is predicated true or false.

3.6 CPU Datapath

This chapter focuses on the CPU, providing information about the data paths, including the functional units, general purpose register files and control registers.

Terminology:

- Scalar instructions: Instructions which utilize only 32-bit or 64-bit resources (operands or destinations)
- Vector instructions: Instructions which utilize 512-bit resources (operands or destinations)

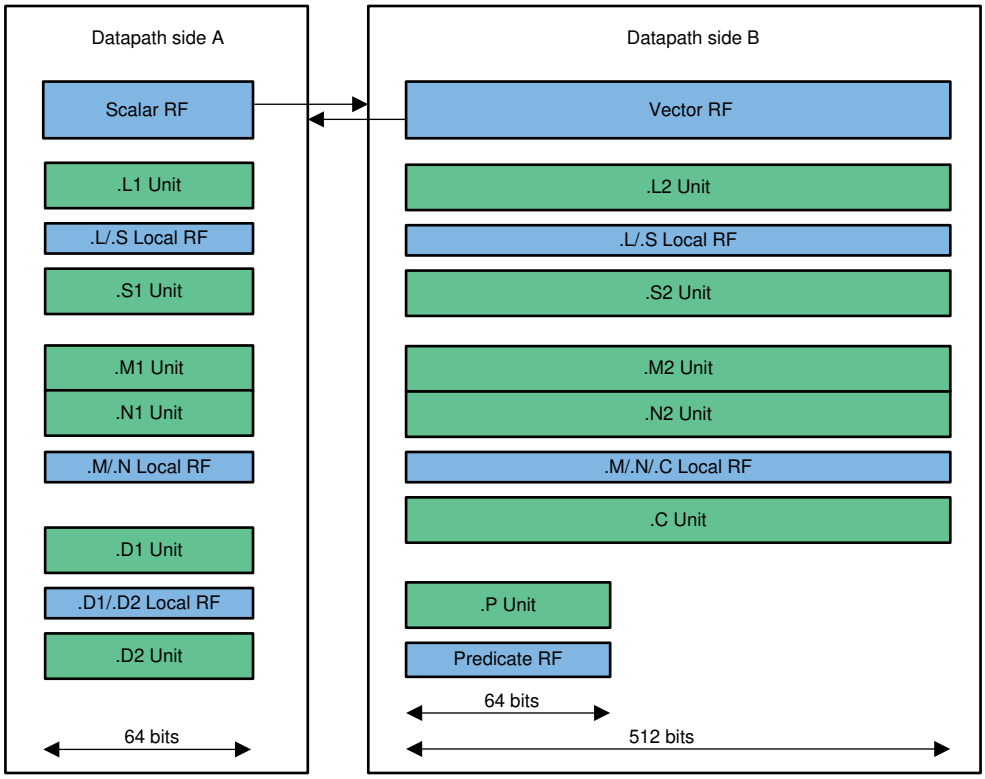


Figure 3-13. C7x CPU Block Diagram

The components of the data path for the CPU are shown in Figure 3-13.

These components consist of:

- Global and Local general-purpose register files
- 12 functional units (.L1, .S1, .M1, .N1, .D1, .L2, .S2, .M2, .N2, .D2, .C, .P)
- One 64-bit load-from-memory data path
- One 512-bit load-from-memory data path
- One 64-bit store-from-memory data path
- One 512-bit store-to-memory data path
- Two 40-bit data physical address paths
- Two 512-bit read data streams from Streaming Engine

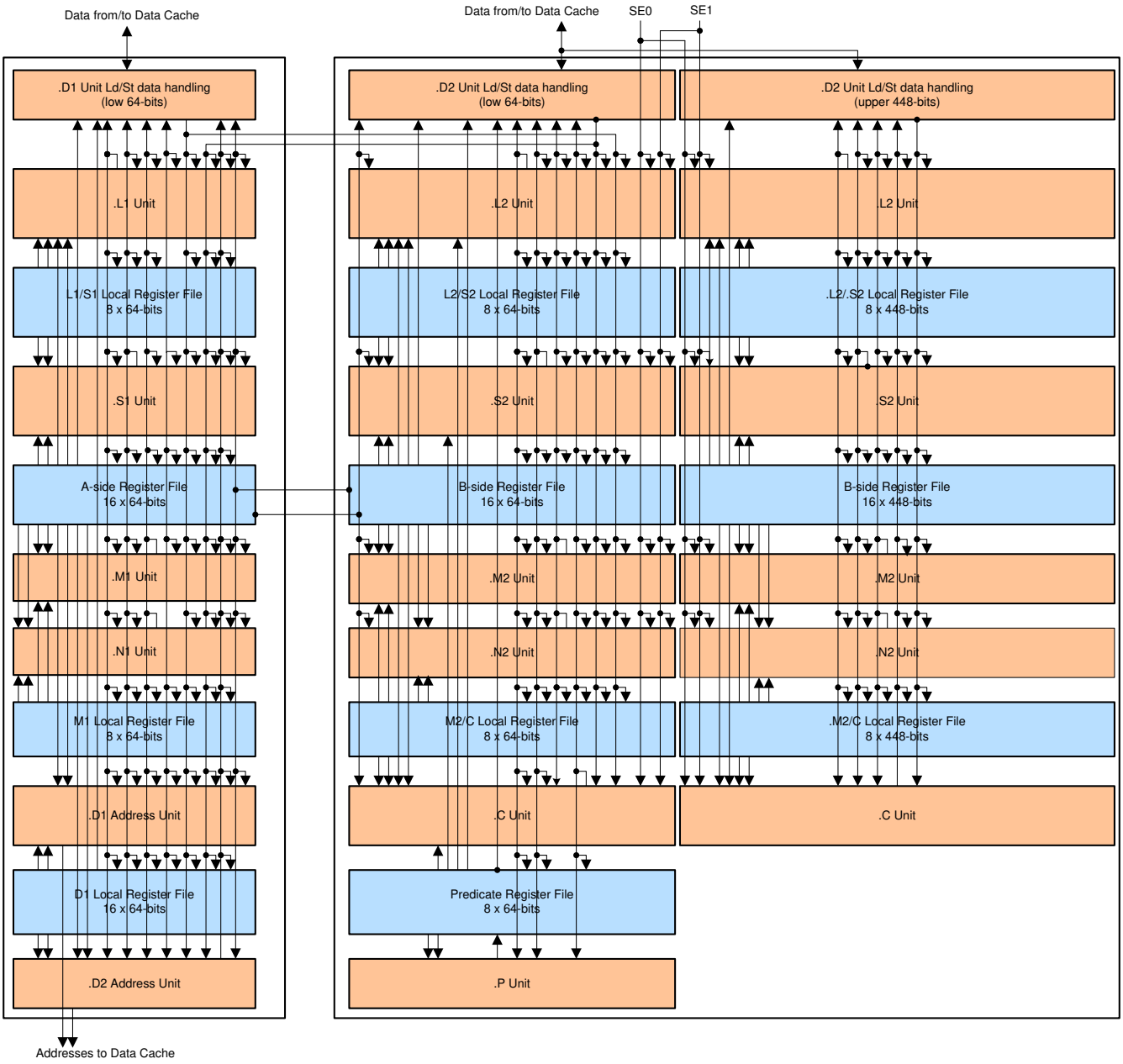


Figure 3-14. CPU Data Paths

3.6.1 General Purpose Register Files Overview

C7x register set is comprised of the following components:

- Global Scalar Register File (GRF): All scalar C7x instructions which are issued to side A data path can read or write to this GRF directly. Scalar instructions which are issued to side B data path can read this GRF via the cross path but cannot write to the GRF.
- Global Vector Register File (VRF): All C7x instructions which are issued to side B data path can read or write to this VRF directly. A side B scalar instructions can also access the low 32 or 64 bits of a VRF register as sources or destination. Instructions which are issued to side A data path can only read the VRF via the cross path but cannot write to the VRF.
- Local Vector Register File (LRF): Local to the functional units, can only be read by its corresponding functional unit(s). However, it can be written by any functional units on the same data path side. Similar to the VRF, any scalar instructions can also access the low 32 or 64 bits of a LRF register as sources or destination

- Predicate Register File (PRF): The predicate register file contains the results from vector comparison operations and is used by the vector selection instructions as well as vector predicated store instructions. A small subset of special instructions can also read directly from predicate registers, performs operations and write back to a predicate register directly.

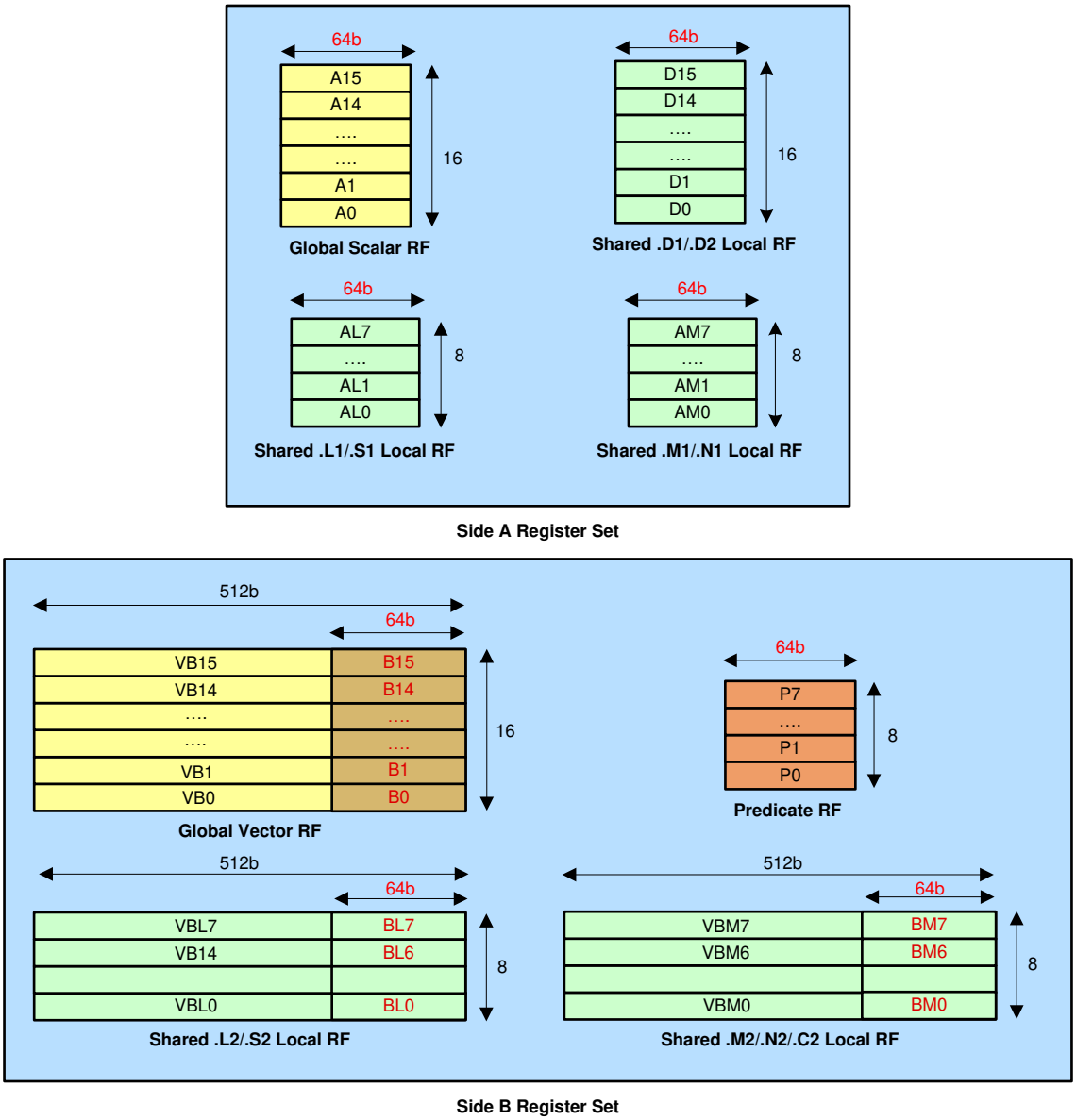


Figure 3-15. C7x General Purpose Register File Set

3.6.2 Register Definitions

3.6.2.1 Scalar Registers

The scalare registers are 64-bits wide. The scalar registers can be accessed by any scalar instructions. The 64-bit scalar registers can also be accessed by vector instructions, with the restrictions as described in Section 3.6.3.4.2.

3.6.2.2 Vector Registers

A vector register contains 512-bit data. Vector registers can only be accessed as a whole by vector instructions.

3.6.3 General Purpose Register File Details

3.6.3.1 Global Scalar Register File (GRF)

There are 16 independent scalar registers; each register is 64-bit wide. Each register can be read as 64-bits scalar data. However, writes are always 64-bit, with zero-extended to fill up to 64-bit if needed. The scalar register files support data ranging in size from 8-bit through 64-bit data. C7x CPU does not support 128-bit data type.

- Addressable as 64-bit scalar data
- Can be read from or write to by any side A functional units
- Can be read from side B functional units via the cross path
- Assembly notation
  - A15-A0: 16 64-bit scalar registers

3.6.3.2 Global Vector Register File (VRF)

There are 16 global vector registers; each register is 512 bits wide:

- Addressable as low 64-bit scalar data, or 512-bit vectors
- Can be read from or write to by any side B functional units
- The low 64-bit can be read from side A functional units via the cross path
- Assembly notations:
  - VB15-VB0: 512-bit vector registers. The vector registers can be accessed as a whole by a vector instruction. The low 64 bits of each of these registers can also be used as an operand or destination for a scalar instruction as described below.
  - B15-B0: low 64-bits scalar portion of VB15-VB0. They can only be used as operand or destination of a 64-bit scalar instruction.

3.6.3.3 Local Register Files (LRF)

There are 3 separate set of local register files for each data path side. Each set dedicated to particular functional units:

3.6.3.3.1 .M and .C Shared Local Register FIle (LRFM)

There are eight local vector registers shared between .M and .C functional units. For side A, each register is 64-bits wide, and for side B, each register is 512-bit wide.

- Addressable low 64-bit scalars or 512-bit vectors
- Can only be read from the same side .M and .C units. Any unit on the same data path side can write to it.
- Assembly notations:
  - AM7-AM0: side A, 8 64-bit registers
  - VBM7-VBM0: side B, 8 512-bit vector registers. The vector registers can be accessed as a whole by a vector instruction. The low 64 bits of each of these registers can also be used as an operand or destination for a scalar instruction
  - BM7-BM0: low 64-bits scalar portion of VBM15-VBM0. They can only be used as operand or destination of a 64-bit scalar instruction.

3.6.3.3.2 .L and .S Shared Local Register File (LRFL)

There are eight local vector registers shared between .L and .S functional units. For side A, each register is 64-bits wide, and for side B, each register is 512-bit wide.

- Addressable as 64-bit scalars or 512-bit vectors
- Can only be read from the same side .L and .S units. Any unit on the same data path side can write to it.
- Assembly notations:
  - AL7-AL0: side A, 8 64-bit registers
  - VBL7-VBL0: side B, 8 512-bit vector registers. The vector registers can be accessed as a whole by a vector instruction. The low 64 bits of each of these registers can also be used as an operand or destination for a scalar instruction
  - BL7-BL0: low 64-bits scalar portion of VBL7-VBL0. They can only be used as operands or destination of a 64-bit scalar instruction.

3.6.3.3.3 .D Unit Local Register (LRFD)

There are sixteen 64-bit local registers, dedicated to .D1 and .D2 functional units. The .D unit local registers can only be used as base register or offset addresses for load/store instructions, not as store data. The local registers can also be used for regular .D1 and .D2 instructions.

- Can only be read by.D1 and .D2 units. Any unit on side A can write to it.
- Assembly notations
  - D15-D0: 16 64-bit wide local registers. They can be used as operands or destination of .D1 and .D2 instructions, including load or store instructions.

3.6.3.3.4 Predicate Register File (PRF)

There are eight registers in the Predicate Register File. Each bit of a Predication Register (PR) controls a byte of a vector data. Since a vector is 512-bits, the width of a predicate register equals 64 bits

- Can only be read by .P, .L2 and .S2 unit • Can only be written by .P, .L2, .S2 and .C2 units
- Assembly notations
  - P7-P0: 8 64-bit wide predicate registers

The predicate register file can be written by vector comparison operations to store the results of the vector compares. A small set of instructions can operate directly on the predicate register values and write back to the predicate register file in one cycle. In addition, there are instructions which can transfer values between the global register files and the predicate register file. Transfers between the predicate register file and the unit local register files are not supported

3.6.3.4 Restrictions

3.6.3.4.1 Instruction Restrictions

- Scalar instructions can read from or write to global scalar register or the lower 64 bits of any vector register, global or local.
- Vector instructions can read from or write to any global vector registers or local vector registers.
- Vector instructions can also read from the global scalar registers (A7-A0) via the cross path. The upper bits are zero-extended to fit the vector operand width.

3.6.3.4.2 Register File Restrictions

- Global Scalar Register File:
  - Can be read from or write to by all side A functional units
  - Can be read from by side B functional units via the cross path
  - Only 32-bit or 64-bit reads, and 64-bit write
  - Any vector instructions on side B can also read 64-bit data from the global scalar registers via the cross path and zero-extended the upper 192-bit to form an input vector
- Global Vector Register File:
  - Can be read from or write to by all side B functional units
  - Low 64-bit can be read from by side A functional units via the cross path
  - Only low 64-bit or whole 512-bit accesses
- Local Register Files:
  - Can be written to by same side functional units
  - Can only be read from its own functional units
  - Only low 64-bit or whole 512-bit accesses

3.6.3.4.3 Forwarding Restrictions

- Forwarding from a 64-bit scalar register to another 64-bit scalar register is allowed
- Forwarding from one vector register to another vector register is allowed
- Forwarding from a 64-bit scalar register to a vector register is allowed, with the upper bits of the vector register keep the current values
- Forwarding from a vector register to a 64-bit scalar register is allowed, with the upper bits of the vector register ignored
- Forwarding between upper bits of a vector register and 64-bit scalar register is not allowed

3.6.3.5 Functional Units

Compared to the C66x, the C7x functional units support wider vectors and more comprehensive ISA.

3.6.3.5.1 .L Unit (.L1, .L2)

.L1 and .L2 are variations of the same underlying .L unit. The principal difference between the .L1 and .L2 is the difference in their widths.

.L1 is the scalar version of the .L unit inside data path side A and .L2 is the vector version of the .L unit inside data path side B.

The .L1 unit accepts up to two 64-bit operands and produce one 64-bit result and the .L2 unit accepts up to two vector operands and produce one vector result.

Each .L unit shares the same local register file with the .S unit of the same side.

Besides supporting the legacy C66x .L and .S instructions, the .L units also support these major operations:

- 64-bit add/subtract operations
- 32-bit min/max operations
- 8-bit SIMD instructions: sadd4, min4, max4 etc.
- Circular min/max operations
- Various new move instructions, including between vector predicate register file and global register file
- Half-precision floating point conversion instructions
- Vector SIMD of legacy and new instructions

3.6.3.5.2 .S Unit (.S1, .S2)

.S1 and .S2 are variations of the same underlying .S unit. The principal difference between the .S1 and .S2 is the difference in their widths.

.S1 is the scalar version of the .S unit inside data path side A and .S2 is the vector version of the .S unit inside data path side B.

The .S1 unit accepts up to two 64-bit operands and produce one 64-bit result and the .S2 unit accepts up to two vector operands and produce one vector result.

Each .S unit shares the same local register file with the .L unit of the same side.

The .S units support similar set of instructions like the .L units. The main difference with the .L unit is the .S units include support for the floating point reciprocal instructions.

3.6.3.5.3 .M Unit (.M1, .M2)

.M1 and .M2 are variations of the same underlying .M unit. The principal difference between the .M1 and .M2 is the difference in their widths.

.M1 is the scalar version of the .M unit inside data path side A and .M2 is the vector version of the .M unit inside data path side B.

The .M1 unit accepts up to two 64-bit operands and produces one 64-bit result. The .M2 unit accepts two vector operands and produces one vector result.

The .M unit shares the same local register file of the .C unit on the same side.

Besides supporting the legacy C66x instructions, .M unit also supports these new operations, for examples:

- 8-bits multiply operations
- Complex dot product operations
- 32 & 64-bits bit count operations
- Complex conjugate multiply operations
- Vector and Vector SIMD of legacy and new instructions
- Bit-wise Logical Operations, moves, as well as adds and subtracts

3.6.3.5.4 .N Unit (.N1, .N2 Unit)

Similar to the .M units, the .N1 and .N2 units contain multiplier hardware to calculate multiply, dot products and similar computations. The .N units can execute independent instructions and has their own issue slots.

However, some instructions which require vector-pair resources can occupy both the .M and its corresponding .N unit. These instructions are called “dual-issued” instructions, and they use the muliplier hardware of .M unit as well as .N unit muliplier hardware to produce the result.

3.6.3.5.5 .C Unit

The .C unit accepts two vector operands and produces one vector result.

These are the classes of instructions supported by the .C unit:

- WCDMA "Rake" and "Search" instructions. Performs up to 512 2-bit PN * 8-bit I/Q complex multiplies per clock cycle
- 8-bit and 16-bit Sum-of-Absolute-Difference calculations - up to 512 SADs per clock cycle (64x improvement over c66x)
- Horizontal add and horizontal min/max instructions
- Vector permutes instructions: VPERM, VDEAL, VSHFL etc

The .C unit also contains 4 vector control registers. These control registers are used to control the operations of certain .C unit instructions. See Section 3.7.43 for more details.

3.6.3.5.6 .D Unit (.D1, .D2)

The .D1 and .D2 unit is used for address calculations and is expanded to accept scalar operands up to 64-bits and produces scalar result up to 64-bits.

The .D1 unit performs scalar load or store up to 64-bits, and the .D2 unit performs vector load or store up to 512-bits. Each unit can send out one load or store request every clock cycle along with the 40-bit physical address to L1D. Load or store data width can be varied and up to 512-bit.

Each unit also includes additional hardware to perform data manipulations such as swapping, pack and unpack on the load and store data to reduce workloads on the other units.

Besides supporting legacy C66x instructions, the .D units also support these new operations:

- 64-bit SIMD arithmetic operations
- 64-bit bit-wise logical operations
- Scalar and vector load and store data manipulations

For .D1 unit, the base and offset values for address calculation can come from the global scalar register file or the shared .D unit local register file. The .D1 unit can write to the global scalar register file and all side A local register files.

For .D2 unit, the base and offset values for address calculation can come from the global scalar register file or the shared .D unit local register file. The .D2 unit can write to the global scalar register file and all side A local register files.

The .D units also contain micro-TLB block to perform address translation from 48-bit virtual address to 40-bit physical address.

To eliminate the need for compiler to worry about the T-path, any load instructions which load more than 64-bits of data from memory can only be issued on the .D1 unit.

shows the load and store instructions supported on .D1 and .D2 units

Table 3-11. Supported load and store combinations

.D1	.D2
Load <= 64-bits to A or B side	Load <= 64-bits to A or B side
Load > 64-bits to B side only	Store > 64-bits from B side only
Store <= 64-bits from A side or B side (via xpath)	Store <= 64-bits from A side (via xpath) or B side

Any combination of load/store operations between .D1 and .D2 listed above is valid and can be issued in parallel.

Note

Vector load and duplicate instructions, which only require 64-bits data or less from memory are allowed to be issued on .D2 unit.

3.6.3.5.7 .P Unit

The C7x CPU also contains a small control unit which performs basic operations on vector predication registers. The .P unit has direct access to the vector predication registers. The predication control unit performs different bit operations on the predication registers such as AND, ANDN, OR, XOR, NOR, BITR, NEG, SET, BITCNT, RMBD, BIT Decimate and Expand.

3.6.3.5.8 Constant Extension Logic

Constant extension logic is used to provide the upper 27 bits to be concatenated with the 5-bit constant which already encoded inside the instructions to form a 32-bit constant. For example, to move 32-bit constant into a register, one MVK instruction will be issued. Within the instruction opcode, there is a constant extension bit, if set, would signal that the constant embedded within the MVK instruction will be concatenated with the adjacent constant extension slot from the same execute packet. These two constants will be concatenated inside the constant extension logic to form a 32-bits constant.

The C7x CPU can extend up to two constants per cycle, so up to two constant extensions can be issued for every execute packet. Each constant extension is associated with defined set of instruction slots:

- CSTX0: associated with .L1, .D1 data, .S2, .D2 offset, .M2, .N2, .B, .C
- CSTX1: associated with .L2, .D2 data, .S1, .D1 offset, .M1, .N1

Where:

- .Dx data: only uses for store-a-constant instructions as the source constant to store out
- .Dx offset: src2 field - constant offset in address calculation for load/store, ADDAx instructions and as constant source for arithmetic instructions.

For example:

- CXT0:
  - .D2 unit uses it as constant offset in address calculation for load/store, ADDAx instructions and as constant for arithmetic instructions.
  - .D1 uses it as constant data for Store-a-constant instruction.
- CXT1:
  - .D1 uses it as constant offset in address calculation for load/store, ADDAx instructions and as constant for arithmetic instructions.
  - .D2 uses it as constant data for Store-a-constant instruction.

The constant logic does not read or write any architectural registers.

Most instructions can only utilize 1 constant extension at the time, except for the MVK instruction which move 64 bit constants and STK (Store-a-constant) instructions. They are special cases and need to utilize both constant extensions simultaneously.

For the case of MVK-64-bit, the 64-bit constant is formed by putting the constant extensions in the order of CSTX1[26:0] & CSTX0[26:0] & 10-bit embedded constant from the instruction encoding itself.

The STK instructions use one constant extension as offset for address calculation and the other constant extension for the constant to store out as data.

The table below describes the constant operand interpretations for different instruction and data types.

Table 3-12. Constant Operand Rules

Instruction Type	Operand Type	Constant Length	Action
Scalar	B/H/W/D	5 bits	Sign extended to 64 bits

Table 3-12. Constant Operand Rules (continued)

Instruction Type	Operand Type	Constant Length	Action
Scalar	B/H/W/D	32 bits	Sign extended to 64 bits
Vector	B/H/W/D	5 bits	Sign extended to data type width and replicateE.g., if B, extend to 8-bits and replicate 8-bits across whole vector
Vector	B/H/W	32 bits	Replicate 32-bit constant across each 32-bit lane (Compiler must replicate constant for B/H across the 32-bit lane)
Vector	D	32 bits	Sign-extend 32-bit constant to 64-bits and replicate 64-bit data across all D lanes

Note

Compiler can put a 32-bit repeating pattern of bytes in an VADDB-k, if useful. Hardware won't know or care. Compiler should support both vaddb(const char, vector) as well as vaddb(const char4, vector)

Note

Vector logical operations are treated as if the type is "W".

3.6.3.6 Register File Cross Paths

Similar to the C66x, there are two 64-bit cross path which allow functional units from one data path to access a 64-bit operand from the opposite side global register file. The cross paths can not access the local register files directly.

Each cross path sends 64-bit data from the global register file of one side to source 1 operand of a functional unit of the other side.

3.6.3.7 Galois Field

Modern digital communication systems typically make use of error correction coding schemes to improve system performance under imperfect channel conditions. The scheme most commonly used is the Reed-Solomon code, due to its robustness against burst errors and its relative ease of implementation.

The DSP contains Galois field multiply hardware that is used for Reed-Solomon encode and decode functions. To understand the relevance of the Galois field multiply hardware, it is necessary to first define some mathematical terms.

Two kinds of number systems that are common in algorithm development are integers and real numbers. For integers, addition, subtraction, and multiplication operations can be performed. Division can also be performed, if a nonzero remainder is allowed. For real numbers, all four of these operations can be performed, even if there is a nonzero remainder for division operations.

Real numbers can belong to a mathematical structure called a field. A field consists of a set of data elements along with addition, subtraction, multiplication, and division. A field of integers can also be created if modulo arithmetic is performed.

An example is doing arithmetic using integers modulo 2. Perform the operations using normal integer arithmetic and then take the result modulo 2. Table 3-13 illustrates addition, subtraction, and multiplication modulo 2.

Table 3-13. Modulo 2 Arithmetic

Addition				Subtraction				Multiplication		
+	0	1		-	0	1		x	0	1
0	0	1		0	0	1		0	0	0
1	1	0		1	1	0		1	0	1

Note that addition and subtraction results are the same, and in fact are equivalent to the XOR (exclusive-OR) operation in binary. Also, the multiplication result is equal to the AND operation in binary. These properties are

unique to modulo 2 arithmetic, but modulo 2 arithmetic is used extensively in error correction coding. Another more general property is that division by any nonzero element is now defined. Division can always be performed, if every element other than zero has a multiplicative inverse:

$x \times x^{-1} = 1$

Another example, arithmetic modulo 5, illustrates this concept more clearly. The addition, subtraction, and multiplication tables are given in Table 3-14.

Table 3-14. Modulo 5 Arithmetic

Addition							Subtraction							Multiplication					
+	0	1	2	3	4		-	0	1	2	3	4		×	0	1	2	3	4
0	0	1	2	3	4		0	0	4	3	2	1		0	0	0	0	0	0
1	1	2	3	4	0		1	1	0	4	3	2		1	0	1	2	3	4
2	2	3	4	0	1		2	2	1	0	4	3		2	0	2	4	1	3
3	3	4	0	1	2		3	3	2	1	0	4		3	0	3	1	4	2
4	4	0	1	2	3		4	4	3	2	1	0		4	0	4	3	2	1

In the rows of the multiplication table, element 1 appears in every nonzero row and column. Every nonzero element can be multiplied by at least one other element for a result equal to 1. Therefore, division always works and arithmetic over integers modulo 5 forms a field. Fields generated in this manner are called finite fields or Galois fields and are written as GF(X), such as GF(2) or GF(5). They only work when the arithmetic performed is modulo a prime number.

Galois fields can also be formed where the elements are vectors instead of integers if polynomials are used. Finite fields, therefore, can be found with a number of elements equal to any power of a prime number. Typically, we are interested in implementing error correction coding systems using binary arithmetic. All of the fields that are dealt with in Reed Solomon coding systems are of the form $GF(2^m)$. This allows performing addition using XORs on the coefficients of the vectors, and multiplication using a combination of ANDs and XORs.

A final example considers the field $GF(2^3)$, which has 8 elements. This can be generated by arithmetic modulo the (irreducible) polynomial $P(x) = x^3 + x + 1$. Elements of this field look like vectors of three bits. Table 3-15 shows the addition and multiplication tables for field $GF(2^3)$.

Note that the value 1 (001) appears in every nonzero row of the multiplication table, which indicates that this is a valid field.

The channel error can now be modeled as a vector of bits, with a one in every bit position that an error has occurred, and a zero where no error has occurred. Once the error vector has been determined, it can be subtracted from the received message to determine the correct code word.

The Galois field multiply hardware on the DSP is named GMPY4. The GMPY4 instruction performs four parallel operations on 8-bit packed data on the .M unit. The Galois field multiplier can be programmed to perform all Galois multiplies for fields of the form $GF(2^m)$, where m can range between 1 and 8 using any generator polynomial. The field size and the polynomial generator are controlled by the Galois field polynomial generator function register (GFPGFR).

In addition to the GMPY4 instruction, the C674x DSP has the GMPY instruction that uses the GPLY control register as a source for the polynomial.

The GFPGFR, shown in Section 3.7.40, contains the Galois field polynomial generator and the field size control bits. These bits control the operation of the GMPY4 instruction. GFPGFR can only be set via the MVC instruction. The default function after reset for the GMPY4 instruction is field size = 7h and polynomial = 1Dh

3.6.3.7.1 Special Timing Consideration

If the next execute packet after an MVC instruction that changes the GFPGFR value contains a GMPY4 instruction, then the GMPY4 is controlled by the newly loaded GFPGFR value.

Table 3-15. Modulo Arithmetic for Field $GF(2^3)$

Addition								
+	000	001	010	011	100	101	110	111

Table 3-15. Modulo Arithmetic for Field GF(2^3) (continued)

Addition								
000	000	001	010	011	100	101	110	111
001	001	000	011	010	101	100	111	110
010	010	011	000	001	110	111	100	101
011	011	010	001	000	111	110	101	100
100	100	101	110	111	000	001	010	011
101	101	100	111	110	001	000	011	010
110	110	111	100	101	010	011	000	001
111	111	110	101	100	011	010	001	000
Multiplication								
×	000	001	010	011	100	101	110	111
000	000	000	000	000	000	000	000	000
001	000	001	010	011	100	101	110	111
010	000	010	100	110	011	001	111	101
011	000	011	110	101	111	100	001	010
100	000	100	011	111	110	010	101	001
101	000	101	001	100	010	111	011	110
110	000	110	111	001	101	011	010	100
111	000	111	101	010	001	110	100	011

3.7 CPU Control Registers

CPU control registers are physically resided inside the CPU core.

Table 3-16 lists the control registers contained inside the C7x CPU.

Table 3-16. CPU Control Registers

Acronym	Register Name	Section
CID	CPU Identification Register	"CPU Identification Register (CPUID)" on page 1-40
DNUM	DSP Core Number Register	"DSP Core Number Register (DNUM)" on page 1-42
GTSC	Global Time-stamp Counter Register	"Global Time Stamp Counter (GTSC)" on page 1-43
TSC	Time-stamp Counter Register	"Time Stamp Counter (TSC)" on page 1-43
STSC	Shadow Time-stamp Counter Register	"Shadow Time Stamp Counter" on page 1-44
TSR	Task State Register	"Task State Register (TSR)" on page 1-46
RP	Return Pointer Register	"Return Pointer Register (RP)" on page 1-49
PC	Program Counter, E1 phase	"E1 Phase Program Counter (PC)" on page 1-49
BPCFG	Branch Predictor Configuration Register	"Branch Predictor Configuration Register (BPCFG)" on page 1-50
FPCR	Floating-point Configuration Register	"Floating Point Configuration Register (FPCR)" on page 1-50
FSR	Flag Status Register	"Flag Status Register (FSR)" on page 1-51
EPRI	Event Priority Register	"Event Priority Register (EPRI)" on page 1-52
EER	Event Enable Register	"Event Enable Register (EER)" on page 1-53
EESETR	Event set register	"Event Enable Set Register (EESETR)" on page 1-54
EECLR	Event clear register	"Event Enable Clear Register (EECLR)" on page 1-54
DEPR	Debug Event Priority Register	"Debug Event Priority Register (DEPR)" on page 1-55
EFR	Event Flag Register	"Event Flag Register (EFR)" on page 1-56
EFSETR	Event Flag Set Register	"Event Flag Set Register (EFSETR)" on page 1-56
EFCLR	Event Flag Clear Register	"Event Flag Clear Register (EFCLR)" on page 1-57
UFCMR	User Flag Clear Mask Register	"User Flag Clear Mask Register (UFCMR)" on page 1-58

Table 3-16. CPU Control Registers (continued)

Acronym	Register Name	Section
ESTP_SS	Interrupt Service Table Pointer Register, Secure Supervisor Privilege	"Event Service Table Pointer Registers (ETSP_SS/ ETSP_S/ESTP_GS)" on page 1-58
ESTP_S	Interrupt Service Table Pointer Register, Root Supervisor Privilege	"Event Service Table Pointer Registers (ETSP_SS/ ETSP_S/ESTP_GS)" on page 1-58
ESTP_GS	Interrupt Service Table Pointer Register, Guest Supervisor Privilege	"Event Service Table Pointer Registers (ETSP_SS/ ETSP_S/ESTP_GS)" on page 1-58
AHPEE	Active Highest Priority Enabled Event Register	"Active Highest Priority Enabled Event Registers (AHPEE)" on page 1-59
PHPEE	Pending Highest Priority Enabled Event Register	"Pending Highest Priority Enabled Event Registers (PHPEE)" on page 1-59
ECSP_S	Exception Context Save Pointer Register, Root Supervisor Privilege	"Exception Context Save Pointer Registers (ECSP_SS/ECSP_S/ECSP_GS)" on page 1-60
ECSP_GS	Exception Context Save Pointer Register, Guest Supervisor Privilege	"Exception Context Save Pointer Registers (ECSP_SS/ECSP_S/ECSP_GS)" on page 1-60
ECSP_SS	Exception Context Save Pointer Register, Secure Supervisor Privilege	"Exception Context Save Pointer Registers (ECSP_SS/ECSP_S/ECSP_GS)" on page 1-60
TCSP	Task Context Save Pointer Register	"Task Context Save Pointer Registers (TCSP)" on page 1-60
RXMR	Return Execution Mode Register	"Returning Execution Mode Register (RXMR_S, RXMR_SS)" on page 1-60
SPBR	Stack Pointer Boundary Register	"Stack Pointer Boundary Register (SPBR)" on page 1-61
IESET	Internal Event Set Register	"Internal Event Set Register (IESET)" on page 1-62
EDR	Event Dropped Register	"Event Dropped Register (EDR)" on page 1-62
ECLMR	Event Claimed Exception Register	"Event Claimed Exception Registers (ECLMR)" on page 1-63
EASGR	Event Assigned Exception Register	"Event Assigned Exception Registers (EASGR)" on page 1-63
UEMR	User Mask Enable Register	"User Mask Enable Register (UEMR)" on page 1-64
GMER	Guest Mode Enable Register	"Guest Mode Enable Register (GMER)" on page 1-64
TCR	Test Counter Register	"Test Counter Register (TCR)" on page 1-65
TCCR	Test Counter Control Register	"Test Counter Control Register (TCCR)" on page 1-66
IERR	Internal Exception Report Register	"Internal Exception Report Register (IERR)" on page 1-67
IEAR	Internal Exception Address Register	"Internal Exception Address Register (IEAR)" on page 1-69
IEDR	Internal Exception Data Register	"Internal Exception Data Register (IEDR)" on page 1-69
IESR	Internal Exception Fault Status Register	"Internal Exception Fault Status Register (IESR)" on page 1-70
GFPGFR	Galois field multiply control register	"Galois Field Polynomial Generator Function Register (GFPGFR)" on page 1-70
GPLY	GMPY polynomial register	"GMPY Polynomial register (GPLY)" on page 1-71
LTBR0-3	Lookup Table Base Address configuration registers	".Look Up Table Base Address Registers (LTBR0-3)" on page 1-71
LTCR0-3	Lookup Table configuration registers	"Look Up Table and Histogram Configuration Registers (LTCR0-3)" on page 1-71
LTER	Lookup Table Enable register	"Look Up Table Enable Register (LTER)" on page 1-73
CUCR0-3	.C unit Control Registers	".C Unit Control Registers (CUCR0-CUCR3)" on page 1-74
STRACR	Streaming Address Generator Control Register	"Streaming Address Control Registers (STRACR0 -3)" on page 1-75

Table 3-16. CPU Control Registers (continued)

Acronym	Register Name	Section
STRACNTR	Streaming Address Generator Counter Register	"Streaming Address Count Registers (STRACNTR0-3)" on page 1-76

3.7.1 Pipeline/Timing of CPU Control Register Accesses

MVC instructions which transfer data from general registers to control registers are either single-cycle or two-cycle instructions, depending on whether the read source is control registers or general register. For all cases, read access of the explicitly named registers occurs in the E1 pipeline phase, and write occurs in either E1 or E2, depending on the specific MVC instruction. In all cases, the source register content is read, moved through the appropriate functional unit, and written to the destination register in the E1 pipeline phase.

Table 3-17. TBD

Pipeline Stage	E1	E2
Read	src2	
Written	dst	dst
Unit in use	S/C/M	S

3.7.2 CPU Identification Register (CPUID)

The CPU identification register (CPUID) contains the CPU ID and its revision number. The CPUID is shown in Figure 1-16 and described in Table 1-17.

RESET -x = value is indeterminate after reset

Table 3-18. CPU Identification Register (CID)

63																32
Reserved																
R-0																
31				28	27											16
SCHEME				BU	CPU ID											
R-x[1]				R-x[1]	R-x[1]											
15			11	10		8	7		6	5						0
RTL(R)				MAJOR(X)			CUSTOM			MINOR(Y)						
R-x[1]				R-x[1]			R-x[1]			R-x[1]						

Table 3-19. CPUID Register Field Descriptions

Bit	Field	Type	Reset	Description
63-32	RESERVED	R	0	
31-30	SCHEME	R	X	Used to distinguish between old scheme and new scheme. The difference of two schemes are the BU field. C7x CPU ties this field to 2'h1 0h = Old scheme (HL01, HL08) 1h = New scheme, reset value 2h - 3h = Reserved
29-28	BU	R	X	Business Indicator. C7x CPU ties this field to 2'h2. 0h = DSPS 1h = WTBU 2h = Processors 3h = Reserved
27-16	CPUID	R	X	0-FFh = Identifies the CPU of the device. Not writable by the MVC instruction. For C711, the CPU ID is 0x60C0

Table 3-19. CPUID Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
15-11	RTL(R)	R	X	0h = RTL version. R as described in PDR with additional clarifications/definitions below. Must be easily ECO-able or controlled during fabrication. Ideally through a top level metal mask or e-fuse. This number is maintained/ owned by IP design owner. RTL follows a numbering such as X.Y.R.Z R changes ONLY when: (1) PDS uploads occur which may have been due to spec changes (2) Bug fixes occur (3) Resets to '0' when X or Y changes. Design team has an internal 'Z' (customer invisible) number which increments on every drop that happens
10-8	MAJOR(X)	R	X	0h = Major Revision. X as described in PDR with additional clarifications/definitions below. This number is owned/maintained by IP specification owner. X is part of IP numbering X.Y.R.Z. X changes ONLY when: (1) There is a major feature addition. An example would be adding Master Mode to Utopia Level2. The Func field (or Class/Type in old PID format) will remain the same. X does NOT change due to: (1) Bug fixes (2) Change in feature parameters.
7-6	CUSTOM	R	X	0h = Indicates a special version for a particular device. Consequence of use may avoid use of standard Chip Support Library (CSL) / Drivers. 0 if non-custom.
5-0	MINOR(Y)	R	X	0h = Minor Revision. Y as described in PDR with additional clarifications/definitions below. This number is owned/maintained by IP specification owner. Y changes ONLY when: (1) Features are scaled (up or down). Flexibility exists in that this feature scalability may either be represented in the Y change or a specific register in the IP that indicates which features are exactly available. (2) When feature creeps from Is-Not list to Is list. But this may not be the case once it sees silicon; in which case X will change. Y does NOT change due to: (1) Bug fixes (2) Typos or clarifications (3) major functional/feature change/addition/deletion. Instead these changes may be reflected via R, S, X as applicable. Spec owner maintains a customer-invisible number 'S' which changes due to: (1) Typos/clarifications (2) Bug documentation. Note that this bug is not due to a spec change but due to implementation. Nevertheless, the spec tracks the IP bugs. An RTL release (say for silicon PG1.1) that occurs due to bug fix should document the corresponding spec number (X.Y.S) in its release notes.

3.7.3 DSP Core Number Register (DNUM)

Multiple CPU may be used in a system. The DSP core number register (DNUM) provides an identifier to shared resources in the system to identify which CPU is accessing those resources. The contents of this register are set to a specific value (depending on the device) at reset. See your device-specific data manual for the reset value of this register. The DNUM is shown in Figure 1-17.

S = See the device-specific data manual for the default value of this field after reset

Table 3-20. DSP Core Number Register (DNUM)

63 24	23 16	15 8	7 0
Reserved	Cluster number	Corepac number	CPU number
R-0	R-S	R-S	R-S

Table 3-21. DNUM Register Field Descriptions

Bit	Field	Type	Reset	Description
63-24	RESERVED	R	0	
23-16	CLUSTER_NUM	R	S	0h = The cluster number the DSP belongs to.
15-8	COREPAC_NUM	R	S	0h = The CorePac number the DSP belongs to within the specified cluster.
7-0	CPU_NUM	R	S	0h = The CPU number within the specified CorePac and cluster.

3.7.4 Global Time Stamp Counter (GTSC)

The TI Keystone3 architecture provides a global timestamp scheme. This 64-bit global timestamp is broadcast across the chip that a debugger can use for coarse-grained profiling, and correlation of trace sources. At the compute cluster level, the system timestamp is generated at the SOC clock rate before broadcasting to each C7x CPU in the cluster.

The C7x CPU supports connection to the compute cluster level global timestamp source therefore user can have access to the global timestamp counter value by executing MVC instructions.

The GTSC register is accessed as a single 64-bit read-only control register as shown below:

Table 3-22. Global Time Stamp Counter (GTSC)

63#unique_64/unique_64_Connect_42_GUID-90312DA4-1702-4EE0-ACA0-E668392DEB0F 0
Global Time Stamp Counter
R-S

Table 3-23. GTSC Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	GTSC	R	S	S = See the device-specific data manual for the default value of this field after reset

3.7.5 Time Stamp Counter (TSC)

The C7x CPU also provides a local time stamp counter register and the corresponding shadow register. To maintain consistency when reading global timestamp and local offset timestamp values, (GTSC and TSC are two separate registers), the user should use MVCI instruction which reads both GTSC and TSC in parallel.

The C7x CPU contains a free running 64-bit time stamp counter (TSC) that advances each CPU clock under normal operation. The TSC register is meant to be used to time interval counter using the GTSC register as the reference counter. Therefore, the TSC register is only synchronized to the 64-bit GTSC value at the power-on reset, but the TSC register can be reset to all 0's by executing MVC to GTSC instruction to write 0 to the GTSC.

In multi-core devices, these timestamp counters enable users to monitor the synchronization between tasks running on different cores. By monitor the TSCs, user would be able to detect when the task on one core has completed before kicking off the task on another core.

All supervisor and user privileges can read the TSC counter. Only supervisor can reset TSC by using MCVI instruction.

TSC register is accessed as a single 64-bit read-only control register as shown below

Table 3-24. Time Stamp Counter (TSC)

63 0

Table 3-24. Time Stamp Counter (TSC) (continued)

Time Stamp Counter
R-S

Table 3-25. TSC Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	TSC	R	S	S = See the device-specific data manual for the default value of this field after reset

3.7.6 Shadow Time Stamp Counter (STSC)

To maintain atomicity, the C7x CPU also provides the shadow TSC (STSC) register. The STSC snapshots the TSC value at the same cycle when GTSC is read by the SW. This way the hardware guarantees both global timestamp counter and local stamp counter values are sampled at the same cycle. STSC is a single 64-bit read-only control register which has the same format as TSC with reset value of all 0's.

All supervisor and user privileges can read the STSC counter.

Table 3-26. Shadow Time Stamp Counter (STSC)

63#unique_66/unique_66_Connect_42_GUID-706926DE-7FDB-4F15-965D-AB75B11C7461	0
Shadow Time Stamp Counter	
R-S	

Table 3-27. STSC Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	STSC	R	S	S = See the device-specific data manual for the default value of this field after reset

3.7.6.1 Timestamp Counter Initialization Example

The local timestamp counter TSC is initialized after reset or by software to all 0's:

1. The TSC counter is enabled right after reset and starts advance each CPU clock cycle right after the reset is de-asserted.
2. The C7x CPU provides TSC reset on-demand by executing MVC read of GTSC value.

By executing the GTSC read instruction with TSC reset option on, the C7x CPU completes three functions:
MVK .S1 0x1, A10 // Set the reset TSC option bit MVCI .S1 GTSC, A10, A0 // Read GTSC, TSC then reset TSC

- Samples 64-bit GTSC bus before writing into A0 register
- Reads TSC register value at the same cycle and write it into the shadow register
- STSC
- Reset TSC register to all 0's

> **CAUTION**
>
> Note that TSC can only be reset when the CPU is in supervisor mode. User mode will not reset TSC value.

3.7.6.2 Disabling Time Stamp Counters

Once enabled, counting cannot be disabled under program control. Countering is disabled/restarted in the following cases:

- After POR reset, the counter restarts
- When the CPU is fully powered down

3.7.6.3 Reading Time Stamp Counters

Reading the full 64-bit count takes one MVC instructions. There are additional functions for user to read and reset individual or multiple timestamp counters in one shot. The table below shows example instructions and their effects on the GTSC, TSC and STSC registers .

Table 3-28. TBD

Instruction	Privilege	A0	GTSC	STSC	TSC
MVCI .S1 GTSC, A10, A0; A10=0'b1	Supervisor	GTSC value	Continue counting	Updated TSC value	Reset to all 0's
	User	Not allowed since user cannot reset TSC, takes exception			
MVCI .S1 GTSC, A10, A0; A10=0'b0	Any	GTSC value	Continue counting	No change	Continue counting as is
MVC .S1 GTSC, A0;	Any	GTSC value	Continue counting	No change	Continue counting as is
MVC .S1 TSC, A0;	Any	TSC value	Continue counting	No change	Continue counting as is
MVC .S1 STSC, A0;	Any	STSC value	Continue counting	No change	Continue counting as is

```
Example 1-1        Example code to read GTSC and TSC at the same cycle then reset TSC to 0's
MVK .S1 0x1, A10;
MVCI .S1 GTSC, A10, A0; // Has to be executed in supervisor mode since reseting TSC.
MVC .S1 STSC, A1;
```

```
vExample 1-2        Example code to read GTSC and TSC at the same cycle, no reset of TSC
MVK .S1 0x0, A10;
MVCI .S1 GTSC, A10, A0; // Can executed in any privilege level.
MVC .S1 STSC, A1;
```

3.7.7 Task State Register (TSR)

The task state register (TSR) contains all of the status bits that determine or indicate the current execution environment. TSR is saved in the event of an interrupt or exception to the ITSR or NTSR, respectively. All bits are readable by the MVC instruction. The TSR is shown in Figure 1-21 and described in Table 1-20.

Table 3-29. Task State Register (TSR)

63#unique_70/unique_70_Connect_42_GUID-D380849C-76F9-4F06-91A6-E97B7B1C56C9 57	56	55 53	52	51 49	48
Reserved	EN	Reserved	HWA0_Present	Reserved	HWA0

47 44	43	42	41	40	39 34	33	32
Reserved	SA3	SA2	SA1	SA0	Reserved	SE1	SE0

31 29	28	27 26	25	24	23 19	18 17	16
Reserved	DBGM	SUDEN	GEE	PROT	MCOLOR	Reserved	COP

15 8	7 4	3	2 0
COP	Reserved	HDL	CXM

Table 3-30. TSR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-57	RESERVED	R	0	
56	EN	R	0	Endian Mode 0h = Little Endian 1h = Big Endian
55-53	RESERVED	R	0	

Table 3-30. TSR Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
52	HWA_PRESENT	R	0	Indicates if there is a HWA (Hardware Accelerator) attached to the CPU 0 = No HWA 1 = HWA Present
51-49	RESERVED	R	0	
48	HWA0	R	0	Hardware Accelerator 0 is active. Not writable by the MVC instruction 0h = Hardware Accelerator 0 is closed. This bit is set to 0 at reset and upon executing HWACLOSE instruction 1h = Hardware Accelerator 0 is opened. This bit is set to 1 upon executing HWAOPEN instruction
47-44	RESERVED	R	0	
43	SA3	R	0	Streaming Address 3 is active. Not writable by the MVC instruction. 0h = Streaming Address 3 is closed. This bit is set to 0 at reset and upon executing SACLOSE 3 instruction 1h = Streaming Address 3 is open and active. This bit is set to 1 upon executing SAOPEN 3 instruction
42	SA2	R	0	Streaming Address 2 is active. Not writable by the MVC instruction. 0h = Streaming Address 2 is closed. This bit is set to 0 at reset and upon executing SACLOSE 2 instruction 1h = Streaming Address 2 is open and active. This bit is set to 1 upon executing SAOPEN 2 instruction
41	SA1	R	0	Streaming Address 1 is active. Not writable by the MVC instruction. 0h = Streaming Address 1is closed. This bit is set to 0 at reset and upon executing SACLOSE 1 instruction 1h = Streaming Address 1 is open and active. This bit is set to 1 upon executing SAOPEN 1 instruction
40	SA0	R	0	Streaming Address 0 is active. Not writable by the MVC instruction. 0h = Streaming Address 0 is closed. This bit is set to 0 at reset and upon executing SACLOSE 0 instruction 1h = Streaming Address 0 is open and active. This bit is set to 1 upon executing SAOPEN 0 instruction
39-34	RESERVED	R	0	
33	SE1	R	0	Streaming Engine 1 is active. Not writable by the MVC instruction. 0h = Streaming Engine 1is closed. This bit is set to 0 at reset and upon executing SECLOSE 1 instruction 1h = Streaming Engine 1is open and active. This bit is set to 1 upon executing SEOPEN 1 instruction
32	SE0	R	0	Streaming Engine 0 is active. Not writable by the MVC instruction. 0h = Streaming Engine 0 is closed. This bit is set to 0 at reset and upon executing SECLOSE 0 instruction 1h = Streaming Engine 0 is open and active. This bit is set to 1 upon executing SEOPEN 0 instruction
31-29	RESERVED	R	0	
28	DBGM	R	0	Debug access mask, writable by DDBG and EDBG instructions 0h = Enable debug accesses 1h = Disable debug accesses

Table 3-30. TSR Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
27-26	SUDEN	R	0	Secure User Debug Enable -- writable by instruction ESUDBG, DSUDBG 0h = Reserved 1h = SUIDEN: Secure User Invasive Debug Enable. This field can only be set by executing ESUDDBG instruction; and de-asserted by executing DSUDBG instruction 2h = SUNIDEN: Secure User Non-invasive Debug Enable . This field can only be set by executing ESUDBG instruction; and de-asserted by executing DSUDBG instruction. 3h = Reserved
25	GEE	R	0	Global Event Enable. • This bit can be written by hardware or through execution of DINT or RINT instruction. • Using MVC instruction to access this field is not allowed and will trigger exception • GEE is set to 0 after reset, meaning all external events are disabled after reset. • Program should execute RINT to enable any events to be taken. • If the CPU is in non-secure mode, all non-secure interrupts are masked by priority scheme. • In secure mode, GEE will disable or enable secure events, all non-secure events are disabled. 0h = Events at current execution mode (indicated by TSR.CXM) are disabled. All events at lower execution mode are disabled by priority rule. 1h = Events at current execution mode (indicated by TSR.CXM) are enabled. All events at lower execution mode are disabled by priority rule.
24	PROT	R	0	Pipeline Mode bit. Not writable by the MVC instruction. Can only be set or cleared by executing PROT, PROTCLR or UNPROT instructions. Default value is set to 1. 0h = Unprotected 1h = Protected
23-19	COLOR	R	0	Current Memory Color Tag. This field can only be updated by executing the instruction MTAG. 0h = Memory Color Tag 0 1h = Memory Color Tag 1
18-17	RESERVED	R	0	

Table 3-30. TSR Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
16-8	COP	R	0	Current Operating Priority. This field can be written by hardware through execution of SETCOP instruction. • COP is set by hardware as -1 at POR reset to disable all internal and external events to be taken. • SW should execute SETCOP to enable user tasks and event handling. • Using MVC instruction to access this field is not allowed and will trigger exception FFh = Default Task Mode 0h-E0h = Event Priority number when TSR.EVT = 1'b1 1FFh = Double fault or right after POR reset. This state can only be derived from double fault state, meaning no event could trigger CPU to switch from normal mode to triple fault mode without going through double fault mode first. 1FEh = Triple fault, or Machine fault
7-4	RESERVED	R	0	
3	HDL	R	0	Current Processing Mode. Indicates currently executing event handling codes. • Hardware automatically modifies this field upon entering event service routines. • Software can modify this field by executing RETE with proper permission • Using MVC instruction to access this field is not allowed and will trigger exception 0h = Program Processing 1h = Event Handler Processing
2-0	CXM	R	0	Current execution mode. These bits reflect the current execution mode of the execute pipeline. The CXM field is not writable by the MVC instruction; It is updated automatically by the CPU upon these events: reset, exceptions, system call, return from events, returns from system call. CXM is initialized to value of 3'h5 (Secure Supervisor) after reset. The bits are interpreted as follow: 0h = Guest User (GU) 1h = Guest Supervisor (GS) 2h = Root User (U) 3h = Root Supervisor (S) 4h = Secure User (SU) 5h = Secure Supervisor (SS) 6h = Reserved 7h = Reserved

3.7.8 Return Pointer Register (RP)

The return pointer register (RP) contains the return pointer that directs the CPU to the proper location to continue program execution after return from a function call. Executing RET instruction causes a branch to be taken using the address in RP.

The RP register is updated automatically by hardware under two circumstances:

- When a CALL instruction is executed, the CPU writes the PC of the first execute packet after the execute packet of the CALL instruction into the RP register

- When triple fault occurs, the CPU will not recover to pre-fault program flow. The CPU will not push current context in any context saving region, but only write the current PC into the RP register.

Programmer can write a value to RP register using MVC instruction. The MVC provided value is checked for address errors before committing to the RP register

Table 3-31. RP Register Field Descriptions

Bit	Field	Type	Reset	Description
63-49	Reserved	R	0	
48-0	RP	R/W	X	TBD

Table 3-32.

Bit Fields	Check Performed	Notes
63-48	16'h0000 or 16'hFFFF	Sign-extension check to ensure bits 63 to 49 are sign
extension of bit 48. If the check fails, the write will not be		
committed. A write error exception will be		
triggered.		
1-0	2'h0	HW checks whether the 2 least significant bits are 2'h0 since
the RP should align to the word boundary. If the check fails, the write will not be		
committed. A write error exception will be		
triggered.		

3.7.9 E1 Phase Program Counter (PC)

The E1 phase program counter (PC), shown in Figure 1-23, contains the 48-bit virtual address of the fetch packet in the E1 pipeline phase.

Table 3-33. PC Register Field Descriptions

Bit	Field	Type	Reset	Description
63-49	Reserved	R	0	
48-0	PC	R	X	TBD

3.7.10 Branch Predictor Configuration Register (BPCFG)

Branch Predictor Configuration Register control the parameters used by the Branch Predictors.

Table 3-34. BPCFG Register Field Descriptions

Bit	Field	Type	Reset	Description
63-1	Reserved	R	0	
0	BPOFF	R/W	0	Disables Branch Predictor Logic 0h = Enables Branch Predictor 1h = Disable Branch Predictor

3.7.11 Floating Point Configuration Register (FPCR)

The floating-point status register (FPCR) contains field that contain the rounding mode for floating point operations. FPCR is shown in Figure 1-65 and described in Table 1-64

Table 3-35. FPCR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-5	Reserved	R	0	
4	FTZ	R/W	1	0h = IEEE 754 compliance mode. 1h = Flush to zero mode. Denorm values are treated as zeros. This is the default mode upon reset.

Table 3-35. FPCR Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
3-2	Reserved	R	0	
1-0	RMODE	R/W	0	Rounding mode for floating point operations 0h = Round toward nearest representable floating-point number 1h = Round toward 0 (truncate) 2h = Round toward infinity (round up) 3h = Round toward negative infinity (round down)

3.7.12 Flag Status Register (FSR)

The flag status register (FSR) contains fields that show the status of fixed point saturation operations and floating point operations such as underflow, overflow, NaNs, denormalized, infinity and inexact results. Each 8-bit segments of the FSR contains the status of the operations of each 64-bit vector slice.

Table 3-36. Flag Status Register (FSR)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SAT_7	UNORD_7	DEN_7	INEX_7	UNDER_7	OVER_7	DIV0_7	INVAL_7	SAT_6	UNORD_6	DEN_6	INEX_6	UNDER_6	OVER_6	DIV0_6	INVAL_6
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SAT_5	UNORD_5	DEN_5	INEX_5	UNDER_5	OVER_5	DIV0_5	INVAL_5	SAT_4	UNORD_4	DEN_4	INEX_4	UNDER_4	OVER_4	DIV0_4	INVAL_4
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SAT_3	UNORD_3	DEN_3	INEX_3	UNDER_3	OVER_3	DIV0_3	INVAL_3	SAT_2	UNORD_2	DEN_2	INEX_2	UNDER_2	OVER_2	DIV0_2	INVAL_2
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SAT_1	UNORD_1	DEN_1	INEX_1	UNDER_1	OVER_1	DIV0_1	INVAL_1	SAT_0	UNORD_0	DEN_0	INEX_0	UNDER_0	OVER_0	DIV0_0	INVAL_0
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
LEGEND: R = Readable by the MVC instruction; W = Writable by the MVC instruction; -n = value after reset															

Table 3-37. FSR Register Field Descriptions

Bit	Field	Type	Reset	Description
7	SAT	R/W	0	Result is saturated -- Fixed Point Only 0h = Result is saturated 1h = Result is not saturated
6	UNORD	R/W	0	Compare involves sources contain NaN 0h = does not contains NaN 1h = contain NaN
5	DEN	R/W	0	Sources contain denormalized number 0h = does not contains denormalized number 1h = contains denormalized number
4	INEX	R/W	0	Inexact results status for .S1 0h = 1h = Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
3	UNDER	R/W	0	Result underflow status for .S1 0h = Result does not underflow 1h = Result underflow

Table 3-37. FSR Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
2	OVER	R/W	0	Result overflow status for .S1 0h = Result does not overflow 1h = Result overflows
1	DIV0	R/W	0	Source to reciprocal/divide operation 0h = 0 is not source to reciprocal/divide operation 1h = 0 is source to reciprocal/divide operation
0	INVAL	R/W	0	Sources contain SNAN 0h = A signed NaN (SNaN) is not a source 1h = A signed NaN (SNaN) is a source

3.7.13 Event Priority Register (EPRI)

The Event Priority Register (EPRI) enables and disables individual events.

EPRI register is not accessible in User mode.

EPRI is an indexed register. For the enabled events, EPRI[n] contains the priority level of the event n. The indexing rule of EPRI[n] registers are different from other index registers who are indexed by the number of 64-bit arrays. The index number of EPRI[n] is the event number indicated by the CRINDEX register. Since the event number is passed to corresponding EPRI by writing into the register CRINDEX, one needs to guarantee the atomicity between the CRINDEX update and EPRI[CRINDEX] update. When changing EPRI[n], programmer should disable all events by set ting GEE = 0x0, then write the event number in the index register CRINDEX, then update the EPRI[CRINDEX] value before enabling all events again.

Table 3-38. EPRI Register Field Descriptions

Bit	Field	Type	Reset	Description
63-8	RESERVED	R	0	
7-5	PRI	R	7h	Event priority. Indicates the priority level of the corresponding event. n is the event number. An event priority may be manually set by using MVC instruction with owner privilege according to scope rule. 0h = Event is not claimed to be secure event 1h = Event is claimed to be secure event
4-0	Reserved	R	Fh	

3.7.14 Event Enable Register (EER)

The event enable register (EER) shows whether the individual event is enabled or disabled. It is a read-only status register, using MVC to modify the bit value directly will be ignored.

To set the bits in EER, one need to set the corresponding bits in EESETR register. To clear the bits in EER, one need to set the corresponding bits in EECLR register.

Table 3-39. EER Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	EVTn	R	0	Event enable. An event triggers event handling only if the corresponding bit is set to 1. 0h = Event is disabled 1h = Event is enabled

3.7.15 Event Enable Set Register (EESETR)

The Event Enable Set register (EESETR) allows programmer to enable corresponding event(s) in the Event Enable register (EER). Writing a 1 to any of the bits in EESETR causes the corresponding Event Enable bit (EER.EVTn) to be set in EER. Writing a 0 to any bit in EESETR has no effect. EESETR cannot set any bit in EER without proper ownership defined by the combination of ECLMR and EASGR.

Table 3-40. EESETR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	EVTn	W	0	Event enable Set 0h = Corresponding Event Enable bit (EER.EVTn) is not set 1h = Corresponding Event Enable bit (EER.EVTn) is set

3.7.16 Event Enable Clear Register (EECLR)

The Event Enable Clear register (EECLR) allows programmer to disable corresponding events in the Event Enable Register (EER). Writing a 1 to any of the bits in EECLR causes the corresponding Event Enable bit (EE.EVTn) to be masked in EER. Writing a 0 to any bit in EECLR has no effect.

EECLR cannot clear any bit in EER without proper ownership defined by the combination of ECLMR and EASGR

Table 3-41. EECLR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	EVTn	W	0	Event enable Set 0h = Corresponding Event Enable bit (EER.EVTn) is not set 1h = Corresponding Event Enable bit (EER.EVTn) is set

3.7.17 Debug Event Priority Register (DEPR)

Debug event priority register allows debugger to set the CPU execution priority when the CPU is halted in debug mode.

Table 3-42. DEPR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-8	RESERVED	R	0	
7-5	PRI	R	7h	Event priority. Indicates the priority level of the corresponding event. n is the event number. An event priority may be manually set by using MVC instruction with owner privilege according to scope rule. 0h = Event is not claimed to be secure event 1h = Event is claimed to be secure event
4-0	RESERVED	R	Fh	

3.7.18 Event Flag Register (EFR)

The Event Flag Register, EFR, contains the status of currently pending events. Each corresponding bit in the EFR is set to 1 by hardware when that event occurs, otherwise, the bits are cleared to 0. To check the status of events, MVC instruction can be used to read the EFR.

All privilege may read EFR. No privilege level may write EFR directly. Programs can only modify the contents of EFR indirectly by writing to the Event Flag Set Register (EFSETR) or Event Flag Clear Register (EFCLR) to set or clear the corresponding bits in EFR.

EFR is claim-scoped. Root Supervisor only sees events not claimed by Secure Supervisor in ECLMR; the rest read as 0.

Guest Supervisor only sees events assigned to it by EASGR; the rest read as 0. User code (SU, U, GU) can not set or clear EFR. Any attempt to do so results in an privilege exception. EFR is an indexed register.

Table 3-43. EFR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	EVTn	R	0	Event flag. Indicates the status of the corresponding external events. An event flag may be manually set by setting the corresponding bit in the Event Flag Set Register (EFSETR) or manually cleared by setting the corresponding bit in the Event Flog Clear Register (EFCLR). 0h = Event is not pending 1h = Event has occurred and is still pending

3.7.19 Event Flag Set Register (EFSETR)

The Event Flag Set register (EFSETR) allows programmer to enable corresponding event(s) in the Event Flag Register (EFR). Writing a 1 to any of the bits in EFSETR causes the corresponding Event Flag bit (EFR.EVTn) to be set in EFR. Writing a 0 to any bit in EFSETR has no effect.

EFSETR is claim-scoped. Secure Supervisor can set or clear any pending event. Root Supervisor can set or clear any event not claimed by Secure Supervisor. Guest Supervisor can set or clear any event assigned to it.

Table 3-44. EFSETR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	EVTn	W	0	Event flag set. 0h = Corresponding Event Flag bit (EFR.EVTn) is not set 1h = Corresponding Event Flag bit (EFR.EVTn) is set

3.7.20 Event Flag Clear Register (EFCLR)

The Event Flag Clear register (EFCLR) allows programmer to disable corresponding events in the Event Flag Register (EFR).

Writing a 1 to any of the bits in EFCLR causes the corresponding Event Flag bit (EE.EVTn) to be masked in EFR.

Writing a 0 to any bit in EFCLR has no effect.

EFCLR is claim-scoped. Secure Supervisor can set or clear any pending event. Root Supervisor can set or clear any event not claimed by Secure Supervisor. Guest Supervisor can set or clear any event assigned to it.

Table 3-45. EFCLR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	EVTn	W	0	Event flag clear 0h = Corresponding Event Flag bit (EFR.EVTn) is not set 1h = Corresponding Event Flag bit (EFR.EVTn) is set

3.7.21 User Flag Clear Mask Register (UFCMR)

The User Flag Clear Mask Register (UFCMR) when enabled, allows all permissions to read and clear the corresponding EFR bit. The write is masked by the ECLMR/EASGR depending on the security at the setup time.

Writing a 1 to any of the bits in UFCMR causes the corresponding Event Flag bit (EE.EVTn) to be masked in EFR.

Writing a 0 to any bit in UFCMR has no effect.

Table 3-46. UFCMR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	EVTn	W	0	Event flag clear 0h = Corresponding Event Flag bit (EFR.EVTn) is not set 1h = Corresponding Event Flag bit (EFR.EVTn) is set

3.7.22 1.4.7.22 Event Service Table Pointer Registers (ETSP_SS/ETSP_S/ESTP_GS)

The C7x Core supports 6 different privilege levels: Secure Supervisor (SS), Secure User (SU), Root Supervisor (S), Root User (U), Guest Supervisor (GS), and Guest User (GU).

Interrupts and exception are handled while the processor are in supervisor modes. Therefore, most interrupts/exception related control registers have 3 copies: one is used to control and/or record status during SS mode of operation, one is used to control and/or record status during S mode of operation, and another one is used to control and/or record status during GS mode of operation.

The Event Service Table Pointer registers (ESTP_SS, ESTP_S and ESTP_GS) are used to locate the event service routine (ESR) during the respective privilege mode. The ESTB field identifies the base portion of the address of the event service table (EST) and the HPEINT field identifies the specific event and locates the specific fetch packet within the EST. The ESTP is shown in Figure 1-36 and described in Table 1-34. The ISTP registers are not accessible in User modes.

Table 3-47. ESTP_SS/ESTP_S/ESTP_GS Register Field Descriptions

Bit	Field	Type	Reset	Description
63-49	RESERVED	R	0	
48-11	ISTB	R/W	S	FFFFh = Interrupt service table base portion of the IST address. This field is cleared to a device-specific default value on reset; therefore, upon startup the IST must reside at this specific address. See the device-specific data manual for more information. After reset, you can relocate the IST by writing a new value to ISTB. If relocated, the first ISFP (corresponding to RESET) is never executed via interrupt processing, because reset clears the ISTB to its default value. S = See the device-specific data manual for the default value of this field after reset.
10-5	HPEINT	R	0	
4-0	0	R	0	0 Cleared to 0 (fetch packets must be aligned on 8-word (32-byte) boundaries).

3.7.23 Active Highest Priority Enabled Event Registers (AHPEE)

The Active Highest Priority Enabled Event Register (AHPEE) contain the highest event number which is in serviced. The AHPEE registers are shown in Figure 1-23.

Table 3-48. AHPEE Register Field Descriptions

Bit	Field	Type	Reset	Description
63-6	RESERVED	R	0	
5-0	NUM	R	0	

3.7.24 Pending Highest Priority Enabled Event Registers (PHPEE)

The Pending Highest Priority Enabled Event Register (PHPEE) contain the highest event number which is waiting to be serviced. The PHPEE registers are shown in Figure 1-23.

Table 3-49. PHPEE Register Field Descriptions

Bit	Field	Type	Reset	Description
63-6	RESERVED	R	0	
5-0	NUM	R	0	

3.7.25 Exception Context Save Pointer Registers (ECSP_SS/ECSP_S/ECSP_GS)

The Exception Context Save Pointer registers (ECSP_SS, ECSP_S, and ECSP_GS) contain the pointer that directs the CPU to the memory location where the machine contexts are stored upon entering and/or returning from interrupts in order to save and restore the correct machine states.

Table 3-50. ECSP_SS/ECSP_S/ECSP_GS Register Field Descriptions

Bit	Field	Type	Reset	Description
63-49	RESERVED	R	0	
48-0	ECSP	R/W	X	

3.7.26 Task Context Save Pointer Registers (TCSP)

The Task Context Save Pointer register (TCSP) contain the pointer that directs the CPU to the memory location where the machine contexts are stored upon entering and/or returning from events in order to save and restore the correct machine states while in user mode.

Table 3-51. TCSP Register Field Descriptions

Bit	Field	Type	Reset	Description
63-49	RESERVED	R	0	
48-0	TCSP	R/W	X	

3.7.27 Returning Execution Mode Register (RXMR_S, RXMR_SS)

In C7x Event Handling Architecture, the event entrance mechanism enables the CPU hardware to push the preempted task or event handler contexts onto the corresponding stacks. In the event exit/return mechanism, a reserve operation is executed by the hardware to automatically unstacking the preempted context. Only key information missing is which stack to recover from. C7x gives programmer full control to decide which program/handler to return to in the event exist process. To be precise, only privileged supervisor can redirect the event exist to a different privilege level. A control register Returning Execution Mode Register (RXMR) is dedicated for programmer to tell the hardware which execution mode will be resumed upon event return. By sampling RXMR register, the hardware is able to select the corresponding context stack to restore the context.

The lower three bits are the resumed TSR.CXM field which has the same encoding as TSR.CXM encoding.

RXMR register is privileged and is only provided to Secure and Root Supervisors:

- RXMR_SS -- Secure Supervisor Returning Execution Mode Register. The valid returning execution modes in this register are:
  - SU
  - S, U
  - GS, GU
- RXMR_S -- (Root) Supervisor Returning Execution Mode Register. The valid returning execution modes in this register are:
  - U
  - GS, GU

During event handling, both hardware and software has accesses to the RXMR_SS and RXMR_S registers with proper privileges.

- At event entrance, the hardware writes the current TSR.CXM into the corresponding RXMR_S/SS register
- At event return, program can write the RXMR_SS/S register with a different returning execution mode to redirect the exits.

Table 3-52. RXMR_S, RXMR_SS Register Field Descriptions

Bit	Field	Type	Reset	Description
63-35	RESERVED	R	0	
34-32	CXM.RETS	W	0	Returning Execution Mode when RETS is executed
31-3	RESERVED	R	0	
2-0	CXM_RETE	W	0	Returning Execution Mode when RETE is executed

3.7.28 Stack Pointer Boundary Register (SPBR)

Stack pointer boundary. It can be used to compare with the Stack Pointer (normally A15 register) to detect stack overflow.

Table 3-53. SPBR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-49	RESERVED	R	0	
48-0	SPBR	R/W	X	

3.7.29 Internal Event Set Register (IESET)

The Internal Event Set Register (IESET) allows programmer to set synchronous or asynchronous internal events by setting a bit in the IESET register using MVC instruction.

Table 3-54. IESET Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	EVTn	R	1	Setting an internal event 0h = Internal Event EVTn is not set 1h = Internal Event EVTn is set

3.7.30 Event Dropped Register (EDR)

EDR register captures dropped events. If an event is in pending state, the corresponding EFR bit is 1'b1. If the same event is raised again by the source when the event is still in pending mode, that event is dropped. This means the first call of the event has not been taken by the CPU yet, the second call of the same event is issued and dropped. When event is dropped, the CPU will send an event-dropped event to notify the system interrupt controller.

Table 3-55. EDR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-6	RESERVED	R	0	
5-0	NUM	R	0	

3.7.31 Event Claimed Exception Registers (ECLMR)

The Event Claim Register (ECLMR) allows Secure Supervisor (SS) to claim certain interrupt/exception events as secure. Events marked as secure in ECLMR will only be delivered to SS.

The ECLMR has one bit per event number, as follows.

The width of ECLMR depends on the number of events supported on a given C7x implementation. For an implementation that supports N interrupt numbers, bits 0 through N-1 contain valid data. ECLMR is an indexed register.

After POR reset, all events are claimed to be secure by default.

Table 3-56. ECLMR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	INTn	R	1	Event claimed by secure supervisor. Indicates the event of the corresponding number is secure. An event can only be claimed by setting the corresponding ECLMR.INTn bit by using executing MVC in SS mode. 0h = Event is not claimed to be secure event 1h = Event is claimed to be secure event

3.7.32 Event Assigned Exception Registers (EASGR)

The Event Assign Register (EASGR) allows Root Supervisor (S) to assign events to the currently active Guest Supervisor (GS). Events assigned to GS in EASGR will only be delivered to GS, and will remain pending when the CPU is in S or SS operating modes. After the secure supervisor claims its share of the event in the event input array, the rest would be split between non-secure root supervisor and non-secure guest supervisors. In a virtualized system, the currently active GS should handle the events meant for it to handle without root supervisor intervention. To make this happen, EASGR register is set to enable the events matching

the corresponding assigned bit to be delivered to current GS. The EASGR bit field is shown in the diagram as follows.

Masking all events which are enabled by User. Only supervisor mode can write to this register.

Table 3-57. EASGR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	INTn	R	1	Event is assigned to current active GS in virtualized system. Indicates the event of the corresponding number is virtual. An event can be assigned by setting the corresponding EASGR.INTn bit by executing MVC in S mode. 0h = Event is not assigned to the currently active GS 1h = Event is assigned to the currently active GS

3.7.33 User Mask Enable Register (UEMR)

Masking all events which are enabled by User. Only supervisor mode can write to this register.

Table 3-58. UEMR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-1	RESERVED	R	0	
0	UME	W	0	User Mask Enable 0h = Don't mask User-Enabled Event 1h = Mask User-Enabled Event

3.7.34 Guest Mode Enable Register (GMER)

Enable Guest Supervisor (GS) to perform SECCALL instruction. Only Secure or Root Supervisor mode can write to this register.

Table 3-59. GMER Register Field Descriptions

Bit	Field	Type	Reset	Description
63-1	RESERVED	R	0	
0	SECCALLEN	W	0	Enable SECCAL Operation 0h = GS cannot execute SECCALL 1h = GS can execute SECCALL

3.7.35 Test Counter Register (TCR)

C7x provides test counter register for the users to test their software. With this register, the user is able to intentionally insert exception trigger in his/her software code at any given instruction boundary by setting the count value. When the program stream is interrupted, the programmer can test the atomic execution of the protected critical section code. This part of protected code are not supposed to be exposed unintentionally, otherwise, the delicate internal data structure could be at rick to be corrupted. One way to test software on very possible SW boundary is to put the TCR write instruction in a loop. In each loop, the counter value increments by 1 or by a fixed amount. Each time when the counter counts to zero, a count-to-zero event is triggered at the instruction one EP more than last time the event occurred.

The TCR feature is summarized as below:

- TCR is an 20-bit counter. This makes sure the counter can count up to 1 million cycles before expiring
- After counter is loaded, it starts to count down starting at the second execution packet at each execution cycle
- The counter doesn’t count when the CPU is stalled (ctl_stall_exe_regs)
- When the counter is count down to 0, an internal count-to-zero event is generated
- This is one time counter, no HW automatic reload
- The counter never stops counting unless it counts to zero. Even an different event interrupts the thread which sets up the counter, the counting continues until it reaches zero.

- The TCR provides configuration capability through TCR Configuration Register (TCCR). Using TCCR register, the programmer can:
  - Assign event number to count-to-zero event by writing into the 6-bit TCCR.EVTNUM field
  - Assign priority number to count-to-zero event has no difference than other event, programmer writes priority value into the corresponding EPRI[EVTNUM] register.
  - The code running in supervisor mode can grant TCR write/initiation permission to its corresponding user privilege
    - SS writes enable/disable SU permission
    - S writes enable/disable U permission
    - GS writes enable/disable GU permission
- When TCR counts to zero, the current TSR.COP has lower priority than TCCR.PRIORITY, the test counter event will be triggered
  - If the TCR counts inside a lower priority ISR, when the TCR counts to zero. The hardware will trigger count-to-zero event, which has higher priority than currently running ISR. Therefore the currently running ISA will be preempted.
  - When TCR counts to zero, the current TSR.COP has higher or same priority than TCCR.PRIORITY, the count-to-zero event will be postponed, and the corresponding event flag bit in the EFR register is set. Later on, when the postponed/flagged count-to-zero event becomes the highest pending event, it will be taken like any other external event.

3.7.36 Test Counter Control Register (TCCR)

Test counter control register (TCCR) is provided for user to control how TCR can initiated and when TCR counts to zero, how count-to-zero event can be triggered.

Table 3-60. TCCR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-17	RESERVED	R	0	
16	UE	W	0	User Mode Enable 0h = User mode is not allowed to initiate TCR 1h = User mode is allowed to initiate TCR
15-14	RESERVED	R	0	
13-8	EVTINDX	W	0	Event index. Used when to even number > 63
7-6	RESERVED	R	0	
5-0	EVTNUM	W	0	Event number used for count-to-zero event. If count-to-zero event is pend due to higher TSR.COP level, EFR[EVTNUM] bit will be set to 1

3.7.37 Internal Exception Report Register (IERR)

The internal exception report register (IERR) contains flags that indicate the cause of the internal exception. In the case of simultaneous internal exceptions, the same flag may be set by different exception sources. In this case, it may not be possible to determine the exact causes of the individual exceptions. The IERR is shown in Figure 1-50 and described in Table 1-42.

The IERR is not accessible in User mode.

Table 3-61. IERR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-15	RESERVED	R	0	

Table 3-61. IERR Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
14	MMX	R/W	0	MMA execution exception 0h = Exception related to MMA execution is not the cause 1h = Exception related to MMA execution is the cause: • MMA instruction is executed when MMA is not present in the cluster • MMA instruction is executed when MMA has not been opened by HWAOPEN • MMA returns error status
13	ADX	R/W	0	Address Out-of-Range exception 0h = Exception due to out-of-range address is not the cause 1h = Exception due to out-of-range address is the cause: • Out-of-range address on L1D memory address • Out-of-range address on CR memory address
12	EXX	R/W	0	Execution exception 0h = Exception due to execution of an instruction is not the cause 1h = Exception due to execution of an instruction is the cause: • Executing PROT-mode only instructions in UNPROT mode • Division by zero
11	SEX	R/W	0	Streaming Engine exception 0h = Exception reported by Streaming Engine is not the cause 1h = Exception reported by Streaming Engine is the cause: • Access to L2 memory returns error status • Streaming Engine reports internal errors
10	DPX	R/W	0	Data Fetch exception 0h = Exception due to fetching data is not the cause 1h = Exception due to fetching data is the cause: • Access to L1D memory returns error status
9-8	RESERVED	R	0	
7	PRX	R/W	0	Privilege exception 0h = Privilege exception is not the cause. 1h = Privilege exception is the cause: • MVC to internal control registers causes permission violation • MVC to external control registers (ECRs) causes permission violation • Invalid Permissions for given opcode • Illegal RETE/RETS execution • Using ECSP addressing mode as a user
6	RAX	R/W	0	Resource access exception 0h = Resource access exception is not the cause. 1h = Resource access exception is the cause: • Accessing Streaming Engine registers when stream is not open • Executes SESAVE or SERSTR while stream is open • Executes LUT/HIST instructions when access bit is not enabled • External Control Register returns non-zero status, except for permission violation • Executes DSWBP instruction when SWBP is not enabled

Table 3-61. IERR Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
5	RCX	R/W	0	Resource conflict exception 0h = Resource conflict exception is not the cause. 1h = Resource conflict exception is the cause: • Conflict due to writing to same register in the same clock cycle
4	OPX	R/W	0	Illegal opcode exception 0h = Illegal opcode exception is not the cause. 1h = Illegal opcode exception is the cause: • Invalid opcodes within a format • Greater than 64-bit stores on D1 unit or greater than 64-bit loads on D2 unit • Illegal instructions on scalar side and on the vector side • Addressing mode is not 3'h0 or 3'h4 for PFS instruction • Addressing mode is not 3'h0 or 3'h4 for ASW, ASD, CASW, CASD instructions
3	EPX	R/W	0	Execute packet exception 0h = Execute packet exception is not the cause. 1h = Execute packet exception is the cause: • Unrecognized or Reserved Instruction Format
2	RESERVED	R	0	
1	IFX	R/W	0	Instruction fetch exception 0h = Exception due to fetching instructions is not the cause 1h = Exception due to fetching instructions is the cause: • Access to L1I memory returns error status
0	PFX	R/W	0	Page Fault exception 0h = Page Fault exception is not the cause 1h = Page Fault exception is the cause

3.7.38 Internal Exception Address Register (IEAR)

This register captures the virtual address of the most recent load or store which experience a corresponding exception.

- If a fetch packet causes page fault or instruction fetch exception, the address of that fetch packet is captured
- If an load or store instruction caused a page fault, its virtual address is captured to the IEAR.
- The address of the first instruction in the execute packet which causes EPX, OPX, RCX, RAX, PRX, SEX, EXX, ADX, MMX exceptions

Table 3-62. IEAR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-49	RESERVED	R	0	
48-32	TFA	R/W	X	
31-0	IEAR	R/W	X	

3.7.39 Internal Exception Data Register (IEDR)

This register captures the data of the most recent MVC instruction which experiences a corresponding exception.

Table 3-63. IEDR Register Field Descriptions

Bit	Field	Type	Reset	Description
63-0	DATA	R/W	X	

3.7.40 Galois Field Polynomial Generator Function Register (GFPGFR)

The Galois field polynomial generator function register (GFPGFR) controls the field size and the Galois field polynomial generator of the Galois field multiply hardware. The content of a 32-bit element of GFPGFR is shown in Figure 1-54 and described in Table 1-43.

Table 3-64. GFPGFR Register Field Descriptions

Bit	Field	Type	Reset	Description
31-27	RESERVED	R	0	
26-24	SIZE	R/W	7h	0 - 7h = Field size
23-8	RESERVED	R	0	
7-0	POLY	R/W	1Dh	0 - FFh = Polynomial generator

3.7.41 GMPY Polynomial Register (GPLY)

Each scalar GMPY instruction (see GMPY) uses a 32-bit polynomial when the instruction is executed on the .C unit. The GPLY register, Figure 1-55, is a vector SIMD 256-bit wide register, with each 32-bit element contain the polynomial for the corresponding GMPY/GMPY4 operation in the vector SIMD VGMPY/VGMPY4 instructions. The content of one 32-bit element of GPLY register when used as GMPY polynomial is shown below.

Table 3-65. GPLY Register Field Descriptions

Bit	Field	Type	Reset	Description
31-0	32BIT_POLYNOMIAL	R/W	0	

3.7.42 .Look Up Table Base Address Registers (LTBR0-3)

The Look Up Table Base Address registers contain the base addresses of the L1D SRAM regions configured as tables. There are total of 4 Look Up Table Base Address registers in the C7x CPU. The table base address must be 128-bytes aligned since the L1D cache line is 128-bytes wide, therefore the 7 Least Significant Bits of the base address is always read out as 0.

Table 3-66. LTBR0-3 Register Field Descriptions

Bit	Field	Type	Reset	Description
63-16	RESERVED	R	0	
15-7	BASE_ADDRESS	R/W	X	
6-0	RESERVED	R	0	

3.7.43 Look Up Table and Histogram Configuration Registers (LTCR0-3)

The Look Up Table Configuration Registers stores the control information of the table lookup and histogram instruction. There are 4 LTCR registers, corresponds with the 4 LTBR registers.

Table 3-67. LTCR0-3 Register Field Descriptions

Bit	Field	Type	Reset	Description
63-29	RESERVED	R	0	
28	VCOP	R	0	0h = Native c7x memory bank configuration (16 banks x 64-bits) 1h = Legacy VCOP EVE memory bank configuration (8 banks x 32-bis)
27-26	RESERVED	R	0	
25-24	PROMO	R	0	00h = Type promotion mode of LUTRD returned data 01h = No promotion 10h = Promote 2x: Bytes to Half-words, Half-words to Words, Words to Double Words 11h = Promote 4x: Bytes to Words, Half-words to Double-Words Promote 8x: Bytes to Double-Words

Table 3-67. LTCR0-3 Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
23-16	TABLE_SIZES	R	0	Size of the SRAM memory allocated for each set of table 0h = 0.0 KBytes 1h = 1.0 KBytes 2h = 2.0 KBytes 3h = 4.0 KBytes 4h = 8.0 KBytes 5h = 16.0 KBytes 6h = 32.0 KBytes 7h - FFh = Reserved. Read as 0.
15-14	RESERVED	R	0	
13-11	WSIZE	R	0	Weight Sizes 0h = Byte 1h = Half Word 2h-7h = Reserved. Read as 0.
10-8	INTERPOLATION	R/W	0	Successive numbers of elements are also written to allow interpolation of different data points 0h = No interpolation, only indexed element per table is written 1h = Returns 2 elements per table 2h = Returns 4 elements per table 3h = Returns 8 elements per table 4h-7h = Reserved. Read as 0.
7	SAT	R/W	0	Histogram bin entries are saturated to min/max values of its data type if enabled 0h = Non-saturate 1h = Saturate
6	SIGNED	R/W	0	Element data is signed or unsigned 0h = Unsigned 1h = Signed
5-3	ESIZE	R/W	0	Input Element Sizes 0h = Byte 1h = Half Word 2h = Word 3h-7h = Reserved. Read as 0.
2-0	NTBL	R/W	0	Number of Table to be looked up in parallel 0h = 1 table 1h = 2 tables 2h = 4 tables 4h = 8 tables 5h = 16 tables 5h-7h = Reserved. Read as 0.

3.7.44 Look Up Table Enable Register (LTER)

The Look Up Table Enable Register specifies the operations allowed for a particular LUT/HIST set. When the LTER bits are set, read or write operation via MVC of the corresponding LTBR and LTCR are allowed. Besides controlling the programmability of LTBR and LTCR, the LTER bit fields also restricts the execution of LUTRD and LUTWR/LUTINIT/HIST/WHIST instructions. Only when enabled by LTER bit fields, corresponding look up tables and histogram instructions are allowed to be executed. Otherwise, exception is raised.

Table 3-68. LTCR0-3 Register Field Descriptions

Bit	Field	Type	Reset	Description
63-14	RESERVED	R	0	

Table 3-68. LTCR0-3 Register Field Descriptions (continued)

Bit	Field	Type	Reset	Description
13-12	LUTE3	R/W	0	Permission for table 3set up. Can only be written by supervisor 0h = No LUT operation allowed 1h = MVC read from LTBR, LTCR and execution of LUTRD instruction allowed, exception raises if encounter LUTWR, LUTINIT, HIST, WHIST or MVC write to LTBR, LTCR 2h = MVC read from LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed. Exception raises if encounter MVC write to LTBR, LTCR 3h = MVC read from and write to LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed
11-10	RESERVED	R	0	
9-8	LUTE2	R/W	0	Permission for table 2 set up. Can only be written by supervisor 0h = No LUT operation allowed 1h = MVC write to LTBR, LTCR MVC read from LTBR, LTCR and execution of LUTRD instruction allowed, exception raises if encounter LUTWR, LUTINIT, HIST, WHIST or MVC write to LTBR, LTCR 2h = MVC read from LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed. Exception raises if encounter 3h = MVC read from and write to LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed
7-6	RESERVED	R	0	
5-4	LUTE1	R/W	0	Permission for table 1 set up. Can only be written by supervisor 0h = No LUT operation allowed 1h = MVC write to LTBR, LTCR MVC read from LTBR, LTCR and execution of LUTRD instruction allowed, exception raises if encounter LUTWR, LUTINIT, HIST, WHIST or MVC write to LTBR, LTCR 2h = MVC read from LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed. Exception raises if encounter 3h = MVC read from and write to LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed
3-2	RESERVED	R	0	
1-0	LUTE0	R/W	0	Permission for table 0 set up. Can only be written by supervisor 0h = No LUT operation allowed 1h = MVC write to LTBR, LTCR MVC read from LTBR, LTCR and execution of LUTRD instruction allowed, exception raises if encounter LUTWR, LUTINIT, HIST, WHIST or MVC write to LTBR, LTCR 2h = MVC read from LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed. Exception raises if encounter 3h = MVC read from and write to LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed

3.7.45 .C Unit Control Register (CUCR0-CUCR3)

The .C unit contains up to 4 dedicated control registers, CUCR0 to CUCR3. This set of control registers can be used as operand in certain .C unit instructions to control the instruction behaviors. Specifically, they can be used as:

- CUCR0 to CUCR3 can be used as the controls for the general permutation instruction (VPERM)

- CUCR0 to CUCR3 can be used as masks for DOTPM and SADM instructions
- CUCR0 (alias GPLY) can also be used to store the polynomials for GMPY instructions
- CUCR1 (alias GFPGFR) can also be used to store the Galois field polynomial generator function

The .C unit control registers can be read and written by issuing a MVC instruction to the .C unit.

3.7.46 Streaming Address Control Registers (STRACR0-3)

The STRACR0-STRACR3 contain the parameters, such as the element counts, loop dimensions, the access length and other flags, to start streaming address calculation. They have similar fields as defined in the Streaming Engine definition template (Streaming Engine Spec, section 1.3.5).

Table 3-69. Streaming Address Configuration Registers (STRACR)

511 448	447 416	415 384	383 352	351 320	319 288	287 256
FLAGS[63:0]	Reserved	DIM5	DIM4	DIM3	DIM2	DIM1
255 224	223 192	191 160	159 128	127 96	95 64	63 32	31 0
DECDIM2_WIDTH	DECDIM1_WIDTH	ICNT5	ICNT4	ICNT3	ICNT2	ICNT1	ICNT0

Table 3-70. Stream Fields

Field Name	Description	Size
ICNT0	Total loop iteration count for level 0 (innermost)	32 bits
ICNT1	Total loop iteration count for level 1	32 bits
ICNT2	Total loop iteration count for level 2	32 bits
ICNT3	Total loop iteration count for level 3	32 bits
ICNT4	Total loop iteration count for level 4	32 bits
ICNT5	Total loop iteration count for level 5 (outermost)	32 bits
DECDIM1_WIDTH	Tile width of DECDIM1. Use together with DECDIM1 flags to specify vertical strip mining feature	32bits
DECDIM2_WIDTH	Tile width of DECDIM2. Use together with DECDIM2 flags to specify vertical strip mining feature	32bits
DIM1	Signed dimension for loop level 1, in elements	32 bits
DIM2	Signed dimension for loop level 2, in elements	32 bits
DIM3	Signed dimension for loop level 3, in elements	32 bits
DIM4	Signed dimension for loop level 4, in elements	32 bits
DIM5	Signed dimension for loop level 5, in elements	32 bits
FLAGS	Stream modifier flags	64 bits

Table 3-71. Streaming Address Configuration Register FLAGS Field

Bit	Field	Type	Reset	Description
63-54	RESERVED	R	0	
53-51	DEC_DIM2	TBD	TBD	
50-48	DEC_DIM1	TBD	TBD	
47-27	RESERVED	R	0	
26-24	DIMFMT	TBD	TBD	
23-15	RESERVED	R	0	
14-12	VECLEN	TBD	TBD	

Table 3-71. Streaming Address Configuration Register FLAGS Field (continued)

Bit	Field	Type	Reset	Description
11-0	RESERVED	R	0	

DIMFMT Details The DIMFMT field allows the programmer to combine 16-bit fields in the stream definition template into 32-bit fields. This increases the range of loop iteration counts and array dimensions the template can express, at the expense of total number of loop dimensions.

DIMFMT indicates which loop levels to combine in order to provide a larger ICNT field and larger DIM field. When DIMFMT instructs the streaming engine to join two consecutive loop levels, the lower numbered level takes over the combined count. The higher numbered level becomes disabled, and behaves as if it has a loop count of 1.

The following table summarizes which loop counts are active and which ones are not for each encoding of DIMFMT. Inactive levels donate their count bits to the level below them, and otherwise act as if that level's loop count was exactly 1.

Table 3-72. Loop Level Configuration vs. DIMFMT

DIMFMT	Dimensions	ICNT5	ICNT4	ICNT3	ICNT2	ICNT1	ICNT0
000b	3	Inactive	32-bit	Inactive	32-bit	Inactive	32-bit
001b	4	Inactive	32-bit	Inactive	32-bit	16-bit	16-bit
010b	4	Inactive	32-bit	16-bit	16-bit	Inactive	32-bit
011b	5	Inactive	32-bit	16-bit	16-bit	16-bit	16-bit
100b	Reserved						
101b	Reserved						
110b	6	16-bit	16-bit	16-bit	16-bit	16-bit	16-bit
111b	6	16-bit	16-bit	16-bit	16-bit	16-bit	16-bit

3.7.47 Streaming Address Count Register (STRACNTR0-3)

The STRACNTR contains the intermediate element counts of all loop levels. Each time a streaming load or store instruction with advancing option [SA++] is executed, the corresponding element count is reduced by a VECLEN amount of elements. When the element count of a loop becomes zero, the address of the element of the next loop is computed using the next loop dimension.

Table 3-73. Streaming Address Count Register (STRACNTR)

511 480	479 448	447 416	415 384	383 352	351 320	319 288	287 256
DEC_DIM2_CNT	DEC_DIM1_CNT	CNT5	CNT4	CNT3	CNT2	CNT1	CNT0
255 192		191 160	159 128	127 96	95 64	63 32	31 0
Reserved		I5 Offset	I4 Offset	I3 Offset	I2 Offset	I1 Offset	Current Offset

Executing a STRAOPEN instruction will set the ICNT fields in STRACNTR to the values contained in the ICNT fields of the STRACR.

While a stream is open, functional access to the associated STRACR and STRACNTR registers are disallowed.

Note

Motivation is to limit corner cases between SA access for addressing (e.g. *D0[SA0], *D0[SA0++], ADDAx) near MVC operations, and the values visible to programmers. Need to decide whether to generate a fault or just return zeros for MVC operations. Debugger access still required for CCS

3.7.48 Extended Control Registers

Extended Control Registers are not physically resided inside the CPU. They can be part of the Streaming Engine, L1I, L1D, Emulation or other modules inside the CorePac. They can be accessed by issuing a MVC to .S1 unit with the appropriate register address.

Table 3-74. Extended Control Registers

Acronym	Register Name	Section
Debug CRs	Emulation Control Registers	Emulation chapter
L1D CRs	L1D Control Registers	Memory System, L1D chapter
L1I CRs	L1I Control Registers	Memory System, L1I chapter
L2 CRs	L2 Control Registers	Memory System, L2 chapter
SE0 CRs	Streaming Engine 0 Control Registers	CPU, Streaming Engine chapter
SE1 CRs	Streaming Engine 1 Control Registers	CPU, Streaming Engine chapter
CMMU CRs	Central Memory Management Control Registers	Memory Management
L1I uTLB CRs	L1I micro-TLB Control Registers	Memory System, L1I chapter
L1D uTLB CRs	L1D micro-TLB Control Registers	Memory System, L1I chapter
SE0 uTLB CRs	Streaming Engine 0 micro-TLB Control Registers	CPU, Streaming Engine chapter
SE1 uTLB CRs	Streaming Engine1 micro-TLB Control Registers	CPU, Streaming Engine chapter
System CRs	System Control Registers	Integration, System chapter
PBIST CRs	PBIST Control Registers	Integration, Pbist chapter

3.7.48.1 Pipeline/Timing of Extended Control Register Accesses

The extended control registers can be accessed via regular MVC instruction. The latency of such access is varied and based on ECR protocol specified in the Extended Configuration Register Interface specification.

3.7.49 1.4.7.50 Control Register Address Mapping

3.7.49.1 CPU Control Registers

Table below lists the register addresses for accessing the internal CPU control register file and which functional unit can access them. Each control register is accessed by the MVC instruction. Majority of control registers can be read from and write to by issuing a MVC to the functional unit .S1. However, some specialized control registers which govern the behaviors of a particular functional unit are resided in that functional unit and can be accessed by issuing MVC to that functional unit such as the CUCR registers which are resides in the .C unit.

See the MVC instruction description for information on how to use this instruction.

Additionally, some of the control register bits are specially accessed in other ways. For example, arrival of an external event, triggers the setting of flag bit EFRm. Subsequently, when that event is processed, this triggers the clearing of EFRm and the clearing of the global event enable bit (GEE) in the TSR. Similarly, saturating instructions like SADD set the SAT (saturation) bit in the flag status register (FSR).

Read and Write access to the control registers requires correct privileges to be met.

Table 3-75. MVC Register Addresses for Accessing CPU Control Registers

Acronym# unique_112/ unique_112_Connect_42_GUID-E038F6C0-E503-47CF-AE20-9C4E084054B6	Register Name	ECR address	Issue Slot	Width	Secure Supervisor	Secure User	Supervisor	User	Guest Supervisor	Guest User
CPUID	CPU ID register	12'h000	.S1	64	RO	RO	RO	RO	RO	RO
PMR	Power Management register	12'h001	.S1	64	RW	RO	RO	RO	RO	RO

Table 3-75. MVC Register Addresses for Accessing CPU Control Registers (continued)

Acronym# unique_112/ unique_112_Connect_42_GUID-E038F6C0-E503-47CF-AE20-9C4E084054B6	Register Name	ECR address	Issue Slot	Width	Secure Supervisor	Secure User	Supervisor	User	Guest Supervisor	Guest User
DNUM	DSP core number register	12'h002	.S1	64	RO	RO	RO	RO	RO	RO
TSC	Time-stamp counter register	12'h003	.S1	64	RO	RO	RO	RO	RO	RO
TSR	Task state register	12'h004	.S1	64	RO	RO	RO	RO	RO	RO
RP	Return pointer register	12'h005	.S1	64	RW	RW	RW	RW	RW	RW
BPCR	Branch Predictor Control Register	12'h006	.S1	64	RW	RO	RW	RO	RW	RO
FPCR	Floating-point control register	12'h007	.S1	64	RW	RW	RW	RW	RW	RW
FSR	Flag status register	12'h008	.S1	64	RW	RW	RW	RW	RW	RW
ECLMR	Event Claim Register	12'h009	.S1	64	RW	RO	RO	RO	RO	RO
EASGR	Event Assign Register	12'h00A	.S1	64	RW	RO	RW	RO	RO	RO
EPRI	Event Priority Register Event priority Register	12'h00B	.S1	64	RW	RO	RW	RO	RW	RO
EER	Event Enable Register	12'h00C	.S1	64	RO	RO	RO	RO	RO	RO
EESET	Event Enable Set Register	12'h00D	.S1	64	WO	X	WO	X	WO	X
EECLR	Event Enable Clear Register	12'h00E	.S1	64	WO	X	WO	X	WO	X
DEPR	Debug Event Priority Register	12'h00F	.S1	64	RO	RO	RO	RO	RO	RO
EFR	Event Flag Register	12'h010	.S1	64	RO	RO	RO	RO	RO	RO

Table 3-75. MVC Register Addresses for Accessing CPU Control Registers (continued)

Acronym# unique_112/ unique_112_Connect_42_GUID-E038F6C0-E503-47CF-AE20-9C4E084054B6	Register Name	ECR address	Issue Slot	Width	Secure Supervisor	Secure User	Supervisor	User	Guest Supervisor	Guest User
EFSET	Event Flag Set Register	12'h011	.S1	64	WO	X	WO	X	WO	X
EFCLR	Event Flag Clear Register	12'h012	.S1	64	WO	X	WO	X	WO	X
IESET	Internal Exception Event Set Register	12'h013	.S1	64	RW	RW	RW	RW	RW	RW
ESTP_SS	Event Service Table Pointer Register, Secure Supervisor	12'h014	.S1	64	RW	RO	RO	RO	RO	RO
ESTP_S	Event Service Table Pointer Register, Supervisor	12'h015	.S1	64	RW	RO	RW	RO	RO	RO
ESTP_GS	Event Service Table Pointer Register, Guest Supervisor	12'h016	.S1	64	RW	RO	RW	RO	RW	RO
EDR	Event Dropped Register	12'h017	.S1	64	RO	RO	RO	RO	RO	RO
ECSP_SS	Event Context Save Pointer, Secure Supervisor	12'h018	.S1	64	RW	RO	RO	RO	RO	RO
ECSP_S	Event Context Save Pointer, Supervisor	12'h019	.S1	64	RW	RO	RW	RO	RO	RO
ECSP_GS	Event Context Save Pointer, Guest Supervisor	12'h01A	.S1	64	RW	RO	RW	RO	RW	RO

Table 3-75. MVC Register Addresses for Accessing CPU Control Registers (continued)

Acronym# unique_112/ unique_112_Connect_42_GUID-E038F6C0-E503-47CF-AE20-9C4E084054B6	Register Name	ECR address	Issue Slot	Width	Secure Supervisor	Secure User	Supervisor	User	Guest Supervisor	Guest User
TCSP	Task Context Save Pointer	12'h01B	.S1	64	RW	RO	RW	RO	RW	RO
RXMR_SS	Returning Execution Mode Register, Secure Supervisor	12'h01C	.S1	64	RW	RO	RO	RO	RO	RO
RXMR_S	Returning Execution Mode Register, Supervisor	12'h01D	.S1	64	RW	RO	RW	RO	RO	RO
AHPEE	Highest Priority Enabled Event, Currently in service	12'h01E	.S1	64	RO	RO	RO	RO	RO	RO
PHPEE	Pending HPEE	12'h001F	.S1	64	RO	RO	RO	RO	RO	RO
IPE	Inter-processor Events Register	12'h020	.S1	64	WO	X	WO	X	WO	X
IERR	Internal exception report register	12'h021	.S1	64	RW	RO	RW	RO	RW	RO
IEAR	Internal Exception Address Register	12'h022	.S1	64	RW	RO	RW	RO	RW	RO
IESR	Internal exception Status Register	12'h023	.S1	64	RW	RO	RW	RO	RW	RO
IEDR	Internal Exception Data Register	12'h024	.S1	64	RW	RO	RW	RO	RW	RO
TCR	Test Count register	12'h025	.S1	64	RW	RW	RW	RW	RW	RW
TCCR	Test Count Config register	12'h026	.S1	64	RW	RO	RW	RO	RW	RO

Table 3-75. MVC Register Addresses for Accessing CPU Control Registers (continued)

Acronym# unique_112/ unique_112_Connect_42_GUID-E038F6C0-E503-47CF-AE20-9C4E084054B6	Register Name	ECR address	Issue Slot	Width	Secure Supervisor	Secure User	Supervisor	User	Guest Supervisor	Guest User
GMER	Guest Mode Enable Register	12'h027	.S1	64	RW	RO	RW	RO	RO	RO
UMER	User Mask Enable Register	12'h028	.S1	64	RW	RO	RW	RO	RW	RO
SPBR	Stack pointer boundary Register	12'h029	.S1	64	RW	RW	RW	RW	RW	RW
LTBR0	Lookup Table Base Address register 0	12'h02A	.S1	64	RW	RW	RW	RW	RW	RW
LTBR1	Lookup Table Base Address register 1	12'h02B	.S1	64	RW	RW	RW	RW	RW	RW
LTBR2	Lookup Table Base Address register 2	12'h02C	.S1	64	RW	RW	RW	RW	RW	RW
LTBR3	Lookup Table Base Address register 3	12'h02D	.S1	64	RW	RW	RW	RW	RW	RW
LTCR0	Lookup Table configuration register 0	12'h02E	.S1	64	RW	RW	RW	RW	RW	RW
LTCR1	Lookup Table configuration register 1	12'h02F	.S1	64	RW	RW	RW	RW	RW	RW
LTCR2	Lookup Table configuration register 2	12'h030	.S1	64	RW	RW	RW	RW	RW	RW
LTCR3	Lookup Table configuration register 3	12'h031	.S1	64	RW	RW	RW	RW	RW	RW
LTER	Lookup Table Enable register	12'h032	.S1	64	RW	RO	RW	RO	RW	RO

Table 3-75. MVC Register Addresses for Accessing CPU Control Registers (continued)

Acronym# unique_112/ unique_112_Connect_42_GUID-E038F6C0-E503-47CF-AE20-9C4E084054B6	Register Name	ECR address	Issue Slot	Width	Secure Supervisor	Secure User	Supervisor	User	Guest Supervisor	Guest User
DBGCTXT	Debug Context (Overlay) Register	12'h033	.S1	64	RW	RW	RW	RW	RW	RW
PC_PROF	PC_PROF Register	12'h034	.S1	64	RO	RO	RO	RO	RO	RO
GPLY	Galois Polynomial Register	12'h035	.S1	64	RW	RW	RW	RW	RW	RW
GFPGFR	Galois Field Polynomial Generator Function Register	12'h036	.S1	64	RW	RW	RW	RW	RW	RW
GTSC	Global Time Stamp Counter	12'h037	.S1	64	RO	RO	RO	RO	RO	RO
STSC	Shadow Time Stamp Counter	12'h038	.S1	64	RO	RO	RO	RO	RO	RO
UFCMR	User Flag Clear Mask Register	12'h039	.S1	64	RW	RO	RW	RO	RW	RO
CUCR0	.C unit Control Register 0	16'h1000	.C	512	RW	RW	RW	RW	RW	RW
CUCR1	.C unit Control Register 1	16'h1001	.C	512	RW	RW	RW	RW	RW	RW
CUCR2	.C unit Control Register 2	16'h1002	.C	512	RW	RW	RW	RW	RW	RW
CUCR3	.C unit Control Register 3	16'h1003	.C	512	RW	RW	RW	RW	RW	RW
STRACR0	Streaming address generator configuration register 0	16'h1004	.C	512	RW	RW	RW	RW	RW	RW
STRACNTR0	Streaming address generator counter register 0	16'h1005	.C	512	RW	RW	RW	RW	RW	RW

Table 3-75. MVC Register Addresses for Accessing CPU Control Registers (continued)

Acronym# unique_112/ unique_112_Connect_42_GUID-E038F6C0-E503-47CF-AE20-9C4E084054B6	Register Name	ECR address	Issue Slot	Width	Secure Supervisor	Secure User	Supervisor	User	Guest Supervisor	Guest User
STRACR1	Streaming address generator configuration register 1	16'h1006	.C	512	RW	RW	RW	RW	RW	RW
STRACNTR1	Streaming address generator counter register 1	16'h1007	.C	512	RW	RW	RW	RW	RW	RW
STRACR2	Streaming address generator configuration register 2	16'h1008	.C	512	RW	RW	RW	RW	RW	RW
STRACNTR2	Streaming address generator counter register 2	16'h1009	.C	512	RW	RW	RW	RW	RW	RW
STRACR3	Streaming address generator configuration register 3	16'h100A	.C	512	RW	RW	RW	RW	RW	RW
STRACNTR3	Streaming address generator counter register 3	16'h100B	.C	512	RW	RW	RW	RW	RW	RW
SA0	Streaming Address Offset Register 0	16'h100C	.C	64	X	X	X	X	X	X
SA1	Streaming Address Offset Register 1	16'h100D	.C	64	X	X	X	X	X	X
SA2	Streaming Address Offset Register 2	16'h100E	.C	64	X	X	X	X	X	X
SA3	Streaming Address Offset Register 3	16'h100F	.C	64	X	X	X	X	X	X

Table 3-75. MVC Register Addresses for Accessing CPU Control Registers (continued)

Acronym# unique_112/ unique_112_Connect_42_GUID-E038F6C0-E503-47CF-AE20-9C4E084054B6	Register Name	ECR address	Issue Slot	Width	Secure Supervisor	Secure User	Supervisor	User	Guest Supervisor	Guest User
PSA0	Streaming Address Predicate Register 0	16'h1010	.P	64	RO	RO	RO	RO	RO	RO
PSA1	Streaming Address Predicate Register 1	16'h1011	.P	64	RO	RO	RO	RO	RO	RO
PSA2	Streaming Address Predicate Register 2	16'h1012	.P	64	RO	RO	RO	RO	RO	RO
PSA3	Streaming Address Predicate Register 3	16'h1013	.P	64	RO	RO	RO	RO	RO	RO
ILCNT	Inner Loop Counter Register	12'h0F0	.S1	64	X	X	X	X	X	X
ILCNT_INIT	Inner Loop Counter Initial Value Register	12'h0F1	.S1	64	X	X	X	X	X	X
OLCNT	Inner Loop Counter Initial Value Register	12'h0F2	.S1	64	X	X	X	X	X	X
EPISKEW	EPISKEW	12'h0F3	.S1	64	X	X	X	X	X	X
LCNTFLAG	16-bit predicate flags	12'h0F4	.S1	64	X	X	X	X	X	X

3.7.49.2 CPU Control Register Bit Field Summary

The table below have the summary off the bit fields and related information for every internal control registers inside the C7x CPU

Table 3-76. TBD

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
CPUID	RSVD	63	32	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
CPUID	SCHEME	31	30	RO	RO	RO	RO	RO	RO	RO	RO	1h	Indicates the old scheme vs. new scheme
CPUID	BU	29	28	RO	RO	RO	RO	RO	RO	RO	RO	2h	Business Indicator

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
CPUID	FUNC	27	16	RO	RO	RO	RO	RO	RO	RO	RO	C0h	Function indicates a software compatible module family
CPUID	RTL	15	11	RO	RO	RO	RO	RO	RO	RO	RO	0h	RTL version
CPUID	MAJOR	10	8	RO	RO	RO	RO	RO	RO	RO	RO	0h	Major revision
CPUID	CUSTOM	7	6	RO	RO	RO	RO	RO	RO	RO	RO	0h	Indicates a special version for a particular device
CPUID	MINOR	5	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Minor revision
PMR	RSVD	63	6	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
PMR	PWRD	5	0	RW	RO	RW	RO	RO	RO	RO	RO	0h	Power down mode field
DNUM	RSVD	63	24	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
DNUM	CLUSTRNUM	23	16	RO	RO	RO	RO	RO	RO	RO	RO	0h	Identifies the cluster number of the device
DNUM	COREPACNUM	15	8	RO	RO	RO	RO	RO	RO	RO	RO	0h	Identifies the corepac number within the cluster of the device
DNUM	CPUNUM	7	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Identifies the DSP core number within the cluster and the corepac of the device
TSC	TSC	63	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Time Stamp Counter
TSR	RSVD6	63	57	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
TSR	EN	56	56	RO	RO	RO	RO	RO	RO	RO	RO	0h	Endian mode
TSR	RSVD5	55	53	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
TSR	HWA0_PRESENT	52	52	RO	RO	RO	RO	RO	RO	RO	RO	0h	HWA0 Present at SOC
TSR	RSVD4	51	49	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
TSR	HWA0	48	48	RO	RO	RO	RO	RO	RO	RO	RO	0h	HWA0 Active
TSR	RSVD_SA7	47	47	RO	RO	RO	RO	RO	RO	RO	RO	0h	Streaming Write 7 is active - writable by SAOPEN/ SACLOSE only
TSR	RSVD_SA6	46	46	RO	RO	RO	RO	RO	RO	RO	RO	0h	Streaming Write 6 is active - writable by SAOPEN/ SACLOSE only
TSR	RSVD_SA5	45	45	RO	RO	RO	RO	RO	RO	RO	RO	0h	Streaming Write 5 is active - writable by SAOPEN/ SACLOSE only

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
TSR	RSVD_SA4	44	44	RO	RO	RO	RO	RO	RO	RO	RO	0h	Streaming Write 4 is active - writable by SAOPEN/ SACLOSE only
TSR	SA3	43	43	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	0h	Streaming Write 3 is active - writable by SAOPEN/ SACLOSE only
TSR	SA2	42	42	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	0h	Streaming Write 2 is active - writable by SAOPEN/ SACLOSE only
TSR	SA1	41	41	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	0h	Streaming Write 1 is active - writable by SAOPEN/ SACLOSE only
TSR	SA0	40	40	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	0h	Streaming Write 0 is active - writable by SAOPEN/ SACLOSE only
TSR	RSVD_SE7	39	39	RO	RO	RO	RO	RO	RO	RO	RO	0h	Streaming Engine 7 is active - writable by SEOPEN/ SECLOSE only
TSR	RSVD_SE6	38	38	RO	RO	RO	RO	RO	RO	RO	RO	0h	Streaming Engine 6 is active - writable by SEOPEN/ SECLOSE only
TSR	RSVD_SE5	37	37	RO	RO	RO	RO	RO	RO	RO	RO	0h	Streaming Engine 5 is active - writable by SEOPEN/ SECLOSE only
TSR	RSVD_SE4	36	36	RO	RO	RO	RO	RO	RO	RO	RO	0h	Streaming Engine 4 is active - writable by SEOPEN/ SECLOSE only
TSR	RSVD_SE3	35	35	RO	RO	RO	RO	RO	RO	RO	RO	0h	Streaming Engine 3 is active - writable by SEOPEN/ SECLOSE only
TSR	RSVD_SE2	34	34	RO	RO	RO	RO	RO	RO	RO	RO	0h	Streaming Engine 2 is active - writable by SEOPEN/ SECLOSE only
TSR	SE1	33	33	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	0h	Streaming Engine 1 is active - writable by SEOPEN/ SECLOSE only

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
TSR	SE0	32	32	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	0h	Streaming Engine 0 is active - writable by SEOPEN/ SECLOSE only
TSR	RSVD3	31	29	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
TSR	DBGM	28	28	R,IW	R,IW	R,IW	R,IW	R,IW	R,IW	R,IW	R,IW	0h	Debug access mask. 0: enables debug accesses, 1: disables debug accesses - writable by DDBG and EDBG instructions
TSR	SUDEN	27	26	R,IW	RO	R,IW	RO	RO	RO	RO	RO	0h	Secure User Debug Enable -- writable by instruction RETE only. [26] : invasive debug enable (SUIDEN), [27] : non-invasive debug enable (SUNIDEN). Secure supervisor writable only.
TSR	GEE	25	25	R, IW	RO	R,IW	RO	R,IW	RO	R, IW	RO	0h	Global Event Enable -- Can only be modifed by REVT/DEVT/ RETE; Will only disable interrupts destined for own executing level
TSR	PROT	24	24	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	1h	Pipeline Mode - writable by PROT/UNPROT/ PROTCLR only
TSR	MCOLOR	23	19	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	R, IW	0h	Memory Tag Color, writable by MTAG instruction only
TSR	RSVD2	18	17	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
TSR	COP	16	8	R, IW	R, IW	R, IW	RO	R, IW	RO	R, IW	RO	1FFh	Current operating priority
TSR	RSVD1	7	5	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
TSR	PFE	4	4	RO	RO	RO	RO	RO	RO	RO	RO	0h	Speculative Page Fault Enable
TSR	HDL	3	3	RO	RO	RO	RO	RO	RO	RO	RO	0h	Current processing mode, 0- Program thread mode; 1-Event handler mode

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
TSR	CXM	2	0	RO	RO	RO	RO	RO	RO	RO	RO	5h	Current execution mode, reset value is 0x5
RP	RP	63	2	RW	RW	RW	RW	RW	RW	RW	RW	0h	Return Pointer
RP	RSV	1	1	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
RP	SYSCALL	0	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	System call: SYSCALL/ ROOTCALL/ SECCALL executed, 0-Normal Call; 1-System Call
BPCR	RSVD	63	1	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
BPCR	BPOFF	0	0	RW	RO	RW	RO	RW	RO	RW	RO	0h	Turn off Branch Predictor if = 1
FPCR	RSVD2	63	9	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
FPCR	IDIV0_EN	8	8	RW	RW	RW	RW	RW	RW	RW	RW	0h	Integer Divide by 0 Exception Enable
FPCR	RSVD1	7	5	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
FPCR	FTZ	4	4	RW	RW	RW	RW	RW	RW	RW	RW	1h	Flush-to-zero mode
FPCR	RSVD0	3	2	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
FPCR	RMODE	1	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Floating Point Rounding Mode
FSR	SAT_7	63	63	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is saturated Fixed Point Flag, vector_slice7
FSR	UNORD_7	62	62	RW	RW	RW	RW	RW	RW	RW	RW	0h	Compare result is unordered Floating Point Flag, vector_slice7
FSR	DEN_7	61	61	RW	RW	RW	RW	RW	RW	RW	RW	0h	Source is a Denorm Floating Point Flag, vector_slice7
FSR	INEX_7	60	60	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is inexact Floating Point Flag, vector_slice7
FSR	UNDER_7	59	59	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is underflow Floating Point Flag, vector_slice7
FSR	OVER_7	58	58	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is overflow Floating Point Flag, vector_slice7
FSR	DIV0_7	57	57	RW	RW	RW	RW	RW	RW	RW	RW	0h	Divide by zero Floating Point Flag, vector_slice7

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
FSR	INVAL_7	56	56	RW	RW	RW	RW	RW	RW	RW	RW	0h	Invalid Operations Floating Point Flag, scalar slice || vector_slice7
FSR	SAT_6	55	55	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is saturated Fixed Point Flag, vector_slice6
FSR	UNORD_6	54	54	RW	RW	RW	RW	RW	RW	RW	RW	0h	Compare result is unordered Floating Point Flag, vector_slice6
FSR	DEN_6	53	53	RW	RW	RW	RW	RW	RW	RW	RW	0h	Source is a Denorm Floating Point Flag, vector_slice6
FSR	INEX_6	52	52	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is inexact Floating Point Flag, vector_slice6
FSR	UNDER_6	51	51	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is underflow Floating Point Flag, vector_slice6
FSR	OVER_6	50	50	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is overflow Floating Point Flag, vector_slice6
FSR	DIV0_6	49	49	RW	RW	RW	RW	RW	RW	RW	RW	0h	Divide by zero Floating Point Flag, vector_slice6
FSR	INVAL_6	48	48	RW	RW	RW	RW	RW	RW	RW	RW	0h	Invalid Operations Floating Point Flag, scalar slice || vector_slice6
FSR	SAT_5	47	47	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is saturated Fixed Point Flag, vector_slice5
FSR	UNORD_5	46	46	RW	RW	RW	RW	RW	RW	RW	RW	0h	Compare result is unordered Floating Point Flag, vector_slice5
FSR	DEN_5	45	45	RW	RW	RW	RW	RW	RW	RW	RW	0h	Source is a Denorm Floating Point Flag, vector_slice5
FSR	INEX_5	44	44	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is inexact Floating Point Flag, vector_slice5

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
FSR	UNDER_5	43	43	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is underflow Floating Point Flag, vector_slice5
FSR	OVER_5	42	42	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is overflow Floating Point Flag, vector_slice5
FSR	DIV0_5	41	41	RW	RW	RW	RW	RW	RW	RW	RW	0h	Divide by zero Floating Point Flag, vector_slice5
FSR	INVAL_5	40	40	RW	RW	RW	RW	RW	RW	RW	RW	0h	Invalid Operations Floating Point Flag, scalar slice || vector_slice5
FSR	SAT_4	39	39	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is saturated Fixed Point Flag, vector_slice4
FSR	UNORD_4	38	38	RW	RW	RW	RW	RW	RW	RW	RW	0h	Compare result is unordered Floating Point Flag, vector_slice4
FSR	DEN_4	37	37	RW	RW	RW	RW	RW	RW	RW	RW	0h	Source is a Denorm Floating Point Flag, vector_slice4
FSR	INEX_4	36	36	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is inexact Floating Point Flag, vector_slice4
FSR	UNDER_4	35	35	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is underflow Floating Point Flag, vector_slice4
FSR	OVER_4	34	34	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is overflow Floating Point Flag, vector_slice4
FSR	DIV0_4	33	33	RW	RW	RW	RW	RW	RW	RW	RW	0h	Divide by zero Floating Point Flag, vector_slice4
FSR	INVAL_4	32	32	RW	RW	RW	RW	RW	RW	RW	RW	0h	Invalid Operations Floating Point Flag, scalar slice || vector_slice4
FSR	SAT_3	31	31	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is saturated Fixed Point Flag, vector_slice3

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
FSR	UNORD_3	30	30	RW	RW	RW	RW	RW	RW	RW	RW	0h	Compare result is unordered Floating Point Flag, vector_slice3
FSR	DEN_3	29	29	RW	RW	RW	RW	RW	RW	RW	RW	0h	Source is a Denorm Floating Point Flag, vector_slice3
FSR	INEX_3	28	28	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is inexact Floating Point Flag, vector_slice3
FSR	UNDER_3	27	27	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is underflow Floating Point Flag, vector_slice3
FSR	OVER_3	26	26	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is overflow Floating Point Flag, vector_slice3
FSR	DIV0_3	25	25	RW	RW	RW	RW	RW	RW	RW	RW	0h	Divide by zero Floating Point Flag, vector_slice3
FSR	INVAL_3	24	24	RW	RW	RW	RW	RW	RW	RW	RW	0h	Invalid Operations Floating Point Flag, scalar slice || vector_slice3
FSR	SAT_2	23	23	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is saturated Fixed Point Flag, vector_slice2
FSR	UNORD_2	22	22	RW	RW	RW	RW	RW	RW	RW	RW	0h	Compare result is unordered Floating Point Flag, vector_slice2
FSR	DEN_2	21	21	RW	RW	RW	RW	RW	RW	RW	RW	0h	Source is a Denorm Floating Point Flag, vector_slice2
FSR	INEX_2	20	20	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is inexact Floating Point Flag, vector_slice2
FSR	UNDER_2	19	19	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is underflow Floating Point Flag, vector_slice2
FSR	OVER_2	18	18	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is overflow Floating Point Flag, vector_slice2

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
FSR	DIV0_2	17	17	RW	RW	RW	RW	RW	RW	RW	RW	0h	Divide by zero Floating Point Flag, vector_slice2
FSR	INVAL_2	16	16	RW	RW	RW	RW	RW	RW	RW	RW	0h	Invalid Operations Floating Point Flag, scalar slice || vector_slice2
FSR	SAT_1	15	15	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is saturated Fixed Point Flag, vector_slice1
FSR	UNORD_1	14	14	RW	RW	RW	RW	RW	RW	RW	RW	0h	Compare result is unordered Floating Point Flag, vector_slice1
FSR	DEN_1	13	13	RW	RW	RW	RW	RW	RW	RW	RW	0h	Source is a Denorm Floating Point Flag, vector_slice1
FSR	INEX_1	12	12	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is inexact Floating Point Flag, vector_slice1
FSR	UNDER_1	11	11	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is underflow Floating Point Flag, vector_slice1
FSR	OVER_1	10	10	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is overflow Floating Point Flag, vector_slice1
FSR	DIV0_1	9	9	RW	RW	RW	RW	RW	RW	RW	RW	0h	Divide by zero Floating Point Flag, vector_slice1
FSR	INVAL_1	8	8	RW	RW	RW	RW	RW	RW	RW	RW	0h	Invalid Operations Floating Point Flag, scalar slice || vector_slice1
FSR	SAT_0	7	7	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is saturated Fixed Point Flag, vector_slice0
FSR	UNORD_0	6	6	RW	RW	RW	RW	RW	RW	RW	RW	0h	Compare result is unordered Floating Point Flag, vector_slice0
FSR	DEN_0	5	5	RW	RW	RW	RW	RW	RW	RW	RW	0h	Source is a Denorm Floating Point Flag, vector_slice0

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
FSR	INEX_0	4	4	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is inexact Floating Point Flag, vector_slice0
FSR	UNDER_0	3	3	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is underflow Floating Point Flag, vector_slice0
FSR	OVER_0	2	2	RW	RW	RW	RW	RW	RW	RW	RW	0h	Result is overflow Floating Point Flag, vector_slice0
FSR	DIV0_0	1	1	RW	RW	RW	RW	RW	RW	RW	RW	0h	Divide by zero Floating Point Flag, vector_slice0
FSR	INVAL_0	0	0	RW	RW	RW	RW	RW	RW	RW	RW	h	Invalid Operations Floating Point Flag, scalar slice || vector_slice0
ECLMR	ECLMR	63	0	RW	RO	RW	RO	RO	RO	RO	RO	FFFF FFFF FFFF FFFFh	Claimed interrupts Register, Secure Supervisor claim only
EASGR	EASGR	63	0	RW	RW	RW	RO	RW	RO	RO	RO	0h	Event Assign Register, Root Supervisor, SS and S can assign
EPRI	RSVD2	63	8	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
EPRI	EPRI	7	5	RW	RW	RW	RO	RW	RO	RW	RO	7h	Event Priority Number
EPRI	RSVD1	4	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
EER	EER	63	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Event Flag register
EESET	EESET	63	0	WO	WO	WO	X	WO	X	WO	X	0h	Evet Set Register
EECLR	EECLR	63	0	WO	WO	WO	X	WO	X	WO	X	0h	Event Clear Register
DEPR	RSVD2	63	8	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
DEPR	DEPR	7	5	RW	RW	RO	RO	RO	RO	RO	RO	0h	Debug Event Priority Number
DEPR	RSVD1	4	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
EFR	EFR	63	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Event Flag register
EFSET	EFSET	63	0	WO	WO	WO	X	WO	X	WO	X	0h	Evet Set Register
EFCLR	EFCLR	63	0	WO	WO	WO	X	WO	X	WO	X	0h	Event Clear Register
IESET	IESET	63	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Internal Event Set Register
ESTP_SS	ESTB	63	0	RW	RO	RW	RO	RO	RO	RO	RO	0h	Interrupt Service Table Pointer

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
ESTP_S	ESTB	63	0	RW	RW	RW	RO	RW	RO	RO	RO	0h	Interrupt Service Table Pointer
ESTP_GS	ESTB	63	0	RW	RW	RW	RO	RW	RO	RW	RO	0h	Interrupt Service Table Pointer
EDR	EDR	63	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Dropped Event Register
ECSP_SS	ECSP	63	17	RW	RO	RW	RO	RO	RO	RO	RO	0h	Event Context Save Pointer
ECSP_SS	NCNT	16	13	RW	RO	RW	RO	RO	RO	RO	RO	0h	Event Context NCNT
ECSP_SS	RSVD	12	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
ECSP_S	ECSP	63	17	RW	RW	RW	RO	RW	RO	RO	RO	0h	Event Context Save Pointer
ECSP_S	NCNT	16	13	RW	RW	RW	RO	RO	RO	RO	RO	0h	Event Context NCNT
ECSP_S	RSVD	12	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
ECSP_GS	ECSP	63	17	RW	RW	RW	RO	RW	RO	RW	RO	0h	Event Context Save Pointer
ECSP_GS	NCNT	16	13	RW	RW	RW	RO	RO	RO	RO	RO	0h	Event Context NCNT
ECSP_GS	RSVD	12	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
TCSP	TCSP	63	17	RW	RW	RW	RO	RW	RO	RW	RO	0h	User Context Save Pointer
TCSP	NCNT	16	13	RW	RW	RW	RO	RO	RO	RO	RO	0h	Event Context NCNT
TCSP	RSVD	12	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
RXMR_SS	RSVD2	63	35	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
RXMR_SS	SYSCALL	34	32	RW	RO	RW	RO	RO	RO	RO	RO	0h	Returning execution mode from System Call when RETS is executed
RXMR_SS	RSVD1	31	4	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
RXMR_SS	HDL	3	3	RW	RO	RW	RO	RO	RO	RO	RO	0h	Returning handler bit
RXMR_SS	CXM	2	0	RW	RO	RW	RO	RO	RO	RO	RO	0h	Returning execution mode
RXMR_S	RSVD2	63	35	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
RXMR_S	SYSCALL	34	32	RW	RW	RW	RO	RW	RO	RO	RO	0h	Returning execution mode from System Call when RETS is executed
RXMR_S	RSVD1	31	4	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
RXMR_S	HDL	3	3	RW	RW	RW	RO	RW	RO	RO	RO	0h	Returning handler bit

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
RXMR_S	CXM	2	0	RW	RW	RW	RO	RW	RO	RO	RO	0h	Returning execution mode
AHPEE	RSVD	63	6	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
AHPEE	NUM	5	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Highest Priority Enabled Event
PHPEE	RSVD	63	6	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
PHPEE	NUM	5	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Highest Priority Enabled Event
IPE	RSVD	63	16	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
IPE	IPE	15	0	WO	X	WO	X	WO	X	WO	X	0h	Inter-Processor Event Register
IERR	IERR	63	0	RW	RW	RW	RO	RW	RO	RW	RO	0h	Internal exception report register
IEAR	IEAR	63	0	RW	RW	RW	RO	RW	RO	RW	RO	0h	Internal Event Address Register
IESR	IESR	63	0	RW	RW	RW	RO	RW	RO	RW	RO	0h	Internal Event Status Register
IEDR	IEDR	63	0	RW	RW	RW	RO	RW	RO	RW	RO	0h	Reserved
TCR	RSVD	63	20	RO	RO	RO	RO	RO	RO	RO	RO	0h	Test Counter Register Count Value
TCR	COUNT	19	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Test Counter Register
TCCR	RSVD3	63	17	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
TCCR	UE	16	16	RW	RW	RW	RO	RW	RO	RW	RO	0h	User mode enable bit
TCCR	RSVD2	15	14	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
TCCR	EVTINDEX	13	8	RW	RW	RW	RO	RW	RO	RW	RO	0h	Event number index group for count-to-zero event triggered
TCCR	RSVD1	7	6	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
TCCR	EVTNUM	5	0	RW	RW	RW	RO	RW	RO	RW	RO	0h	Event number for count-to-zero event triggered
GMER	RSVD	63	1	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
GMER	SECCALLEN	0	0	RW	RW	RW	RO	RW	RO	RO	RO	0h	SECCALL enable 0: GS cannot make SECCALL; 1: GS can make SECCALL
UMER	RSVD	63	1	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
UMER	UMEN	0	0	RW	RW	RW	RO	RW	RO	RW	RO	0h	User Mask Enable Register
SPBR	SPBR	63	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Stack Pointer Boundary
LTBR0	RSVD1	63	16	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTBR0	BASE	15	7	RW	RW	RW	RW	RW	RW	RW	RW	0h	Base Address
LTBR0	RSVD0	6	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTBR1	RSVD1	63	16	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
LTBR1	BASE	15	7	RW	RW	RW	RW	RW	RW	RW	RW	0h	Base Address
LTBR1	RSVD0	6	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTBR2	RSVD1	63	16	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTBR2	BASE	15	7	RW	RW	RW	RW	RW	RW	RW	RW	0h	Base Address
LTBR2	RSVD0	6	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTBR3	RSVD1	63	16	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTBR3	BASE	15	7	RW	RW	RW	RW	RW	RW	RW	RW	0h	Base Address
LTBR3	RSVD0	6	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR0	RSVD2	63	29	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR0	VCOP	28	28	RW	RW	RW	RW	RW	RW	RW	RW	0h	VCOP bit to tell L1D to interpret memory configuration: 0: native c7x, 1: same as EVE
LTCR0	RSVD1	27	26	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR0	PROMO	25	24	RW	RW	RW	RW	RW	RW	RW	RW	0h	Levels of precision promotion: 2b"00 -- No Promote 2b"01 -- Promote_2X -> B->H, H->W, W->D 2b"10 -- Promote_4X -> B->W, H->D 2b"11 -- Promote_8X -> Promote B->D
LTCR0	TSIZE	23	16	RW	RW	RW	RW	RW	RW	RW	RW	0h	Table size
LTCR0	RSVD0	15	14	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR0	WSIZE	13	11	RW	RW	RW	RW	RW	RW	RW	RW	0h	Histogram Weight Sizes
LTCR0	INTER	10	8	RW	RW	RW	RW	RW	RW	RW	RW	0h	Interpolating
LTCR0	SAT	7	7	RW	RW	RW	RW	RW	RW	RW	RW	0h	1: Result is saturated, 0: Result is not saturated
LTCR0	SIGN	6	6	RW	RW	RW	RW	RW	RW	RW	RW	0h	1: Signed, 0: Unsigned
LTCR0	ESIZE	5	3	RW	RW	RW	RW	RW	RW	RW	RW	0h	Element size
LTCR0	NTBL	2	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Number of tables
LTCR1	RSVD2	63	29	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR1	VCOP	28	28	RW	RW	RW	RW	RW	RW	RW	RW	0h	VCOP bit to tell L1D to interpret memory configuration: 0: native c7x, 1: same as EVE
LTCR1	RSVD1	27	26	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
LTCR1	PROMO	25	24	RW	RW	RW	RW	RW	RW	RW	RW	0h	Levels of precision promotion: 2b"00 -- No Promote 2b"01 -- Promote_2X -> B->H, H->W, W->D 2b"10 -- Promote_4X -> B->W, H->D 2b"11 -- Promote_8X -> Promote B->D
LTCR1	TSIZE	23	16	RW	RW	RW	RW	RW	RW	RW	RW	0h	Table size
LTCR1	RSVD0	15	14	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR1	WSIZE	13	11	RW	RW	RW	RW	RW	RW	RW	RW	0h	Histogram Weight Sizes
LTCR1	INTER	10	8	RW	RW	RW	RW	RW	RW	RW	RW	0h	Interpolating
LTCR1	SAT	7	7	RW	RW	RW	RW	RW	RW	RW	RW	0h	1: Result is saturated, 0: Result is not saturated
LTCR1	SIGN	6	6	RW	RW	RW	RW	RW	RW	RW	RW	0h	1: Signed, 0: Unsigned
LTCR1	ESIZE	5	3	RW	RW	RW	RW	RW	RW	RW	RW	0h	Element size
LTCR1	NTBL	2	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Number of tables
LTCR2	RSVD2	63	29	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR2	VCOP	28	28	RW	RW	RW	RW	RW	RW	RW	RW	0h	VCOP bit to tell L1D to interpret memory configuration: 0: native c7x, 1: same as EVE
LTCR2	RSVD1	27	26	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR2	PROMO	25	24	RW	RW	RW	RW	RW	RW	RW	RW	0h	Levels of precision promotion: 2b"00 -- No Promote 2b"01 -- Promote_2X -> B->H, H->W, W->D 2b"10 -- Promote_4X -> B->W, H->D 2b"11 -- Promote_8X -> Promote B->D
LTCR2	TSIZE	23	16	RW	RW	RW	RW	RW	RW	RW	RW	0h	Table size
LTCR2	RSVD0	15	14	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR2	WSIZE	13	11	RW	RW	RW	RW	RW	RW	RW	RW	0h	Histogram Weight Sizes
LTCR2	INTER	10	8	RW	RW	RW	RW	RW	RW	RW	RW	0h	Interpolating
LTCR2	SAT	7	7	RW	RW	RW	RW	RW	RW	RW	RW	0h	1: Result is saturated, 0: Result is not saturated

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
LTCR2	SIGN	6	6	RW	RW	RW	RW	RW	RW	RW	RW	0h	1: Signed, 0: Unsigned
LTCR2	ESIZE	5	3	RW	RW	RW	RW	RW	RW	RW	RW	0h	Element size
LTCR2	NTBL	2	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Number of tables
LTCR3	RSVD2	63	29	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR3	VCOP	28	28	RW	RW	RW	RW	RW	RW	RW	RW	0h	VCOP bit to tell L1D to interpret memory configuration: 0: native c7x, 1: same as EVE
LTCR3	RSVD1	27	26	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR3	PROMO	25	24	RW	RW	RW	RW	RW	RW	RW	RW	0h	Levels of precision promotion: 2b"00 -- No Promote 2b"01 -- Promote_2X -> B->H, H->W, W->D 2b"10 -- Promote_4X -> B->W, H->D 2b"11 -- Promote_8X -> Promote B->D
LTCR3	TSIZE	23	16	RW	RW	RW	RW	RW	RW	RW	RW	0h	Table size
LTCR3	RSVD0	15	14	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserve
LTCR3	WSIZE	13	11	RW	RW	RW	RW	RW	RW	RW	RW	0h	Histogram Weight Sizes
LTCR3	INTER	10	8	RW	RW	RW	RW	RW	RW	RW	RW	0h	Interpolating
LTCR3	SAT	7	7	RW	RW	RW	RW	RW	RW	RW	RW	0h	1: Result is saturated, 0: Result is not saturated
LTCR3	SIGN	6	6	RW	RW	RW	RW	RW	RW	RW	RW	0h	1: Signed, 0: Unsigned
LTCR3	ESIZE	5	3	RW	RW	RW	RW	RW	RW	RW	RW	0h	Element size
LTCR3	NTBL	2	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Number of tables
LTER	RSVD3	63	14	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
LTER	LUTE3	13	12	RW	RW	RW	RO	RW	RO	RW	RO	0h	Enable look up table set 3
LTER	RSVD2	11	10	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
LTER	LUTE2	9	8	RW	RW	RW	RO	RW	RO	RW	RO	0h	Enable look up table set 2
LTER	RSVD1	7	6	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
LTER	LUTE1	5	4	RW	RW	RW	RO	RW	RO	RW	RO	0h	Enable look up table set 1
LTER	RSVD0	3	2	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
LTER	LUTE0	1	0	RW	RW	RW	RO	RW	RO	RW	RO	0h	Enable look up table set 0
DBGCTXT	RSVD	63	32	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
DBGCTXT	DBGCTXT	31	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Debug Context Switch Register

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
PC_PROF	PC_PROF	63	0	RW	RW	RO	RO	RO	RO	RO	RO	0h	PC Profile Register
GPLY	RSVD	63	32	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
GPLY	GPLY	31	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Galois Polynomial Register
GFPGFR	RSVD2	63	27	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
GFPGFR	SIZE	26	24	RW	RW	RW	RW	RW	RW	RW	RW	7h	Field size
GFPGFR	RSVD1	23	8	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
GFPGFR	POLY	7	0	RW	RW	RW	RW	RW	RW	RW	RW	Dh	Polynomial Generator
GTSC	GTSC	63	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Global Time Stamp Counter
STSC	STSC	63	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	Shadow Time Stamp Counter
UFCMR	UFCMR	63	0	RW	RW	RW	RO	RW	RO	RW	RO	0h	User Flag Clear Mask Register
CUCR0	CUCR	511	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Local C unit Ctrl registers data
CUCR1	CUCR	511	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Local C unit Ctrl registers data
CUCR2	CUCR	511	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Local C unit Ctrl registers data
CUCR3	CUCR	511	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	Local C unit Ctrl registers data
STRACR0	RSVD_FLAG3	511	506	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR0	DECDIM2SD	505	504	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2SD field for Vertical strip mine
STRACR0	DECDIM2	503	501	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2 field for Vertical strip mine
STRACR0	DECDIM1SD	500	499	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1SD field for Vertical strip mine
STRACR0	DECDIM1	498	496	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1 field for Vertical strip mine
STRACR0	RSVD_FLAG2	495	475	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR0	DIMFMT	474	472	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIMFMT flags
STRACR0	RSVD_FLAG1	471	463	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR0	VECLEN	462	460	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA VECLENGTH flags
STRACR0	RSVD_FLAG0	459	448	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR0	RSVD1	447	416	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA DIM5 value

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
STRACR0	DIM5	415	384	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM4 value
STRACR0	DIM4	383	352	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM3 value
STRACR0	DIM3	351	320	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM2 value
STRACR0	DIM2	319	288	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM1 value
STRACR0	DIM1	287	256	RW	RW	RW	RW	RW	RW	RW	RW	0h	Reserved
STRACR0	DECDIM2_WIDTH	255	224	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2 width
STRACR0	DECDIM1_WIDTH	223	192	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1 width
STRACR0	ICNT5	191	160	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT5
STRACR0	ICNT4	159	128	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT4
STRACR0	ICNT3	127	96	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT3
STRACR0	ICNT2	95	64	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT2
STRACR0	ICNT1	63	32	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT1
STRACR0	ICNT0	31	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT0
STRACR1	RSVD_FLAG3	511	506	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR1	DECDIM2SD	505	504	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2SD field for Vertical strip mine
STRACR1	DECDIM2	503	501	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2 field for Vertical strip mine
STRACR1	DECDIM1SD	500	499	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1SD field for Vertical strip mine
STRACR1	DECDIM1	498	496	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1 field for Vertical strip mine
STRACR1	RSVD_FLAG2	495	475	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR1	DIMFMT	474	472	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIMFMT flags
STRACR1	RSVD_FLAG1	471	463	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR1	VECLEN	462	460	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA VECLENGTH flags
STRACR1	RSVD_FLAG0	459	448	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR1	RSVD1	447	416	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA DIM5 value

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
STRACR1	DIM5	415	384	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM4 value
STRACR1	DIM4	383	352	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM3 value
STRACR1	DIM3	351	320	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM2 value
STRACR1	DIM2	319	288	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM1 value
STRACR1	DIM1	287	256	RW	RW	RW	RW	RW	RW	RW	RW	0h	Reserved
STRACR1	DECDIM2_WIDTH	255	224	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2 width
STRACR1	DECDIM1_WIDTH	223	192	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1 width
STRACR1	ICNT5	191	160	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT5
STRACR1	ICNT4	159	128	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT4
STRACR1	ICNT3	127	96	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT3
STRACR1	ICNT2	95	64	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT2
STRACR1	ICNT1	63	32	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT1
STRACR1	ICNT0	31	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT0
STRACR2	RSVD_FLAG3	511	506	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR2	DECDIM2SD	505	504	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2SD field for Vertical strip mine
STRACR2	DECDIM2	503	501	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2 field for Vertical strip mine
STRACR2	DECDIM1SD	500	499	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1SD field for Vertical strip mine
STRACR2	DECDIM1	498	496	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1 field for Vertical strip mine
STRACR2	RSVD_FLAG2	495	475	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR2	DIMFMT	474	472	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIMFMT flags
STRACR2	RSVD_FLAG1	471	463	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR2	VECLEN	462	460	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA VECLENGTH flags
STRACR2	RSVD_FLAG0	459	448	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR2	RSVD1	447	416	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA DIM5 value

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
STRACR2	DIM5	415	384	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM4 value
STRACR2	DIM4	383	352	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM3 value
STRACR2	DIM3	351	320	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM2 value
STRACR2	DIM2	319	288	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM1 value
STRACR2	DIM1	287	256	RW	RW	RW	RW	RW	RW	RW	RW	0h	Reserved
STRACR2	DECDIM2_WIDTH	255	224	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2 width
STRACR2	DECDIM1_WIDTH	223	192	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1 width
STRACR2	ICNT5	191	160	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT5
STRACR2	ICNT4	159	128	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT4
STRACR2	ICNT3	127	96	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT3
STRACR2	ICNT2	95	64	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT2
STRACR2	ICNT1	63	32	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT1
STRACR2	ICNT0	31	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT0
STRACR3	RSVD_FLAG3	511	506	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR3	DECDIM2SD	505	504	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2SD field for Vertical strip mine
STRACR3	DECDIM2	503	501	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2 field for Vertical strip mine
STRACR3	DECDIM1SD	500	499	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1SD field for Vertical strip mine
STRACR3	DECDIM1	498	496	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1 field for Vertical strip mine
STRACR3	RSVD_FLAG2	495	475	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR3	DIMFMT	474	472	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIMFMT flags
STRACR3	RSVD_FLAG1	471	463	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR3	VECLEN	462	460	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA VECLENGTH flags
STRACR3	RSVD_FLAG0	459	448	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA flags Reserved
STRACR3	RSVD1	447	416	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA DIM5 value

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
STRACR3	DIM5	415	384	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM4 value
STRACR3	DIM4	383	352	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM3 value
STRACR3	DIM3	351	320	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM2 value
STRACR3	DIM2	319	288	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA DIM1 value
STRACR3	DIM1	287	256	RW	RW	RW	RW	RW	RW	RW	RW	0h	Reserved
STRACR3	DECDIM2_WIDTH	255	224	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM2 width
STRACR3	DECDIM1_WIDTH	223	192	RW	RW	RW	RW	RW	RW	RW	RW	0h	DECDIM1 width
STRACR3	ICNT5	191	160	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT5
STRACR3	ICNT4	159	128	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT4
STRACR3	ICNT3	127	96	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT3
STRACR3	ICNT2	95	64	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT2
STRACR3	ICNT1	63	32	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT1
STRACR3	ICNT0	31	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Loop CNT0
STRACNTR0	DECDIM2_CNT	511	480	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM1_CNT
STRACNTR0	DECDIM1_CNT	479	448	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM2_CNT
STRACNTR0	ICNT5	447	416	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT5
STRACNTR0	ICNT4	415	384	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT4
STRACNTR0	ICNT3	383	352	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT3
STRACNTR0	ICNT2	351	320	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT2
STRACNTR0	ICNT1	319	288	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT1
STRACNTR0	ICNT0	287	256	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT0
STRACNTR0	DECDIM2SD_CNT	255	224	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM2SD_CNT
STRACNTR0	DECDIM1SD_CNT	223	192	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM1SD_CNT
STRACNTR0	I5_OFFSET	191	160	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I5 offset

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
STRACNTR0	I4_OFFSET	159	128	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I4 offset
STRACNTR0	I3_OFFSET	127	96	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I3 offset
STRACNTR0	I2_OFFSET	95	64	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I2 offset
STRACNTR0	I1_OFFSET	63	32	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I1 offset
STRACNTR0	CURR_OFFSET	31	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Current offset
STRACNTR1	DECDIM2_CNT	511	480	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM1_CNT
STRACNTR1	DECDIM1_CNT	479	448	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DEC_DIM_CNT0
STRACNTR1	ICNT5	447	416	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT5
STRACNTR1	ICNT4	415	384	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT4
STRACNTR1	ICNT3	383	352	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT3
STRACNTR1	ICNT2	351	320	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT2
STRACNTR1	ICNT1	319	288	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT1
STRACNTR1	ICNT0	287	256	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT0
STRACNTR1	DECDIM2SD_CNT	255	224	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM2SD_CNT
STRACNTR1	DECDIM1SD_CNT	223	192	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM1SD_CNT
STRACNTR1	I5_OFFSET	191	160	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I5 offset
STRACNTR1	I4_OFFSET	159	128	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I4 offset
STRACNTR1	I3_OFFSET	127	96	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I3 offset
STRACNTR1	I2_OFFSET	95	64	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I2 offset
STRACNTR1	I1_OFFSET	63	32	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I1 offset
STRACNTR1	CURR_OFFSET	31	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Current offset
STRACNTR2	DECDIM2_CNT	511	480	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM1_CNT
STRACNTR2	DECDIM1_CNT	479	448	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DEC_DIM_CNT0

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
STRACNTR2	ICNT5	447	416	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT5
STRACNTR2	ICNT4	415	384	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT4
STRACNTR2	ICNT3	383	352	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT3
STRACNTR2	ICNT2	351	320	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT2
STRACNTR2	ICNT1	319	288	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT1
STRACNTR2	ICNT0	287	256	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT0
STRACNTR2	DECDIM2SD_CNT	255	224	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM2SD_CNT
STRACNTR2	DECDIM1SD_CNT	223	192	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM1SD_CNT
STRACNTR2	I5_OFFSET	191	160	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I5 offset
STRACNTR2	I4_OFFSET	159	128	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I4 offset
STRACNTR2	I3_OFFSET	127	96	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I3 offset
STRACNTR2	I2_OFFSET	95	64	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I2 offset
STRACNTR2	I1_OFFSET	63	32	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I1 offset
STRACNTR2	CURR_OFFSET	31	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Current offset
STRACNTR3	DECDIM2_CNT	511	480	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM1_CNT
STRACNTR3	DECDIM1_CNT	479	448	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DEC_DIM_CNT0
STRACNTR3	ICNT5	447	416	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT5
STRACNTR3	ICNT4	415	384	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT4
STRACNTR3	ICNT3	383	352	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT3
STRACNTR3	ICNT2	351	320	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT2
STRACNTR3	ICNT1	319	288	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT1
STRACNTR3	ICNT0	287	256	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA current loop CNT0
STRACNTR3	DECDIM2SD_CNT	255	224	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM2SD_CNT

Table 3-76. TBD (continued)

Register Name	Bitfield	MSB	LSB	Secure Debug Perms	Non-Secure Debug Perms	Secure Supervisor Perms	Secure User Perms	Supervisor Perms	User Perms	Guest Supervisor Perms	Guest User Perms	Reset Value	Description
STRACNTR3	DECDIM1SD_CNT	223	192	RW	RW	RW	RW	RW	RW	RW	RW	0h	Current DECDIM1SD_CNT
STRACNTR3	I5_OFFSET	191	160	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I5 offset
STRACNTR3	I4_OFFSET	159	128	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I4 offset
STRACNTR3	I3_OFFSET	127	96	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I3 offset
STRACNTR3	I2_OFFSET	95	64	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I2 offset
STRACNTR3	I1_OFFSET	63	32	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA I1 offset
STRACNTR3	CURR_OFFSET	31	0	RW	RW	RW	RW	RW	RW	RW	RW	0h	SA Current offset
SA0	RSVD	63	32	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
SA0	SA_OFFSET	31	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA offset value
SA1	RSVD	63	32	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
SA1	SA_OFFSET	31	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA offset value
SA2	RSVD	63	32	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
SA2	SA_OFFSET	31	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA offset value
SA3	RSVD	63	32	RO	RO	RO	RO	RO	RO	RO	RO	0h	Reserved
SA3	SA_OFFSET	31	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA offset value
PSA0	PSA	63	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA byte enables
PSA1	PSA	63	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA byte enables
PSA2	PSA	63	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA byte enables
PSA3	PSA	63	0	RO	RO	RO	RO	RO	RO	RO	RO	0h	SA byte enables
PC	PC	48	0	RW	RW	X	X	X	X	X	X	0h	Program Counter, read-only in functional mode.
ILCNT	ILCNT	63	0	RO	RO	X	X	X	X	X	X	0h	Inner Loop Count
OLCNT	OLCNT	63	0	RO	RO	X	X	X	X	X	X	0h	Outer Loop Count
ILCNT_INIT	ILCNT_INIT	63	0	RO	RO	X	X	X	X	X	X	0h	Inner Loop Count Initial Value
EPISKEW	RSVD	63	6	RO	RO	X	X	X	X	X	X	0h	Reserved
EPISKEW	EPISKEW	5	0	RO	RO	X	X	X	X	X	X	0h	EPISKEW
LCNTFLAG	FLG	63	0	RO	RO	X	X	X	X	X	X	0h	Loop Counter Flags

3.7.49.3 Extended Control Registers

Extended Control Registers are those registers which are not inside the CPU.

The table below lists the register addresses that MVC instruction used for accessing the extended control registers in different modules outside of the CPU.

Table 3-77. MVC Register Indexes for Accessing the Extended Control Registers

Acronym	ECR Address		Module	Supervisor Read/ Write	User Read/Write
	High	Low			
Debug_CRs inside Thinman	0001xx	xxxxxx	IAP	R,W	X
L1D_CRs	001000	xxxxxx	L1D	R,W	X
L1I_CRs	001001	xxxxxx	L1I	R,W	X
L2_CRs	00101x	xxxxxx	L2	R,W	X
SE0_CRs	001100	xxxxxx	SE	R,W	X
SE1_CRs	001101	xxxxxx	SE	R,W	X
Reserved	00111x	xxxxxx			
CMMU CRs	01000x	xxxxxx	CMMU	R,W	X
L1I_uTLB_CRs	010010	0xxxxx	L1I	R,W	X
L1D_uTLB_CRs	010010	1xxxxx	L1D	R,W	X
SE0_uTLB_CRs	010011	0xxxxx	SE	R,W	X
SE1_uTLB_CRs	010011	1xxxxx	SE	R,W	X
System_CRs	0110xx	xxxxxx	Sys	R,W	X
PBIST_CRs	0111xx	xxxxxx	Sys	R,W	X
CT-SET CRs	1000xx	xxxxxx	Sys	R,W	X
Debug CRs inside Matlock	10011x	xxxxxx	IAP	R,W	X
Debug CRs inside Columbo	101xxx	xxxxxx	IAP	R,W	X
Reserved	11xxxx	xxxxxx			

3.8 Opcode Maps

The 32-bit opcodes are mapped in Section 3.13.

3.9 Delay Slots

The execution of fixed-point and floating-point instructions when the processor is operating in unprotected pipeline mode can be defined in terms of delay slots and functional unit latency.

The number of delay slots is equivalent to the number of additional cycles required after the source operands are read for the result to be available for reading.

The functional unit latency is equivalent to the number of cycles that must pass before the functional unit can start executing the next instruction.

All C7x instructions have a functional unit latency of one cycle (can be fully pipelined).

Table 1-51 shows the number of delay slots associated with each type of instruction.

Table 3-78. Delay Slot and Functional Unit Latency

Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles(1)	Write Cycles	Branch Taken
NOP	0	1			
Store	0	1	i	i	
Load	4	1	i	i, i+4(2)	
Branch	0	1	i(3)		i
Single cycle	0	1	i	i	
2-cycle	1	1	i	i+1	
3-cycle	2	1	i	i+2	

Table 3-78. Delay Slot and Functional Unit Latency (continued)

Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles(1)	Write Cycles	Branch Taken
4-cycle	3	1	i	i+3	

(1) Cycle i is in the E1 pipeline phase.
(2) For loads, any address modification happens in cycle i. The loaded data is written into the register file in cycle (i+4).
(3) The branch to label, branch to IRP, and branch to NRP instructions do not read any general purpose registers.

Note that the delay slot and functional unit latency concepts do not apply when the processor i s operating in protected pipeline mode. During protected pipeline mode, all instructions are considered to have 0 delay slots and execution conforms to the sequential operation model where dependent instructions are guaranteed to be correct, regardless of how many cycles it takes to complete the instructions.

3.10 Parallel Operations

Instructions are always fetched sixteen words at a time. This constitutes a fetch packet. The basic format of a fetch packet is shown in Figure 1-63. Fetch packets are aligned on 512-bit (16-word) boundaries.

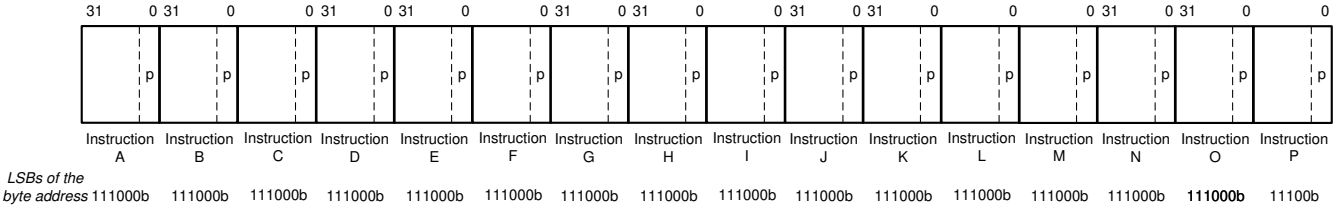


Figure 3-16. Basic Format of a Fetch Packet

The execution of the individual instructions is partially controlled by a bit in each instruction, the p-bit. The p-bit (bit 0) determines whether the instruction executes in parallel with another instruction. The p-bits are scanned from left to right (lower to higher address). If the p-bit of instruction I is 1, then instruction I + 1 is to be executed in parallel with (in the same cycle as) instruction I. If the p-bit of instruction I is 0, then instruction I + 1 is executed in the cycle after instruction I. All instructions executing in parallel constitute an execute packet. Each instruction in an execute packet must use a different functional unit.

In C7x CPU, an execute packet can contain up to sixteen 32-bit wide slots. Each slot in an execute packet must use a different functional unit. A slot can either be a self-contained instruction or is used to expand the constant field of associated instruction(s). Also, a slot can be used as conditional codes to apply to the instructions within the same fetch packet. A fetch packet can contain up to 2 constant extension slots and two condition code extension slots.

There are up to 18 distinct instruction slots. It is up to the compiler to limit the number of parallel instructions to 16 to fit inside one fetch packet.

- .L1 unit
- .S1 unit
- .M1 unit
- .N1 unit
- .D1 unit
- .L2 unit
- .S2 unit
- .M2 unit
- .N2 unit
- .D2 unit
- .C unit
- .P unit
- Unitless instructions
- Branch instructions
- Constant Extension 0
- Constant Extension 1

- Condition Code Extension 0
- Condition Code Extension 1

The last instruction in an execute packet will be marked with its p-bit cleared to zero. There are three types of p-bit patterns for fetch packets. These three p-bit patterns result in the following execution sequences for the instructions:

- Fully serial
- Fully parallel
- Partially serial

Section 3.10.1 through Section 3.10.3 show the conversion of a p-bit sequence into a cycle-by-cycle execution stream of instructions.

3.10.1 Example 1-3 Fully Serial p-Bit Pattern in a Fetch Packet

All the instructions are executed sequentially.

This p-bit pattern:

Figure 3-17. TBD

results in this execution sequence:

Table 3-79. TBD

Cycle/Execute Packet	Instructions
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H
9	I
10	J
11	K
12	L
13	M
14	N
15	O
16	P

3.10.2 Example 1-4 Parallel p-Bit Pattern in a Fetch Packet

First nine instructions in the fetch packet are executed in parallel, and the second 7 instructions are executed in parallel a cycle later.

This p-bit pattern:

Figure 3-18. TBD

results in this execution sequence:

3.10.3 Example 1-5 Partially Serial p-Bit Pattern in a Fetch Packet

This p-bit pattern:

Figure 3-19. TBD

results in this execution sequence:

Table 3-80.

Cycle/Execute Packet	Instructions						
1	A						
2	B						
3	C	D	E	F	G	H	I
4	J	K	L	M			
5	N	O	P				

3.10.3.1 Example Parallel Code

The vertical bars || signify that an instruction is to execute in parallel with the previous instruction. The code for the fetch packet in Section 3.10.3 would be represented as this:

```
         instruction A
         instruction B
          instruction C
||        instruction D
||        instruction E
||        instruction F
||        instruction G
||        instruction H
||        instruction I
          instruction J
||        instruction K
```

```
||        instruction L
||        instruction M
        instruction N
||        instruction O
||        instruction
```

3.11 Branching Into the Middle of an Execute Packet

Branching into the middle of an execute packet is not allowed. For instance, in Section 3.10.3, branching to the address containing instruction D, which is in the middle of the execute packet, is not allowed. If that occurs, results are undefined.

3.12 Conditional Operations

Conditional conditions are encoded differently in C7x than in C66x. In C66x, most instructions contain a 3-bit condition field (CREG field) that specifies the condition register, and a 1-bit field (Z) that specifies a test for zero or nonzero. In C7x, to minimize opcode space, only a number of selected instructions contain the CREGZ field directly in the encoding. These are the instructions which get used the most to alter program control flows such as branches and move instructions. The rest of the instructions do not have the CREGZ field encoded directly, and normally treated as non-conditional. However, if needed, an execute packet can contain two unique 32-bit Condition Code Extension slots which specifies the 4-bit CREGZ fields for the instructions which are in the same execute packet.

In the case that the instruction contains the embedded CREGZ field in its opcode but also get specified to use the Condition Code Extension, the CREGZ field in the Condition Code Extension is used and the embedded CREGZ field is ignored.

Table 1-52 shows the conditional register specified by a 4-bit CREGZ field

Table 3-81. Registers That Can Be Tested by CREGZ field

Specified Conditional Register	creg				z
	Bit:	31	30	29	28
Unconditional		0	0	0	0
Reserved		0	0	0	1
A0		0	0	1	z
A1		0	1	0	z
A2		0	1	1	z
A3		1	0	0	z
A4		1	0	1	z
A5		1	1	0	z
Reserved		1	1	1	x

Table 1-53 and Table 1-54 shows the unit assignment for each 4-bit field inside the two Condition Code Extension slots.

Table 3-82. Condition Code Extension Functional Unit Assignment Slot 0

Extension Bits	[31:28]	[27:24]	[23:20]	[19:16]	[15:12]	[11:8]	[7]	[6:0]
Unit	.L1	.L2	.S1	.S2	.D1	.D2	Resv	0011101

Table 3-83. Condition Code Extension Functional Unit Assignment Slot 1

Extension Bits	[31:28]	[27:24]	[23:20]	[19:16]	[15:12]	[11:8]	[7]	[6:0]
Unit	.M1	.M2	.C	.P	.N1	.N2	Resv	1011101

Conditional instructions are represented in code by using square brackets, [ ], surrounding the condition register name. The following execute packet contains two MV instructions in parallel. The first MV is conditional on A0 being nonzero. The second MV is conditional on A0 being zero. The character ! indicates the inverse of the condition.

[A0] MV .L1 A1, A2 || [!A0] MV .S1 A3, A4

The above instructions are mutually exclusive, only one will execute. If they are scheduled in parallel, mutually exclusive instructions are constrained as described in Section 3.14. If mutually exclusive instructions share any resources as described there, they cannot be scheduled in parallel (put in the same execute packet), even though only one will execute.

The act of making an instruction conditional is called predication and the conditional register is called the predication register. For C7x, predication can be applied to both scalar and vector instructions. However, predication on a vector instruction incurs an additional stall cycle if the predicate value is used in the immediate cycle after it is defined.

3.13 Constant Extensions

The C7x CPU can extend up to two constants per cycle, so up to two constant extensions can be issued for every execute packet. Each constant extension is associated with defined set of instruction slots:

- CSTX0: associated with .L1, .D1 data, .S2, .D2 offset, .M2, .N2, .B, .C
- CSTX1: associated with .L2, .D2 data, .S1, .D1 offset, .M1, .N1

Where:

- .Dx data: only uses for store-a-constant instructions as the source constant to store out
- .Dx offset: src2 field - constant offset in address calculation for load/store, ADDAx instructions and as constant source for arithmetic instructions.

Most instructions can only utilize 1 constant extension at the time, except for the MVK64 instruction which move 64 bit constants and STKD (Store 64-bit constant) instructions. They are special cases and need to utilize both constant extensions simultaneously.

For the case of MVK-32-bit, the 32-bit constant is formed by putting the constant extensions in the order of CSTX0[26:0] & 5-bit embedded constant from the instruction encoding itself

For the case of MVK-64-bit, the 64-bit constant is formed by putting the constant extensions in the order of CSTX1[26:0] & CSTX0[26:0] & 10-bit embedded constant from the instruction encoding itself.

3.14 Resource Constraints

No two instructions within the same execute packet can use the same resources. Also, no two instructions can write to the same register during the same cycle. The following sections describe how an instruction can use each of the resources.

3.14.1 1.4.14.1 Constraints on Instruction Issue Slots

- The Branch-type instructions are always issued at the end of the execute packet
- Instructions which have predication register as destination can only be issued to side B execution units

3.14.2 1.4.14.2 Constraints on Instructions Using the Same Functional Unit

Two instructions using the same functional unit cannot be issued in the same execute packet.

The following execute packet is invalid:

```
ADD .S1     A0, A1, A2     ;.S1 is used for
||  SHR .S1     A3, 15, A4     ;...both instructions
```

The following execute packet is valid:

```
ADD .L1     A0, A1, A2     ;Two different functional
||  SHR .S1     A3, 15, A4     ;...units are used
```

3.14.3 1.4.14.3 Constraints on the Same Functional Unit Writing in the Same Instruction Cycle

On the C7x, each functional unit can only write the result of one instruction per cycle.

For example, even though the functional units can execute instructions with different latencies, two independent writes from the same execution unit to the register file in the same instruction cycle is not supported and will result in an exception and erroneous values being written to the destination registers.

Therefore, the following sequence is invalid since VB3 and VB5 are written by the .S2 unit on the same cycle.

```
VADDSP .S2 VB0, VB1, VB3 ; this instruction has 2 delay slots ;
NOP
VRCPSP .S2 VB3, VB4, VB5; this instruction has one delay slot;
NOP
```

Another example is that even though the .M unit has two write ports to the register file to allow up to 1024-bit vector result, the results of a 512-bit 4-cycle instruction and a 512-bit 2-cycle instruction operating on the same .M unit can not be written in the same instruction cycle.

3.14.4 Constraints on Register Writes

Two instructions cannot write to the same register on the same cycle. Two instructions with the same destination can be scheduled in parallel as long as they do not write to the destination register on the same cycle. For example, an MPY issued on cycle I followed by an ADD on cycle I + 1 cannot write to the same register because both instructions write a result on cycle I + 1. Therefore, the following code sequence is invalid unless a branch occurs after the MPY, causing the ADD not to be issued.

```
MPY .M1 A0, A1, A2
ADD .L1     A4,     A5,     A2
```

However, this code sequence is valid:

```
MPY  .M1 A0, A1, A2
||     ADD      .L1 A4,     A5, A2
```

Figure 1-64 shows different multiple-write conflicts. For example, ADD and SUB in execute packet L1 write to the same register. This conflict is easily detectable.

MPY in packet L2 and ADD in packet L3 might both write to B2 simultaneously; however, if a branch instruction causes the execute packet after L2 to be something other than L3, a conflict would not occur. Thus, the potential conflict in L2 and L3 might not be detected by the assembler. The instructions in L4 do not constitute a write conflict because they are mutually exclusive. In contrast, because the instructions in L5 may or may not be mutually exclusive, the assembler cannot determine a conflict. If the pipeline does receive commands to perform multiple writes to the same register, the result is undefined.

```
Figure 1-64        Examples of the Detectability of Write Conflicts by the Assembler
L1:                    ADD    .L1        A5,    A6,    A7     ; \ detectable, conflict
||                  SUB    .S1        A8,    A9,    A7     ; /
L2:                 MPY        .M1        A0,    A1,    A2     ; \ not detectable
L3:                 ADD        .L1        A3,    A4,    A2     ; /
L4:    [!A0]    ADD    .L1        A5,    A6,    A7    ; \ detectable, no conflict
||    [A0]     SUB    .S1        A8,    A9,    A7     ; /
L5:    [!A1]    ADD    .L1        A5,    A6,    A7    ; \ not detectable
||    [A0]     SUB    .S1        A8,    A9,    A7    ; /
```

3.14.5 Constraints on Cross Paths (1X and 2X)

3.14.5.1 Maximum Number of Accesses to a Register on the Opposite Register File

The C7x cross paths allows multiple reads through one cross path to the same register. For example the following sequence is valid:

```
ADD .L1 A0, B0, A1
 ||     ADD .S1 A2, B0, A2
 ||     ADD .D1 A3, B0, A3
 ||     MPY .M1 A4, B0, A4
```

3.14.5.2 Cross Path Sharing

On the C7x, the same register can be read multiple times through the cross path on the same cycle. The 64-bit cross path can only be used to transport a 64-bit register value.

3.14.6 Cross Path Stalls

The DSP introduces a delay clock cycle whenever an instruction attempts to read a register via a cross path that was updated in the previous cycle. This is known as a cross path stall. This stall is inserted automatically by the hardware, no NOP instruction is needed. It should be noted that no stall is introduced if the register being read has data placed by a load instruction, or if an instruction reads a result one cycle after the result is generated.

Here are some examples:

```
ADD  .S1   A0, A0, A1                        ; / Stall is introduced; A1 is updated
                       ;   1 cycle before it is used as a
ADD  .S2X  A1, B0, B1                        ; \ cross path source
ADD  .S1   A0, A0, A1                        ; / No stall is introduced; A0 not updated
                       ;   1 cycle before it is used as a cross
ADD  .S2X  A0, B0, B1                        ; \ path source
LDW  .D1 *++A0[1], A1                        ; / No stall is introduced; A1 is the load
                       ;   destination
NOP  4                                       ;   NOP 4 represents 4 instructions to
ADD  .S2X  A1, B0, B1                        ; \ be executed between the load and add.
LDW  .D1  *++A0[1], A1                       ; / Stall is introduced; A0 is updated
ADD  .S2X   A0, B0, B1                       ;   1 cycle before it is used as a
                       ; \ cross path source
```

It is possible to avoid the cross path stall by scheduling an instruction that reads an operand via the cross path at least one cycle after the operand is updated. With appropriate scheduling, the DSP can provide one cross path operand per data path per cycle with no stalls. In many cases, the C7x Optimizing Compiler and Assembly Optimizer automatically perform this scheduling.

3.14.7 Crosspath Constraints

VSTINTL[B/B4/H/W/D], VSTPINTL[B/B4/H/W/D], LUTRD, LUTWR, HIST and WHIST instructions, which are issued on .D2 unit, utilize the crosspath resource to read the upper vector side-B operand so instructions from side-A which attempt to read another side B register via crosspath can not be issued in parallel. Note that side-A instruction which read the same side B register via crosspath is allowed.

For example:

- Illegal: VSTINTLB .D2X VB1:VB0, *A0[0] || ADDW .L1X B2, A1, A2
- Legal: VSTINTLB .D2X VB1:VB0, *A0[0] || ADDW .L1X B1, A1, A2

3.14.8 Constraints on Constant Extension Sharing

Each constant extension can only be used by one execution unit per execute packet.

3.14.9 Constraints on Loads and Stores

The base address register and the offset register for .D1 unit can only come from the global scalar register file or the shared .D unit local register file, for all addressing modes.

For .D2 unit, the base address register and the offset register come from the global scalar register file or the shared .D unit local register file for all addressing mode, except post address increment mode. If the addressing mode is post increment mode, the base address register for .D2 unit can only come from the .D unit local register file.

Load and store instructions can use an address pointer from these two register files while loading to or storing data from other register files.

3.14.10 Constraints on Look Up Table and Histogram Instructions

The look up table and histogram instructions can not be issued in parallel with another memory access instructions such as loads or stores.

3.14.11 Constraints on Multicycle NOPs

Two instructions that generate multicycle NOPs cannot share the same execute packet. Instructions that generate a multicycle NOP are:

- NOP n (where n > 1)
- IDLE

Also, a multicycle NOP can not be issued in parallel with a branch instructions.

3.14.12 Constraints on Branch instructions

Branch instructions must always be issued at the beginning of the execution packet, i.e its P-bit is always 0. Also, only one branch instruction can be issued per execute packet, regardless of whether it is predicated true or false.

3.14.13 Constraints on FIR and MATMPY Instructions

The operands for the VFIR and VMATMPY instructions come from the Streaming Engine registers. A permute network then performs necessary data alignment before feeding the operands to the .M2 and .N2 units.

Table 3-84. VFIR and VMATMPY Scheduling Restrictions Related to Streaming Engine Operands

Instruction	Parallel Instructions Using Same SE	Parallel Instructions Using Different SE	Other Comments
VFIR8DS4HD	ILLEGAL	NA / not possible	Dual-issue to .M2 and .N2
VFIR8DS2HD	ILLEGAL	NA / not possible	Dual-issue to .M2 and .N2
VFIR8HD	ILLEGAL	Legal	Dual-issue to .M2 and .N2
VFIR8DS4HW	ILLEGAL	NA / not possible	Dual-issue to .M2 and .N2
VFIR8DS2HW	ILLEGAL	NA / not possible	Dual-issue to .M2 and .N2
VFIR8HW	ILLEGAL	Legal	Dual-issue to .M2 and .N2
VFIR4DS2HW	ILLEGAL	NA / not possible	Dual-issue to .M2 and .N2
VFIR4HW	ILLEGAL	NA / not possible	Dual-issue to .M2 and .N2
VMATMPYHD	ILLEGAL	ILLEGAL	Dual-issue to .M2 and .N2, can use both SE0 and SE1
VMATMPYHW	ILLEGAL	ILLEGAL	Dual-issue to .M2 and .N2, can use both SE0 and SE1

3.14.14 Constraints on MFENCE Instruction

The MFENCE instruction will stall until the completion of all the CPU-triggered memory transactions, including:

- Cache line fills
- Writes from L1D to L2 or from CorePac to MSMC and/or other system endpoints
- Victim write backs
- Block or global coherence operations
- Cache mode changes
- Outstanding XMC prefetch requests

To determine if all the memory transactions are completed, the MFENCE instruction checks an internal busy flag. MFENCE always wait at least 5 clock cycles before checking the busy flag in order to account for pipeline delays.

The following code is legal:

```
STW          A0, *A1
MFENCE                    ; This will wait until the STW write above
                ; has landed in it's final destination
```

During the course of executing a MFENCE operation, any enabled interrupts will still be serviced.

When an interrupt occurs during the execution of a MFENCE instruction, the address of the execute packet containing the MFENCE instruction is saved in IRP or NRP. This forces returning to the MFENCE instruction after interrupt servicing.

The MFENCE instruction can not be issued in parallel with another memory access instruction such as loads, stores, look up table and histogram.

3.14.15 Constraints On Atomic Instructions (CASW, CASD, CASQ, ASW, ASD, FAW, FAD)

The Atomic Instructions can not be issued in parallel with another memory access instruction such as loads, stores, look up table and histogram.

3.14.16 Constraints On IDLE Instruction

An IDLE instruction cannot be placed in parallel with the following instructions:

- NOP n (if n > 1)

IDLE instruction can be placed in parallel with the NOP instruction.

IDLE instruction can not be issued in UNPROT mode. If IDLE instruction is encountered in UNPROT mode, illegal opcode exception is raised.

3.14.17 Constraints on NOP Instruction

A NOPn (with n > 1) instruction cannot be placed in parallel with other instructions.

3.14.18 Constraint on DEVT Instruction

A DEVT instruction cannot be placed in parallel with the following instructions:

- MVC reg, TSR
- REVT
- IDLE
- RETE
- RETI
- RETS
- SYSCALL
- NOP n (if n>1)

3.14.19 Constraint on REVT Instruction

A REVT instruction cannot be placed in parallel with the following instructions:

- MVC reg, TSR
- DEVT
- IDLE
- RETE
- RETI
- RETS
- SYSCALL
- NOP n (if n>1)

3.14.20 Constraints on scalar predication

The following instructions are unconditional and can not be predicated:

- REVT
- DEVT
- PROT
- UNPROT

3.14.21 Constraints on Instructions in UNPROT mode

The following instructions can not be issued in UNPROT mode. Exception is raised if these instructions are encountered during UNPROT mode:

- CALL

- CALLA
- CALLE
- SYSCALL
- ROOTCALL
- SECCALL
- RET
- RETE
- RETS
- IDLE
- DEVT
- REVT

3.15 Addressing Modes

The DSP is fundamentally a load/store architecture. The only way to write to memory is through store operations. However, data from memory can be accessed through normal load operations or streaming engine operation. The DSP can read data directly from the streaming engine and use it directly for computation. There is no specific address mode for streaming engine operation.

The address modes described below applied for load and store instructions.

The C7x CPU supports scaled and nonscaled linear addressing with and without post increment.

Scaled linear mode shifts the src1/cst offset operand to the left by 3, 2, 1, or 0 for double-word, word, half-word, or byte element sizes, respectively, and then performs the addition with the src2 base address. In nonscaled mode, the offset is not shifted before adding to the base register. Scaled offset is denoted as [] and unscaled offset is denoted as ().

Postincrement addressing updates the base register by a specified amount after address calculation. For postincrement addressing, the value of the base register before the addition is the address to be accessed from memory. Post increment operation of the base address register is denoted as baseR++.

3.15.1 Base Address

The base address register is a 64-bit register. Because the virtual address size is 49-bits, the upper 15 bits of this register are not used by the address generation logic or uTLB lookup. However, those bits are checked to make sure they are all 0’s or all 1’s, if not, an address exception is generated.

In c7x, base address register can be A0-A13, D0-D15 scalar registers, collectively denoted as baseR. Program Counter (PC) or Event Context Switch Pointer (ECSP) control registers can also be used as base address register.

3.15.2 Offset Address

Offset can be either a 32-bit scalar register value or a constant. Offset value defaults to 0 when no bracketed register or constant is specified.

Register offset is a 32-bit signed value. This value is scaled by the data type, that is, element size (shift by 0,1,2, or 3 if the element size is Byte, Half-Word, Word or Double-Word respectively) and the result (up to 35-bits result after the shift) is sign-extended the rest of the way to 49 bits. This offset is then added to the base address register.

Load/store instructions can have A8-A15, D0-D15, SA0-SA3, or SA0++ through SA3++ as register offsets. ADDA/SUBA instructions can have A0-A15, D0-D15, SA0-SA3, SA0++ through SA3++ as register offsets. Collectively, valid register offset is denoted as offsetR32.

Constant offset can be a scaled 5-bit unsigned constant or a non-scaled signed 32-bit constant. Scaled mode simply shifts the offset operand to the left by 3, 2, 1, or 0 for double word, word, half word, or byte element sizes, respectively; and then performs an add to the base address register. In non-scaled mode, the constant is not shifted before adding to the base address register.

For the offset address generation options, the result of the calculation is the address to be accessed in memory. For the post-increment addressing, the value of the base register before the addition is the value used in the address calculation.

3.15.3 Addressing Modes

Load and store instruction opcodes contains a 3-bit field to specify the addressing mode to be used for address calculation. Following are the supported addressing modes.

3.15.3.1 Constant Offset Mode

- Non-scaled:
  - baseR(scst32)
  - baseR++(scst32)
  - baseR
  - baseR++
- Scaled:
  - baseR[ucst5]
  - baseR++[ucst5]

3.15.3.2 Register Offset Mode

- baseR[offsetR32]
- baseR++[offsetR32]

3.15.3.3 Program Counter as Base Address Mode (PC-Relative)

- PC[ucst5]
- PC(scst32)

PC-relative references are relative to the PC of the fetch packet containing the reference. This is true for the case of an execute packet spanning fetch packet as well, the PC reference is to the fetch packet which contains the instruction that has the PC relative addressing mode. Specifically, the address that is used for the base address when using the PC relative addressing mode is the address of the fetch packet containing the .D unit instruction that has the PC relative addressing mode.

For example, for this code sequence:

```
LDW .D1 *PC[0x30], A0
|| LDW .D2 *PC[0x34], A1
```

If the instruction on .D1 and the instruction on .D2 are in different fetch packets due to spanning, then they will use different values for the PC.

3.15.3.4 Event Context Stack Pointer as Base Address Mode (ECSP-Relative)

- ECSP[ucst5]
- ECSP(scst32)

Event Context Stack Pointer (ECSP) contains the address used to stack machine status when an event is detected. In this mode, the value contains in the ECSP control register is used as the base address for the corresponding load and store instructions.

3.15.4 Base Address Register Post Increment Operation

Constant and register offset mode include post increment operation on the base address register. Post increment of the base address is denoted as baseR++.

The encoding allow both .D1 and .D2 units to use either scalar register file or local D register file as bases. Therefore, instructions with post-increment addressing modes issued on either .D1 or .D2 can update either scalar register or local D registers.

- A0++[0] means writing A0 with its current value, that is, the write occurs, but the value of the register does not change.
- A0++[N] means increment A0 by N times element size in bytes. The element size is determined by the instruction.
- A0++ means increment A0 by the number of bytes accessed by the instruction. For example, VSTP2B, VSTP4B, VSTP16W with A0++ mode increments A0 by 2, 4, and 64, respectively. Only vector predicated store instructions have this address mode.

- The baseR update does not depend on the contents of the vector predicate register.
- Post increment mode does not apply when baseR is PC or ECSP.

3.15.5 Address Syntax

The address syntax described below applied for normal load and store instructions.

Table 3-85 describes the addressing generator options. The memory address is formed from base address register baseR, PC, or ECSP, and an optional offset that is either a 32-bit register (offsetR32), a 5-bit unsigned constant (ucst5), or a 32-bit signed constant (scst32).

Table 3-85. Indirect Address Generation for Load/Store

Addressing Type	No Modification of Base Address Register	Post-increment of Base Address Register
Register indirect, no offset	baseR[0]	baseR++[0]
Register indirect, offset by access size in bytes	baseR	baseR++
Register relative with 5-bit unsigned constant offset, scaled	baseR[ucst5]	baseR++[ucst5]
Register relative with 32-bit signed constant offset, unscaled	baseR(scst32)	baseR++(scst32)
Register relative with 32-bit register index, scaled	baseR[offsetR32]	baseR++[offsetR32]
Register relative with Streaming Address Generator as register index, scaled	baseR[SA]	baseR[SA++]
Program Counter Register relative with 5-bit unsigned constant offset, scaled	PC[ucst5]	Not Supported
Program Counter Register relative with 32-bit signed constant offset, unscaled	PC(scst32)	Not Supported
Event Context Stack Pointer Register relative with 5-bit unsigned constant offset, scaled	ECSP[ucst5]	Not Supported
Event Context Stack Pointer Register relative with 32-bit signed constant offset, unscaled	ECSP(scst32)	Not Supported

3.15.5.1 Load and Store Instructions

Table 3-86 shows the addressing mode encodings for load and store instructions.

Table 3-86. Addressing Mode Encodings for Load and Store Instructions

ADDR MODE	Load Instructions	Scalar Store Instructions	Vector Store Instructions
000	baseR[ucst5]	baseR[ucst5]	baseR
001	baseR(scst32)	baseR(scst32)	baseR(scst32)
010	baseR[offsetR32]	baseR[offsetR32]	baseR[offsetR32]
011	Reserved	Reserved	Reserved
100	baseR++[ucst5]	baseR++[ucst5]	baseR++
101	baseR++(scst32)	baseR++(scst32)	baseR++(scst32)
110	baseR++[offsetR32]	baseR++[offsetR32]	baseR++[offsetR32]
111	Reserved	Reserved	Reserved

When addressing mode is '000' or '100' for vector stores, the src2 field is used to encode the predication register for vector predicated stores, and the offset value is the number of bytes specified by the instruction opcode.

Addressing mode notes:

- reg32 value is a signed integer, which is sign-extended before being added to base.
- ecst32 is the 32-bit constant formed by appending the 27-bits from the constant extension unit with the 5-bits from the SRC2 field.
- ecst32 is signed, and unscaled.
- address mode = 000 or 100 for VSTx/DVSTx instructions escape to the predicated store space, and SRC2 specifies which predicate register to read from.

- base=01111 is reserved for PC-register. Only mode *PC(scst32) and *PC[ucst5] are supported.
- base=01110 is reserved for ECSP register -- Only mode *ECSP(scst32) and *ECSP[ucst5] are supported.
- The baseR is a 64-bit register. The upper 15 bits of this register are not used by address generation or uTLB lookup. They are checked to make sure they are all 0's or all 1's.
- The offsetR is a 32-bit signed value. This value is scaled by the data type (shift by 0,1,2,3) and that result (up to 35-bits result after the shift) is sign-extended the rest of the way to 49 bits. This offset is then added to the base register.
- For vector loads, the destination is zero-filled if the number of bytes specified by the instructions is smaller than the width of the destination register, that is, VLD4W would zero-filled up to 512-bit vector size, while VLD2W destination is 64bits only so no zero-filled.
- ACSWAPH and ACSWAPW only support mode "000".

3.15.5.1.1 Base Address Register Encodings

The base registers can come from the scalar general registers A0-A13 or local .D units registers D0-D15.

The base register field is encoded as follows:

- 00000 - 01101: A0 through A13
- 01110: ECSP
- 01111: PC
- 10000 - 11111: D0 through D15

3.15.5.1.2 Offset Address Register Encodings

The offset register field in register relative modes (address modes "010" or "110" in Table 3-86) is encoded as follows:

- 00000 – 00011: SA0, SA1, SA2, SA3
- 00100 – 00111: Reserved for SA4 through SA7
- 01000 – 01111: A8 through A15
- 10000 – 11111: D0 through D15

Register relative mode 010 provides scaled offset addressing for all valid src2 encodings— for example, *baseR[A8], *baseR[D10], *baseR[SA0], *baseR[SA1].

Register relative mode 110 with A8 – A15 or D0 – D15 provides scaled post-increment addressing— for example, *baseR++[A11], *baseR++[D3].

Register offset mode 110 with SA provides scaled offset addressing and increments the SA register (not the base)— for exmaple, *baseR[SA0++], *baseR[SA1++].

Note

If an unsupported base or offset combination is detected, the CPU raises an illegal opcode exception. For example, the CPU raises an illegal opcode exception when *A15[A0] or *A14[A1] and so forth, because A15 and A14 are reserved for PC and ECSP control registers relative addressing modes which only support ucst5 or scst32 as offsets.

3.15.5.2 ADDA/SUBA Instructions

ADDA and SUBA instructions perform linear scaled address calculations by adding the base address in src1 with the shifted offset value in src2. ADDA and SUBA instructions only support register offset modes.

Src1 contains the base register and is encoded as follows:

- 000000 - 001101: A0 throughA13
- 001110: ECSP
- 001111: PC
- 010000 - 011111: B0 through B15
- 100000 - 101111: D0 through D15
- 110000 - 111111: Reserved

Src2 contains the offset register and is encoded as follows:

- 000000 - 001111: A0 through A15

- 010000 - 011111: Reserved
- 100000 - 101111: D0 through D15
- 110000 - 110011: SA0 through SA3
- 110100 - 110111: Reserved
- 111000 - 111011: SA0++ through SA3++
- 111100 - 111111: Reserved

3.15.5.3 ADDKPC Instruction

A constant is added to the address of the first instruction of the fetch packet that contains the ADDKPC instruction (PC). The result is placed in the specified destination register.

The supported constant is either an unscaled 32-bit signed value or a scaled 5-bit unsigned value, matching with the PC-Relative addressing mode.

If the E-bit of the source 2 (i.e opcode bit 25th) is one, the ADDKPC uses the 32-bit signed value, which is formed by concatenating 27-bit of an extended constant and 5 remaining bit of src2 field. The signed 32-bit value is then added to the address of the first instruction of the fetch packet that contains the ADDKPC instruction.

If the E-bit of the source 2 is zero, the ADDKPC uses the 5 remaining bits of src2 field as 5-bit unsigned value.The 5-bit unsigned value is then shifted 2 bits to the left, then added to the address of the first instruction of the fetch packet that contains the ADDKPC instruction.

3.15.5.4 Programmer Notes

The address register used for loading or storing a vector is considered to be a byte offset address. No attempt is made by the DSP to force any particular alignment of the memory addresses. For example, if issuing a VLDW *A0,VB0 instruction, the vector starting at the address contained in A0 is loaded, without regard to whether the address in A0 is aligned to a 512-bit boundary or to a word boundary.

However, TI recommends that most data structures are aligned to 64-bit boundaries, so that vector load/store operations' addresses are 64-bit aligned. The reason for this is that the memory system has 64-bit banks of memory, thus maintaining a 64-bit alignment reduces bank conflicts, because this minimizes the amount of memory banks that must be accessed.

3.15.6 Alignment Restrictions

There are no alignment requirements by load and store instructions. If the non-aligned load/store operation crosses a cache line, the CPU inserts an extra stall cycle to handle the non-aligned data.

3.16 Pipeline Operation Modes

The C7x pipeline can operate in two modes: protected mode and unprotected mode. In both modes, branches have zero delay slots. Pipeline protection is enabled or disabled through setting bit 8th in the TSR (TSR.PROTP). Two new instructions are introduced to manipulate the modes: PROT and UNPROT. When TSR.PROTP is not set, executing PROT sets TSR.PROTP. When TSR.PROTP is set, executing UNPROT clears TSR.PROTP. If TSR.PROTP is already set, executing PROT has no visible effect, and likewise, if TSR.PROTP is clear, executing UNPROT has no effect.

3.16.1 Unprotected (Exposed) Pipeline Operation Mode

Unprotected or exposed pipeline mode is the traditional VLIW operation mode. Unprotected pipeline mode requires the programmer or compiler to know the latencies of the instructions, and to insert NO-OPS or other instructions between dependent instructions to ensure correctness. The CPU executes in unprotected mode when bit TSR.PROTP=0.

3.16.2 Protected (Unexposed) Pipeline Operation Mode

Protected or unexposed pipeline operation mode conforms to the sequential operation model where dependent instructions are ensured to be correct, regardless of how many cycles it takes to complete the instructions. The CPU executes in protected mode when bit TSR.PROT=1.

Executing the PROT instruction causes the destination registers of the instructions in the following execute packet to be protected by the pipeline control logic. This means that for an instruction which takes multiple cycles to complete, that if a subsequent instruction attempts to read the destination of the first instruction within the

delay slots of that first instruction, then the CPU pipeline is stalled until the instruction, which writes that the register has completed.

For example:

```
PROT
MPY32 A0, A1, A2 ; Multiply - this instruction takes 4 cycles to execute
ADD A2, A8, A8 ; Accumulate
```

In the above example, the pipeline control logic inserts 3 NOP cycles to allow the MPY32 instruction to complete before the ADD instruction executes. Contrast this behavior to the unprotected pipeline case.

```
UNPROT
MPY32 A0, A1, A2 ; Multiply - this instruction takes 4 cycles to execute
ADD A2, A8, A8 ; Accumulate - Whoops!
```

In the above case, the ADD instruction uses the value present in the A2 register rather than the result of the multiply, because the multiply instruction takes multiple cycles to complete. To make this code behave correctly in the unprotected mode, the programmer or compiler must manually specify the NOP 3.

```
UNPROT
MPY32 A0, A1, A2 ; Multiply - this instruction takes 4 cycles to execute
NOP 3
ADD A2, A8, A8 ; Accumulate - correct this time
```

The C7x is an in-order machine all the way through to the first execution stage. Out-of-order completion is allowed, and is what is protected in the PROT=1 mode. The consequence of this is that when a pipeline dependency is found, all of the functional units and entire execution pipeline from instruction fetch through the E1 phase are stalled. In other words, all the functional units are stalled at the same time, and no units are allowed to advance until the pipeline conflict has resolved. When issuing single cycle instructions, no pipeline protection is required, because program order is ensured to be followed due to the one-stall-all-stall behavior above.

In practice, for typical usage, the pipeline is left in PROTECTED mode while executing straight-line code, and enters UNPROTECTED mode only for software pipelined loops, to take advantage of multiple-assignment code.

Including the protected pipeline model in the C7x allows the compiler to not always have to assume the worst case operand usage when performing certain tasks, such as popping values off the stack before returning. In the unprotected pipeline model, the compiler must ensure that load instructions have completed before the first instruction of the calling function can execute, just in case that very next instruction reads one of the registers which is being restored from the stack. In addition, the protected pipeline model saves code space.

Issuing a PROT and UNPROT instructions within 5 cycles of one-another results in an exception.

Manually manipulating TSR.PROTP using MVC is not allowed, that is, writes to the PROTP bit through the MVC instruction are ignored. Executing PROT, PROTCLR, or UNPROT instructions are required to affect a change in TSR.PROTP.

3.16.3 Predication versus Branching

There is a slight behavioral difference in score boarding with respect to using branches versus using predication for program flow decision making. In the case of predication, the determination of whether to execute the instruction is made when the instruction is in the E1 phase. This means that the predication condition could have been calculated in the immediately previous clock cycle, which does not give enough time to make the decision to stall the pipeline or not. Put another way, the following code always introduces a 3 cycle pipeline stall, regardless of the value of A0.

```
MPY32 A8, A9, A10
[A0] MV A10, A11
```

Because the pipeline protection must determine the stall before the MV instruction enters the E1 phase, the value of A0 is not taken into consideration, and a 3 cycle pipeline stall is always introduced.

However, for contrast, consider the following case.

```
MPY32 A8, A9, A10
[A0] B around_label
MV A10, A11
around_label:
<Continuation>
```

In this case, if the branch is taken, then the pipeline protection mechanism only kicks in if the branch was not taken, and it was predicted to be not taken (for example, a correctly predicted not-taken branch). If the branch is predicted taken, then the MV instruction is not in the pipeline, and the hazard is not seen. If the branch is predicted taken and it turns out that it was not taken, then again, no hazard was seen, and by the time the branch miss has completed, A10 will have the correct value in it, therefore no additional stall cycles are incurred due to hazard detection.

If the branch was incorrectly predicted not-taken, then the hazarding logic does kick in for the execution pipeline at the same time that the branch mis-prediction gets detected. However, the branch mis-prediction correction is not affected by the scoreboarding logic, and the amount of time spent is merely the branch mis-predict penalty: the incorrect hazarding does not introduce any additional stalls over the mis-predict penalty.

3.16.4 Load Data Scoreboarding

In addition to maintaining a scoreboard for providing a protected pipeline, the C7x CPU also maintains a scoreboard to keep track of memory Load operations which have not completed due to unexpected events in the memory system (for example, a cache miss).The DSP keeps track of up to 8 outstanding loads which have not completed before stalling it's pipeline due to L1D read data not being returned. The CPU has three different conditions which can cause a stall due to load instructions and/or L1D responses.

- In protected pipeline mode, if the destination of a load instruction is read as an operand to a subsequent instruction before the 4 cycles it takes for the L1D data cache to return data, then the pipeline is stalled until the L1D data cache returns data.
- In either protected pipeline mode or unprotected mode, if the destination of a load instruction is read as an operand to an instruction, and the L1D data cache indicates that it does not have the data by the 4 cycle L1D cache latency, then the pipeline stalls.
- If the DSP has sent 8 load instructions and data has not been returned for any of them, then the DSP stalls when it encounters the next Load instruction, provided it didn't stall first due to reasons 1 or 2.
- If the L1D data cache indicates that it cannot accept any new load or store instructions, and the DSP encounters a load or store instruction, the DSP stalls until the L1D cache resolves its internal stall.

Store instructions are "fire and forget" from the DSP's perspective and are not tracked. Due to the load scoreboarding behavior, the DSP can accept data returns from the memory system in any order. The DSP sends a transaction ID out with every outstanding load instruction, and L1D returns the corresponding transaction ID back with the load data. Furthermore, the load scoreboarding behavior allows the compiler to hoist load instructions further up in the schedule and to hide L1D cache miss penalties when the compiler knows it has enough other work to do. In addition, this also allows the L1D data cache to support hit-under-miss behavior, again leading to possible performance improvements in code which has a mixture of loads which are likely to miss (such as large database item lookups), and loads that are likely to hit (such as stack accesses).

3.16.5 WAW Hazard Handling

Due to the scoreboarding rules, you can run into the situation in which you have issued a load to a register, say A0, and this load misses L1D and is stuck in the memory system. It is possible for a branch to be executed, and to then encounter another instruction which writes to A0 before A0 is read. In the current implementation, the pipe is stalled until the load data is returned and then the second instruction is allowed to proceed.

In the future, TI may allow the write to A0 to cancel the score boarding on the load to A0. The DSP will still keep track of the load and it still counts against the limit of 8(or 16) outstanding load instructions. This option is open for investigation to determine if it is worthwhile.

For all the loads to be completed before continuing program execution, one must execute an MFENCE operation. Executing an MFENCE ensures that all outstanding load instructions have completed and updated the contents of the DSP's registers, and ensuring that all outstanding store instructions have had their data committed at their respective end-points.

3.16.6 Stall Behaviors - Architecture Stalls versus Implementation Stalls

The scoreboard keeps track of whether a register contains the value that it is supposed to contain or not. If the scoreboard indicates that a register does not yet have the value it is supposed to have, then any reference (read or write) to that register then stalls the pipeline, inserting stalls that should not be “visible” to program execution. Therefore, distinction between “architectural” cycle counts and “implementation” cycle counts is needed. The architectural cycle count is “how many cycles it should take if I ignore physics and everything happens when I said it happened.” The implementation cycle count is the actual number of cycles taken when all the pipeline interlocks, cache misses, branch mis-predictions, and so forth are resolved.

The way to look at the PROT vs UNPROT mode is the effects when the result of that instruction should be available for use.

The basic sequence:

```
arch
cycle                    Instruction
0                    PROT
1                    LDW *A0, A8            <-A8 should be available in cycle 2 (architectural
definition).
2                    ADD  A8, A9, A10     <-Stall occurs because LDW -> use is 5 cycles.
```

Expand the above:

```
arch            impl
cycle            cycle        Instruction
 0            0         PROT
 1            1         LDW *A0, A8        <-A8 should be available in cycle 2 (architectural
definition).
 -            2         (wait for load)
 -            3         (wait for load)
 -            4         (wait for load)
 -            5         (wait for load)
 2            6         ADD  A8, A9, A10
```

If a memory system can be implemented, which returns loads in a single clock cycle, then those “implement cycles 2 through 5" would go away.

The following is a more complicated example where protected loads mixed with unprotected loads:

```
arch
cycle                    Instruction
  0                    PROT
  1                    LDW *A0, A8                <-    A8 should be available in cycle 2
(architectural definition).
  2                    UNPROT
  3                    LDW *A1, A9                    <- Unprotected load - result is scheduled to
land in architectural cycle # 8
  4                    ADD  A8, A9, A10                <- Stall occurs because of the protected LDW
-> use is 5 cycles.  So, 2 stall cycles
  5                    ADD  A9, 0, A11
  6                    ADD  A9, 0, A12
  7                    ADD  A9, 0, A13
  8                    ADD  A9, 0, A14                    <- First time that reading A9 should get
the value from the LDW *A1
```

The expanded view:

```
arch                impl
cycle                cycle            Instruction
  0                0            PROT
  1                1            LDW *A0, A8                <- A8 should be available in cycle 2
(architectural definition).
  2                2            UNPROT
  3                3            LDW *A1, A9                <- Unprotected load - result is scheduled
to land in architectural cycle # 8
  -                4            (wait for load)
  -                5            (wait for load)                <- Waiting for A8 to get updated by
the protected load
```

```
  4                6            ADD  A8, A9, A10
  5                7            ADD  A9, 0, A11
  6                8            ADD  A9, 0, A12                <- A9 is NOT updated here (4
Implementation cycles after the load)
  7                9            ADD  A9, 0, A13
  8                10            ADD  A9, 0, A14                <- First time that reading A9 should
get the value from the LDW *A1
```

With a hypothetical implementation that can return the first “protected” LDW in a single clock, the implementation cycles 4 and 5 would go away. But, as defined by the architecture, no implementation can allow the 2nd LDW to return until arch cycle 8.

Also, a similar situation occurs when a load misses L1D: even in UNPROT mode, the pipeline still advances until a following instruction tries to use the register.

3.16.7 Crossing Between PROT and UNPROT

The following examples highlight the differences involving multi-cycle instruction results when crossing between the two execution modes.

From protected to unprotected mode:

```
MPYWW   M0,X0,L1
      MPYWW   M1,X0,L1
      MPYWW   M2,X0,L1
      UNPROT
      ADDW    L0,L1,X0
```

In this case, the result of the MPYWW M2,X0,L1 is protected because it entered execution while in protected mode. The ADDW instruction gets L1 source from the result of MPYWW M2,X0,L1 instruction.

From unprotected to protected:

```
MPYWW   M0,X0,L1
      MPYWW   M1,X0,L1
      MPYWW   M2,X0,L1
      PROT
      ADDW    L0,L1,X0
```

In this case, the result of MPYWW M2,X0, L1 is unprotected because it entered execution while in unprotected mode. The ADDW instruction gets the L1 source from the result of MPYWW M0,X0,L1 instruction.

To cleanly exit the loop before entering the protected mode, either of the following code section is needed:

```
      MPYWW   M0,X0,L1
      MPYWW   M1,X0,L1
      MPYWW   M2,X0,L1
      NOP     2
      PROT
      ADDW    L0,L1,X0
```

or

```
MPYWW   M0,X0,L1
      MPYWW   M1,X0,L1
      MPYWW   M2,X0,L1
      PROT
      NOP     2
      ADDW    L0,L1,X0
```

3.16.8 Crossing From UNPROT to PROT Mode With PROTCLR Instruction

Similar to PROT instruction, the PROTCLR instruction changes the operation mode to protected mode. The PROTCLR instruction has the following properties:

- PROTCLR instruction annuls all pending writes which has life-time > 0 before PROT takes affect. (Life-time: remaining arch. cycles to be executed).
- PROTCLR turns instructions-with-latency which is parallel with it into single cycle instructions.

- Both PROT and PROTCLR instructions protect “on-reads” instruction packets after it, meaning the data consumed by instructions following a PROT or PROTCLR instruction are ensured to be the data produced last.

The following example explains what would happen with a sequence of load data across PROTCLR boundary:

```
VLW *D++, A0   <- complete both data and address updates
VLW *D++, A1   <- complete both data and address updates
VLW *D++, A2   <- complete both data and address updates
VLW *D++, A3   <- discard data, complete address updates
VLW *D++, A4   <- discard data, complete address updates
VLW *D++, A5   <- discard data, complete address updates
VLW *D++, A6   <- discard data, complete address updates
VLW *D++, A7   <- discard data, complete address updates
PROTCLR || LDW *D++ , A7 <- complete both data and address updates since this LDW is turned into
single-cycle
ADD  A7,     <- A7 is protected "on-reads" here
```

3.16.9 Exception Detection

Due to nature of the protected mode, certain instruction sequences which do not cause exception in the unprotected mode now can cause an exception. For instance, a multi-cycle instruction which is issued in parallel with another single cycle instruction can cause multiple write exception if they both trying to write into the same register:

```
PROT
MPY32 A0, A1, A2
|| ADD A3, A4, A2
```

3.17 Nested Loop Predication

3.17.1 Overview

In many typical DSP applications, loops comprise a majority of the number of cycles or MIPS. Because of this, performance of loops can greatly affect the performance of the entire application. Many of these loops are nested loops or multi-dimensional loops with both an inner and an outer loop. Some common examples are FIR and IIR filters, FFT, and vision code. To optimize these nested loops, it is necessary to consider not only the inner loop performance but also the outer loop performance, especially when the inner loop count is small for execution of each outer loop. Two standard techniques used to optimize nested loops on the highly parallel C7x architecture is software pipelining and loop collapsing.

Software pipelining involves initiating new iterations of the loop before previous iterations have completed to obtain high throughput. This implies there are some cycles (loop prolog) to begin executing, or pipe up, of each inner loop and some more cycles to pipe down the loop (loop epilog). These cycles are incurred for each outer loop execution so they can affect performance, especially when the inner loop count is small. The more deeply pipelined the DSP is, the more cycles will be required for the prolog and epilog.

Loop collapsing allows for pipelining around outer loops as well as inner loops, which can lead to better efficiency. However, conventional loop flattening algorithms can introduce unnecessary long-latency feedback paths that limit the effectiveness of pipelining. One approach to nested loops is to collapse multiple nested loops into a single loop. Code that was in the outer loop is predicated by the loop counters so as to execute only on iterations that correspond to back edges of the original outer loop.

3.17.2 Nested Loop Optimization in Compiler Scheme

3.17.2.1 Nested Loop Code

In describing the nested loop mechanism, the example code uses a general conceptual model of a nested loop with one inner loop and one outer loop.

```
for (j = 0; j<ICNT_outer; ++j)
{
Preloop;
for (i = 0; i <ICNT_inner; ++i)
{
Body_inner;
```

```
}
Postloop;
}
```

The preloop and postloop statements represent code that is executed before and after the inner loop. Preloop code typically consists of initialization of accumulators, and/or loads of coefficients or filter taps that are invariant within the loop. Postloop code typically consists of consolidating partial sums and/or storing results. In most cases, there is little if any preloop or postloop code, and the overhead of the outer loop comes mainly from the loop control itself: the counter increment, comparison, and branching. The C7x NLC architecture assumes loops that are perfectly nested and the numbers of iterations are loop-invariant values defined outside the loop. If a loop uses a counter that starts at a value other than 0 or increments by a stride other than 1, the compiler is expected to scale and offset the counter accordingly. The C7x CPU also assumes that the counters are used only for counting and are not otherwise referenced. Finally, the C7x CPU assumes that the body of the loop executes at least once; that is, none of the trip counts should be initialized as 0 to start with.

3.17.2.2 Software Pipelining

Software pipelining is a family of compiler methods for scheduling and resource allocation that exploit the fact that while the number of parallel operations available in a single iteration of a loop is often limited, operations from later iterations can be executed before earlier iterations have completed as long as all dependencies are respected. The result of applying pipelining to a loop is a steady state schedule and a prologue and epilogue. The basic terminology and abbreviation of pipelining is listed in Table 3-87. TI uses the following abbreviations (which are more completely defined in succeeding paragraphs) throughout the text.

Table 3-87. Software Pipelining Nomenclature

Abbreviation	Meaning
Iteration Interval (II)	Initiation Interval is the interval (in instruction cycles) between successive iterations of the loop
Stage	A stage is the code executed in one iteration interval
Dynamic length (dynlen)	Dynamic length is the length (in instruction cycles) of a single iteration of the loop. It is therefore equal to the number of stages times to the iteration interval.
Kernel	The kernel is the period when the loop is executing in a steady state with the maximum number of loop iterations executing simultaneously
Prolog	The prolog is the period before the loop reaches the kernel in which the loop is winding up. The length of the prolog will by the dynamic length minus the iteration interval (dynlen - ii).
Epilog	The epilog is the period after the loop leaves the kernel in which the loop is winding down. The length of the prolog will by the dynamic length minus the iteration interval (dynlen - ii)
RecII	Recurrence Initiation Interval

The nested loop controller (NLC) is a hardware mechanism to facilitate low-overhead loop collapsing by off-loading the computations associated with nested loop counters and predicates.

3.17.3 Nested Loop Counter Logic

The compiler handles perfectly nested loops efficiently, but if there is outer loop code in between the inner loop and outer loop, the efficiency reduced due to the predication generation and management only by the software. Hardware nested loop counter (NLC) provides a hardware mechanism to implement the counters and predicates discussed above to reduce the overhead of doing it programmatically through explicit instructions. Two major functions hardware help the most are:

- Decrement and compare for each predicate
- Lifetime management of the predicates

The hardware NLC features includes:

- The nested loop counters as control registers
- NLC counter logic performing the increments, comparisons, and resets
- Based on the counter comparison, generates predication for the preloop, the postloop, and the collapsed loop branch condition to control the loop.
- Hardware also keeps the predication lifetime history for software to manage these predicates.

3.17.3.1 Nested Loop Predication Instructions

The NLC logic consists of four operations.

3.17.3.1.1 NLCINIT

NLCINIT instruction initializes both ILCNT and OLCNT register and provides a 6-bit Epilog Skew flag parameter (EPISKEW), which indicates number of additional taken branches. An instruction example is shown below:

```
NLCINIT .S1 A0, A1, #Flags;
// A0 is the 64-bit initial inner loop count
// A1 is the 64-bit initial outer loop count
// Flag parameter is 6-bit EPISKEW value
```

Executing this instruction causes these effects:

- Writes the EPISKEW parameter, this number has a range of -1 to 30.
- Initializes the ILCNT and OLCNT registers.
- Initializes the inner loop predicate flag registers to 16'b0000_0000_0000_0001.
- Initializes the outer loop predicate flag registers to 16'b0000_0000_0000_0000.

3.17.3.1.2 TICK

TICK instruction advances the counters and updates the predicate flag registers. The counters are advanced 'odometer style'; that is, when inner loop counter reaches its trip count, it resets to zero and the outer counter is increment by one.

C7x CPU expects TICK to be the first instruction in the software pipelined loop.

3.17.3.1.3 GETP (Get Predicate Instruction)

GETP instruction provides an NLC query that retrieves the predicate of each level. In a given iteration, retrieving the predicate prior to the TICK operation produces the preloop predicate (the predication register value for the preloop instructions). Retrieve the predicate following the TICK produces the postloop predicate (the predicate register value for the postloop instructions).

```
bool NLC_getp(int level, int iteration_offset = 0);
```

A GETP issued after initialization but prior to the first TICK returns 'true'. Such a GETP corresponds to a preloop predicate, so this behavior enables the preloops to execute on the first iteration.

The level parameter specifies the loop level. The iteration offset is an index into the predicate FIFO, to facilitate overlapping lifetimes in a software pipeline. Therefore, the GETP instruction provides a constant operand (no register version):

```
GETP [fifo offset const];
```

Unlike MVC instruction, GETP instruction can be executed on .S1, .L1 and .M1 units instead of only on .S1.

Offset value specifies how many TICKs to go back to retrieve the predicate.

3.17.3.1.4 BNL (Branch Nested Loop Instruction)

Executing BNL instruction causes the CPU to take a branch after EPISKEW cycles run out, after OLCNT and ILCNT = 1. Every TICK reduces inter loop count value by 1.

The total number of ticks is calculated by this equation:

```
Total number of TICKs = OLCNT value * ILCNT value + EPISKEW
```

Hardware compares the current counts of ticks to the total number of TICKS calculated in the equation above before taking the branch. Branch taking conditions are straight forward:

- Branch is true, if the ticks counted < total number of ticks
- Branch is false, if the ticks counted >= total number of ticks

The definition of counters enables the user to load the counters with N, instead of (N-1).

BNL instruction is a conditional instruction which can be predicated. This is expected to be used for early exit condition to break into the outer loop or all levels of loops. Usually, BNL is the last instruction in the loop.

3.17.3.2 Nested Loop Count Registers

3.17.3.2.1 Inner Loop Count Register (ILCNT)

This register is 64-bit wide and only visible through debugger. MVC instruction cannot access it.

When an event is taken, ILCNT is automatically saved by hardware when the event handler is entered. When the event handler returns, the ILCNT content is restored by hardware from the corresponding context saving region.

The ILCNT register is initialized by executing an NLCINIT instruction.

3.17.3.2.2 Outer Loop Count Register: OLCNT

This register is 64-bit wide and only visible through debugger. MVC instruction cannot access it.

When an event is taken, OLCNT is automatically saved by hardware when the event handler is entered. When the event handler returns, the OLCNT content is restored by hardware from the corresponding context saving region.

The OLCNT register is initialized by executing NLCINIT instruction.

3.17.3.2.3 EPISKEW Register: EPISKEW

EPISKEW register keeps the 6-bit EPISKEW for BNL predication purposes.

This register is only visible through debugger. MVC instruction cannot access it.

3.17.3.2.4 ILCNTFLG[i] (Inner Loopf Counter Flag Register)

These are single bit indexed registers which keep the one bit predication value per register for inner loop predication. The index of the register is the offset value. There are 16 of these registers. The index of the ILCFLAG is provided by the offset of GETP instruction.

3.17.3.2.5 OLCNTFLG[i] (Outer Loop Counter Flag Register)

These are single bit indexed registers which keep the one bit predication value per register for outer loop predication. The index of the register is the offset value. There are 16 of these registers. The index of the OLCFLAG is provided by the offset of GETP instruction.

3.17.4 Privilege and Security Model

3.17.4.1 Privilege and Security Model Summary

The C7x provides native support of multiple privilege levels. The C7x privilege model enables operating systems to isolate application-level tasks from each other and itself. It also provides separate secure and non-secure partitions, allowing a secure execution environment to run alongside a non-secure environment.

The C7x provides additional facilities to support virtualization by managing guest operating systems running under a host operating system or a thin layer of root supervisor runtime.

3.17.4.1.1 C7x Privilege Model Overview

The C7x architecture defines a set of privilege levels (exception levels):

- Secure root supervisor
- Secure root user
- Non-secure root supervisor
- Non-secure root user
- Non-secure guest supervisor
- Non-secure guest user

The six privilege levels embody three aspects of privilege: secure level, supervisory level, and root/guest level. These three concepts are orthogonal. Table 3-88 illustrates the relationship and purpose of each privilege level. C7x does not support virtualized secure privilege level. This results in 6 distinct privilege levels.

Table 3-88. Privilege Levels Supported by C7X Virtualization Support

	Secure	Non-Secure	
		Root (Supervisor)	Guest
Supervisor	Secure kernel / Trusted execution environment	Host operating system	Guest operating system
User	Secure services	Host applications	Guest applications

The following sections describe the orthogonal privilege axes.

3.17.4.1.1.1 Supervisor versus User Axis

This privilege axis defines the separation between operating system and application. The operating system, such as an RTOS or another operating environment, manages the resources of the machine, scheduling tasks, processing interrupt events and so forth. Code that directly manages the machine resources operates at the root supervisor privilege level. Code that manages the virtual machine resources operates at the guest supervisor privilege level.

Individual tasks rely on the operating system to manage machine or virtual machine resources. Each task should only be able to access the resources granted to it by the operating system. To enforce this separation, tasks operate at the more restricted user privilege level.

User code interacts with supervisor code through a narrow gateway: system calls through the instruction. Otherwise, interrupts and exceptions provide the only other mechanisms to leave User mode and enter supervisor mode. The interrupt and exception sections describe these transitions more fully.

3.17.4.1.1.2 Secure versus Non-secure Axis

Some device applications require certain system level guarantees and capabilities that the traditional supervisor versus user privilege axis cannot provide. These include, but are not limited to:

- Secure boot. Secure boot verifies that a given boot image was provided by the original equipment manufacturer and has not been tampered with, usually by verifying the image against encryption key signatures.
- Trusted identity. A given product may be required to identify itself uniquely, and prove that identity.
- Rights management. Data or content may be bound to a specific device.
- Sensitive secrets. The device may need to store sensitive information, such as session keys or credit card numbers.
- Proprietary algorithms. Some customers may develop algorithms that they wish to protect against reverse engineering.

In each of these cases, a normal supervisor cannot provide the necessary protection.

The secure versus non-secure privilege axis provides an additional, orthogonal mechanism to separate security-sensitive code from the rest of the operating environment.

System integrators can build secure systems on top of this privilege access by providing at least the following additional system resources:

- A secure boot ROM to establish a root of trust.
- Secure firewalls to isolate regions of memory as accessible to secure privilege levels only.
- Some form of secret key accessible only from secure privilege levels to validate code before handing control from secure boot ROM to booted code.

C7x security support works by restricting access both to sensitive secrets, and to hardware features that could subvert security, such as certain cache configuration and maintenance operations, power-down modes, and so forth. Not all devices or systems require security support. The C7x provides mechanisms to open these resources to non-secure code on devices that do not require secure functionality.

C7x security may block access to resources non-secure code wishes to manage. Recoverable exceptions allow the secure operating environment to trap and emulate access to these restricted resources. The secure privilege levels do not virtualize non-secure privilege levels. Secure code intercepts non-secure access to some resources solely to close security holes and enforce software-defined policies for those resources.

3.17.4.1.1.3 Root versus Guest Supervisor

C7x virtualization support extends the privilege model by adding a third privilege axis: root versus guest. The root supervisor privilege level is a strict superset of supervisor as found on devices that lack virtualization support. Root supervisor gains access to additional resources intended to control the behavior of a guest operating system:

- Stage 2 virtual address translation control, if supported by CMMU
- Mechanisms to control delivery of interrupts and exceptions to host and guest
- Trap-and-emulate support for registers reserved for root supervisor
- Low overhead paravirtualization support

These additional resources exist solely to support virtualization. Devices that do not support virtualization lack these resources.

3.17.4.1.1.4 Root versus Guest User

Root user and guest user look nearly identical. The only difference is that root supervisor directly manages root user tasks, while guest supervisor directly manages guest user tasks. As a result, the CPU delivers system calls and exceptions generated by root user to root supervisor, and delivers system calls and exceptions generated by guest user to guest supervisor.

3.18 Instruction Set Architecture

3.18.1 Definition

- Scalar instructions: instructions which use 32-bit or 64-bit resources only (operands or destinations). If a scalar instruction produces a result less than 64 bits, its result will be zero extended to fill up 64-bit scalar value. The only exception is the scalar loads, for which the results would be sign-extended to 64-bit value.
- Vector instructions: instructions which use 512-bit resources (operands or destinations).

The C7x ISA is developed with the following goals:

- Increase orthogonality to ease programming, that is, the same instruction should have all SIMD variations.
- Increase general computing power
- Increase general data bandwidths and data usage flexibility
- Specialized instructions to accelerate important algorithms:
  - Motion search/stereo vision
    - Vector sum-of-absolute-differences with sliding windows
    - Vector complex dot products, conjugate complex dot products
  - LTE/WiMax/TD-SCDMA PHY
    - Vector matrix multiplies
    - Vector min/max instructions with decision bits
    - Horizontal min/max
- Sort/Search
  - Horizontal min/max instructions
  - Vector min/max instructions with decision bits
- Viterbi
  - Circular maximum instructions
- W-CDMA/FFT/IFFT
  - Correlation instructions with sliding windows
- Floating point
  - Conversion instructions between half-precision floating point and other data formats

3.18.2 Instruction Syntax and Naming Conventions

The C7x instruction mnemonics follow the basic structure:

Vector	Core Function	Input Precision	Output Precision

All vector and double vector instructions have the prefix of “V” to denote that this is a vector operation. The scalar instructions do not have the prefix “V”.

Depending on the precision of input and output elements, most instructions have suffixes containing the input precision and output precision. If both input and output precision of the elements are the same, then only one precision suffix character is listed if there is no potential of confusion.

- B: 8-bit precision
- H: 16-bit precision
- W: 32-bit precision
- D: 64-bit precision

Table 3-89. Examples of Instruction Names

Instruction names	Vector	Core Function	Input Precision	Output Precision
ADDW	No	Addition	32-bit	32-bit
ADDD	No	Addition	64-bit	64-bit
MPYWW	No	Multiply	32-bit	32-bit
MPYWD	No	Multiply	32-bit	64-bit
VMPYWD	Yes	Multiply	32-bit	64-bit

3.18.3 Instruction Operation and Execution Notation

Table 3-90 explains the symbols used in the instruction descriptions.

Table 3-90. Instruction Operation and Execution Notations

Symbol	Meaning
abs(x)	Absolute value of x
and	Bitwise AND
-a	Perform 2s-complement subtraction using the addressing mode defined by the AMR
+a	Perform 2s-complement addition using the addressing mode defined by the AMR
bi	Select bit i of source/destination b
bit_count	Count the number of bits that are 1 in a specified byte
bit_reverse	Reverse the order of bits in a 32-bit register
byte0	8-bit value in the least-significant byte position in 32-bit register (bits 0-7)
byte1	8-bit value in the next to least-significant byte position in 32-bit register (bits 8-15)
byte2	8-bit value in the next to most-significant byte position in 32-bit register (bits 16-23)
byte3	8-bit value in the most-significant byte position in 32-bit register (bits 24-31)
bv2	Bit vector of two flags for s2 or u2 data type
bv4	Bit vector of four flags for s4 or u4 data type
by..z	Selection of bits y through z of bit string b
cond	Check for either creg equal to 0 or creg not equal to 0
creg	3-bit field specifying a conditional register, see Section 3.6 on page 3-14
cstn	n-bit constant field (for example, cst5)
dint	64-bit integer value (two registers)
dst_e	lsb32 of 64-bit dst (placed in even-numbered register of a 64-bit register pair)
dst_h	msb8 of 40-bit dst (placed in odd-numbered register of 64-bit register pair)
dst_l	lsb32 of 40-bit dst (placed in even-numbered register of a 64-bit register pair)
dst_o	msb32 of 64-bit dst (placed in odd-numbered register of 64-bit register pair)
dst_0 or src_0	32-bit value in the least-significant position in 128-bit quad register
dst_1 or src_1	32-bit value in the next to least-significant 32-bit word position in 128-bit quad register
dst_2 or src_2	32-bit value in the next to most-significant 32-bit word position in 128-bit quad register
dst_3 or src_3	32-bit value in the least-significant position in 128-bit quad register
dwdst	64-bit register pair result
dwop	64-bit register pair operand
dws4	Four packed signed 16-bit integers in a 64-bit register pair
dwu4	Four packed unsigned 16-bit integers in a 64-bit register pair

Table 3-90. Instruction Operation and Execution Notations (continued)

Symbol	Meaning
gmpy	Galois Field Multiply
i2	Two packed 16-bit integers in a single 32-bit register
i4	Four packed 8-bit integers in a single 32-bit register
int	32-bit integer value
lmb0(x)	Leftmost 0 bit search of x
lmb1(x)	Leftmost 1 bit search of x
long	40-bit integer value
lsbn or LSBn	n least-significant bits (for example, lsb16)
msbn or MSBn	n most-significant bits (for example, msb16)
nop	No operation
norm(x)	Leftmost nonredundant sign bit of x
not	Bitwise logical complement
op	Opfields
or	Bitwise OR
R	Any general-purpose register
ROTL	Rotate left
sat	Saturate
sbyte0	Signed 8-bit value in the least-significant byte position in 32-bit register (bits 0-7)
sbyte1	Signed 8-bit value in the next to least-significant byte position in 32-bit register (bits 8-15)
sbyte2	Signed 8-bit value in the next to most-significant byte position in 32-bit register (bits 16-23)
sbyte3	Signed 8-bit value in the most-significant byte position in 32-bit register (bits 24-31)
scstn	n-bit signed constant field
se	Sign-extend
sint	Signed 32-bit integer value
slong	Signed 40-bit integer value
sdword	Signed 64-bit integer value
slsb16	Signed 16-bit integer value in lower half of 32-bit register
smsb16	Signed 16-bit integer value in upper half of 32-bit register
src1_e or src2_e	lsb32 of 64-bit src (placed in even-numbered register of a 64-bit register pair)
src1_h or src2_h	msb8 of 40-bit src (placed in odd-numbered register of 64-bit register pair)
src1_l or src2_l	lsb32 of 40-bit src (placed in even-numbered register of a 64-bit register pair)
src1_o or src2_o	msb32 of 64-bit src (placed in odd-numbered register of 64-bit register pair)
s2	Two packed signed 16-bit integers in a single 32-bit register
s4	Four packed signed 8-bit integers in a single 32-bit register
-s	Perform 2s-complement subtraction and saturate the result to the result size, if an overflow occurs
+s	Perform 2s-complement addition and saturate the result to the result size, if an overflow occurs
ubyte0	Unsigned 8-bit value in the least-significant byte position in 32-bit register (bits 0-7)
ubyte1	Unsigned 8-bit value in the next to least-significant byte position in 32-bit register (bits 8-15)
ubyte2	Unsigned 8-bit value in the next to most-significant byte position in 32-bit register (bits 16-23)
ubyte3	Unsigned 8-bit value in the most-significant byte position in 32-bit register (bits 24-31)
ucstn	n-bit unsigned constant field (for example, ucst5)
uint	Unsigned 32-bit integer value
ulong	Unsigned 40-bit integer value
ullong	Unsigned 64-bit integer value
ulsb16	Unsigned 16-bit integer value in lower half of 32-bit register
umsb16	Unsigned 16-bit integer value in upper half of 32-bit register
u2	Two packed unsigned 16-bit integers in a single 32-bit register

Table 3-90. Instruction Operation and Execution Notations (continued)

Symbol	Meaning
u4	Four packed unsigned 8-bit integers in a single 32-bit register
x clear b,e	Clear a field in x, specified by b (beginning bit) and e (ending bit)
x ext l,r	Extract and sign-extend a field in x, specified by l (shift left value) and r (shift right value)
x extu l,r	Extract an unsigned field in x, specified by l (shift left value) and r (shift right value)
x set b,e	Set field in x to all 1s, specified by b (beginning bit) and e (ending bit)
xdwop	64-bit register pair operand that can optionally use cross path
xint	32-bit integer value that can optionally use cross path
xor	Bitwise exclusive-ORs
xsint	Signed 32-bit integer value that can optionally use cross path
xslsb16	Signed 16 LSB of register that can optionally use cross path
xsmsb16	Signed 16 MSB of register that can optionally use cross path
xs2	Two packed signed 16-bit integers in a single 32-bit register that can optionally use cross path
xs4	Four packed signed 8-bit integers in a single 32-bit register that can optionally use cross path
xuint	Unsigned 32-bit integer value that can optionally use cross path
xulsb16	Unsigned 16 LSB of register that can optionally use cross path
xumsb16	Unsigned 16 MSB of register that can optionally use cross path
xu2	Two packed unsigned 16-bit integers in a single 32-bit register that can optionally use cross path
xu4	Four packed unsigned 8-bit integers in a single 32-bit register that can optionally use cross path
□	Assignment
+	Addition
++	Increment by 1
×	Multiplication
-	Subtraction
==	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<<	Shift left
>>	Shift right
>>s	Shift right with sign extension
>>z	Shift right with a zero fill
~	Logical inverse
&	Logical AND

3.18.4 Instruction Set

The following are an overview of the different instruction classes.

3.18.4.1 Flow Control Instructions (Branches, Calls, Returns, Systems Calls)

These instructions are used to alter the program flow.

3.18.4.1.1 Branches (BA, B, BE)

The C7x CPU supports a different variation of branches:

- Absolute branch (BA), where the branch destination is specified by a register
- Unconditional relative branch with 19-bit signed constant displacement (B)
- Conditional relative branch with 19-bit signed constant displacement (B)
- Conditional relative branch using extended 46-bit signed constant as displacement (BE)

3.18.4.1.2 Calls (CALLA, CALL, CALLE)

Calls and returns are used for context switching during exception handling.

- Absolute call (CALLA)
- Unconditional relative call with 19-bit signed constant displacement (CALL)
- Conditional relative call with 19-bit signed constant displacement (CALL)
- Conditional relative call using extended 46-bit signed constant as displacement (CALLE)

3.18.4.1.3 System Call (SYSCALL), Root Call (ROOTCALL), Security Call (SECCALL)

The SYSCALL/ROOTCALL/SECCALL instructions are used to generate software-triggered events. There are three instructions to direct the service request to the proper privilege level.

Typically, SYSCALL is used with an embedded OS/RTOS, where the application task running in a unprivileged/ lower privileged execution state can request services from the OS, which runs in the higher privileged execution state. The SYSCALL event mechanism provides the transition from unprivileged to privileged or lower privileged to higher privileged.

In addition, the SYSCALL mechanism is useful as a gateway for application tasks to access various services (including OS services, or other API functions such as semaphore_pend function provided by TI-RTOS) because an application task can request a service without knowing the actual program memory address of the service. It only must know the SYSCALL service number, the input parameter, and the return results.

Syntax and priorities – These instructions have the following syntax:

- SYSCALL <10-bit constant>
- ROOTCALL <10-bit constant>
- SECCALL <10-bit constant>

The 10-bit immediate value is required for this instruction. It is an 10-bit constant. The value itself does not affect the behavior of these call events, but the hardware can use this number to find the handler's entry point address.

These calls inherit the priority level from the caller process. For example, if a user task at COP = 0xFF makes a SYSCALL. The CPU transition in supervisor with COP = 0xFF still when running SYSCALL handler.

3.18.4.1.3.1 SYSCALL Entry Process

The SYSCALL exception is generated when executing the SYSCALL instructions. An immediate value is required for these instructions, which works as an vector offset parameter-passing method. The SYSCALL exception handler entry point can be generated in hardware by concatenating the address as below.

```
SYSCALL #parameter; // SYSCALL vector address = ESTP + 0x1000 + #parameter * 64Bytes;
```

SYSCALL can be executed from every privilege level. If supervisor makes SYSCALL, hardware treats it as if it's a branch to the corresponding entry point.

3.18.4.1.3.2 ROOTCALL Entry Process

The ROOTCALL can only be executed in guest supervisor or root supervisor privilege level. ROOTCALL vector generation uses ESTP_S register, which is also used when root user (U) makes a SYSCALL to root supervisor (S), as described above. To differentiate the entry points of the SYSCALL to the root supervisor from the ROOTCALL to the root supervisor, the C7x provides separate vector table offset addresses for SYSCALL and ROOTCALL. This way, the root supervisor can keep para-virtualization code separated from the rest of system related code in it's memory space.

In C7x, the vector table offset of root supervisor system calls are:

- SYSCALL Offset = 0x1000 ;
- ROOTCALL Offset = 0x2_0000;

3.18.4.1.3.3 SECCALL Entry Process

The SECCALL can be executed in root supervisor, permission-granted guest supervisor, or secure supervisor privilege level. SECCALL vector generation uses ESTP_SS register, which is also used when secure user (SU) makes a SYSCALL to secure supervisor (SS).

To differentiate the entry points of the SYSCALL to the secure supervisor from the SECCALL to the secure supervisor, the C7x provides separate vector table offset addresses for SYSCALL and SECCALL. This way, the secure supervisor can keep secure monitor code for a virtualized system from secure use initiated system code in it's memory space.

In C7x, the vector table offset of secure supervisor system calls are:

- SYSCALL Offset = 0x1000;
- SECCALL Offset = 0x2_0000;

3.18.4.1.4 Returns From Events (RETE)

RETE instruction is executed at the end of the event handler to trigger the event return mechanism. As a result, only supervisor (SS, S or GS) code may execute RETE. If user code executes this instruction, the CPU raises a privilege exception.

In the event handler, by using ECSP addressing mode, the software can read the context saving stack content at a fixed offset location, such as the return pointer (RP) value and the saved TSR snapshot. The RETE instruction takes two opcode fields, one is the return pointer and the other the saved TSR value.

3.18.4.1.4.1 CPU Execution Privilege Transition on Event Return

Event return could also change the CPU's current execution mode. Most of the time, the event return transitions the CPU from handler privilege back to the requestor privilege or the preempted program privilege. But the supervisor has the option to switch to a different mode allowed which may not have relationship with the event entrance. The C7x provides this flexibility to the software through RXMR register accesses.

The legal privilege transition is forced by the hardware, according to Table 3-91. If any privilege code attempts to transition into a higher privilege level by executing RETE, the CPU raises a privilege exception. This could happen if the RXMR.CXM holds a higher privilege level than current execution privilege when RETE instruction is executed.

Table 3-91. Valid Privilege Transitions on RETE

Initial Privilege Level	Allowed Privilege after Event Return
Guest Supervisor (GS)	GS, GU
(Root) Supervisor (S)	S, U, GS, GU
Secure Supervisor (SS)	SS, SU, S, U, GS, GU

3.18.4.1.5 Returns from System Call (RETS)

RETS instruction is executed at the end of the system call handler to trigger the system call return mechanism. The program the CPU returns back to could be:

- The program initiated by the system call
- The program the supervisor invokes to service the system call

The supervisor which services the system call controls the returned program selection, by programing the RXMR.SYSCALL and RP registers before executing RETS instruction.

3.18.4.1.6 Return From Subroutine (RET)

Branches to the location are contained in Return Pointer (RP) register. This instruction can only be executed in protected mode. If this instruction is issued in unprotected mode, an exception is generated. In protected mode, no change in execution mode or control registers occurs when this instruction is executed.

3.18.4.2 Pipeline Control Instructions (PROT, UNPROT, PROTCLR)

The C7x pipeline can operate in two modes: protected mode and unprotected mode. In both modes, branches have zero delay slots. Pipeline protection is enabled or disabled through setting bit 8th in the TSR (TSR.PROT). Three new instructions are introduced to manipulate the modes: PROT, PROTCLR, and UNPROT. When TSR.PROT is not set, executing PROT and PROTCLR sets TSR.PROT. When TSR.PROT is set, executing UNPROT clears TSR.PROT. If TSR.PROT is already set, executing PROT has no visible effect. Likewise, if TSR.PROT is not set, executing UNPROT has no effect.

3.18.4.2.1 Instruction Setting Unprotected Pipeline Mode (UNPROT)

Unprotected (exposed) pipeline mode is the traditional VLIW operation mode. Unprotected pipeline mode requires the programmer or compiler to know the latencies of the instructions and to insert NO-OPS or other instructions between dependent instructions to ensure correctness. The CPU executes in unprotected mode when bit TSR.PROT=0.

Executing the UNPROT instruction sets the TSR.PROT to zero.

3.18.4.2.2 Instruction Setting Protected Pipeline Mode (PROT)

Protected (unexposed) pipeline operation mode conforms to the sequential operation model where dependent instructions are ensured to be correct, regardless of how many cycles it takes to complete the instructions. The CPU executes in protected mode when bit TSR.PROT=1.

Executing the PROT instruction causes the destination registers of the instructions in the following execute packet to be protected by the pipeline control logic. This means that for an instruction which takes multiple cycles to complete, that if a subsequent instruction attempts to read the destination of the first instruction within the delay slots of that first instruction, then the CPU pipeline is stalled until the instruction which writes that register has completed.

3.18.4.2.3 PROTCLR

Similar to PROT instruction, the PROTCLR instruction changes the operation mode to protected mode. The PROTCLR instruction has the following properties:

- PROTCLR instruction annuls all pending writes which has life-time > 0 before PROT takes affect. (Life-time: remaining arch. cycles to be executed).
- PROTCLR turns instructions-with-latency, which is parallel with it, into single-cycle instructions.

Both PROT and PROTCLR instructions protect “on-reads” instruction packets after it, meaning the data consumed by instructions following a PROT or PROTCLR instruction are ensured to be the data produced last.

For more details on PROT and UNPROT operation mode, refer to Section 3.5.

3.18.4.3 NOP Instruction

NOP instruction performs no operation for a specified cycle duration. The 6-bit constant field (UCST6) is encoded as idle cycle count-1. For UCST6+1 cycles, no operation is executed.

The maximum value for count is 16. NOP with no operand is treated like NOP 1 with UCST6 field encoded as 000000.

Multi-cycle NOP instructions (NOP with count greater than 1) can not be issued in parallel with any other instructions.

3.18.4.4 IDLE Instruction

IDLE instruction performs an infinite multi-cycle NOP that can be terminated upon servicing an event. The IDLE instruction can not be issued in parallel with other instructions.

The IDLE instruction monitor idle status of L1I, L1D, SE, CMMU blocks and send out an overall idle status to power down controller by monitoring the following:

- Loads/stores: The CPU keeps track of load/store operations to know when they have completed.
- \ Instruction fetch: The CPU tells the PMC to stop instruction prefetching and keeps track of when the PMC has stopped fetching new instructions.
- Streaming engine: The CPU monitors the TSR.SE_ACTIVE bits to ensure all streaming engines have been closed. Also, the SE sends the idle signal to the CPU to let the CPU knows that all transactions that it initiates to L2 have completed.

3.18.4.5 Floating Point Instructions

The C7x supports single precision, double precision floating point operations, and half-precision conversions. The C7x has two modes: flush-to-zero mode and IEEE-754 standard mode.

3.18.4.5.1 Flush-to-Zero Mode

The flush-to-zero mode treats all the subnormal operands as zeros, and produces zeros if the result is in the subnormal range. This is the default mode upon reset.

3.18.4.5.2 IEEE-754 Standard Mode

When the IEEE-mode is enabled, the floating point operation sub-normal handling are in full compliance with the IEEE-754 Standard.

The flush-to-zero mode has latency advantage over the IEEE-mode when doing multiplication and some conversion instructions. The extra cycles require no software consideration and are handled by hardware automatically.

Table 3-92 shows the floating point instructions which have latency differences between flush-to-zero mode versus IEEE-754 mode.

Table 3-92. Instructions With Latency Difference Between Flush-to-Zero vs. IEEE-754 Modes

Instructions	Flush-to-Zero Mode Latency	IEEE-754 Mode Latency
MPYSP	4	5
VMPYSP	4	5
MPYDP	4	5
VMPYDP	4	5
VCMPYSP	4	5
VMPYSP2DP	4	5
VMATMPYSP	4	5
VOPMATMPYSP	4	5
VSPDPH	4	5
VSPDPL	1	3
VHPSP	1	3

3.18.4.5.3 Floating Point Configuration Register (FPCR)

The floating-point status register (FPCR) is used to control the floating point operations. FPCR is shown in Figure 3-20 and described in Table 3-93.

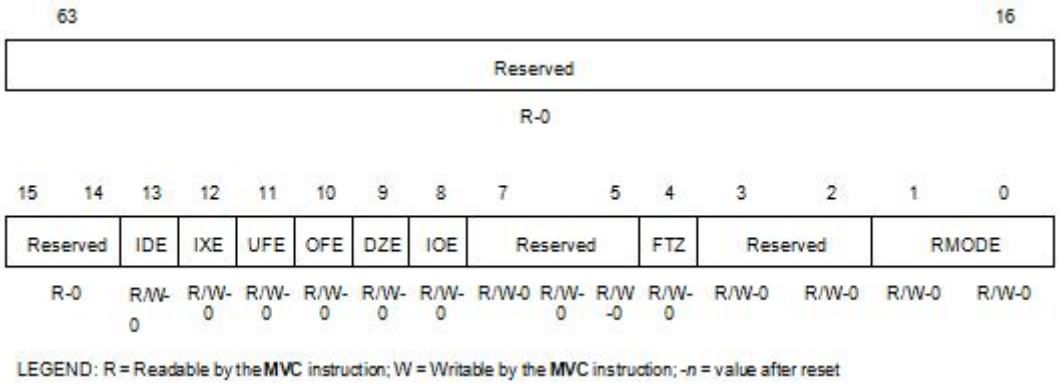


Figure 3-20. Floating Point Configuration Register (FPCR)

Table 3-93. Floating Point Configuration Register (FPCR) Field Descriptions

Bit	Field	Description
63-14	Reserved	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect
13	IDE	Input Denormal Exception Enable
12	IXE	Inexact Operation Exception Enable
11	UFE	Underflow Exception Enable
10	OFE	Overflow Exception Enable

Table 3-93. Floating Point Configuration Register (FPCR) Field Descriptions (continued)

Bit	Field	Description
9	DZE	Divide By Zero Exception Enable
8	IOE	Invalid Operation Exception Enable
7-5	Reserved	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect
4	FTZ	Flush to zero mode. Sub-normal values are treated as zeros
3-2	Reserved	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect
1-0	RMODE	0-3h - Rounding mode for floating point operations 0h = Round toward nearest representable floating-point number 1h = Round toward 0 (truncate) 2h = Round toward infinity (round up) 3h = Round toward negative infinity (round down)

3.18.4.5.4 Flag Status Register (FSR)

The flag status register (FSR) contains fields that contain the status of saturation operations and floating point operations such as underflow, overflow, NaNs, subnormal, infinity and inexact results. FSR is shown in Figure 3-21 and described in Table 3-94.

For a SIMD vector floating point operations, the status produced by individual operation are OR'ed together to produce one final status flag.

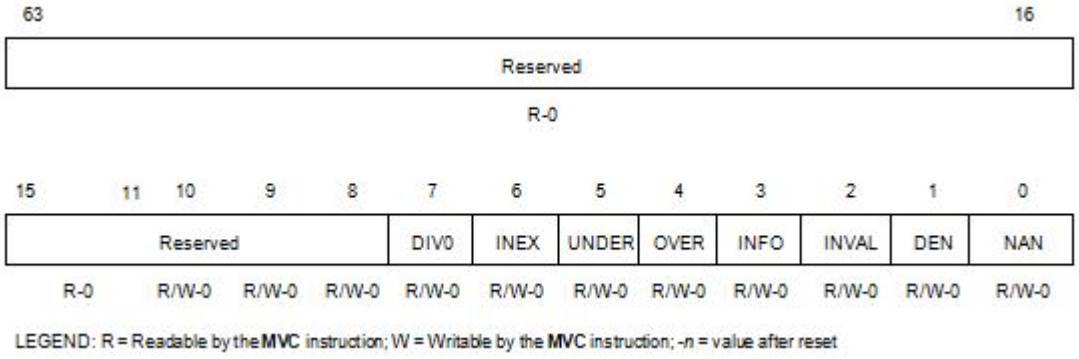


Figure 3-21. Flag Status Register (FSR)

Table 3-94. Flag Status Register (FSR) Field Descriptions

Bit	Field	Description
63-8	Reserved	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7	SAT	Result is saturated -- Fixed Point Only 0h = Result is saturated 1h = Result is not saturated
6	UNORD	Compare involves sources contain NaN 0h = Does not contain NaN 1h = Contains NaN
5	DEN	Sources contain subnormal number 0h = Does not contain subnormal number 1h = Contains subnormal number
4	INEX	Inexact results status for .S1 1h = Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
3	UNDER	Result underflow status for .S1 0h = Result does not underflow 1h = Result underflow
2	OVER	Result overflow status for .S1 0h = Result does not overflow 1h = Result overflows

Table 3-94. Flag Status Register (FSR) Field Descriptions (continued)

Bit	Field	Description
1	DIV0	Source to reciprocal/divide operation 0h = 0 is not source to reciprocal/divide operation 1h = 0 is source to reciprocal/divide operation
0	INVAL	Sources contain SNAN 0h = A signed NaN (SNaN) is not a source 1h = A signed NaN (SNaN) is a source

3.18.4.6 Memory Fence Instructions (MFENCE, MFENCEST, MTAG)

The C7x supports two fencing instructions.

3.18.4.6.1 MFENCE k

Stalls the instruction fetch pipeline until all pending load and store operations with the same memory tag color specified by the constant k are completed.

- k= 00000b: stall until all memory transactions with memory tag color = 0 completed
- k= 00001b: stall until all memory transactions with memory tag color = 1 completed
- k= 00010b-11110b: reserved
- k= 11111b: stall until all memory transactions, regardless of memory tag color, completed

3.18.4.6.2 MFENCEST k

Stalls the instruction fetch pipeline until all pending store operations with the same memory tag color specified by the constant k are completed.

- k= 00000b: stall until all store transactions with memory tag color = 0 completed
- k= 00001b: stall until all store transactions with memory tag color = 1 completed
- k= 00010b-11110b: reserved
- k= 11111b: stall until all store transactions regardless of memory tag color completed

The memory tag color is stored in the TSR.MCOLOR field of the TSR register. The TSR.MCOLOR is read by every load and store operation, to know which tag color it belongs to.

The instruction MTAG is used to modify the TSR.MCOLOR field.

MFENCE and MFENCEST are issued on the .D1 unit only.

3.18.4.6.3 MTAG Instruction (MTAG)

The MTAG instruction changes the memory tag color for memory fence actions. This instruction changes the memory tag color which is recorded in the TSR.MCOLOR field.

- k= 00000b: Change the memory tag color to 0
- k= 00001b: Change the memory tag color to 1
- k= 00010b-11110b: Reserved
- k= 11111b: Flipping the memory tag color

The new color specified by the MTAG instruction is available for the instructions in the following execute packet. The instructions in parallel with the MTAG instruction still use the current color in TSR.MCOLOR.

3.18.4.6.4 Fencing with Streaming Engine

The SEOPEN instruction creates an implicit MFENCE operation to ensure it has the latest data before the streaming engine starts fetching data from memory system. When CPU opens a stream, the streaming engine starts to fence on the selected fence color and waits for the respected fence IDLE signals from DMC. If during this wait for IDLE state from DMC, if the CPU took an interrupt and the streaming engine goes into frozen state, the CPU then issues a STRSAVE, STRRSTR, and upon return from interrupt, the SE should fence on all tag colors, not just the fence color from SEOPEN, because the streaming engine does not maintain SEOPEN fence color state.

3.18.4.6.5 Fencing with IDLE Instruction

The IDLE instruction creates an implicit MFENCE operation to ensure all memory operations are completed before the processor gets to IDLE mode.

3.18.4.7 Atomic Instructions

The C7x defines a new atomic instruction family to replace the C66x LL/SL/CL combination. The new atomic instructions are CASW and CASD, supporting different data types: W(ord) and D(ouble Word). All atomic instructions must have their addresses aligned to their element sizes.

3.18.4.7.1 Atomic Compare and Swap (CASW, CASD)

This instruction sends out two data elements to store data to memory. The top data element is then compared to the existing value at the specified memory location. If it is EQUAL to the memory location value, then the lower data element is written to the memory location. The instruction must then return the value that was previously stored in the memory location. From CPU perspective, this instruction looks like a store and a load to the CPU.

3.18.4.7.2 Atomic Swap (ASW, ASD)

This instruction is similar to the above atomic compare and swap instructions; however, there is no comparison and the data is always swapped.

3.18.4.8 Loads Instructions

The C7x can perform one scalar load and one vector load in parallel. The supported addressing modes for load instructions are described in Section 3.15.

3.18.4.8.1 Scalar Loads

Scalar loads can load up to 64 bits of data or less at a time. The scalar load instructions have an embedded predication field and can be predicated by [A0], ![A0], [A1] registers, or are unconditional.

3.18.4.8.2 Vector Loads

The C7x ISA defines a vector load as a load up to 512 bits of data. All vector load instructions have 5-cycle latency, the same latency as scalar loads.

All vector load/store operations are aware of the endian mode of the DSP. In general, the lowest address is always placed in the SIMD lane containing bit zero, and the highest memory address is placed in the SIMD lane containing bit 512.

3.18.4.8.3 Speculative Load Instructions

Speculative loads are loads which do not signal an exception even if they violate memory protection. Speculative loads allow the compiler to issue a load before knowing if the address is good. When a protection or page-not-mapped faults are encountered, a speculative load returns value of zeros. The speculative loads still stall.

The following are current handling for speculative loads:

- All speculative loads return zeros if they encounter protection violations. If the speculative load is to an unmapped page table, page-faults are signaled to the O/S, which decides to bring the page in if it is not mapped.
- If the speculative load is to a memory-mapped I/O space, the memory system counts it as a failure and drops the requests.
- All vector loads are assumed to be speculative loads only.

3.18.4.8.4 Unpacking Loads

Vector load operations can perform unpacking from one size to another (with sign- or zero-extension) and vector store operations can perform packing, reducing the pressure on the .L and .S units. The unpacking loads always convert from a data size to the next larger power-of-two data size. For example, unpack 8-bit data into 16-bit lanes. Conversely, the packing stores always convert from a data size to the next smaller power-of-two data size.

3.18.4.8.5 Load and Swap

In certain cases, the incoming data may be stored in a format that is different than what the C7x ISA prefers. A common example of this is complex data types. Incoming complex data may come in a format where the imaginary portion is packed in the MS word of the data, and the real portion is stored in the Least Significant word. The C7x CPU hardware assumes that the real part is in the Most Significant Word (MSW) and the imaginary is in the Least Significant Word (LSW). The “load and swap” instructions swap every other element

during the load. These loads are also endian aware, for example if you are using a format that contains 16-bit real and 16-bit imaginary parts, then you must use the VLDH or the VLDHSWAPB instructions to load the data or load the data and swap every half-word, respectively.

3.18.4.8.6 Load and Duplicate

Load an element and duplicate it into all the other elements of a vector or double vector.

3.18.4.8.7 Data Prefetch Instruction (PFS)

The C7x defines a new instruction to support prefetching data into L1D. It is called prefetch scalar instruction (PFS). The PFS instruction brings in one cache line at a time. The instruction has no delay slots, and from a pipeline perspective is most similar to a store.

3.18.4.9 Store Instructions

3.18.4.9.1 Scalar Store Instructions

The C7x can perform one scalar store and one vector store in parallel. The supported addressing modes for store instructions are described in Section 3.15.

3.18.4.9.2 Vector Store Instructions

The C7x ISA defines vector load/store operations up to 512 bits. All vector loads and vector stores must have their addresses aligned to the corresponding element sizes.

All vector load/store operations are aware of the endian mode of the DSP. In general, the lowest address is always placed in the SIMD lane containing bit zero, and the highest memory address are placed in the SIMD lane containing bit 512.

3.18.4.9.3 Vector Predicate Stores

Vector predicate stores conditionally store the bytes based on byte enables from a predicate register. Each bit of the predicate register determines which byte in the vector source register to be stored.

3.18.4.9.4 Packing Stores

Packing stores have three variations:

- Pack the lower halves of the data together.
- Pack the upper halves of the data together.
- Pack the upper halves of the data together after shifting left by 1 (for handling signed*signed results).

The constant or register given in a “post increment” mode specifies the number of elements that the base pointer should be incremented by. For example, VLDW *A8[4]++, VA0 loads the vector data from address *A8[4] + 0x10.

3.18.4.9.5 Store a Predicate Register (STPREDB, STPREDH, STPREDW, STPREDD)

The store a predicate instruction reads a predicate register and stores it out to memory.

- STPREDB: the entire predicate register is written out to memory. STPREDB instruction is equivalent to the below code sequence:

```
STPREDB P0, *D0[0] == MVPB P0, A0; STD A0, *D0[0]
```

- STPREDH: OR’ed the bits of each two bit groups of the predicate register to form one bit, then the result bits are packed together and written out to memory. The STPREDH is equivalent to the following instruction sequence:

```
STPREDH P0, *D0[0] == MVPH P0, A0; STW A0, *D0[0]
```

- STPREDW: OR’ed the bits of each four bit groups of the predicate register to form one bit, then the result bits are packed together and written out to memory. The STPREDW is equivalent to the following instruction sequence:

```
STPREDW P0, *D0[0] == MVPW P0, A0; STH A0, *D[0]
```

- STPREDD: OR'ed the bits of each eight bit groups of the predicate register to form one bit, then the result bits are packed together and written out to memory. The STPREDD is equivalent to the following instruction sequence:

```
        STPREDD P0, *D0[0] == MVPD P0, A0; STB A0, *D0[0]
```

3.18.4.10 Memory Operation Ordering

Between load and store operations which are issued in parallel, the ordering listed in Table 3-95 is observed.

Table 3-95. Execution Order of Memory Operation

.D1	.D2	Order
LD	LD	.D1, then .D2
LD	ST	.D1, then .D2
ST	LD	.D2, then .D1
ST	ST	.D1, then .D2
LD	VST	.D2, then .D1
ST	VST	.D2, then .D1

3.18.4.11 Streaming Engine Instructions

3.18.4.11.1 Stream Open (SEOPEN)

The stream open instruction (SEOPEN) is used to initialize the streaming engine with commands to start address calculations, and send requests to level 2 memory to fetch data into the streaming engine. SEOPEN instruction flushes the streaming engine internal states before receiving new data.

```
SEOPEN     base_address, stream number, template
```

Example of SEOPEN instruction assembler syntax:

```
SEOPEN D0, 0, VB0
```

Note

Open streaming engine stream 0 at the starting address specified by D0. The SE0 template is read from VB0, TSR.SE0 bit is set to 1.

When open, an instruction can reference the streaming engine data holding registers as SE0, SE1, or SE0++ and SE1++.

The SEOPEN instruction sends out the TSR.MCOLOR field to the streaming engine to specify which memory color the stream is fenced on.

```
TSR.MCOLOR definition:
000000        == fence on color 0
000001        == fence on color 1
000010-111110 == Reserved
111111        == fence on all colors
```

Store data from store instructions with the specified memory color issued before the SEOPEN are seen by the streaming engine. Store data from store instructions issued in parallel or after the SEOPEN instruction may not be seen, and the result of SE read from the same address is undefined.

3.18.4.11.2 Stream Open Short Cut (SEOPENSC)

The DSP, CPU, and streaming engine support a small set of short-cut instructions for common stream types, as shown in Table 3-96. These stream short-cuts do not require a separate stream template vector register. Each short-cut stream sends out its template encoding according to Table 3-97.

```
SEOPENSC            base_address, stream_number, shortcut_operations
```

Table 3-96. Stream Open Short-Cut Instructions

Shortcut_Operations Encoding	Short-cut Mnemonic	Element Size	Promotion
000000	SEOPENB	1 byte	None
000001	SEOPENBH	1 byte	Sign extend to half-words
000010	SEOPENBUH	1 byte	Zero extend to half-words
000011	SEOPENH	Half-words (2 bytes)	None
000100	SEOPENHW	Half-words (2 bytes)	Sign extend to words
000101	SEOPENHUW	Half-words (2 bytes)	Zero extend words
000110	SEOPENW	Words (4 bytes)	None
000111	SEOPENWD	Words (4 bytes)	Sign extend to double-words
001000	SEOPENWUD	Words (4 bytes)	Zero extend to double-words
001001	SEOPEND	Double-words (8 bytes)	None

Concretely, each of the above instructions sends one of the following templates to open the stream.

Table 3-97. Explicit Template Encodings for Stream Short Cut Instructions

Instruction	Bits [511:448](1)	Bits[447:32](2)	Bits[31:0](3)
SEOPENB	0x00000000_00006000	All zeros	0xFFFFFFFF
SEOPENBH	0x00000000_00005500	All zeros	0xFFFFFFFF
SEOPENBUH	0x00000000_00005100	All zeros	0xFFFFFFFF
SEOPENH	0x00000000_00005001	All zeros	0xFFFFFFFF
SEOPENHW	0x00000000_00004501	All zeros	0xFFFFFFFF
SEOPENHUW	0x00000000_00004101	All zeros	0xFFFFFFFF
SEOPENW	0x00000000_00004002	All zeros	0xFFFFFFFF
SEOPENWD	0x00000000_00003502	All zeros	0xFFFFFFFF
SEOPENWUD	0x00000000_00003102	All zeros	0xFFFFFFFF
SEOPEND	0x00000000_00003003	All zeros	0xFFFFFFFF

(1) Corresponds to FLAGS[63:0] - setting up a 1-dimensional stream.
(2) Corresponds to DIM5 down to DIM1, and ICNT5 down to ICNT1 (where non-active ICNT's are not used).
(3) Corresponds to ICNT0.

3.18.4.11.3 Stream Close (SECLOSE)

SECLOSE closes a stream and marks a stream inactive, flushing any outstanding activity. Any further references to the stream trigger exceptions. SECLOSE also allows a program to prematurely terminate one or both streams.

STRCLOSE does not stall the DSP CPU, and does not wait for any stream to finish sending requests. To synchronize the DSP CPU with a no-data stream, programs should issue an appropriate MFENCE operation.

The CPU and streaming engine permit SECLOSE on an inactive or frozen stream. This forces the streaming engine into the inactive state if it is currently frozen. This allows cleanup code to blindly issue SECLOSE for both streams to put the CPU in a known state.

```
SECLOSE     stream_number
```

Example of SECLOSE instruction assembler syntax:

```
SECLOSE 0
```

Note

Closing streaming engine stream 0. TSR.SE0 bit is set to 0.

3.18.4.11.4 Stream Break Instruction (SEBRK)

Stream breaks allow exiting early from a level of loop nest within a stream. SEBRK takes the following arguments:

```
SEBRK               levels_to_break, stream number
```

Table 3-98. SEBRK Arguments

Argument	Description
stream_number	Which stream to trigger a break on (0 or 1)
levels_to_break	Number of loop levels to break out of (0, 1, 2, 3, 4 or 5)

Issuing a SEBRK causes the streaming address generator to skip all remaining elements for the corresponding number of loop levels.

SEBRK5 ends the stream but does not close it.

3.18.4.12 Saving and Restoring Streams: SESAVE / SERSTR

The SESAVE and SERSTR instructions give software the ability to context switch a stream. At a high level, SESAVE captures enough stream state from the streaming engine to allow restoring the stream to its current position in the future. Conversely, SERSTR copies the saved stream state back into the streaming engine, so that it can resume where it left off.

Programs can also use SESAVE to inspect the state of a stream that triggered an exception. This provides a clean mechanism for servicing page faults.

```
SESAVE              segment_number, stream_number, stream_save_register
    SERSTR              segment_number, stream_number, stream_save_register
```

Table 3-99. SESAVE / SERSTR Arguments

Argument	Description
segment_number	Which state segment to operate on (0, 1, 2, or 3)
stream_number	Which stream to operate on (0 or 1)
stream_save_reg	Vector to save stream state to or restore stream state from

SESAVE copies the current state of a frozen or inactive stream into the CPU registers. This includes whether the stream was frozen or inactive, all of the stream parameters, and all of the current loop counts and pointers associated with the stream. SERSTR copies this state back into a stream. They provide a context switch mechanism for the streaming engine.

Note

The stream engine reuses and captures each SERSTR segment into the same ‘Segment 0’ hardware vector register each time a SERSTR command is issued, before moving them to internal registers where they belong. Thus, because ‘Segment 0’ register is being updated and overridden after each SERSTR with data from Segment 1, then 2, and then 3, it is important for software to ensure that the last SERSTR command is issued to ‘Segment 0’, so that it is the last vector data captured in register. In other words, the ordering for a SERSTR command when any SERSTR has started, should be to restore Segment 3, 2, 1 in any order, and then the last SERSTR command should be to restore Segment 0. This MUST be done after any SERSTR commands have started on any other Segment, to restore Segment 0. There is, however, no ordering requirements for SESAVE.

Note

When the stream engine is in the “frozen” state, it can restart from its previous position in the stream before it was frozen, when it moves to the “active” state (TSR.SEn = 1), even in the absence of an explicit SESAVE and SERSTR. In this scenario, hardware copies all the current states and counts and restarts the stream from where the last CPU fetch left of.

The streaming engine architecture defines the save/restore record as 4 vector-sized segments. The architecture does not, however, define the exact format or contents of these vectors. The exact contents of the save/restore record may vary from generation to generation.

The architecture ensures that saving the save/restore record saves sufficient context to allow restoring the stream engine state at some arbitrary point in the future. This enables an operating system to context-switch the streaming engine between multiple tasks.

Figure 3-22 provides the layout for the current generation. This definition may change in future architecture revisions. Programs that wish to operate across device generations—such as programs that do not wish to identify individual versions of the streaming engine by its SEn_PID—should save and restore all four segments, despite the fact the present generation does not use the fourth segment.

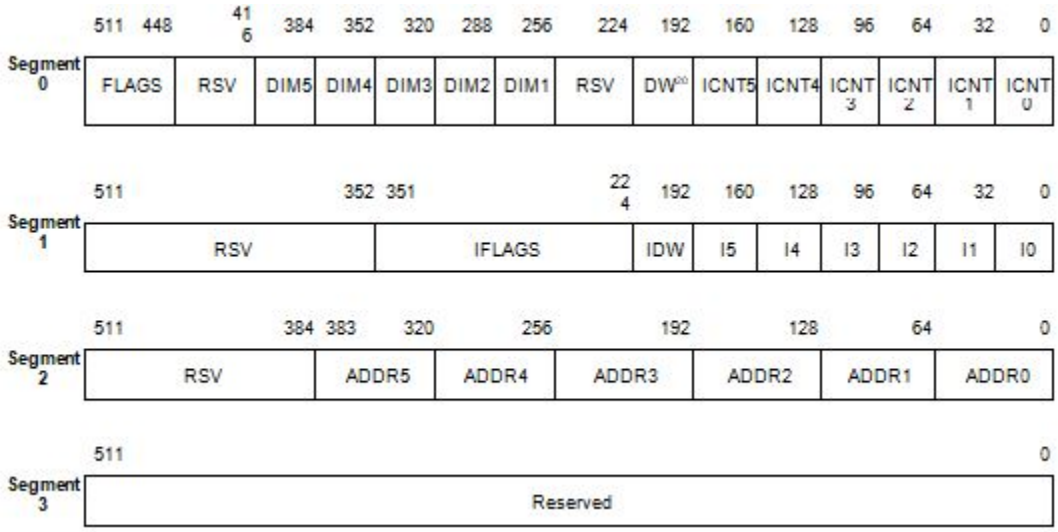


Figure 3-22. Stream Save and Restore Format

Table 3-100 describes all the save/restore fields in the current generation. The first segment shares its definition with the stream template, whereas the streaming engine’s internal state and microarchitectural details define the second and third segment.

Table 3-100. Stream Save/Restore Field Definitions

Field Name	Description
Segment 0	Saved stream template for this stream(1)
I0	Remaining iterations for loop level 0 (innermost)
I1	Remaining iterations for loop level 1
I2	Remaining iterations for loop level 2
I3	Remaining iterations for loop level 3
I4	Remaining iterations for loop level 4
I5	Remaining iterations for loop level 5 (outermost)
IDW	Remaining iterations for DECDIM_WIDTH
IFLAGS	Internal stream engine flags and state(2)
ADDR0	Saved address for loop level 0(3)
ADDR1	Saved address for loop level 1(3)
ADDR2	Saved address for loop level 2(3)
ADDR3	Saved address for loop level 3(3)
ADDR4	Saved address for loop level 4(3)
ADDR5	Saved address for loop level 5(3)

(1) For dataless streams, this takes on an internal format. For short-cut streams, it should resemble the reference short-cut templates in Table 3-97.

(2) IFLAGS are internal stream engine register states which are returned to CPU during a SESAVE. Upon return from a context switch, the CPU should restore, through SERSTR, the IFLAGS as they are with no modifications. This ensures the stream engine can switch to its original state and continue the stream after coming back from any context switch.

(3) To be consistent with the CPU's address handling, ADDR0 through ADDR5 are sign-extended to 64 bits. That is, bits [63:49] are the same as bit [48] of the address. Internally, the steaming engine only uses address bits [48:0]. Also see"Streaming Engine Error Syndromes" on page 1-91, for bit [48] address rollover checks.

3.18.4.12.1 SEOPEN Follow By SEOPEN

If there is a SEOPEN executed while the corresponding SE already opened, the SE logic terminates the current SE stream and opens a new stream.

3.18.4.12.2 Scalar Predication on Instructions with SE Operands

Any instruction which use SE data as operands, such as SE0, SE1, SE0++, and SE1++, can be predicated as normal. For example:

```
Legal: VADDW .L2 SE0++, SE1++, VB0
Legal: [A0] VADDW .L2 SE0++, SE1++, VB0
```

3.18.4.13 Stream Vector Data Registers: SE0 and SE1

The streaming engine conveys the data for both streams through two special 512-bit vector registers, SE0 and SE1.

Programs access streams through these special registers. To access the contents of the stream, programs substitute one of these special register names in place of a CPU register.

Table 3-101 defines these registers.

Table 3-101. Stream Vector Data Registers

Stream Number	Stream Vector Register Name	Contents
0	SE0	Data at the head of stream 0
1	SE1	Data at the head of stream 1

Any number of instructions can include a vector reference in a given execute packet. The CPU places no restrictions on the number of stream vector references that occur in parallel.

The streaming engine vector data registers may be used in place of vector registers as input to nearly any instruction, with the following restrictions:

- It may not be used as a destination argument for an instruction, including combined source/destination arguments.
- It may not be used as a source argument for a store instruction.
- It may not be used as an argument for any instruction on the A-side of the machine.

Other than those restrictions, the SE0 and SE1 registers behave as any other vector register.

SE0 and SE1 may even be used with scalar instructions on the B-side. Scalar instructions see the lowest 64 bits of the stream data.

Note

Certain DSP CPU instructions in the VFIR and VMATMPY family of instructions place additional restrictions on SE0 and SE1. See the description of these instructions in the CPU instruction set for further details.

The special registers listed above always refer to the vector at the top of the stream. Programs can re-read the same top of stream data as many times as necessary. By default, reading these special registers does not advance the stream.

To advance a stream, the streaming engine provides an alternate stream reference encoding that tells the streaming engine to advance to the next 512-bit bundle. The assembly syntax represents stream advance with a post-increment suffix, as shown in Table 3-102.

Table 3-102. Stream Vector Data Registers with Stream Advance

Syntax	Meaning
SE0++	Returns the same data as SE0, and then advances stream to the next 512 bit bundle.
SE1++	Returns the same data as SE1, and then advances stream to the next 512 bit bundle.

Stream advances always advance by 512 bits, regardless of which register name the program uses to signal the advance.

At most one stream reference per execute packet can request a stream advance on a given stream. If multiple parallel instructions attempt to advance the same stream in the same cycle, the stream only advances by one vector. Stream references to different streams in the same execute packet can advance those streams in parallel.

For more details on the streaming engine instructions and its operations, refer the C7x Streaming Engine Specification document.

3.18.4.14 Cache Operation Instructions (BLKCMO and BLKPLD)

These instructions provide help to the programmer to maintain and preload caches to attain optimal cache performances.

3.18.4.14.1 Block Cache Maintenance Operation: BLKCMO

The DSP CPU architecture defines a set of block cache maintenance instructions that operate on a contiguous block of addresses.

It performs block cache maintenance operation on a contiguous block of addresses. This instruction can only be issued on .D2 unit and uses the streaming engine 0 interface.

Block cache maintenance operations take the following general form:

```
BLKCMO              base_address_reg, optype, byte_count_reg
```

Table 3-103. BLKCMO Instruction Arguments

Argument	Description		
base_address_reg	Starting virtual address for the block coherence operation (64 bits)		
	Block cache maintenance operation type. Valid operation types include:		
	DCCU	S	Data Cache Clean to Point of Unification, Shareable
	DCCIU	S	Data Cache Clean and Invalidate to Point of Unification, Shareable
	DCIU	S	Data Cache Invalidate to Point of Unification, Shareable
	DCCC	S	Data Cache Clean to Point of Coherence, Shareable
	DCCIC	S	Data Cache Clean and Invalidate to Point of Coherence, Shareable
	DCIC	S	Data Cache Invalidate to Point of Coherence, Shareable
optype			
	DCCUN	S	Data Cache Clean to Point of Unification, Non-Shareable
	DCCIU	NS	Data Cache Clean and Invalidate to Point of Unification, Non-Shareable
	DCIU	NS	Data Cache Invalidate to Point of Unification, Non-Shareable
	DCCC	NS	Data Cache Clean to Point of Coherence, Non-Shareable
	DCCIC	NS	Data Cache Clean and Invalidate to Point of Coherence, Non-Shareable
	DCIC	NS	Data Cache Invalidate to Point of Coherence, Non-Shareable
byte_count_reg	Total number of bytes to operate on (32 bits)		

Block cache maintenance operations move lines out of one or more caches, leaving the lines either clean or invalid.

3.18.4.14.2 Block Cache Preload: BLKPLD

The DSP CPU architecture defines a set of block cache preload instructions that operate on a contiguous block of addresses. Block cache preload operations take the following general form:

```
BLKPLD              base_address_reg, optype, byte_count_reg
```

Table 3-104. BLKPLD Instruction Arguments

Argument	Description		
base_address_reg	Starting virtual address for the block coherence operation (64 bits)		
optype	Block cache preload type. Valid operation types include:	L2R	Preload to L2 for reading
		L2W	Preload to L2 for writing
		L3R	Preload to L3 for reading
		L3W	Preload to L3 for writing
byte_count_reg	Total number of bytes to operate on (32 bits)		

Preload instructions serve as a hint to the caches. Caches may ignore the hint.

3.18.4.15 Streaming Address Generators

In many algorithms such as sorting, FFT, video compression and computer vision, and data are processed in terms of blocks. Therefore, the abilities to generate both read and write access patterns in multi-dimensions are needed for speeding up these algorithms. The address calculation logic of the streaming engine performs multi-dimensional address calculations to provide streaming data into the C7x CPU. However, the streaming address generators perform multi-dimensional address calculations to be used as offset addresses for load and store instructions in the C7x CPU. The streaming address generators follow similar semantics as the streaming engine's address generation logic. This generation of the C7x devices support up to 4 identical streaming address generators, denoted as SA0 to SA3. Each address generator is configured and operated using its own set of control registers STRACR and STRACNTR.

Only load, stores, and address calculation instructions (ADDAB, ADDAH, ADDAW, ADDAD, SUBAB, SUBAH, SUBAW, SUBAD) can use the SA0-SA3 as an operand.

The generated offsets SA0-SA3 are scaled to the element sizes and must be paired with a base address to form the full address. The element size information comes from the opcode of the load, store, or ADDA/SUBA instructions. Like regular load and store instructions, the base address can comes from any .D unit local registers or any global scalar registers.

A store or load which takes any SA0-SA3 value as offset value is referred to as “streaming load instruction” or “streaming load instruction”.

The streaming address generators include the following components.

3.18.4.15.1 Streaming Address Open Instruction (SAOPEN)

SAOPEN instruction starts a new address generation sequence for the specified streaming address generator. When the SAOPEN instruction is executed, any load or store instructions with address modes of “010” or “110” and the offset register field value of 0-3 uses the offset value calculated by the corresponding address generators SA0-SA3.

SAOPEN takes a stream number and a register value. Upon executing SAOPEN, the register value is copied into the corresponding STRACR register to be used as the address template, the ICNT fields of the corresponding STRCNTR register are initialized and the corresponding TSR.SA bit is set to 1.

```
SAOPEN     template, stream number
```

Example of SAOPEN instruction assembler syntax:

```
SAOPEN VB0, 0
```

Note

Content of VB0 is copied into the STRACR0, initializes the ICNT fields of STRCNTR0, and sets TSR.SA0 to 1.

3.18.4.15.2 Streaming Address Close Instruction (SACLOSE)

SACLOSE instruction explicitly closes the specified streaming address generation operations. Executing SACLOSE resets the corresponding STRACR and STRACNTR to their default values and resets the corresponding TSR.SA bit to 0. Addressing through SA is illegal when the stream is closed.

```
SACLOSE     stream_number
```

3.18.4.15.3 SAOPEN Follows by SAOPEN Before SACLOSE

If there is a SAOPEN executed while the SA already opened, the SA logic terminates the current SA operations and opens a new SA stream.

3.18.4.15.4 Streaming Address Break Instruction (SABRK)

Stream breaks allow exiting early from a level of loop nest within a stream. SABRK takes the following arguments:

```
SABRK               levels_to_break, stream number
```

Table 3-105. SABRK Arguments

Argument	Description
stream_number	Which streaming address to trigger a break on (0, 1, 2 or 3)
levels_to_break	Number of loop levels to break out of (1, 2, 3, 4 or 5)

Issuing a SABRK causes the streaming address generator to skip all remaining elements for the corresponding number of loop levels.

SABRK5 ends the stream but does not close it.

3.18.4.15.5 Streaming Address Control Registers (STRACR0 -3)

The STRACR0-STRACR3 contain the parameters, such as the element counts, loop dimensions, the access length and other flags, to start streaming address calculation. They have similar fields as defined in the streaming engine definition template.

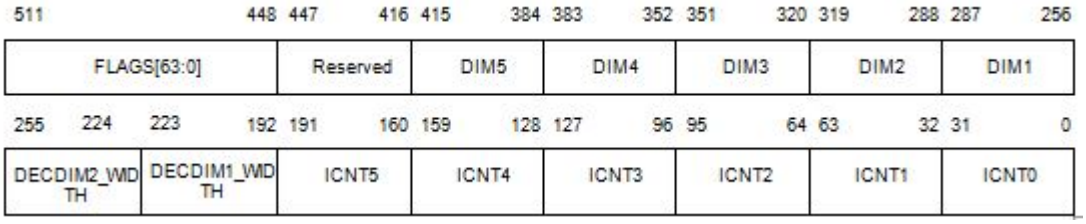


Figure 3-23. Streaming Address Configuration Registers

Table 3-106. Stream Fields

Field Name	Description	Size
ICNT0	Total loop iteration count for level 0 (innermost)	32 bits
ICNT1	Total loop iteration count for level 1	32 bits
ICNT2	Total loop iteration count for level 2	32 bits
ICNT3	Total loop iteration count for level 3	32 bits
ICNT4	Total loop iteration count for level 4	32 bits
ICNT5	Total loop iteration count for level 5 (outermost)	32 bits
DECDIM1_WIDTH	Tile width of DECDIM1. Use together with DECDIM1 flags to specify vertical strip mining feature	32bits
DECDIM2_WIDTH	Tile width of DECDIM2. Use together with DECDIM2 flags to specify vertical strip mining feature	32bits
DIM1	Signed dimension for loop level 1, in elements	32 bits
DIM2	Signed dimension for loop level 2, in elements	32 bits
DIM3	Signed dimension for loop level 3, in elements	32 bits
DIM4	Signed dimension for loop level 4, in elements	32 bits
DIM5	Signed dimension for loop level 5, in elements	32 bits
FLAGS	Stream modifier flags	64 bits

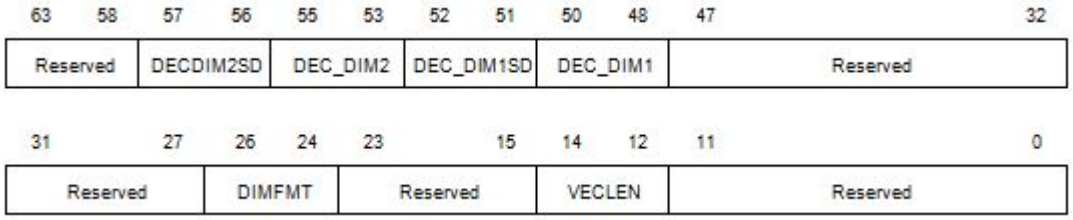


Figure 3-24. Streaming Address Configuration Register FLAGS Field

Streaming address flag field definitions:

- DIMFMT: Number of dimensions
  - 000b = 1-Dimensional stream
  - 001b = 2-Dimensional stream
  - 010b = 3-Dimensional stream
  - 011b = 4-Dimensional stream
  - 100b = 5-Dimensional stream

  - 101b = 6-Dimensional stream
  - 110b-111b = Reserved
- VECLEN: Number of elements get advanced per access
  - 000b = 1 element
  - 001b = 2 elements
  - 010b = 4 elements
  - 011b = 8 elements
  - 100b = 16 elements
  - 101b = 32 elements
  - 110b = 64 elements
  - 111b = Reserved
- DEC_DIM1: Data strip mining decrement dimension for DECDIM1_WIDTH count
  - 000b = Normal operating mode
  - 001b = Strip mining on DIM1 (DECDIM1_WIDTH decrements when entering DIM1)
  - 010b = Strip mining on DIM2 (DECDIM1_WIDTH decrements when entering DIM2)
  - 011b = Strip mining on DIM3 (DECDIM1_WIDTH decrements when entering DIM3)
  - 100b = Strip mining on DIM4 (DECDIM1_WIDTH decrements when entering DIM4)
  - 101b = Strip mining on DIM5 (DECDIM1_WIDTH decrements when entering DIM5)
  - 110b = Reserved
  - 111b = Reserved
- DEC_DIM2: Data strip mining decrement dimension for DECDIM2_WIDTH count
  - 000b = Normal operating mode
  - 001b = Strip mining on DIM1 (DECDIM2_WIDTH decrements when entering DIM1)
  - 010b = Strip mining on DIM2 (DECDIM2_WIDTH decrements when entering DIM2)
  - 011b = Strip mining on DIM3 (DECDIM2_WIDTH decrements when entering DIM3)
  - 100b = Strip mining on DIM4 (DECDIM2_WIDTH decrements when entering DIM4)
  - 101b = Strip mining on DIM5 (DECDIM2_WIDTH decrements when entering DIM5)
  - 110b = Reserved
  - 111b = Reserved

3.18.4.15.6 Streaming Address Count Register (STRACNTR)

The STRACNTR contains the intermediate element counts of all loop levels. Each time a streaming load or store instruction with advancing option [SA++] is executed, the corresponding element count is reduced by a VECLEN amount of elements. When the element count of a loop becomes zero, the address of the element of the next loop is computed using the next loop dimension.

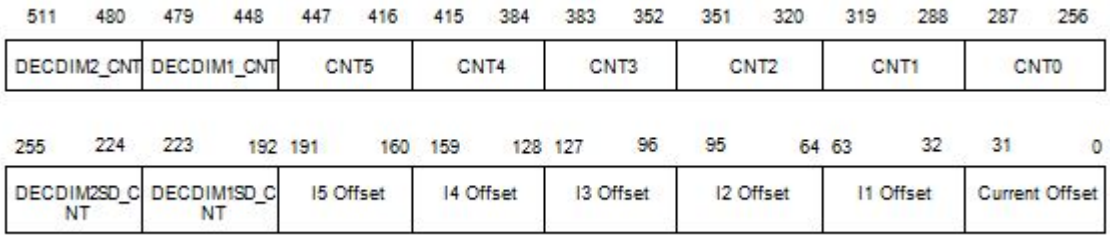


Figure 3-25. Streaming Address Count Register (STRACNTR)

Executing a STRAOPEN instruction sets the CNT fields in STRACNTR to the values contained in the ICNT fields of the STRACR.

While a stream is open, functional access to the associated STRACR and STRACNTR registers are disallowed.

Note

The motivation is to limit corner cases between SA access for addressing (such as *D0[SA0], *D0[SA0++], ADDAx) near MVC operations, and the values visible to programmers. The user must decide whether to generate a fault or just return zeros for MVC operations. Debugger access is still required for CCS.

3.18.4.15.7 Streaming Load / Store Instructions

Streaming load or store instructions are regular load or store instructions which use the address generated by the streaming address generators as the offset. The full address is computed by combing the offset with a base address. Similar to regular load and store instructions, the base address can come from a global scalar register or from a .D unit local register. The offset generated by the streaming address generators are contained in registers [SA0], [SA1], and so forth. By default, reading the [SA] registers do not advance the offset calculations, and the [SA] registers can be re-read with the same value as many time as needed. A different encoding for streaming load and store instructions are used to advance the [SA] registers to the next offset calculation, and are denoted as [SA0++], [SA1++], and so forth.

Element size comes from the load or store instructions that use the [SA] offset, but the number of elements accessed is specified by the VECLEN field of the streaming address configuration registers (STRACR.VECLEN), regardless of the access size specified by the load or store instructions.

The streaming address generators generate offsets, so you can use multiple base addresses with the same offsets and different element sizes.

- Stream advance always advances by VECLEN elements.
- Instruction containing SA advance may be scalar-predicated. If the predicate is false, SA does not advance.
- At most one increment per SA per cycle, regardless of how many are specified in parallel:
  - Legal: SA0++ || SA0++ or SA1++ || SA1++ – , and so forth
  - Legal: SA0++ || SA1++
- SA returns zeros if it advances past the number of available offsets.

Example of streaming load or store instruction assembler syntax:

```
VST8W VB0, *D0[SA0++]
```

Note

Store 8-word elements in VB0 to the memory location specified by Base Register D0 and the offset calculated by the streaming address generator 0, then advance to the next offset calculation by a VECLEN number of word elements. In the case that the number of word elements remaining to be accessed is less than that specified by the instruction, only the remaining number of elements are written to memory.

```
VST8W VB0, *D1[SA0]
```

Note

Store 8-word elements in VB0 to the memory location specified by Base Register D1 and the offset calculated by the streaming address generator 0, then advance to the next offset calculation by a VECLEN number of word elements. In the case that the number of word elements remaining to be accessed is less than that specified by the instruction, only the remaining number of elements are written to memory.

3.18.4.15.8 Streaming Loads and Stores Encoding

Streaming loads and streaming stores only have 2 addressing modes: 010 and 110. The offset register field in src2 is encoded as follows:

- 00000 – 00011: SA0 to SA3

- 00100 – 00111: Reserved for SA4 through SA7
- 01000 – 01111: A8 through A15
- 10000 – 11111: D0 through D15
- Mode 010 provides scaled offset addressing for all valid src2 encodings—for example, *base[A8], *base[D10], *base[SA0], *base[SA1].
- Mode 110 with A8 – A15 or D0 – D15 provides scaled post-increment addressing for all valid src2 encodings —for example, *base++[A11], *base++[D3].
- Mode 110 with SA provides scaled offset addressing and increments the SA (not the base)—for example, *base[SA0++], *base[SA1++].

3.18.4.15.9 Programming Models

The streaming address generators compute a sequence of offsets for elements of a stream. The full address is then calculated by adding the base value with the generated offset to form a pointer walking through memory. A multiple-level loop nest controls the path the pointer takes. The iteration count for a loop level indicates the number of times that level repeats. The dimension gives the distance between pointer positions for consecutive iterations of that loop level.

For a basic, linear address generator configuration, the innermost loop computes the offsets of physically contiguous elements from memory. Its implicit dimension is 1 element. The pointer itself moves from element to element in consecutive, increasing order. In each level outside the inner loop, the loop moves the pointer to a new location based on the size of that loop level's dimension.

The address generators support address calculation mode of forward linear stream.

- Forward streams start at an offset of 0.
- Matches *p++ convention of SE

Forward stream addressing: the following code illustrates the basic algorithm for calculating the offsets of a 6 level forward loop nest. Forward linear stream addressing algorithm:

```
// ptr is an element pointer
current_offset = 0;
for (int i5 = 0; i5 < ICNT5; i5++) {
i5_offset = current_offset;
  for (int i4 = 0; i4 < ICNT4; i4++) {
    i4_offset = current_offset;
    for (int i3 = 0; i3 < ICNT3; i3++) {
        i3_offset = current_offset;/
      for (int i2 = 0; i2 < ICNT2; i2++) {
            i2_offset = current_offset;
        for (int i1 = 0; i1 < ICNT1; i1++) {
            i1_offset = current_offset;
            for (int i0 = 0; i0 < ICNT0; i0 += VECLEN) {
            current_offset += VECLEN;
            sa_offset = current_offset;
            if ((ICNT0 - i0) < _VECLEN) {
                pred = (1 << (ICNT0 - i0)) - 1;
            } else {
                pred = (1 << VECLEN) -1;}
            }//end i0
          current_offset = i1_offset + DIM1;
        }//end i1
        current_offset = i2_offset + DIM2;
        }//end i2
      current_offset = i3_offset + DIM3;
    }//end i3
    current_offset = i4_offset + DIM4;
}//end i4
  current_offset = i5_offset + DIM5;
}//end i5
```

This form of addressing allows programs to specify regular paths through memory in a small number of parameters. Table 3-107 defines these parameters more explicitly.

Table 3-107. Addressing Parameters for a Basic Stream

Parameter	Definition
ICNT0	Number of iterations for the innermost loop dimension, loop level 0. At this level, all elements are physically contiguous. That is, DIM0 = 1. In Data Strip Mining Mode, ICNT0 is used as the initial total “actual width” of the frame
ICNT1	Number of iterations for the first level above the innermost loop, loop level 1.
DIM1	Number of elements between starting points for consecutive iterations of loop level 1.
ICNT2	Number of iterations for loop level 2.
DIM2	Number of elements between starting points for consecutive iterations of loop level 2.
ICNT3	Number of iterations for loop level 3.
DIM3	Number of elements between starting points for consecutive iterations of loop level 3.
ICNT4	Number of iterations for loop level 4.
DIM4	Number of elements between starting points for consecutive iterations of loop level 4.
ICNT5	Number of iterations for loop level 5.
DIM5	Number of elements between starting points for consecutive iterations of loop level 5.

3.18.4.15.10 Predications Handling

The CPU’s vector predicates provide a mechanism for ignoring portions of a vector in certain operations, such as vector-predicated stores. This naturally fits with the notion of valid and invalid element lanes maintained by the streaming address generator. All streaming store instructions are considered as vector-predicated stores.

The unit maintains the counts of the remaining valid elements in streaming address predicate registers PSA0-PSA3. These predicate registers are used for streaming store instructions to generate the byte write enables to send to L1D. When a streaming store instruction is executed, the vector predicate value from the PSA register is read, converted to byte valid/invalid status, and sent out to L1D. This is similar to normal vector predicated store, but the predicate in this case is from the PSA, instead of the PRF.

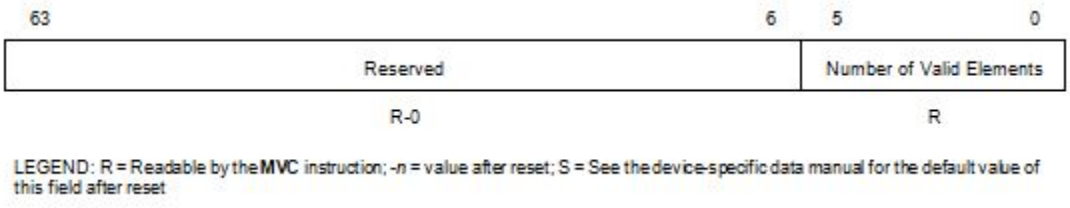


Figure 3-26. Predicate Streaming Address Register

The streaming address predicates are generated every time a new stream is opened, or when a stream reads with advancement occurs.

- SA generates predicates in the following cases:
  - SAOPEN
  - Streaming load or store instructions with SA0++/SA1++/SA2++/SA3++
  - Addressing modes
- Predicate is ‘element wise’ for the next VECLEN elements.
  - VECLEN is power of 2 from 1 to 64
- The SA predicates are generated depending on the configuration of the DEC_DIM field as follows:
  - Predicates always fill the LSBs of the associated predicate registers.
  - The predicates are calculated as CNT0 - VECLEN when in the final loop.
  - The vector predicate is applied when:
    - CNT0 is less than or equal VECLEN
    - CNT0 is saturated at zero
    - CNT0 gets reloaded from the template ICNT0 when the count of the dimension specified by DEC_DIM or higher is reloaded.
- Implicit SA predication on streaming vector stores

  - Streaming vector store gets predicate information directly from the corresponding streaming address predicate register PSA.
  - Streaming vector store translates the PSA predicate value to byte enables as necessary, according to the element type specified by the store instruction.
  - For streaming vector store and pack instructions, the byte enables generated from the PSA register are also packed in the same way as the store data.
  - Only the vector store examines the LSBs of the corresponding streaming address predicate register.
  - If the vector store has fewer elements than the CNT0, the upper predicate bits are ignored.
  - If the vector store has more elements than the CNT0, the upper predicate bits are implicit 0.
- Scalar predication on streaming load/store with advancement
  - Only advanced when scalar predication is true.
- Two taken streaming load/store with advancements in parallel:
  - Address is only advanced once.
- Predication from PSAs can also be applied for scalar streaming stores.

3.18.4.15.11 Stores with Streaming Address Predicate Registers Examples

TBD

3.18.4.15.12 Interrupts, Exceptions, System Call Handling

When encountering an interrupt/exception/system call event, the following steps are taken:
- Streaming address active bits in TSR get set to 0.
- Streaming address generation states in STRACR and STRACNTR registers do not get zeroed.
- Handler can use MVC instruction to save and restore streaming address states as needed.
- Return from interrupt/exception/system call restores TSR, which reactivate a stream if it was previously active.

3.18.4.16 Parallel Table Lookup and Histogram Instructions

The C7x CPU has dedicated instructions to perform table look up and histogram operations. The tables are mapped into L1D memory space. The tables can be loaded either through the DMA port, by LUTINIT instruction, or by normal store instructions to the memory space containing the tables. The C7x CPU supports up to 4 separate sets of parallel look up tables. Within a set, up to 16 tables can be looked up in parallel with byte, half word, or word element sizes. The tables can be accessed with independent index addresses.

The base address for a set of parallel tables is contained in 1 of the 4 Look Up Table Base Address Registers (see Figure 3-27). The look up table instructions contain the set number to identify which base address registers to use as the base address.

Lookup table and histogram instructions treat all data as in little endian mode.

Note

The table base address must be aligned to the table size specified in the LTCR register.

The LTCR Control Register (see Figure 3-28) is used to configure the tables. LTCR specifies the number of parallel tables, the data size of the bin elements, and whether the elements are signed or unsigned data.

3.18.4.16.1 Look Up Table Base Address Registers (LTBR0-3)

The Look Up Table Base Address registers contain the offsets of the L1D SRAM regions configured as tables. There are total of 4 Look Up Table Base Address registers in the C7x CPU. The LTBR registers can only be written by supervisors.

The table base address must be 128 bytes aligned because the L1D cache line is 128 bytes wide, therefore the 7 LSBs of the base address is always read out as 0.

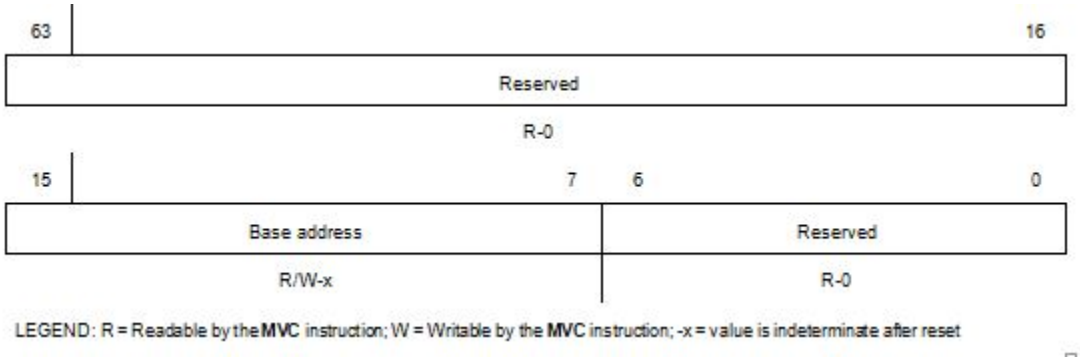


Figure 3-27. Look Up Table Base Address Register

3.18.4.16.2 Look Up Table and Histogram Configuration Registers (LTCR0-3)

The Look Up Table and Histogram Configuration Registers store the control information of the table lookup and histogram instructions. There are 4 LTCR registers, corresponding with the 4 LTBR registers. The LTCR registers can only be written by supervisors.

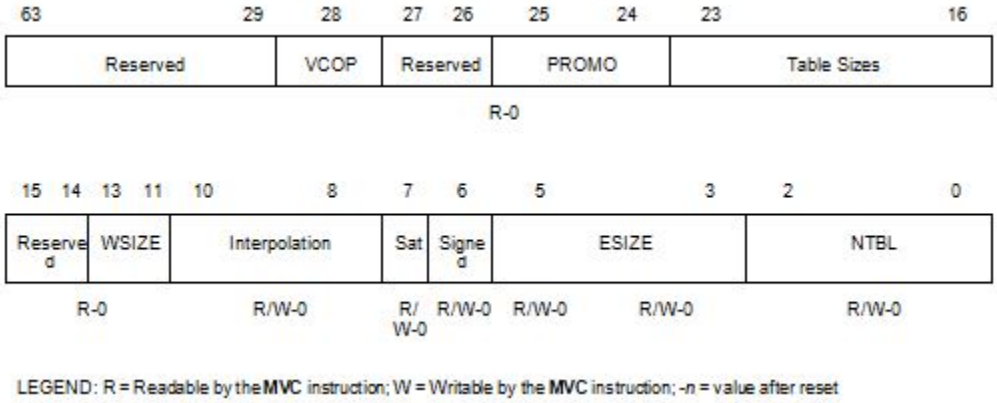


Figure 3-28. Look Up Table and Histogram Configuration Register (LTCR)

Table 3-108. Look Up Table and Histogram Configuration Register Field Descriptions

Bit	Field	Description
63-29	Reserved	Reserved. Read as 0.
28	VCOP	0 = Native c7x memory bank configuration (16 banks x 64-bits) 1 = Legacy VCOP EVE memory bank configuration (8 banks x 32-bis)
27-26	Reserved	Reserved. Read as 0.
25-24	PROMO	Type promotion mode of LUTRD returned data 00 = No promotion 01 = Promote 2x: Bytes to half-words, half-words to words, words to double words 10 = Promote 4x: Bytes to words, half-words to double-words 11 = Promote 8x: Bytes to double-words
23-16	Table Sizes	Size of the SRAM memory allocated for each set of table 00000000 = Reserved. Table elements read as 0. Memory location data unchanged if write to. 00000001 = 1.0 KBytes 00000010 = 2.0 KBytes 00000011 = 4.0 KBytes 00000100 = 8.0 KBytes 00000101 = 16.0 KBytes 00000110 = 32.0 KBytes 00000111-11111111 = Reserved. Table elements read as 0. Memory location data unchanged if write to.
15-14	Reserved	Reserved. Read as 0

Table 3-108. Look Up Table and Histogram Configuration Register Field Descriptions (continued)

Bit	Field	Description
13-11	WSIZE	Weight Sizes 000 = Byte 001 = Half Word 010-111 = Reserved. Read as 0
10-8	Interpolation	Successive numbers of elements are also written to allow interpolation of different data points 000 = No interpolation, only indexed element per table is written 001 = Returns 2 elements per table 010 = Returns 4 elements per table 011 = Returns 8 elements per table 100-111 = Reserved. Read as 0
7	Sat	Histogram bin entries are saturated to min/max values of its data type if enabled 1 = Saturate 0 = Non-saturate
6	Signed	Element data is signed or unsigned 1 = Signed 0 = Unsigned
5-3	ESIZE	Input Element Sizes 000 = Byte 001 = Half Word 010 = Word 011-111 = Reserved. Read as 0
2-0	NTBL	Number of Table to be looked up in parallel 000 = 1 table 001 = 2 tables 010 = 4 tables 011 = 8 tables 100 = 16 tables 101-111 = Reserved. Read as 0

3.18.4.16.3 Look Up Table Enable Register (LTER)

The Look Up Table Enable Register specifies the operations allowed for a particular LUT/HIST set.

When the LTER bits are set, read or write operations through MVC of the corresponding LTBR and LTCR are allowed in user mode. Besides controlling the programmability of LTBR and LTCR, the LTER bit fields also restricts the execution of LUTRD and LUTWR/LUTINIT/HIST/WHIST instructions in user mode. Only when enabled by LTER bit fields are the corresponding look up tables and histogram instructions are allowed to be executed in user mode; otherwise, an exception is raised.

LTER does not affect LUT/HIST operations in supervisor modes. LUT/HIST operations are allowed, regardless of the status of LTER bits in supervisor mode.

LTER can only be written during supervisor modes.

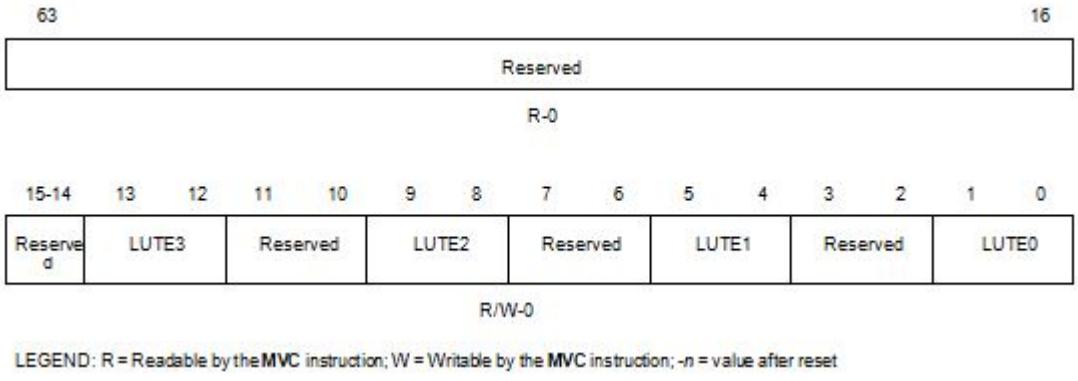


Figure 3-29. Look Up Table Enable Register

Table 3-109. Look Up Table Enable Register

Bit	Field	Description
63-14	Reserved	Reserved. Read as 0.
12-13	LUTE3	Permission for table 3set up. Can only be written by supervisor 11 = MVC read from and write to LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed 10 = MVC read from LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed. Exception raises if encounter MVC write to LTBR, LTCR 01 = MVC read from LTBR, LTCR and execution of LUTRD instruction allowed, exception raises if encounter LUTWR, LUTINIT, HIST, WHIST or MVC write to LTBR, LTCR 00 = No LUT operation allowed
10-11	Reserved	Reserved. Read as 0.
8-9	LUTE2	Permission for table 2 set up. Can only be written by supervisor 11 = MVC read from and write to LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed 10 = MVC read from LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed. Exception raises if encounter MVC write to LTBR, LTCR 01 = MVC read from LTBR, LTCR and execution of LUTRD instruction allowed, exception raises if encounter LUTWR, LUTINIT, HIST, WHIST or MVC write to LTBR, LTCR 00 = No LUT operation allowed
6-7	Reserved	Reserved. Read as 0.
4-5	LUTE1	Permission for table 1set up. Can only be written by supervisor 11 = MVC read from and write to LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed 10 = MVC read from LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed. Exception raises if encounter MVC write to LTBR, LTCR 01 = MVC read from LTBR, LTCR and execution of LUTRD instruction allowed, exception raises if encounter LUTWR, LUTINIT, HIST, WHIST or MVC write to LTBR, LTCR 00 = No LUT operation allowed
3-2	Reserved	Reserved. Read as 0.
0-1	LUTE0	Permission for table 0 set up. Can only be written by supervisor 11 = MVC read from and write to LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed 10 = MVC read from LTBR, LTCR and execution of LUTRD, LUTWR, LUTINIT, HIST, WHIST instructions allowed. Exception raises if encounter MVC write to LTBR, LTCR 01 = MVC read from LTBR, LTCR and execution of LUTRD instruction allowed, exception raises if encounter LUTWR, LUTINIT, HIST, WHIST or MVC write to LTBR, LTCR 00 = No LUT operation allowed

3.18.4.16.4 Interpolation Feature

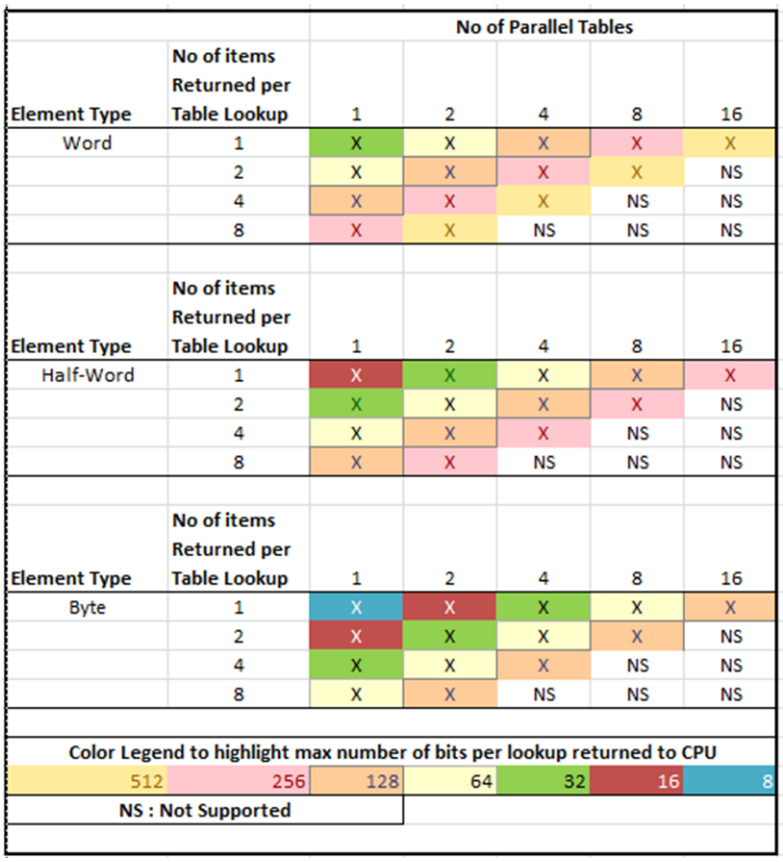
The interpolation bit of the LTCRs is used to enable or disable the interpolation mode.

In non-interpolation mode (LTCR.Interpolation bits = “000”), only one requested element per table is returned. In interpolation mode (LTCR.Interpolation field != “000”), a number of successive elements beyond the indexed element are also returned and written to the destination register.

Figure 3-30 describes the maximum number of elements which can be returned from L1D memory for different table types during interpolation mode.

For example, if the element type is word, and the number of parallel tables is 8, then a total of 2 words per table are returned from L1D. On the other hand, if the element type is word, and the number of parallel tables is 4, a total of 4 words per table are returned.

The bit field LTCR.Interpolation is used to specify how many successive elements to be written to the destination register. The value of LTCR.Interpolation bit field can not exceed the maximum number of elements which can be returned by L1D.



		No of Parallel Tables				
Element Type	No of items Returned per Table Lookup	1	2	4	8	16
Word	1	X	X	X	X	X
	2	X	X	X	X	NS
	4	X	X	X	NS	NS
	8	X	X	NS	NS	NS
Element Type	No of items Returned per Table Lookup	1	2	4	8	16
Half-Word	1	X	X	X	X	X
	2	X	X	X	X	NS
	4	X	X	X	NS	NS
	8	X	X	NS	NS	NS
Element Type	No of items Returned per Table Lookup	1	2	4	8	16
Byte	1	X	X	X	X	X
	2	X	X	X	X	NS
	4	X	X	X	NS	NS
	8	X	X	NS	NS	NS

Color Legend to highlight max number of bits per lookup returned to CPU						
512	256	128	64	32	16	8

NS : Not Supported

Figure 3-30. Number of Elements Returned for Interpolation for Different Table Types

Note

The number of elements returned from L1D, which is always the max allowed for by the bus, is different than the number of elements written to the register, which is controlled by the interpolation field in LTCR.

3.18.4.16.5 LUTRD Valid Combinations

Each LTCR provides separate fields to specify promotion characteristics, the number of interpolation elements, and number of tables. The combinations of those variables are not always valid or feasible. The rules for creating a valid look up table configuration are:

- The total bits asked to be returned after type promotion or interpolation can not be greater than 512-bit vector size.
- Packing the returning byte and half-word elements after type promotion, when the total bytes returned does not fill up an entire 512-bit vector size are not allowed, except for when number of tables is 1.

Table 3-110 lists the configurations supported by the lookup table read instruction (LUTRD).

Table 3-110. Valid Configurations of Lookup Table Read

Input Element Size	No. of Interpolated Elements	1 Table				2 Tables				4 Tables				8 Tables				16 Tables			
		Output Element Size				Output Element Size				Output Element Size				Output Element Size				Output Element Size			
		B	H	W	D	B	H	W	D	B	H	W	D	B	H	W	D	B	H	W	D
Byte	1	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	No	Yes	Yes	No	No	Yes	Yes	No	No	No	NA
	2	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	No	Yes	Yes	No	No	Yes	NA	No	Yes	NA	NA
	4	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	No	Yes	NA	No	Yes	NA	NA	Yes	NA	NA	NA
	8	Yes	Yes	Yes	Yes	No	No	Yes	NA	No	Yes	NA	NA	Yes	NA	NA	NA	NA	NA	NA	NA

Table 3-110. Valid Configurations of Lookup Table Read (continued)

Half-word	1	NA	Yes	Yes	Yes	NA	No	Yes	Yes	NA	No	Yes	Yes	NA	No	Yes	Yes	NA	No	Yes	NA
	2	NA	Yes	Yes	Yes	NA	No	Yes	Yes	NA	No	Yes	Yes	NA	No	Yes	NA	NA	Yes	NA	NA
	4	NA	Yes	Yes	Yes	NA	No	Yes	Yes	NA	No	Yes	NA	NA	Yes	NA	NA	NA	NA	NA	NA
	8	NA	Yes	Yes	Yes	NA	No	Yes	NA	NA	Yes	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
Word	1	NA	NA	Yes	Yes	NA	NA	Yes	Yes	NA	NA	Yes	Yes	NA	NA	Yes	Yes	NA	NA	Yes	NA
	2	NA	NA	Yes	Yes	NA	NA	Yes	Yes	NA	NA	Yes	Yes	NA	NA	Yes	NA	NA	NA	NA	NA
	4	NA	NA	Yes	Yes	NA	NA	Yes	Yes	NA	NA	Yes	NA	NA	NA	NA	NA	NA	NA	NA	NA
	8	NA	NA	Yes	Yes	NA	NA	Yes	Yes	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA

3.18.4.16.6 Programming Models

For look up table and histogram instructions, the indices is read from a vector register and sends out to L1D. Each index value is contained in a fixed 32-bit lane of the source vector register, regardless of table parameters. Depending on the number of tables, the 32-bit indices from different lanes are used as specified in Table 3-111.

Table 3-111. Index Positions in Vector Register Source versus Number of Tables

Vector Register Value	Lookup Table Indices	1 Table	2 Tables	4 Tables	8 Tables	16 Tables
Vx[31:0]	index0	Valid	Valid	Valid	Valid	Valid
Vx[63:32]	index1		Valid	Valid	Valid	Valid
Vx[95:64]	index2			Valid	Valid	Valid
Vx[127:96]	index3			Valid	Valid	Valid
Vx[159:128]	index4				Valid	Valid
Vx[191:160]	index5				Valid	Valid
Vx[223:192]	index6				Valid	Valid
Vx[255:224]	index7				Valid	Valid
Vx[287:256]	index8					Valid
Vx[319:288]	index9					Valid
Vx[351:320]	index10					Valid
Vx[383:352	index11					Valid
Vx[415:384]	index12					Valid
Vx[447:416]	index13					Valid
Vx[479:448]	index14					Valid
Vx[511:480]	index15					Valid

For look up table read operations, the returned elements from each table are packed into the least significant byte positions of the destination register. The remaining unused bits in the destination register are filled with zeros.

The C7x CPU also supports 1, 2, 8, or 16 histogram data bins. Data bin values can be signed or unsigned, specified by the LTCR.sign bit field. For weighted histogram instruction, the data or weights is read from a vector register and sent out as store data to L1D. Each byte/half-word weight value is contained in a 32-bit lane of the source vector register and is always treated as signed value. Depending on the number of tables, different weights are used, as specified in Table 3-112. 32-bit weights are not supported.

Table 3-112. Histogram Weight Positions in Vector Register

Histogram Weight	Histogram Weights	1 Table	2 Tables	4 Tables	8 Tables	16 Tables
Vx[31:0]	Weight0	Valid	Valid	Valid	Valid	Valid
Vx[63:32]	Weight1		Valid	Valid	Valid	Valid
Vx[95:64]	Weight2			Valid	Valid	Valid
Vx[127:96]	Weigth3			Valid	Valid	Valid
Vx[159:128]	Weight4				Valid	Valid
Vx[191:160]	Weight5				Valid	Valid
Vx[223:192]	Weight6				Valid	Valid

Table 3-112. Histogram Weight Positions in Vector Register (continued)

Histogram Weight	Histogram Weights	1 Table	2 Tables	4 Tables	8 Tables	16 Tables
Vx[255:224]	Weight7				Valid	Valid
Vx[287:256]	Weight8					Valid
Vx[319:288]	Weight9					Valid
Vx[351:320]	Weight10					Valid
Vx[383:352	Weight11					Valid
Vx[415:384]	Weight12					Valid
Vx[447:416]	Weight13					Valid
Vx[479:448]	Weight14					Valid
Vx[511:480]	Weight15					Valid

3.18.4.16.7 Look Up Tables Read Instruction: LUTRD tbl_index, tbl_set, dst

The LUTRD instruction sends out address bus containing the indices, along with the content of the LTCR control register to L1D and receives back the corresponding table elements. The number of tables, size of the elements, and other parameters are specified by the LTCR control register. The LUTRD instruction returns up to 512 bits total. The requested elements from all tables are packed into the destination register from the least significant position. Figure 3-31 and Figure 3-32 show the examples of LUTRD for 4 parallel tables and half word element configuration, with and without interpolation.

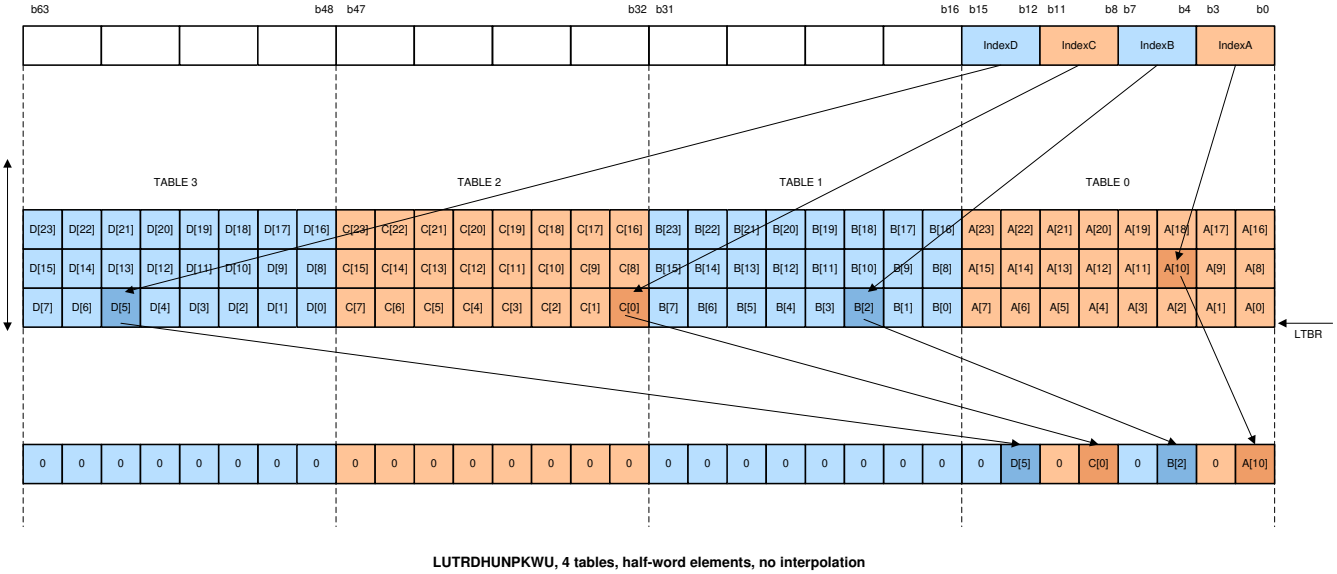


Figure 3-31. LUTRD, 4 Parallel Tables, Word Element Size, No Interpolation

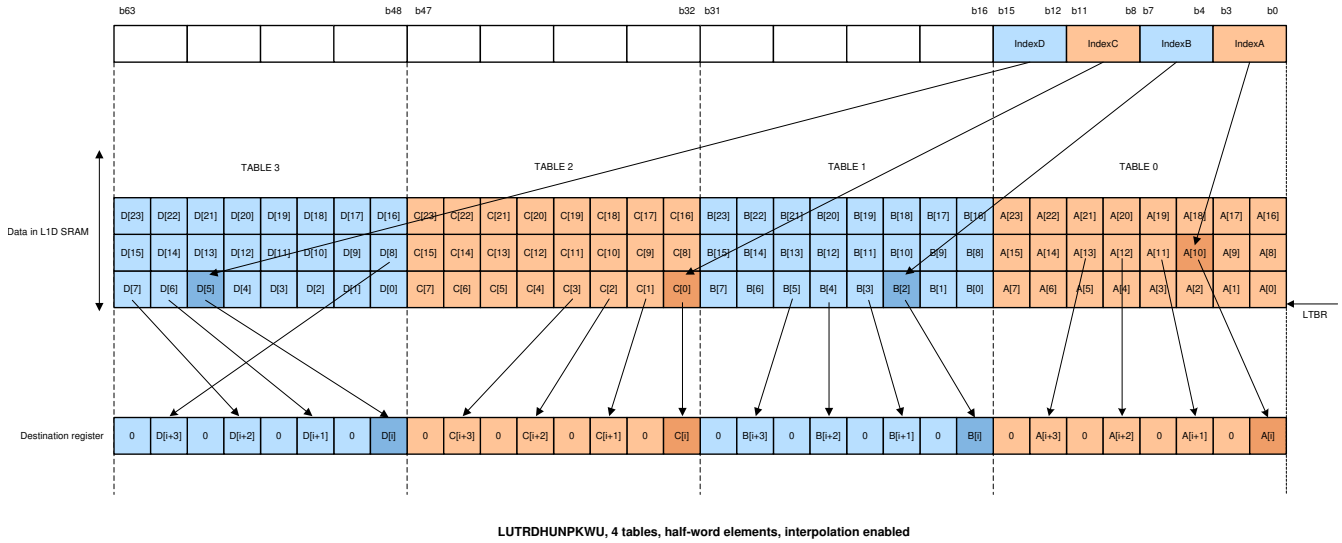


Figure 3-32. LUTRD, 4 Parallel Tables, Word Element Size, 4-Elements Interpolation

3.18.4.16.8 Look Up Tables Read with Unpack Option: LUTRD tbl_index, tbl_set, dst

The LUTRD instructions perform look up table reads and unpack the result elements to the specified precision, if specified in the PROMO field in its corresponding LUTCR.

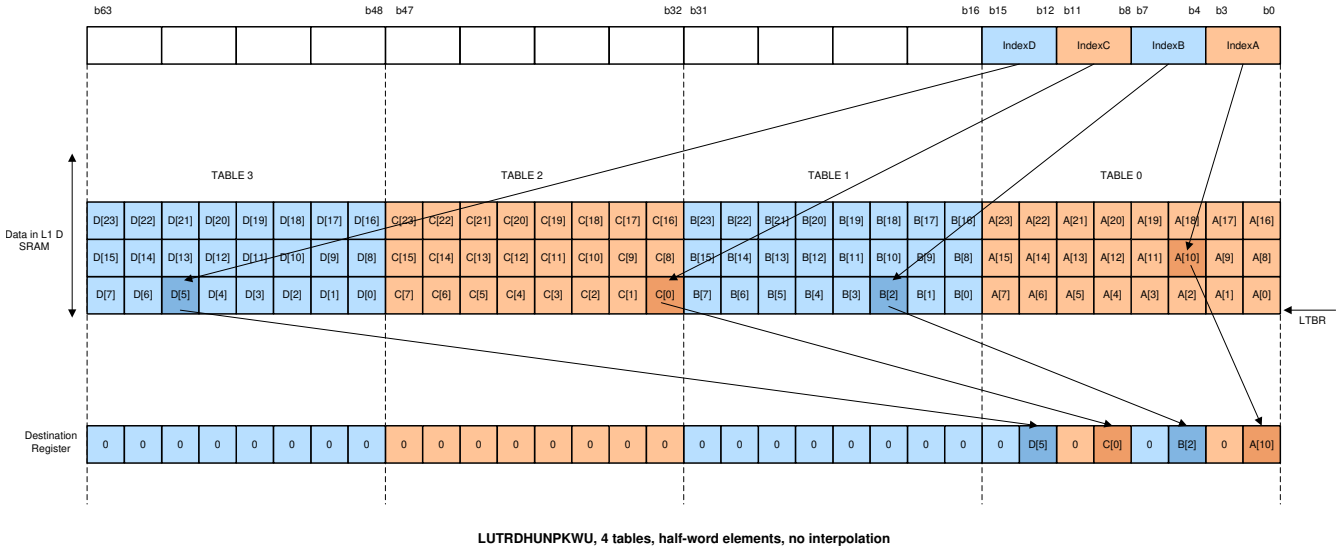
- The sign bit field in LTCR specifies whether the element is sign or zero-extended.
- The ESIZE field in LTCR specifies the source element size.
- In non-interpolation mode, LUTRDUNPK instructions just yield 1 element per table and sign/zero extended it to 64 bits.
- In interpolation mode, LUTRDUNPK performs sign or zero extend the elements to the specified precision.

The following are examples of different variations of look up table read and unpack instruction:

- Lookup Table Read and Unpack to half-words, signed
- Lookup Table Read and Unpack to half-words, unsigned
- Lookup Table Read and Unpack to words, signed
- Lookup Table Read and Unpack to words, unsigned
- Lookup Table Read and Unpack to double-words, signed
- Lookup Table Read and Unpack to double-words, unsigned

If the source element size is equal or larger than the result precision, then no unpack is performed.

Figure 3-33 and Figure 3-34 show the examples of lookup table read and unpack to words, unsigned, for 4 parallel tables and half word element configuration, with and without interpolation.



LUTRDHUNPKWU, 4 tables, half-word elements, no interpolation

Figure 3-33. Lookup Table Read, Half-Words Unpack to Words, Unsigned, 4 Parallel Table

		No of Parallel Tables				
Element Type	No of items Returned per Table Lookup	1	2	4	8	16
Word	1	X	X	X	X	X
	2	X	X	X	X	NS
	4	X	X	X	NS	NS
	8	X	X	NS	NS	NS
Element Type	No of items Returned per Table Lookup	1	2	4	8	16
Half-Word	1	X	X	X	X	X
	2	X	X	X	X	NS
	4	X	X	X	NS	NS
	8	X	X	NS	NS	NS
Element Type	No of items Returned per Table Lookup	1	2	4	8	16
Byte	1	X	X	X	X	X
	2	X	X	X	X	NS
	4	X	X	X	NS	NS
	8	X	X	NS	NS	NS
Color Legend to highlight max number of bits per lookup returned to CPU						
512	256	128	64	32	16	8
NS : Not Supported						

Figure 3-34. Lookup Table Read, Half-Words Unpack to Words, Unsigned, 4 Parallel Table, 4-Elements Interpolation

3.18.4.16.9 Look Up Tables Write Instruction: LUTWR tbl_index, tbl_set, write_data

The LUTWR instruction sends out an address bus containing 32-bit indices, the contents of the LTCR control register, and data bus containing the data to write to the look up tables. The number of tables, size of the elements, and other parameters are specified in the LTCR control register.

3.18.4.16.10 Look Up Table Initialization Instruction: LUTINIT tbl_index, tbl_set, write_data

L1D stores lookup and histogram elements into 16 64-bit physical memory banks or ways. The LUTINIT instruction is used to initialize those elements.

The LUTINIT instruction specifies data for a single way, with no duplication, and L1D replicates this data internally based on the number of ways the lookup table is configured for and stored in L1D SRAM using the entire available bandwidth of 1024 bits. Also, row addressing is needed to populate the entire lookup table. Each subsequent LUTINIT instruction specifies the address of the first element in the row being populated. Table 3-113 summarizes the row addressing for each of the supported LUT configurations, along with the element type.

Table 3-113. Row Addressing used for LUTINIT

No of Ways	Element Type	Index for LUTINT
16	word	0x0,0x2,0x4,0x6,0x8,0xA,0xC,0xE,...
	halfword	0x0,0x4,0x8,0xC,0x10,0x14,0x18,0x1C,...
	byte	0x0,0x8,0x10,0x18,0x20,0x28,0x30,0x38,0x40,...
8	word	0x0,0x4,0x8,0xC,0x10,0x14,0x18,0x1C,...
	halfword	0x0,0x8,0x10,0x18,0x20,0x28,0x30,0x38,0x40,...
	byte	0x0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,0x80,…
4	word	0x0,0x8,0x10,0x18,0x20,0x28,0x30,0x38,0x40,...
	halfword	0x0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,0x80,…
	byte	0x0,0x20,0x40,0x60,0x80,0x0,0xC0,0xE0,0x100,…
2	word	0x0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,0x80,…
	halfword	0x0,0x20,0x40,0x60,0x80,0x0,0xC0,0xE0,0x100,…
	byte	0x0,0x40,0x80,0xC0,0x100,0x140,0x180,0x1C0,0x200,0x240,…
1	word	0x0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,0x80,...
	halfword	0x0,0x20,0x40,0x60,0x80,0x0,0xC0,0xE0,0x100,...
	byte	0x0,0x40,0x80,0xC0,0x100,0x140,0x180,0x1C0,0x200,0x240,...

Note

LUTINIT duplication is only supported for the number of ways = 2, 4 ,8, or 16. For the number of ways = 1, there is no internal duplication done by L1D and only 512 bits can be written every cycle. LUTINIT for number of ways = 1 provides same bandwidth as a vector store.

Table 3-114. Number of Bits Sent Out by LUTINIT for Different Configurations

No of Ways	Element Type	Data Send Out (bits)	Data Written to L1D SRAM (bits)	Expansion Factor
16	word	64	1024	16
	halfword	64	1024	16
	byte	64	1024	16
8	word	128	1024	8
	halfword	128	1024	8
	byte	128	1024	8
4	word	256	1024	4
	halfword	256	1024	4
	byte	256	1024	4
2	word	512	1024	2
	halfword	512	1024	2
	byte	512	1024	2

Table 3-114. Number of Bits Sent Out by LUTINIT for Different Configurations (continued)

No of Ways	Element Type	Data Send Out (bits)	Data Written to L1D SRAM (bits)	Expansion Factor
1	word	512	512	1
	halfword	512	512	1
	byte	512	512	1

LUTINIT for 16-way lookup table – element type word:

```
LUTINIT D0, 0, B0; where D0=0x00 is index of 1st row element, 0 is table set, B0 = data[63:0]
LUTINIT D1, 0, B1; where D1=0x02 is index of 2nd row element, 0 is table set, B1 = data[127:64]
LUTINIT D2, 0, B2; where D2=0x04 is index of 3rd row element, 0 is table set, B2 = data[191:128]
LUTINIT D3, 0, B3; where D3=0x06 is index of 4th row element, 0 is table set, B3 = data[255:192]
```

After execution of these 4 instructions, L1D SRAM is initialized as shown in Figure 3-35.

	L1D SRAM BANKS															
	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
...																
...																
...																
ROW3	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]	D[255:192]
ROW2	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]	D[191:128]
ROW1	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]	D[127:64]
ROW0	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]	D[63:0]
	1024 BITS															

Figure 3-35. 16-Way Lookup Table – Element Type Word

LUTINIT for 1-way lookup table – element type byte:

```
LUTINIT D0, 0, VB0; where D0=0x00 is index of 1st row element, 0 is table set, VB0 = data0[511:0]
LUTINIT D1, 0, VB1; where D0=0x40 is index of 1st row element, 0 is table set, VB1 = data1[511:0]
LUTINIT D2, 0, VB0; where D0=0x80 is index of 1st row element, 0 is table set, VB2 = data2[511:0]
LUTINIT D2, 0, VB0; where D0=0xC0 is index of 1st row element, 0 is table set, VB3 = data3[511:0]
```

After execution of these 4 instructions, L1D SRAM is initialized as shown in Figure 3-36.

	L1D SRAM Banks														
	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1
...															
...															
...															
ROW3															
ROW2															
ROW1	DATA3[511:0]								DATA2[511:0]						
ROW0	DATA1[511:0]								DATA0[511:0]						
	512 bits								512 bits						
	1024 bits														

Figure 3-36. One-Way Lookup Table – Element Type Byte

3.18.4.16.11 Normal Histograms Instruction: HIST hist_index, hist_set

The HIST instruction performs normal histogram operations by incrementing the addressed bin entry by 1. The C7x supports 2, 4, 8, or 16 histogram bin configurations. The number of histograms, size of the elements, and other parameters are specified by the LTCR control register.

3.18.4.16.12 Weighted Histograms Instruction: WHIST hist_index, hist_set, his_weights

Similar to the HIST instruction, but instead of incrementing the addressed bin entry by 1, the WHIST incrementing the addressed bin entry by the weights contained in the weight data bus.

If the SAT bit field in the LTCR is on, the addressed bin storage elements are saturated to min/max values of the element data type, as follows:

- For unsigned byte, bin data is saturated to [0, 0xFF]
- For signed byte, bin data is saturated to [0x80, 0x7F]
- For unsigned half-word, bin data is saturated to [0, 0xFFFF]
- For signed half-word, bin data is saturated to [0x8000, 0x7FFF]
- For unsigned word, bin data is saturated to [0, 0xFFFF FFFF]
- For signed word, bin data is saturated to [0x8000 0000, 0x7FFF FFFF]

There are restrictions on the weight data type:

- Weights are always signed values.
- The weight size is limited to byte and half-word, and must be no bigger then the histogram bin. Thus, for byte-sized bins, you can only have byte-sized weights. For half-word or word-sized bins, you can have byte or half-word sized weights.

3.18.4.16.13 Parallel Scatter Instructions: LUTSCATTER tbl_index, tbl_set, write_data

This is similar to LUTWR, but interprets L1D as a single memory space instead of a number of tables.

- Offset vector is 16 lanes by 16-bit addresses aligned on 32-bit boundaries, same as LUTRD/LUTWR instructions.
- Payload vector is 16 lanes with the element size defined in the LUTCR.ESIZE field.
- LUTCR.VCOP bit to tell L1D to treat memory configuration:
  - 0: native c7x 16x64-bits memory configuration
  - 1: VCOP EVE 8x32-bit memory configuration

Native C7x mode: the LUTSCATTER operation is currently only supported for L1DSRAM space. In C7x, L1DSRAM consists of 16- to 64-bit wide banks. A key limitation for LUTSCATTER is that each of the byte addresses specified should map to different banks in L1DSRAM (16 × 64-bit banks). If multiple addresses map to the same L1D bank, there would be an addressing error generated, and an error signature would be captured inside the L1D status register. The address specified with LUTSCATTER is always a byte address and independent of the type of element (byte, halfword, word) being written. Also, the specified byte address is aligned to the element type by hardware, that is, non-aligned accesses are not supported with the LUTSCATTER operation. The exact mapping of each of the sub-stores (both address and data) to the respective L1D banks or ways is done by DMC and no permute operation is required in software. Native support for the LUTSCATTER operation is only for 16-way mode, and 8, 4 ,2 , and 1-way configurations are not supported.

VCOP EVE mode: the LUTSCATTER operation is defined in EVE the same way as C7x with respect to their memory system architecture, but supports porting EVE code on the C7x memory system due to differences in the EVE and C7x bank structure. The EVE memory system consists of 8- to 32-bit banks, whereas the C7x memory system has 16- to 64-bit banks. One way to port the existing EVE code is to map the EVE PSCATTER instruction on 8-way {way,offset} pairs instead of 8 EVE byte addresses. When EVE byte address are mapped the corresponding way and offset pair for an 8-way lookup operation, they can be used directly to perform the scatter operation on the C7x memory system as a way, and offset values are not impacted by the banking differences between EVE and C7x. Mathematical calculation of the way and offset from the byte address gets involved if done entirely inside CPU, so hardware support is provided for address translation and permute of data, and offset values.

16-way EVE PSCATTER support: Even though EVE only had support for 8-way pscatter, for EVE translation mode in C7x, TI provides support for a 16-way pscatter mode. For this mode, EVE byte address are translated to the way and offset, assuming that in this case EVE has 16- to 32-bit banks. When the way and offset pair values are known, the hardware commits the pscatter instruction on the C7x memory system, assuming the same values of way and offset value.

Software can enable the EVE/Legacy capability mode in the LUTCR register, and hardware provides support for translation of EVE byte addresses to map the scatter operation to C7x.

To enable legacy support, the LTCR.VCOP bit should be set in the LUTCR register; by default, C7x native mode is enabled.

The behavior of LUT base address and LUT size is also applicable for LUTSCATTER instruction. LUT size defines the size of SRAM on which the scatter operation is performed. If the offset specified by LUTSCATTER

overshoots the defined size, hardware aliases or wraps around the offset to contain access within the defined buffer for the pscatter operation.

3.18.4.16.14 Look Up Table and Histogram Instruction Encoding

- For all of the LUT-family instructions that take a vector of indices (LUTRD, LUTWR, HIST, WHIST), the vector of indices specified by src1 is read through the same register file read port as the cross-path operand for an A-side instruction reading a B-side register.
- LUTRD is similar to vector loads, therefore it can only be issued on .D1, and is indicated as .D1X. The “X” applies to the vector of indices and not the data.
- LUTWR, HIST, and WHIST are similar to vector stores, therefore they can only be issued on .D2 and are indicated as .D2X. The “X” applies to the vector of indices and not the data.
- LUTINIT does not take a vector of indices. Its src1 always comes from a scalar register on the A-side (A0-A15, D0-D15). It does not (and cannot) use a cross path for its scalar index operand. It is always on .D2, no X.
- Cross-path sharing between these LUT/HIST instructions and other instructions is allowed. The lower 64 bits of the vector gets shared.
- The outbound data associated with LUT-family instructions that look approximately like a store (LUTINIT, LUTWR, WHIST) use the same resource as a vector store. This is why they are on D2/D2X. The X associated with LUTWR and WHIST only applies to the vector of indices and not the data.
- The inbound data associated with LUTRD uses the same path as a vector load. This is why it is on D1X.

A summary of the src1 encoding for LUT/HIST instructions is listed in Table 3-115.

Table 3-115. SRC1 Encoding for LUT/HIST Instructions

SRC1 Encoding	LUTRD	LUTWR, HIST, WHIST	LUTINIT
Issue Slot	.D1X Only	.D2X Only	.D2 Only
0xxxx	Reserved	Reserved	A0-A15
1xxxx	VB0-VB15	VB0-VB15	D0-D15

3.18.4.16.15 Summary of Look Up Table and Histogram Features

Summary:

- Up to 4 sets of look up table or histogram can be specified simultaneously.
- Look up table elements and histogram data bin can be signed or unsigned, bytes, half-words, or words. These properties are specified by the configuration registers LTCR.
- The base address is contained in the corresponding LUTBR configuration registers.
- Indices are always treated as unsigned values at 32-bit lanes of the source register. However, only the least significant 16 bits of each index is needed due to limited L1D size limit.
- Histogram weights are at 32-bit lanes of the source register.
- Histogram weights can only be signed bytes or signed half-words.
- Histogram bin data saturates to the minimum or maximum values of its data bin type.

3.18.4.17 Correlation Instructions

Correlation instruction naming convention is shown in Table 3-116.

Table 3-116. Correlation Instruction Naming Convention

Full Name	Vector	Base Operation	Masked	Offset	Pos/Neg	Input Ways	Input Precision	Output Ways	Output Precision
VCDOTPM32 OPN32B32H	V	CDOTP	M	32	PN	32	B	32	H

Field description:

- Base operation: Describe the underlined operation of the instruction – CDOTP: Complex dot product
- Masked: Output is masked or not
- Offset: The amount of offset used to slide the multiplier operand
- Pos/Neg: Multiplying in with +/- 1 or +/-j
- Input ways: The number of SIMD input elements
- Input precision width: Precision of input, can be B(yte), H(alf-word), W(ord), D(ouble-word)

- Output ways: The number of SIMD output elements
- Output precision width: Precision of output, can be B(yte), H(alf-word), W(ord), D(ouble-word)

3.18.4.17.1 Sum of Absolute Difference Instructions

Sum of absolute difference (SAD) instructions are mainly used for video motion estimation. It performs subtraction and absolute difference between reference pixels and current image pixels. Input data size is unsigned 8 bit or unsigned 16 bit, and motion estimation output is unsigned 16 bit or 32 bit. In the following sections, different flavors of SAD instructions are explained.

3.18.4.17.2 VSADM8O16B32H: 16 Ways SAD of 8-Bit Input Precision with Mask and 8-Bit Offset and 32 Half-Word Output

This double vector instruction implements a sliding window correlation using the sum of absolute differences of the inputs. The inputs are 8-bit pixel values. The difference is taken between 16 current image pixels in OP2 and 16 reference pixels in OP1, and then the absolute value of each difference, which can be masked off using the mask bit from OP1[143:128], accumulated across 16 pixels. This calculation is repeated for 16 pixels in OP1 against 32 sets of 16 pixels in OP2 with 32 offset values. For example, an 8×8 block search can be implemented from a 16×16 block search by masking off the upper 8 pixels contribution using a mask.

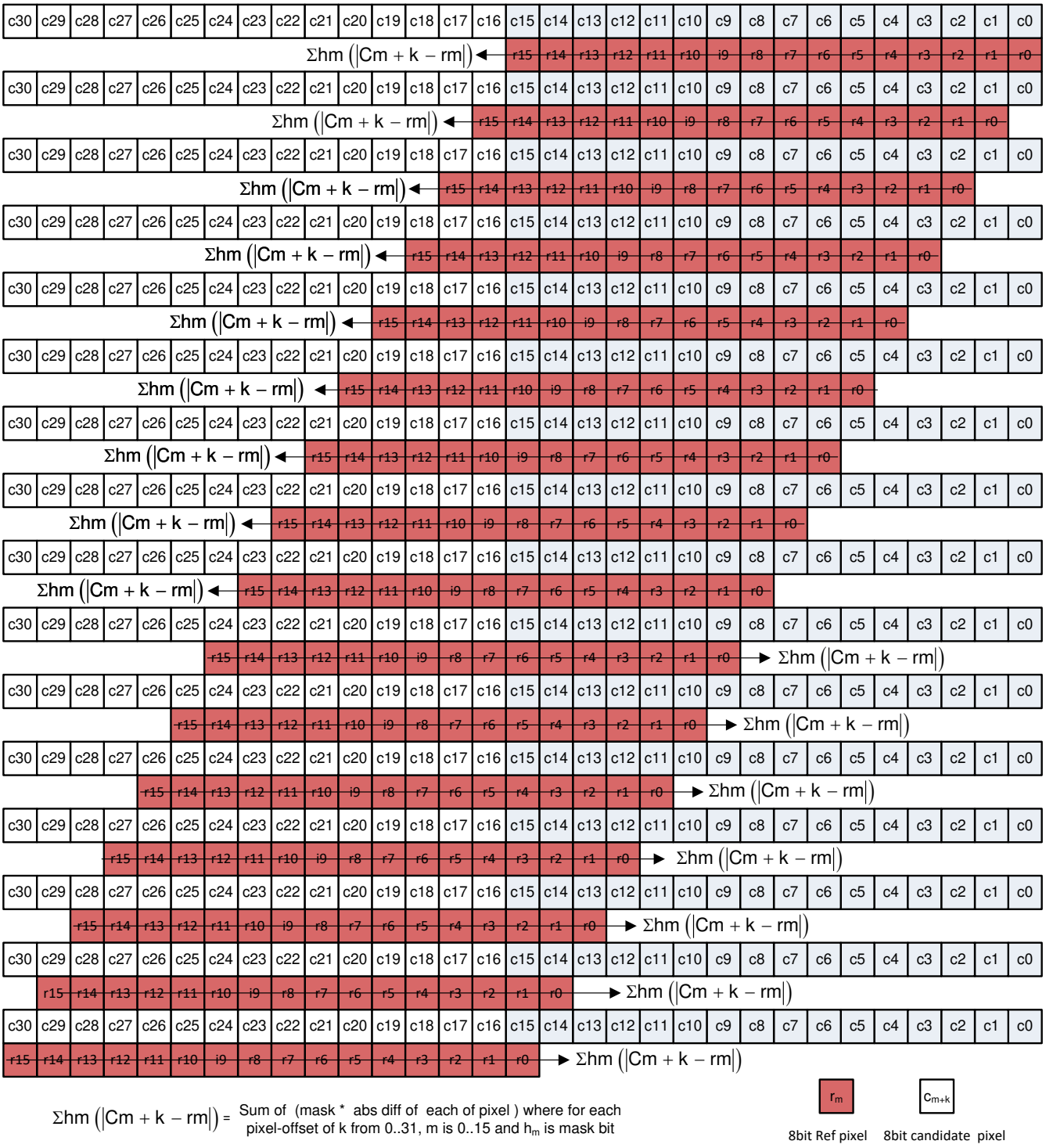


Figure 3-37. Lower Half of the Pixels of VSADM8O16B32H

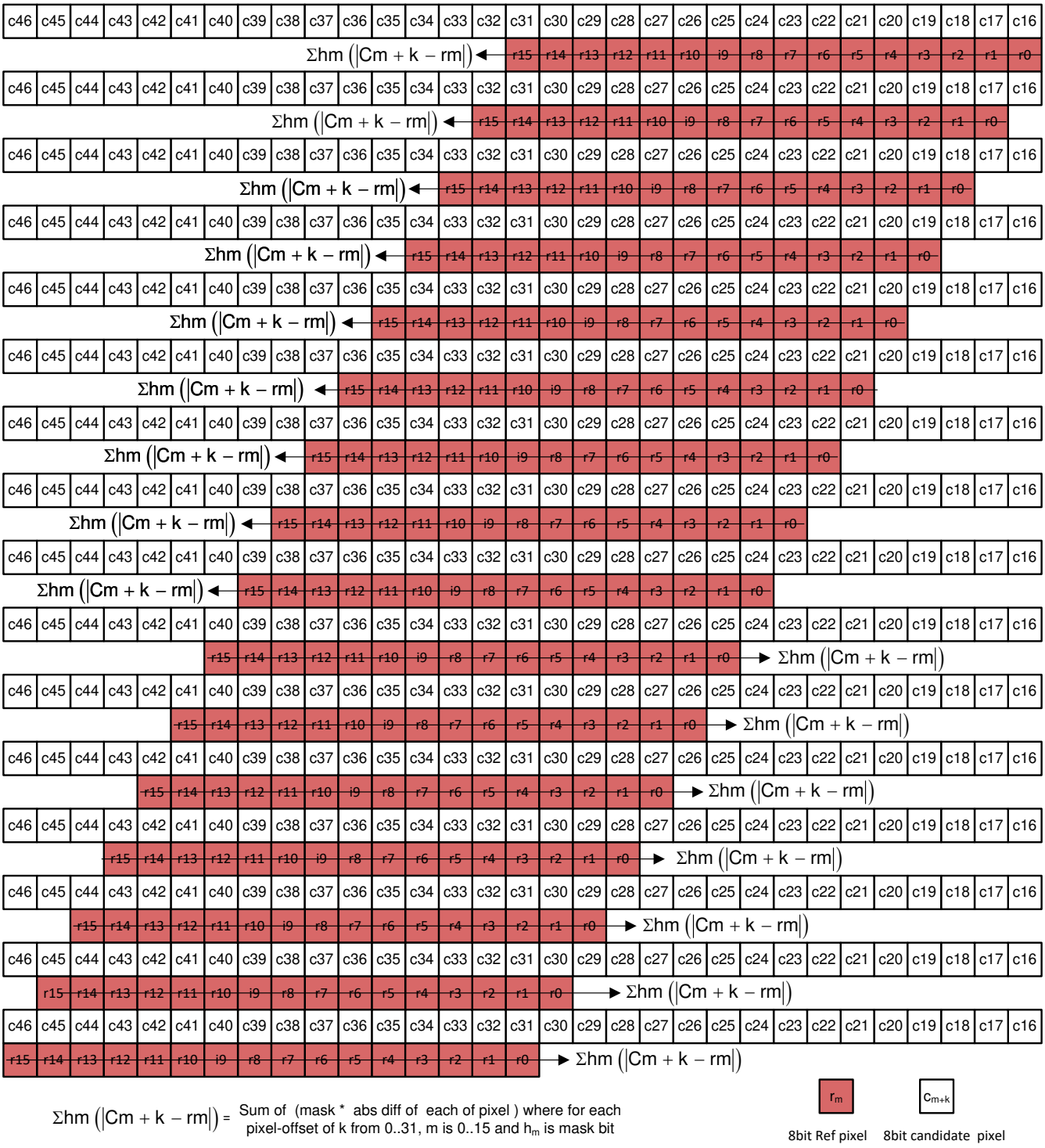


Figure 3-38. Higher Half of the Pixels of VSADM8O16B32H

3.18.4.17.3 VSAD8O16B32H - 16 Ways SAD of 8-Bit Input Precision with 8-Bit Offset and 32 Half-Word Output

This double vector instruction implements a sliding window correlation using the sum of absolute differences of the inputs. The inputs are 8-bit pixel values. The difference is taken between 16 input image pixels in OP2 and 16 reference pixels in OP1, and then the absolute value of this difference is accumulated across 16 pixels. This calculation is repeated for 16 pixels in OP1 against 32 sets of 16 pixels in OP2 with 32 offset values. This instruction is identical to VSADM8O16B32H, except that there are no mask bits applied to the calculations.

3.18.4.17.4 DVSADM16O8H16W: 8 Ways SAD of Half-Word Input Precision with Mask and 16-Bit Offset and 16 Word Outputs

This double vector instruction implements a sliding window correlation using the sum of absolute differences of the inputs. The inputs are 16-bit pixel values. The difference is taken between 8 current pixels in OP2 and 8 reference pixel in OP1[127:0], and then the absolute value of each difference, which can be masked off using mask bits from OP1[135:128], accumulated across 8 pixels. For example, with mask bits a 4×4 block search can be implemented from an 8×8 block search.

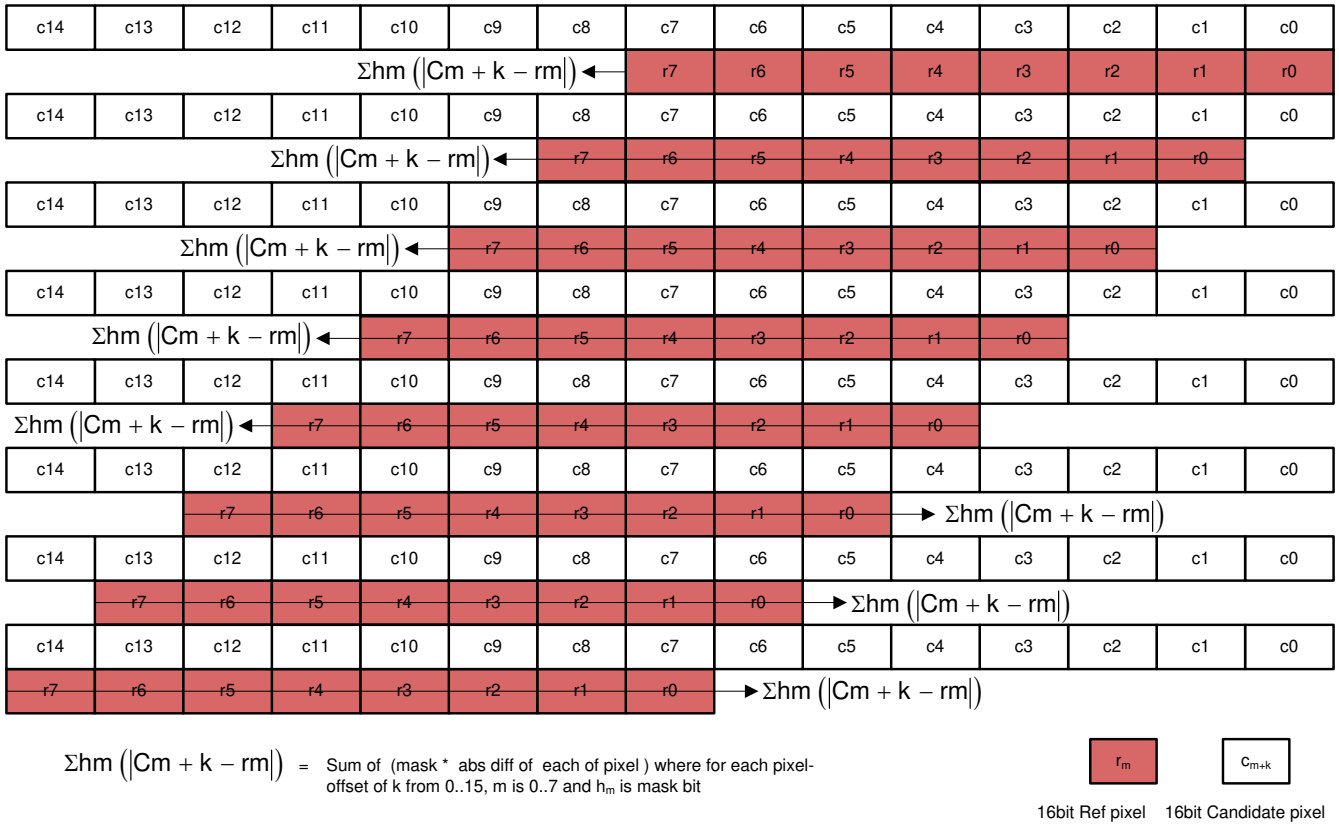


Figure 3-39. Lower Half of the Pixels of VSADM16O8H16W

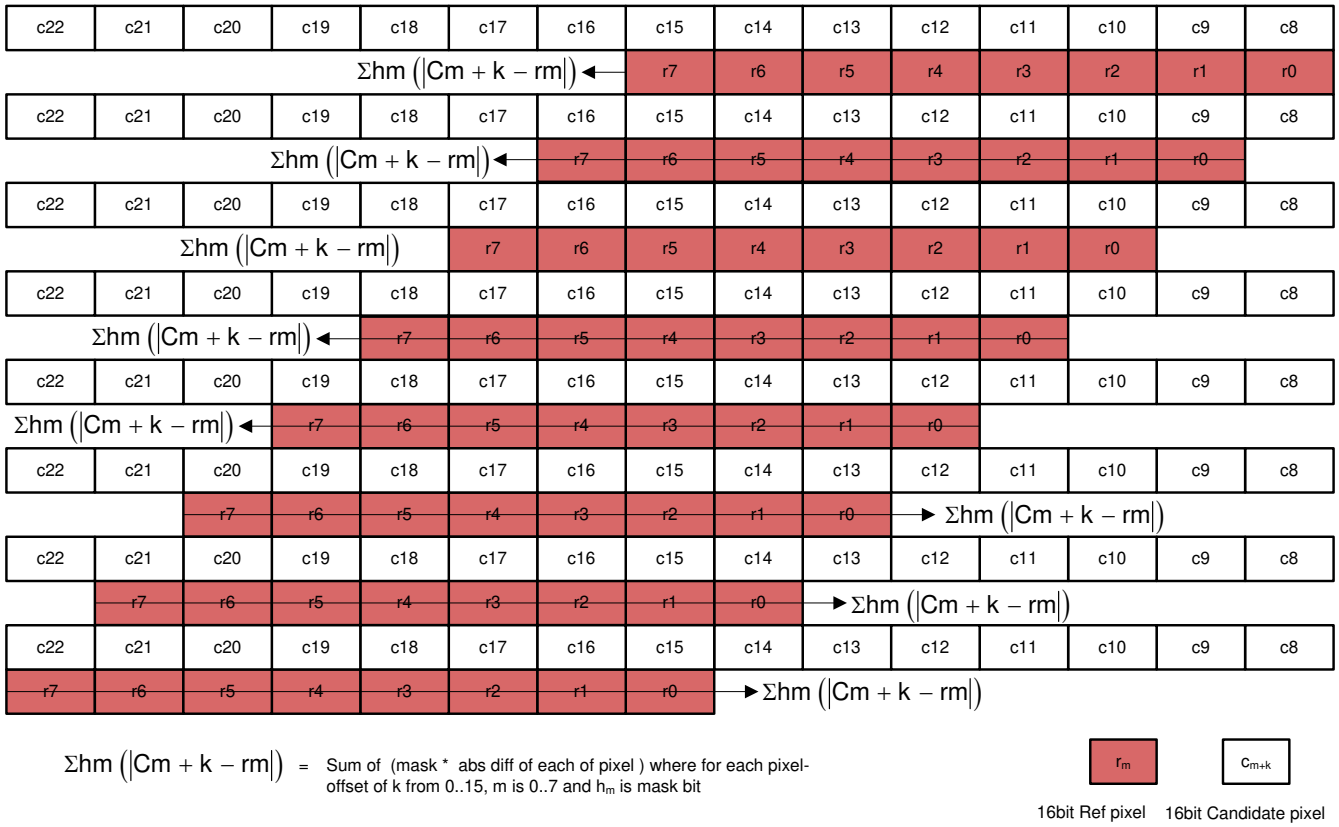


Figure 3-40. Higher Half of the Pixels of VSADM16O8H16W

3.18.4.17.5 DVSAD16O8H16W - 8 Ways SAD of Half-Word Input Precision with 16-Bit Offset and 16 Word Outputs

This double vector instruction implements a sliding window correlation using the sum of absolute differences of the inputs. The inputs are 16-bit pixel values. The difference is taken between 8 input image pixels in OP2 and 8 reference pixel in OP1[127:0], and then the absolute value of this difference is accumulated across 8 pixels. This calculation is repeated for 8 pixels in OP1 against 16 sets of 8 pixels in OP2 with 16 offsets.

This instruction is identical to DVSADM16O8H16W, except that there are no mask bits applied to the calculations.

3.18.4.18 Vector Byte Permute Instructions

3.18.4.18.1 Vector Swap Instructions

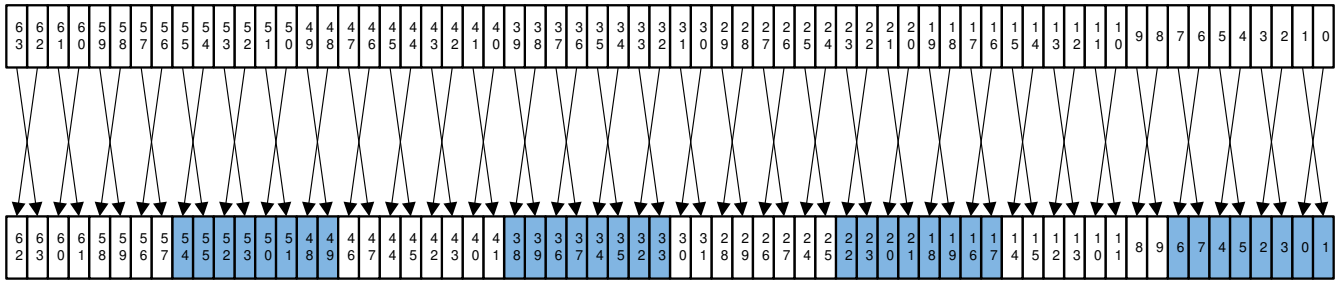
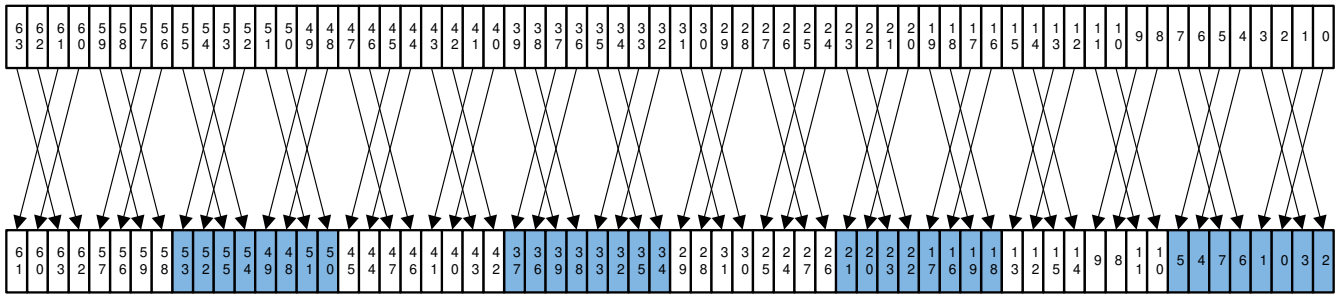
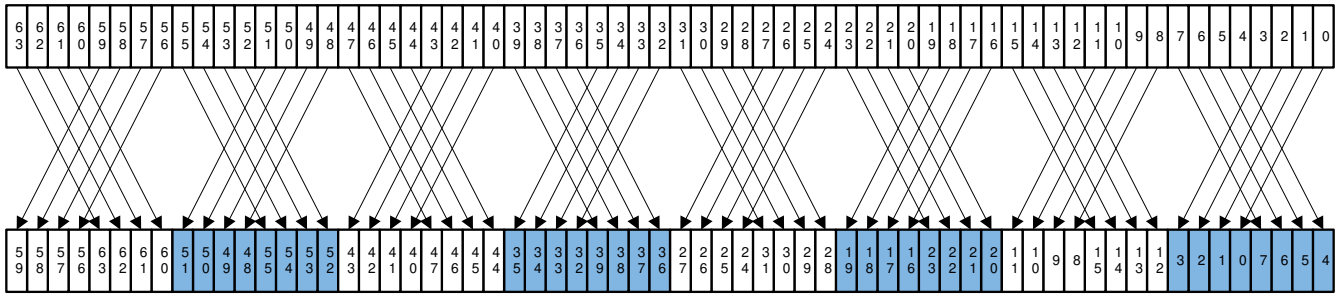


Figure 3-41. VSWAPB



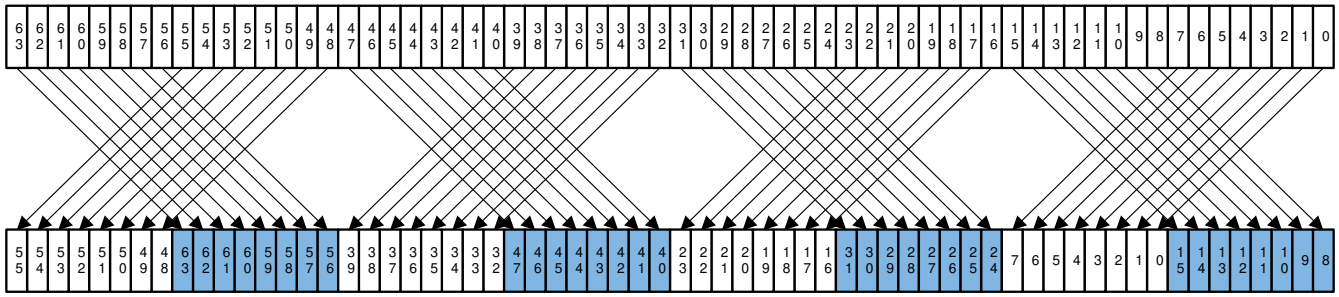
VSWAPH

Figure 3-42. VSWAPH



VSWAPW

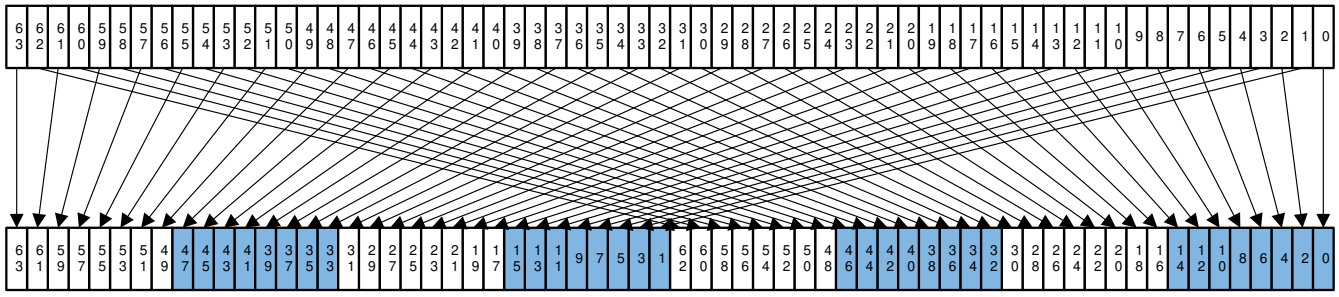
Figure 3-43. VSWAPW



VSWAPD

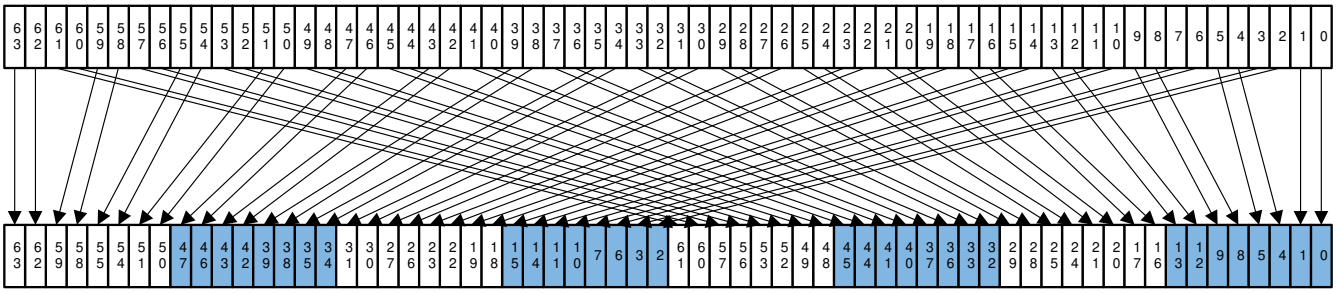
Figure 3-44. VSWAPD

3.18.4.18.2 Vector Deal2 Instructions



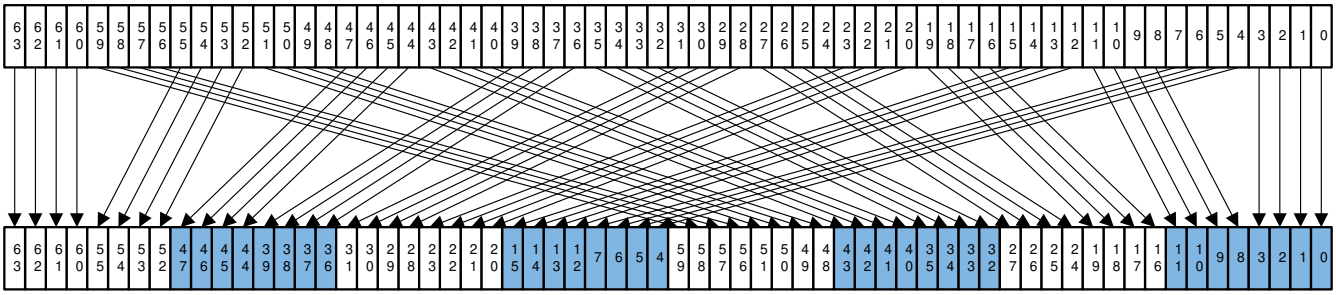
VDEAL2B

Figure 3-45. VDEAL2B



VDEAL2H

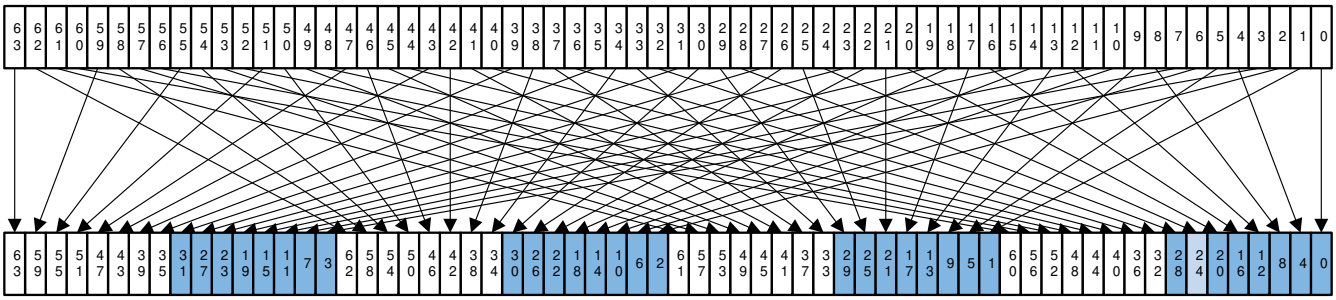
Figure 3-46. VDEAL2H



VDEAL2W

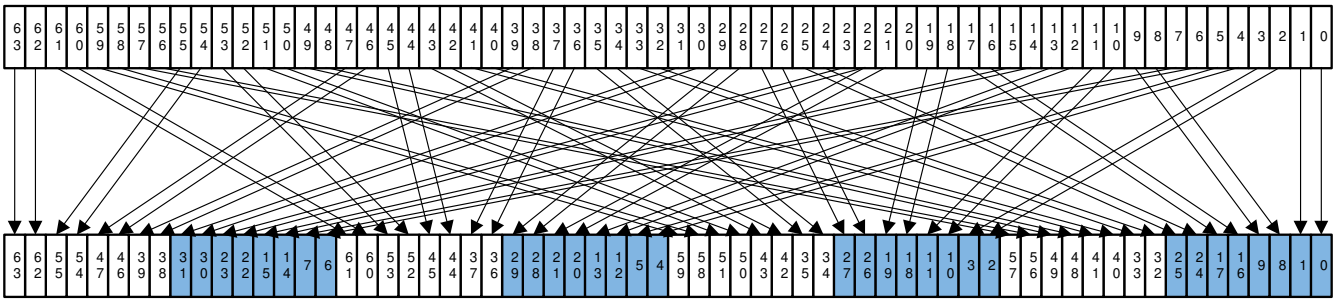
Figure 3-47. VDEAL2W

3.18.4.18.3 Vector Deal4 Instructions



VDEAL4B

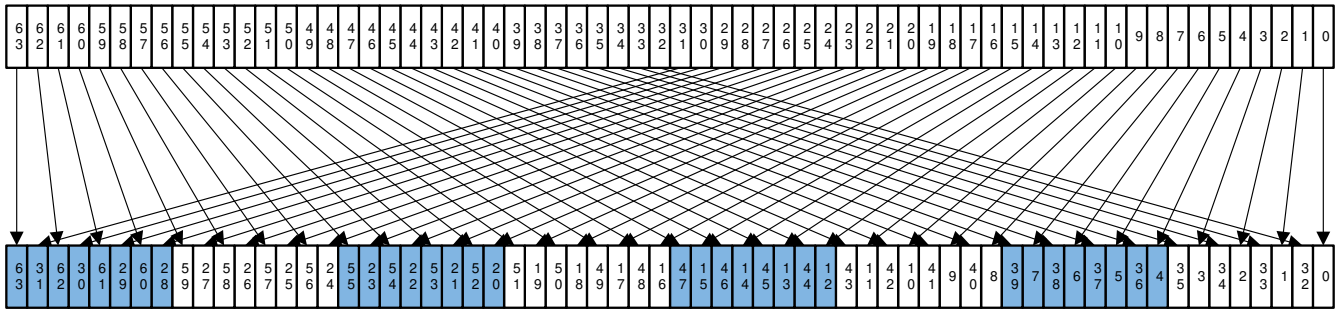
Figure 3-48. VDEAL4B



VDEAL4H

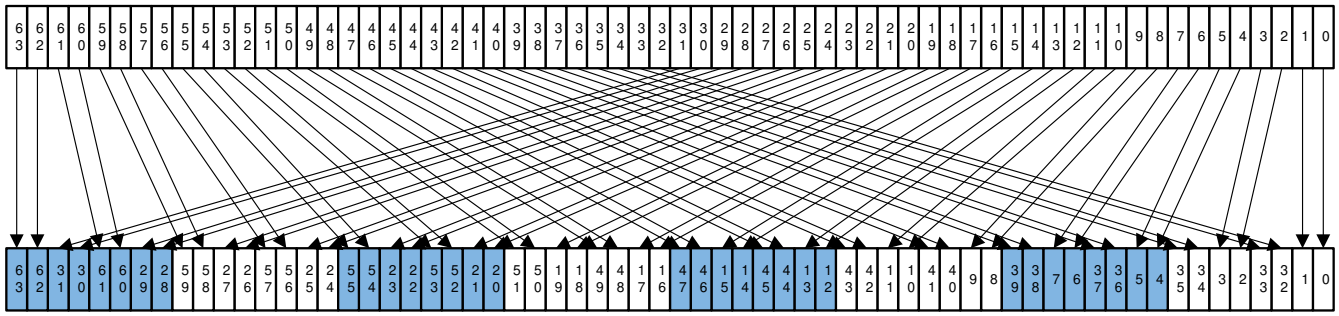
Figure 3-49. VDEAL4H

3.18.4.18.4 Vector Shuffle2 Instructions



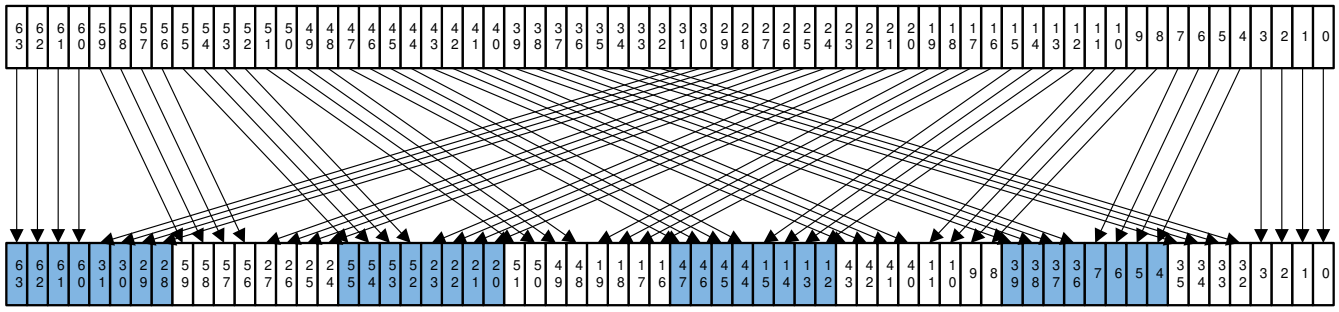
VSHFL2B

Figure 3-50. VSHFL2B



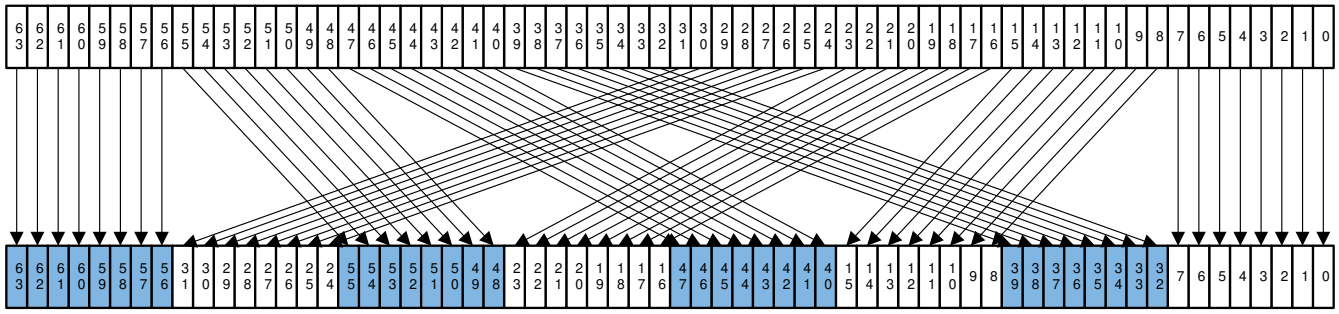
VSHFL2H

Figure 3-51. VSHFL2H



VSHFL2W

Figure 3-52. VSHFL2W



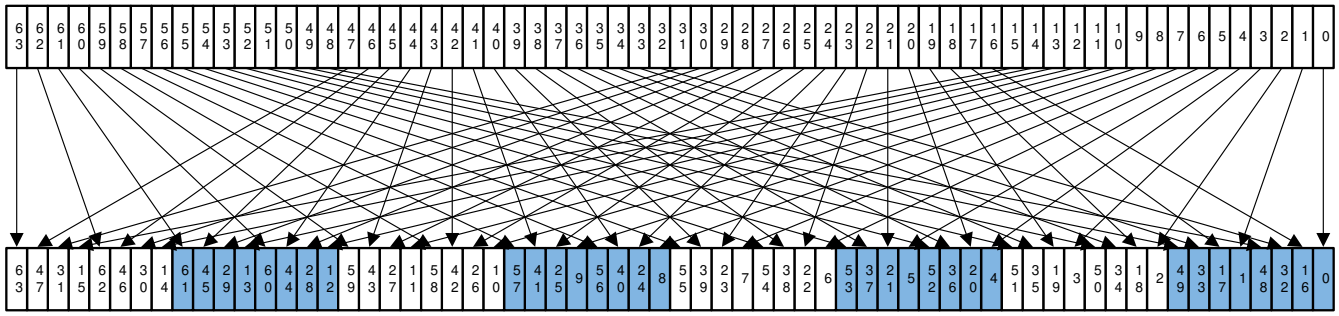
VSHFL2D

Figure 3-53. VSHFL2D

Note

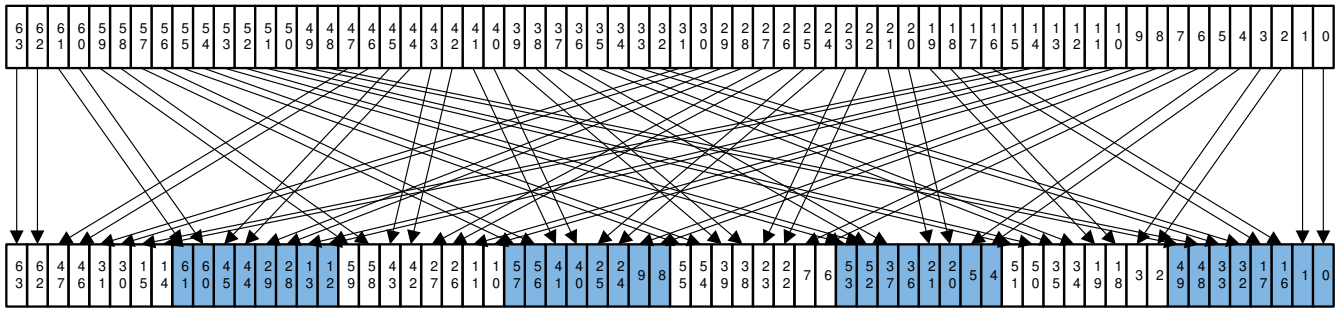
64-byte VSHFL2D is the same as 64-byte VDEAL4D.

3.18.4.18.5 Vector Shuffle4 Instructions



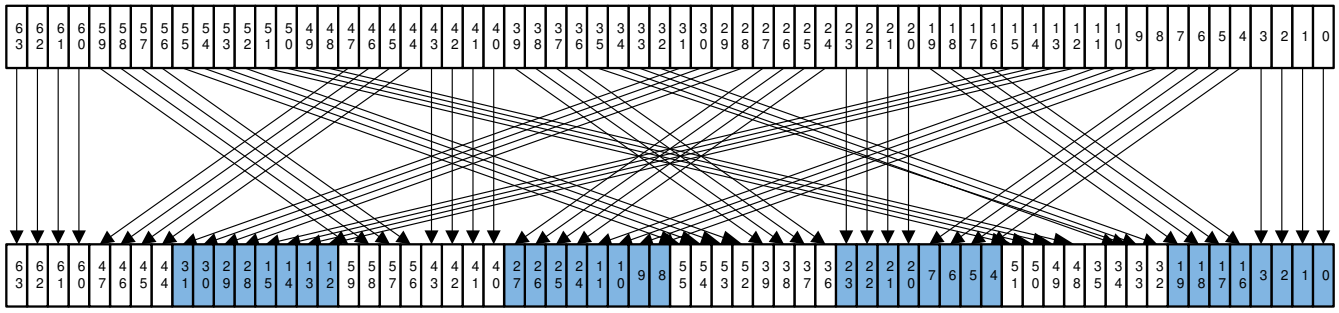
VSHFL4B

Figure 3-54. VSHFL4B



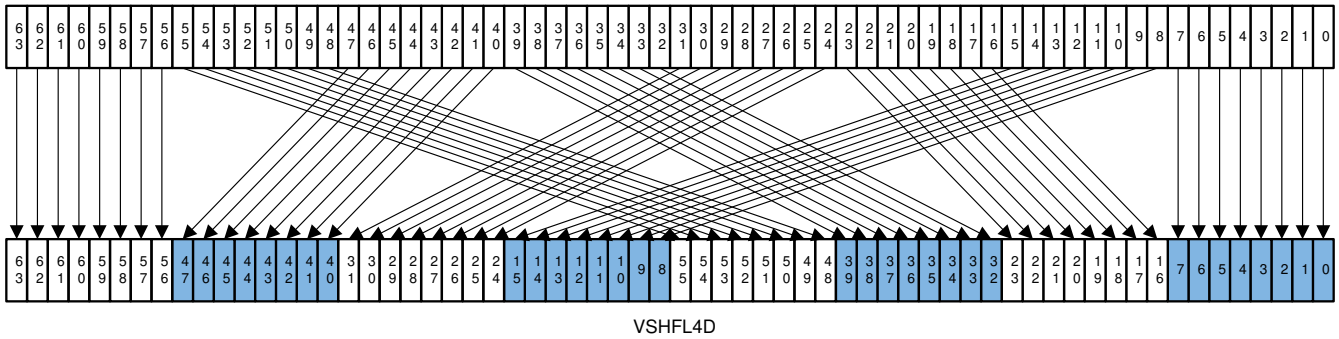
VSHFL4H

Figure 3-55. VSHFL4H



VSHFL4W

Figure 3-56. VSHFL4W



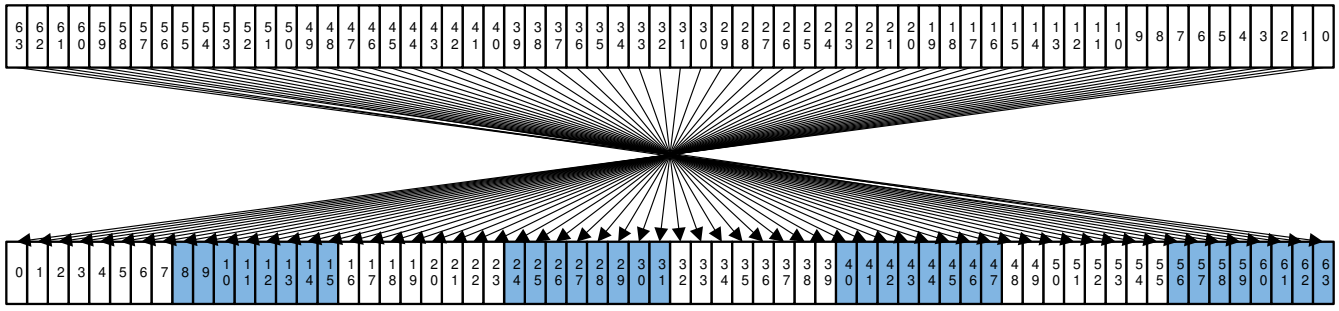
VSHFL4D

Figure 3-57. VSHFL4D

Note

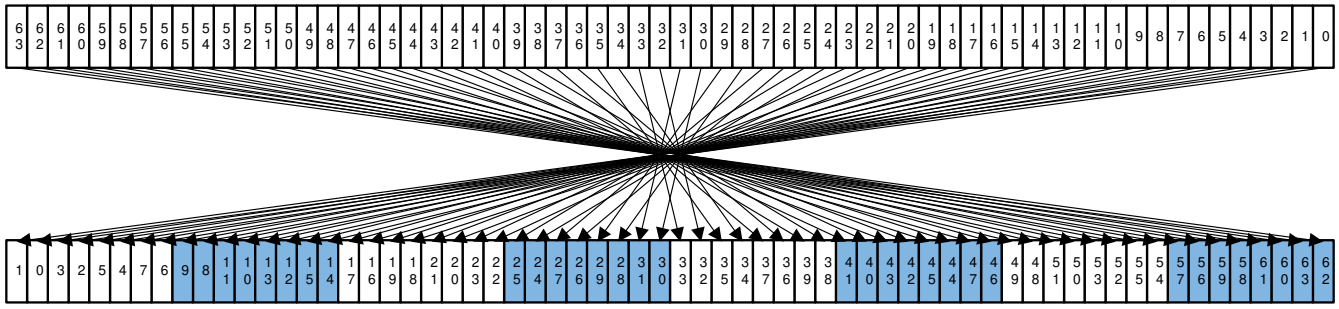
64-byte VDEAL4W is the same as VSHFL4W, and VDEAL4D is the same as VSHFL2D.

3.18.4.18.6 Vector Reverse Instructions



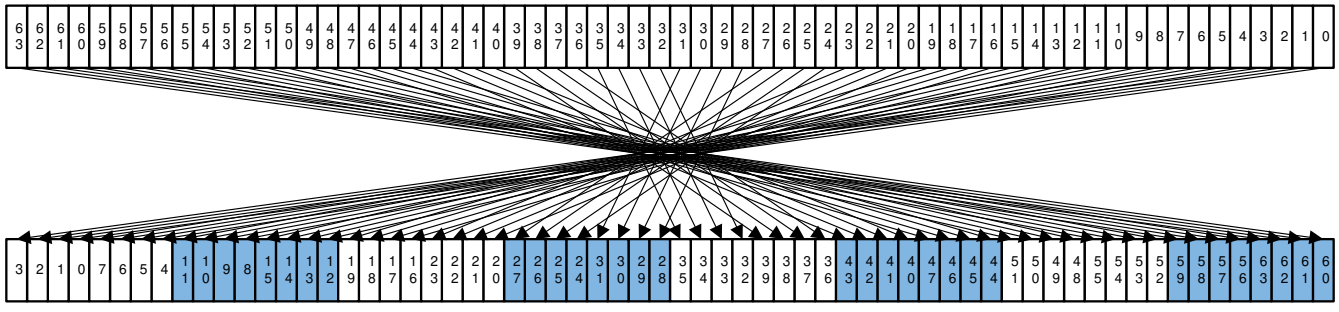
VREVERSEB

Figure 3-58. VREVERSEB



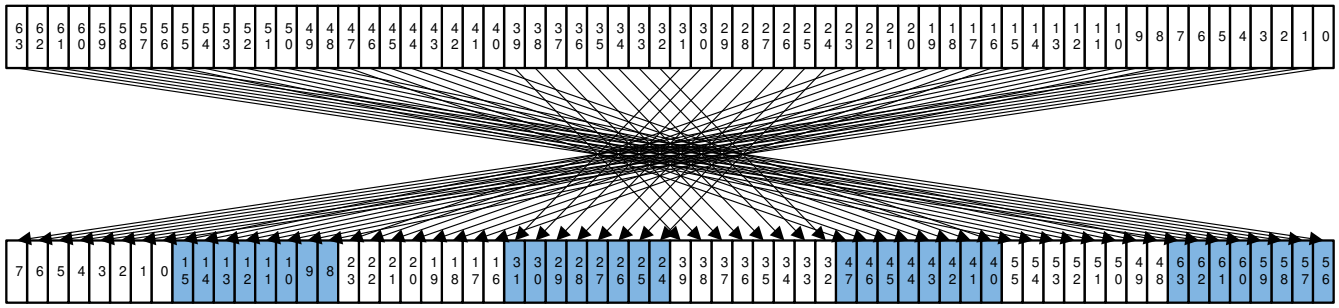
VREVERSEH

Figure 3-59. VREVERSEH



VREVERSEW

Figure 3-60. VREVERSEW



VREVERSED

Figure 3-61. VREVERSED

3.18.4.18.7 General Vector Byte Permute Instruction VPERM

The VPERM instruction enables all byte permute patterns, as well as clear or set. It takes in two vector operands.

The vector from Src2 is the data to be permuted, and the vector is the control bits for the permuted data. The control vector can come from either the Src1 or CUCR registers. Each byte of the permuted data is controlled by the corresponding byte of the control data. The control bytes are encoded as follows:

- $Src1\_byte_{63\text{-}0}[5:0]$: select 1 out of 64 available data bytes from Src2
- $Src1\_byte_{63\text{-}0}[7:6]$:
  - 00= no-change
  - 01= output all 0's for that byte
  - 10= output all 1's for that byte
  - 11= fill with the sign bit (bit 7th) of the byte selected by $Src1\_byte_{63\text{-}0}[5:0]$

3.18.4.18.8 VPERMEE, VPERMEO, VPERMOO

These instructions are similar to VPERM, but the data input is formed by concatenating the even elements of src1 with the even elements of src2. The src1 elements form the upper half and src2 elements form the lower half of the input to be permuted.

The control vector for these class of instructions come from the CUCR control register.

The element sizes can be bytes, half-words, words, or double-words.

3.18.4.19 FIR Instructions

The FIR instructions are dual-issued instructions, meaning that they must be executed on both .M2 and .N2 units. The operands of the FIR instructions can only come from SE0 or SE1 registers.

Because the FIR instruction uses SE0 and SE1 as source registers, another instruction which also uses SE0 or SE1 registers in the same execute packet may not be issued in parallel. Table 3-117 shows the restriction of whether the SE registers can be used in parallel.

Table 3-117. SE Register Usage Restriction for Instructions in Parallel with FIR Instructions

Instruction	Parallel Use of Same SE	Parallel Use of Different SE	Other Comments
VFIR8DS4HD .N2 vec_pair, se_pair, vec_pair	ILLEGAL	NA / not possible	Dual-issue, B-side only
VFIR8DS2HD .N2 vec_pair, se_pair, vec_pair	ILLEGAL	NA / not possible	Dual-issue, B-side only
VFIR8HD .N2 vec_pair, se_single, vec_pair	ILLEGAL	Legal	Dual-issue, B-side only
VFIR8DS4HW .N2 vec_pair, se_pair, vec_single	ILLEGAL	NA / not possible	Dual-issue, B-side only
VFIR8DS2HW .N2 vec_pair, se_pair, vec_single	ILLEGAL	NA / not possible	Dual-issue, B-side only
VFIR8HW .N2 vec_pair, se_single, vec_single	ILLEGAL	Legal	Dual-issue, B-side only

Table 3-117. SE Register Usage Restriction for Instructions in Parallel with FIR Instructions (continued)

Instruction	Parallel Use of Same SE	Parallel Use of Different SE	Other Comments
VFIR4DS2HW .N2 vec_single, se_pair, vec_pair	ILLEGAL	NA / not possible	Dual-issue, B-side only
VFIR4HW .N2 vec_single, se_pair, vec_pair	ILLEGAL	NA / not possible	Dual-issue, B-side only

3.18.4.20 MATMPY Instructions

The MATMPY instructions are dual-issued instructions, meaning that they required to be executed on both .M2 and .N2 units. The operands of the FIR instructions can only come from SE0 and SE1 registers.

Because the MATMPY instruction uses SE0 and SE1 as source registers, another instruction which also uses SE0 or SE1 registers in the same execute packet may not be issued in parallel. Table 3-118 shows the restriction of whether the SE registers can be used in parallel.

Table 3-118. SE Register Usage Restriction for Instructions in Parallel with MATMPY Instructions

Instruction	Parallel Use of Same SE	Parallel Use of Different SE	Other Comments
VMATMPYHD .N2 se_single, se_single, vec_pair	ILLEGAL	ILLEGAL	Dual-issue, B-side only, may use both SE
VMATMPYHW .N2 se_single, se_single, vec_single	ILLEGAL	ILLEGAL	Dual-issue, B-side only, may use both SE

3.18.4.21 Sixteen Elements Sort Instructions

These instructions are useful in sorting elements through the bi-tonic sorting network.

3.18.4.22 Instructions with Variable Latency

These instructions have an architecture cycle latency of 1, regardless of whether it is issued in protected or unprotected mode.

The actual clock cycles needed for producing the quotient or the remainder varies are dependent on the source data. The hardware automatically inserts stall cycles if the result of the DIV or MOD instructions are needed before the computation is finished.

3.18.4.22.1 Divide Instructions (DIVW, DIVUW, DIVDW, DIVUDW)

The C7x supports signed and unsigned divisions between two 32-bit words (DIVW, DIVUW) or division between a 64-bit double word and a 32-bit word (DIVDW, DIVUDW).

The divide by 0 is an special condition. Its handling depends on whether the divide by zero exception is enabled, and is controlled by the bit DIV0EN in the FPCR.

- If FPCR.DIV0EN = 1, take exception and no result is written to the destination.
- If FPCR.DIV0EN=0, then the result is written to the destination as follows:
  - unsigned value divide by 0: Quotient = Largest unsigned integer value, set FSR.SAT = 1
  - positive value divide by 0: Quotient = Largest positive integer value, set FSR.SAT = 1
  - negative value divide by 0: Quotient = Smallest negative integer value, set FSR.SAT=1
  - 0/0: Quotient = 0, set FSR.SAT=1

3.18.4.22.2 Modulus Instructions (MODW, MODUW, MODDW, MODUDW)

Similarly, the C7x supports signed and unsigned modulo operations between two 32-bit words (MODW, MODUW) or between 64-bit double word dividend and 32-bit word divisor (MODDW, MODUDW).

The divide by 0 is an special condition. Its handling depends on whether the divide by zero exception is enabled, and is controlled by the bit DIV0EN in the FPCR.

- If FPCR.DIV0EN = 1, take exception and no result is written to the destination.
- If FPCR.DIV0EN=0, then the result is written to the destination as follows:
  - unsigned value divide by 0: Remainder = 0, set FSR.SAT = 1
  - positive value divide by 0: Remainder = 0, set FSR.SAT = 1
  - negative value divide by 0: Remainder = 0, set FSR.SAT=1
  - zero divide by zero: Remainder = 0, set FSR.SAT=1

3.18.4.23 Debug Instructions

3.18.4.23.1 SWBP – Software Breakpoint

There are two types of SWBP instructions. When the debugger is enabled, the two SWBP instructions have the same behaviors and cause the CPU to halt execution prior to it entering the E1 pipeline phase. The debug logic is signalled that a software breakpoint has been encountered. When the debugger is disabled, the two SWBP instructions are treated differently by the CPU:

- ESWBP: Embedded Software Breakpoint. Treated as a NOP if encountered outside of a debugging session.
- DSWBP: Debugger Software Breakpoint. Treated as illegal opcode exception if encountered outside of a debugging session.

The use of these instructions is intended solely for debug purposes.

3.18.4.23.2 EDBG – Enable Emulator Debug Events

The EDBG instruction enables emulator debug events in the current cycle and sets TSR.DBGM to 0. This instruction executes unconditionally.

3.18.4.23.3 DDBG – Disable Emulator Debug Events

The DDBG instruction disables emulator debug events in the current cycle and sets TSR.DBGM to 1. This instruction executes unconditionally.

3.18.4.23.4 MARK – Mark Current Cycle for Emulator Use

The MARK instruction signals the debug logic that the 5-bit id value is present at the CPU boundary. The 5-bit value is presented in E2.

The MARK instruction with id value of 00000 (MARK 0) when placed in parallel with a LD or ST operation results in the corresponding memory access submitted to the memory system with the interest flag set.

This instruction executes unconditionally.

3.18.4.24 HWA Instructions

The hardware accelerator (HWA) is tightly coupled with the C7x CPU. The HWA accepts commands and operands from the C7x CPU, performs specific computation tasks, and returns the result back to the C7x CPU.

General stages of HWA operation:

1. Starts HWA: The CPU sends the HWAOPEN command to the HWA to start the HWA. The command includes relevant information to program the HWA.
2. Loads operands: A series of HWALD operations to load the HWA matrix operands.
3. Computes: When the HWA operands are loaded, the HWAOP instructions are issued to start computation.
4. Transfers: The computed data is transferred to the HWA internal storage buffers.
5. Receives result: The value in the HWA storage buffers is sent to the C7x CPU to write to its general purpose registers.
6. Closes HWA.

Refer to the corresponding HWA architecture spec, which has details on the supported computations. For instance, the MMA provides specific computations to speed up convolution neural networks algorithms.

3.18.4.24.1 HWAOPEN

HWAOPEN sends the compute template to the HWA to let the HWA know the types of operands and computations to use for subsequent HWA operations.

3.18.4.24.2 HWALDA, HWALDB, HWALDC, HWALDAB, HWALDBC

These various HWALD instructions are used to read values from the GRF, LRFL and SE0, SE1 registers and send it out to the HWA to initialize the HWA operand matrices.

3.18.4.24.3 HWAOP

HWAOP instruction tells the HWA to proceed on the programmed computation.

3.18.4.24.4 HWAXFER

HWAXFER instruction transfers the computed result from the HWA logic to its internal buffers.

3.18.4.24.5 HWARCVS

HWARCVS instruction transfers the value stored in the HWA internal buffers to the C7x CPU general purpose registers.

3.18.4.24.6 HWACLOSE

HWACLOSE closes the HWA operations. All intermediate values inside the HWA are discarded.

More details are available in the MMA spec.

3.18.4.25 Debug Instruction Set

3.18.4.25.1 SWBP – Software Breakpoint

There are two types of SWBP instructions. When the debugger is enabled, the two SWBP instructions have the same behaviors and cause the CPU to halt execution prior to it entering the E1 pipeline phase. The debug logic is signalled that a software breakpoint has been encountered. When the debugger is disabled, the two SWBP instructions are treated differently by the CPU:

- ESWBP: Embedded Software Breakpoint. Treated as a NOP if encountered outside of a debugging session.
- DSWBP: Debugger Software Breakpoint. Treated as illegal opcode exception if encountered outside of a debugging session.

The use of these instructions is intended solely for debug purposes.

3.18.4.25.2 EDBG – Enable Emulator Debug Events

The EDBG instruction enables emulator debug events in the current cycle and sets TSR.DBGM to 0. This instruction executes unconditionally.

3.18.4.25.3 DDBG – Disable Emulator Debug Events

The DDBG instruction disables emulator debug events in the current cycle and sets TSR.DBGM to 1. This instruction executes unconditionally.

3.18.4.25.4 MARK – Mark Current Cycle for Emulator Use

The MARK instruction signals the debug logic that the 5-bit id value is present at the CPU boundary. The 5-bit value is presented in E2.

The id value of 10000 is reserved for the ABORTI instruction which is used to signal the aborting of an ISR.

The MARK instruction with id value of 00000 (MARK 0) when placed in parallel with a LD or ST operation results in the corresponding memory access submitted to the memory system with the interest flag set.

This instruction executes unconditionally.

3.19 Instruction Set Detail Descriptions

This section gives detailed information on the instruction set. Each instruction may present the following information:

- Assembler syntax
- Functional units
- Operands
- Opcode
- Description
- Execution
- Pipeline
- Instruction type
- Delay slots
- Examples

The ADDAB instruction is used as an example to familiarize with the way each instruction is described. The example describes the kind of information in each part of the individual instruction description, and where to obtain more information.

3.19.1 Example

The way each instruction is described.

Syntax EXAMPLE (.unit) src, dst

.unit = .L1, .L2, .S1, .S2, .D1, .D2, .M1, .M2, .N1, .N2, .C, .P

src and dst indicate source and destination, respectively. The (.unit) dictates which functional unit the instruction is mapped to.

A table is provided for each instruction that gives the opcode map fields, units the instruction is mapped to, types of operands, and the opcode.

The opcode shows the various fields that make up each instruction.

This instruction has three opcode map fields: src1, src2, and dst. This operation can be done on .D1 or .D2 (both are specified in the unit column). The s in front of each operand signifies that src1, src2, and dst are all signed values.

In the second group, src1, src2, and dst are sint, ucst5, and sint, respectively. The u in front of the cst5 operand signifies that src1 (ucst5) is an unsigned value. Any operand that begins with x can be read from a register file that is different from the destination register file. The operand comes from the register file opposite the destination, if the x bit in the instruction is set (shown in the opcode map).

Description instruction execution and its effect on the rest of the processor or memory contents are described. Any constraints on the operands imposed by the processor or the assembler are discussed. The description parallels and supplements the information given by the execution block.

3.20 Instruction Opcode Mapping

The general format for a two-source arithmetic instruction has the form shown in Figure 3-62.

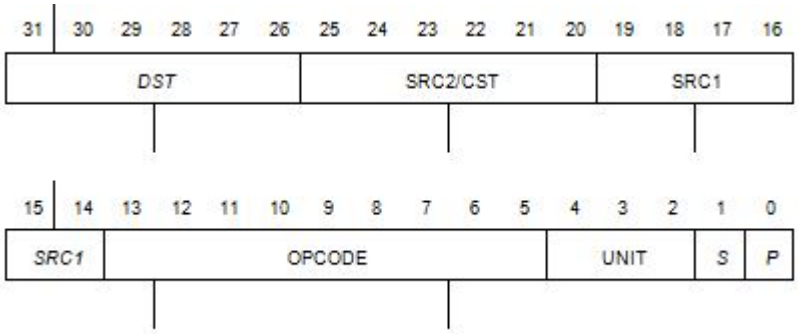


Figure 3-62. General Instruction Format

Where:

- DST: the register to be written to
- SRC2/CST: the source 2 operand, either from a register or a constant
- SRC1/CST: the source 1operand, either from a register or a constant
- OPCODE: identifies the operation
- UNIT: specifies which unit the instruction is executed on
- S (Side bit): specifies which side of the datapath the instruction is executed on
- P (P-bit): specifies whether there is another instruction is executed in parallel

Table 3-119. Unit and Side Field Mapping

Unit and Side Fields (Opcode[4:1])	Unit
0000	.L1
0001	.L2
0010	.S1
0011	.S2

Table 3-119. Unit and Side Field Mapping (continued)

Unit and Side Fields (Opcode[4:1])	Unit
0100	.M1
0101	.M2
0110	.D1-Loads
0111	.C2
1000	.D1-Stores
1001	.D2-Stores
1010	.D2-Loads
1011	CXT0
1100	Reserved
1101	Reserved
1110	CREG0/CREG1, .D1-Arith, .D2-Arith
1111	CXT1, BR, .P

3.20.1 Constant Extension Opcode Map (CSTX0, CSTX1)

The constant extensions are mapped in Figure 3-63 and Figure 3-64.

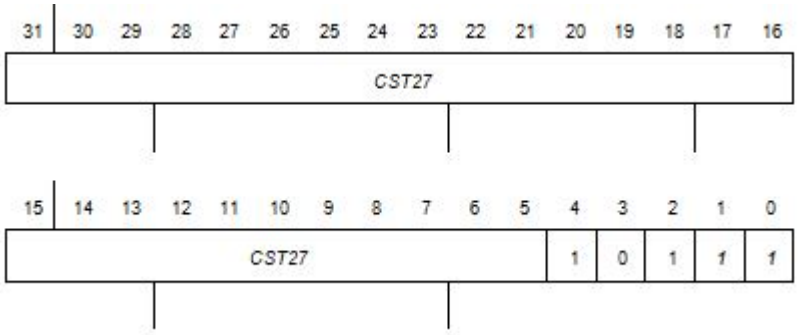


Figure 3-63. Constant Extension 0 (CSTX0)

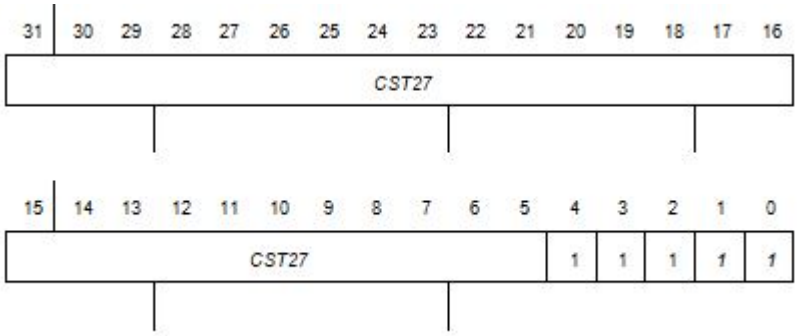


Figure 3-64. Constant Extension 1(CSTX1)

3.20.2 Condition Code Extension Opcode Map (CCEXT0, CCEXT1)

The condition code extensions are mapped in Figure 3-65 and Figure 3-66.

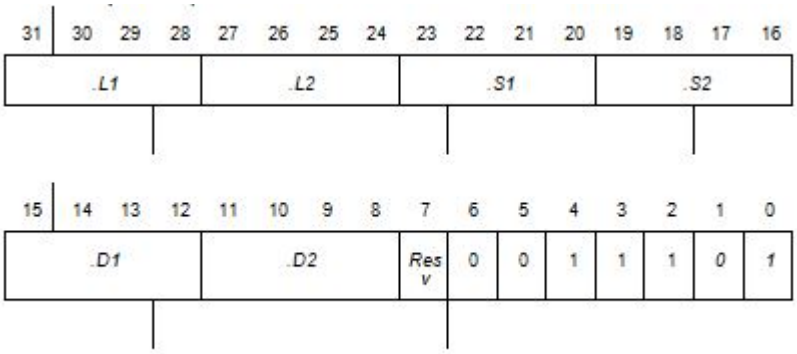


Figure 3-65. Condition Code Extension 0 (CCEXT0)

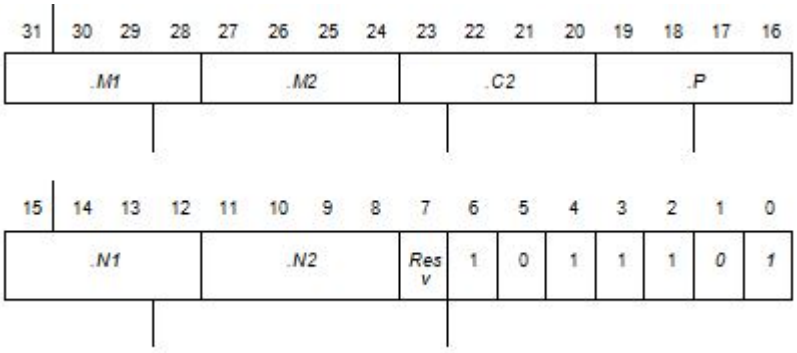


Figure 3-66. Condition Code Extension 1 (CCEXT1)

3.20.3 .L unit Opcode Map (L1S, L2S)

The CPU 32-bit opcodes used in the .L1 and .L2 units are mapped in Figure 3-67, Figure 3-68, and Figure 3-69.

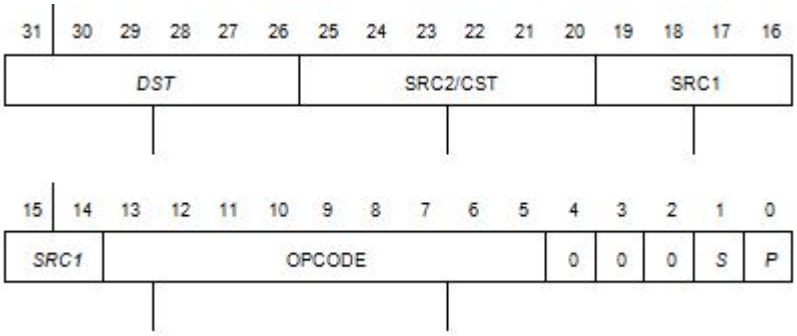


Figure 3-67. L2S Instruction Format

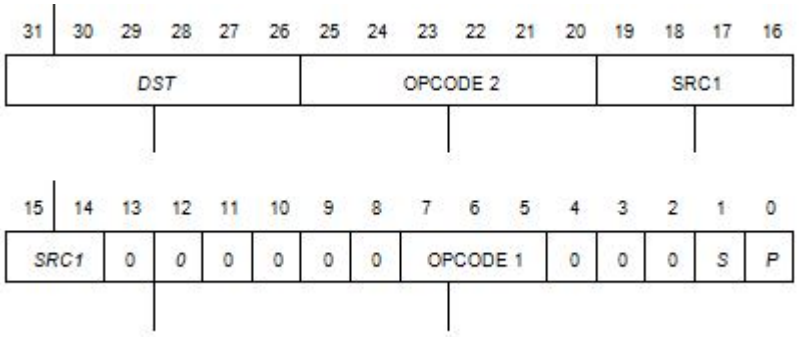


Figure 3-68. L1S Instruction Format

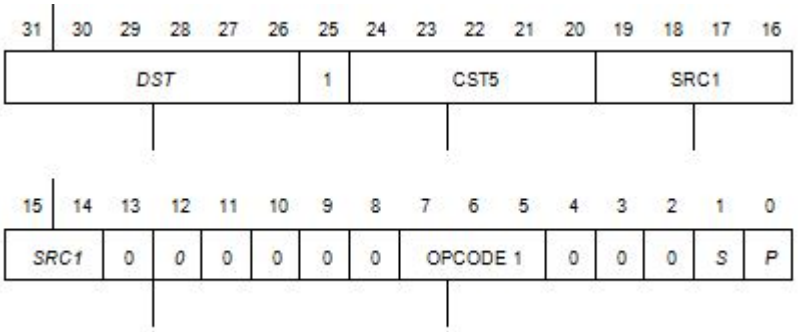


Figure 3-69. L2S_E Instruction Format

Table 3-120. L1S and L2S Formats Source Opcode Encoding (SRC1/SRC2)

SRC Opcodes	Side A	Side B
000000-001111	A0-A15	VB0-VB15
010000-011111	B0-B15	A0-A15
100000-100111	AL0-AL7	VBL0-VBL7
101000-101111	Reserved	Reserved
110000-111011	Reserved	Reserved
111100	Reserved	SE0
111101	Reserved	SE0++
111110	Reserved	SE1
111111	Reserved	SE1++

Table 3-121. L1S and L2S Formats Destination Opcode Encoding (DST)

DST Opcodes	Side A	Side B
000000-001111	A0-A15	VB0-VB15
010000-011111	D0-D15	Reserved
100000-100111	AM0-AM7	VBM0-VBM7
101000-101111	Reserved	Reserved
110000-110111	AL0-AL7	VBL0-VBL7
111000-111111	Reserved	Reserved

3.20.4 .S unit Opcode Map (S2S, S1S)

The CPU 32-bit opcodes used in the .L1 and .L2 units are mapped in Figure 3-70, Figure 3-71, and Figure 3-72.

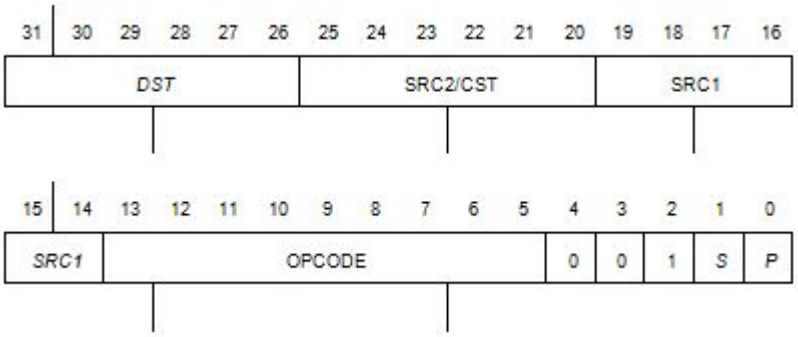


Figure 3-70. S2S Instruction Format

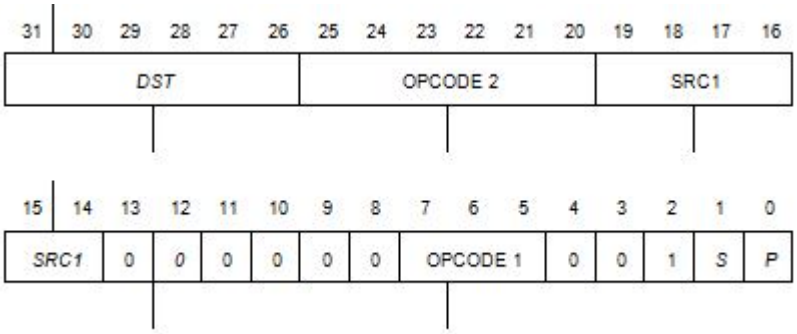


Figure 3-71. S1S Instruction Format

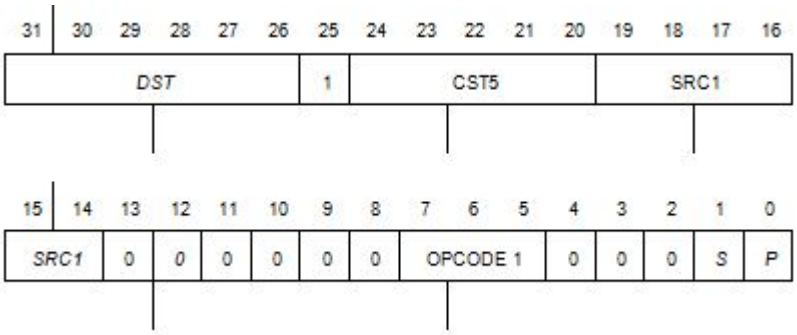


Figure 3-72. S2S_E Instruction Format

Table 3-122. S1S and S2S Formats Source Opcode Encoding (SRC1/SRC2)

SRC Opcodes	Side A	Side B
000000-001111	A0-A15	VB0-VB15
010000-011111	B0-B15	A0-A15
100000-100111	AL0-AL7	VBL0-VBL7
101000-101111	Reserved	Reserved
110000-111011	Reserved	Reserved
111100	Reserved	SE0
111101	Reserved	SE1
111110	Reserved	SE0++
111111	Reserved	SE1++

Table 3-123. S1S and S2S Formats Destination Opcode Encoding (DST)

DST Opcodes	Side A	Side B
000000-001111	A0-A15	VB0-VB15

Table 3-123. S1S and S2S Formats Destination Opcode Encoding (DST) (continued)

DST Opcodes	Side A	Side B
010000-011111	D0-D15	Reserved
100000-100111	AM0-AM7	VBM0-VBM7
101000-101111	Reserved	Reserved
110000-110111	AL0-AL7	VBL0-VBL7
111000-111111	Reserved	Reserved

3.20.5 .M unit Opcode Map (M2S, M1S)

The CPU 32-bit opcodes used in the .M1 and .M2 units are mapped in Figure 3-73 and Figure 3-74.

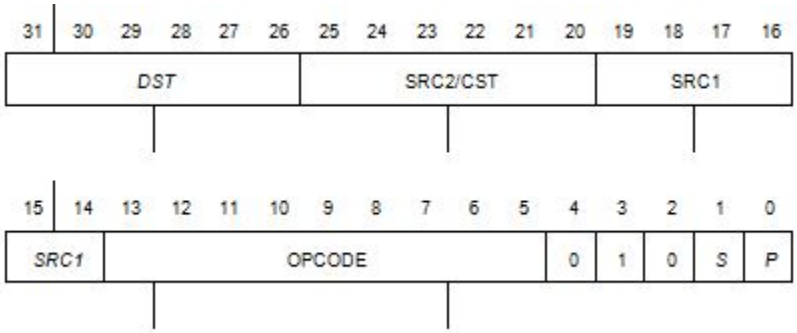


Figure 3-73. M2S Instruction Format

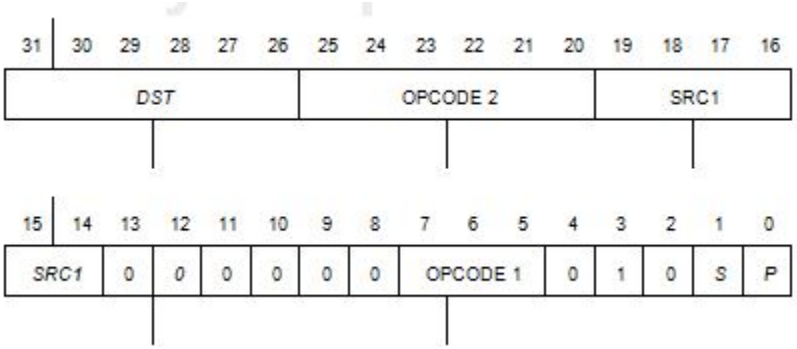


Figure 3-74. M1S Instruction Format

Table 3-124. M1S and M2S Formats Source Opcode Encoding (SRC1/SRC2)

SRC Opcodes	Side A	Side B
000000-001111	A0-A15	VB0-VB15
010000-011111	B0-B15	A0-A15
100000-100111	AM0-AM7	VBM0-VBM7
101000-101111	Reserved	Reserved
110000-111011	Reserved	Reserved
111100	Reserved	Reserved
111101	Reserved	SE0++
111110	Reserved	SE1
111111	Reserved	SE1++

Table 3-125. M1S and M2S Formats Destination Opcode Encoding (DST)

DST Opcodes	Side A	Side B
000000-001111	A0-A15	VB0-VB15
010000-011111	D0-D15	Reserved

Table 3-125. M1S and M2S Formats Destination Opcode Encoding (DST) (continued)

DST Opcodes	Side A	Side B
100000-100111	AM0-AM7	VBM0-VBM7
101000-101111	Reserved	Reserved
110000-110111	AL0-AL7	VBL0-VBL7
111000-111111	Reserved	Reserved

3.20.6 .N unit Opcode Map (N2S, N1S)

The CPU 32-bit opcodes used in the .N1 and .N2 units are mapped in Figure 3-75 and Figure 3-76.

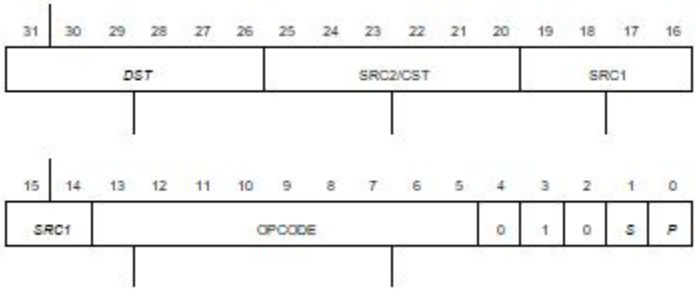


Figure 3-75. N2S Instruction Format

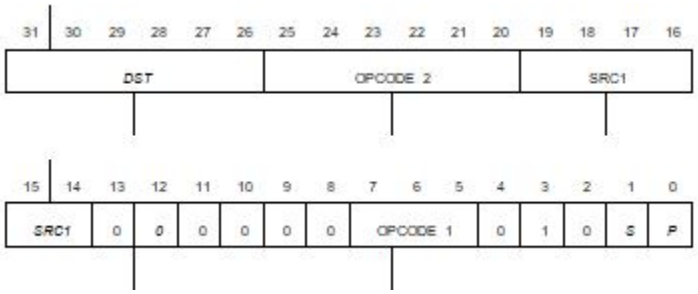


Figure 3-76. N1S Instruction Format

Table 3-126. N1S and N2S Formats Source Opcode Encoding (SRC1/SRC2)

SRC Opcodes	Side A	Side B
000000-001111	A0-A15	VB0-VB15
010000-011111	B0-B15	A0-A15
100000-100111	AM0-AM7	VBM0-VBM7
101000-101111	Reserved	Reserved
110000-111011	Reserved	Reserved
111100	Reserved	SE0
111101	Reserved	SE0++
111110	Reserved	SE1
111111	Reserved	SE1++

Table 3-127. N1S and N2S Formats Destination Opcode Encoding (DST)

DST Opcodes	Side A	Side B
000000-001111	A0-A15	VB0-VB15
010000-011111	D0-D15	Reserved
100000-100111	AM0-AM7	VBM0-VBM7
101000-101111	Reserved	Reserved
110000-110111	AL0-AL7	VBL0-VBL7

Table 3-127. N1S and N2S Formats Destination Opcode Encoding (DST) (continued)

DST Opcodes	Side A	Side B
111000-111111	Reserved	SE0

3.20.7 .C unit Opcode Map (C2S, C1S)

The C-unit is on side B, thus the s-bit of C-unit instructions is always 1.

The CPU 32-bit opcodes used in the .C unit are mapped in Figure 3-77 and Figure 3-78.

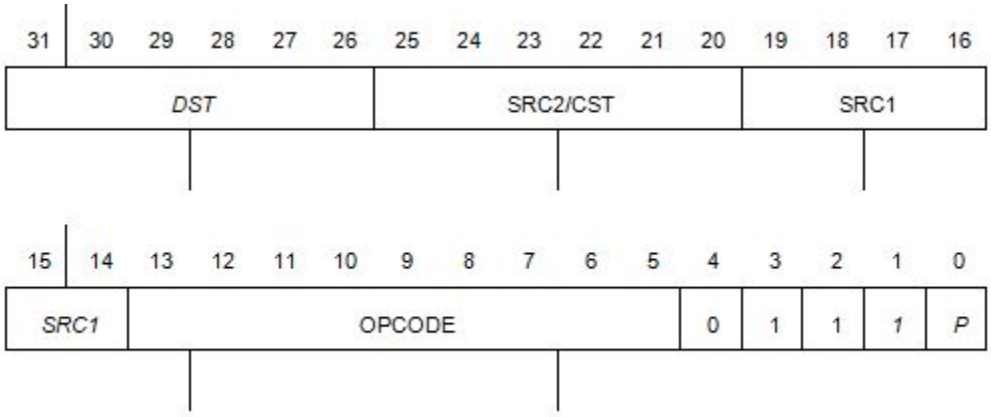


Figure 3-77. C2S Instruction Format

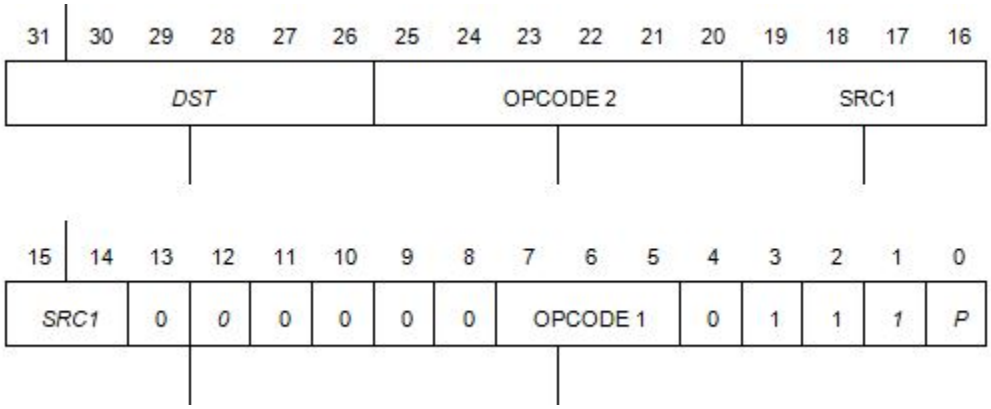


Figure 3-78. C1S Instruction Format

Table 3-128. C1S and C2S Formats Unit Source Opcode Encoding (SRC1/SRC2)

SRC Opcodes	Side B
000000-001111	VB0-VB15
010000-011111	A0-A15
100000-100111	VBM0-VBM7
101000-101111	Reserved
110000-111011	Reserved
111100	SE0
111101	SE0++
111110	SE1
111111	SE1++

Table 3-129. C1S and C2S Formats Unit Destination Opcode Encoding (DST)

DST Opcodes	Side B
000000-001111	VB0-VB15

Table 3-129. C1S and C2S Formats Unit Destination Opcode Encoding (DST) (continued)

DST Opcodes	Side B
010000-011111	Reserved
100000-100111	VBM0-VBM7
101000-101111	Reserved
110000-110111	VBL0-VBL7
111000-111111	SE0

3.20.7.1 C2SM Instruction Format

This format is used for instructions which have CUCR as another source input. Those instructions are the VPERM with CUCR as control register, VDOTPMPN and VSADM instructions.

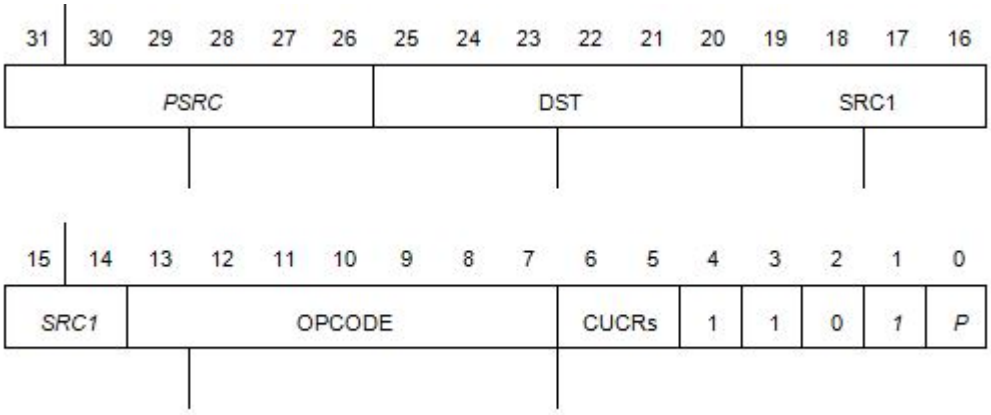


Figure 3-79. C2SM Instruction Format

CUCR field definition:

- 00: CUCR0
- 01: CUCR1
- 10: CUCR2
- 11: CUCR3

3.20.8 .P unit Opcode Map (P2S, P1S)

The P-unit is on side B, thus the s-bit of P-unit instructions is always 1.

The CPU 32-bit opcodes used in the .P unit are mapped in Figure 3-80 and Figure 3-81.

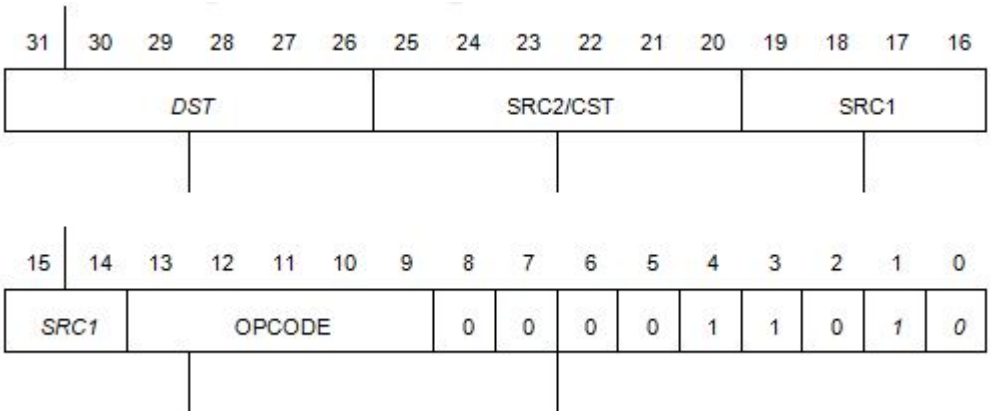


Figure 3-80. P2S Instruction Format

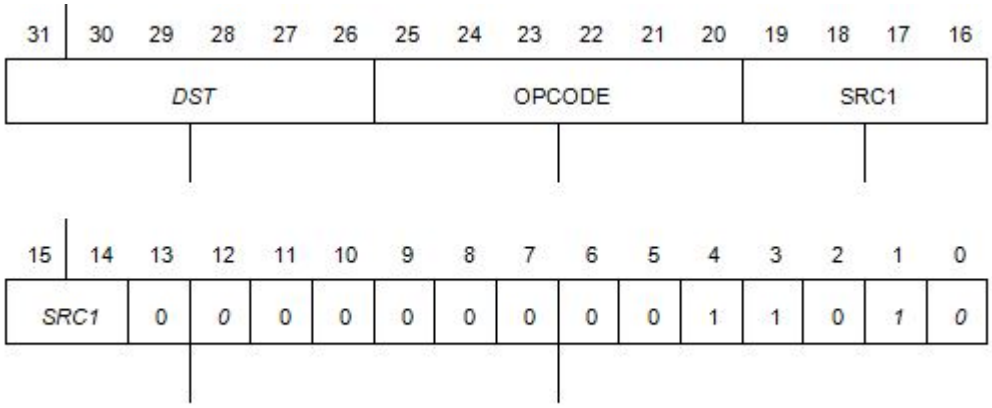


Figure 3-81. P1S Instruction Format

Table 3-130. P1S and P2S Opcode Encoding (SRC1/SRC2)

SRC Opcodes	Side B
000000-000111	P0-P7
001000-111111	Reserved

Table 3-131. P1S and P2S Destination Opcode Encoding (DST)

DST Opcodes	Side B
000000-000111	P0-P7
001000-111111	Reserved

3.20.9 .D unit Opcode Map

Both .D1 and .D2 units are on side A, thus the s-bit of D-unit instructions is always 1.

3.20.9.1 Arithmetic Format D2SSA

The CPU 32-bit opcodes used in the .D1 and .D2 units for arithmetic instructions are mapped in Figure 3-82 and Figure 3-83.

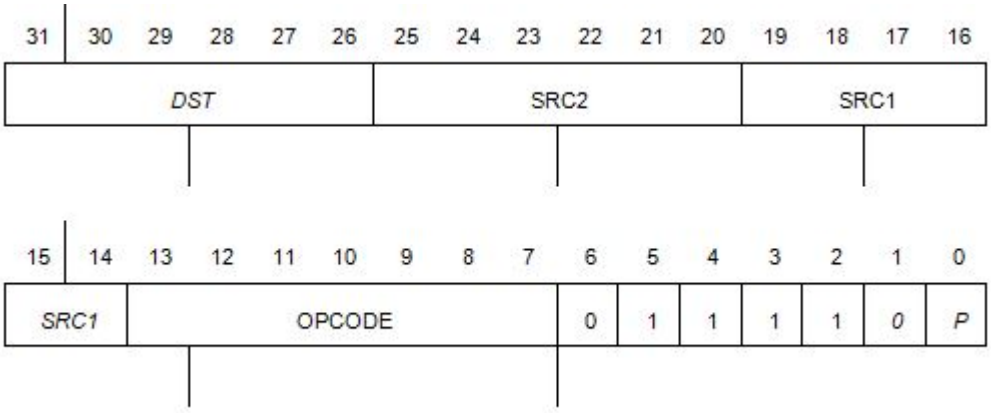


Figure 3-82. .D1 Unit D2SSA Instruction Format

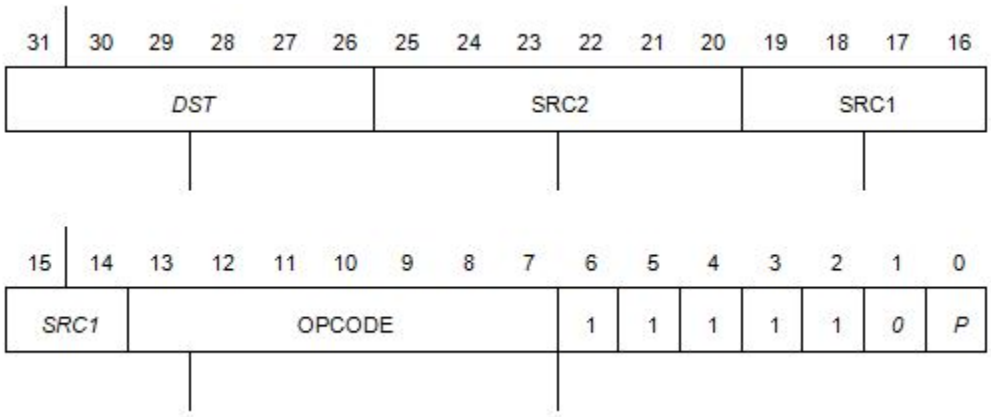


Figure 3-83. .D2 Unit D2SSA Instruction Format

Table 3-132. D2SSA Format Source Opcode Encoding (SRC1/SRC2)

SRC Opcodes	SRC1	SRC2
00000-001111	A0-A15	A0-A15
010000-011111	B0-B15	Reserved
100000-101111	D0-D15	D0-D15
110000	Reserved	SA0
110001	Reserved	SA1
110010	Reserved	SA2
110011	Reserved	SA3
110100-110111	Reserved	Reserved
111000	Reserved	SA0++
111001	Reserved	SA1++
111010	Reserved	SA2++
111011	Reserved	SA3++
111100-111111	Reserved	Reserved

Table 3-133. D2SSA Format Destination Opcode Encoding (DST)

DST Opcodes	Side A	Side B
000000-001111	A0-A15	VB0-VB15
010000-011111	D0-D15	Reserved
100000-100111	AM0-AM7	VBM0-VBM7
101000-101111	Reserved	Reserved
110000-110111	AL0-AL7	VBL0-VBL7
111000-111111	Reserved	SE0

3.20.9.2 Load Instructions D2S Format

The CPU 32-bit opcodes used in the .D unit for load instructions are mapped in Figure 3-84 and Figure 3-85.

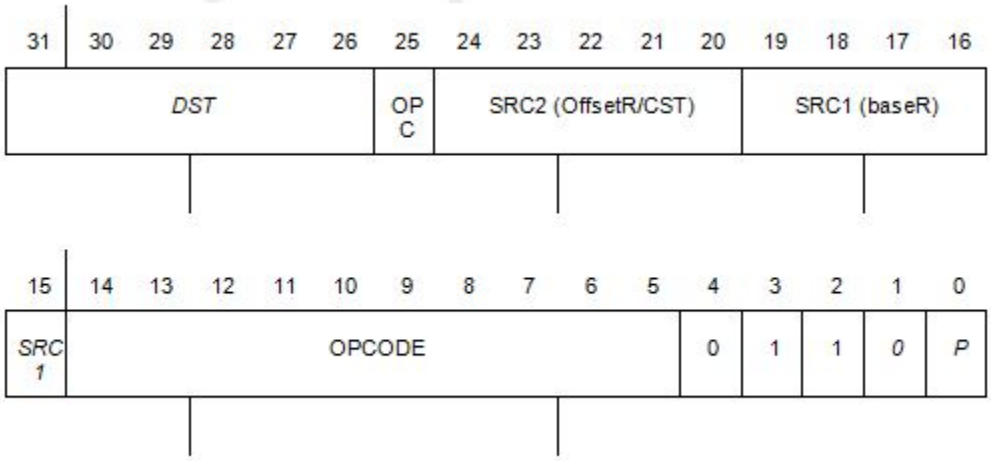


Figure 3-84. .D1 Unit Load Instruction D2S Format

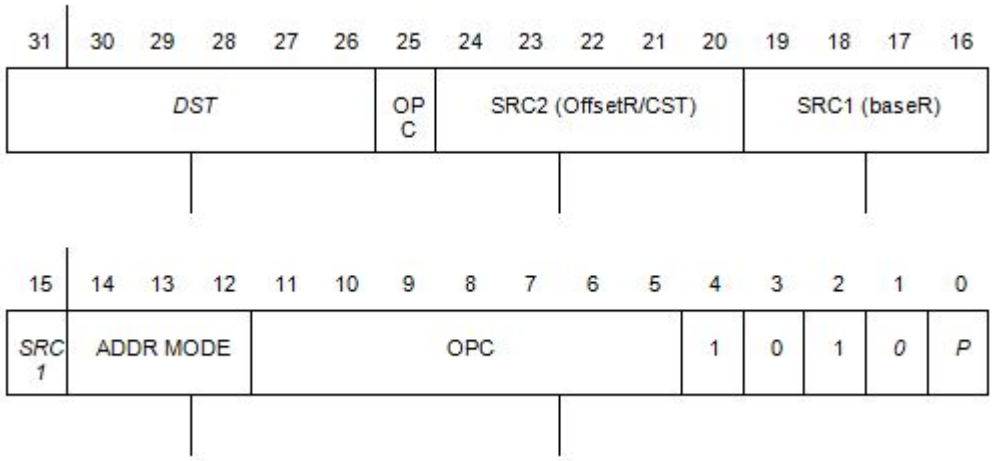


Figure 3-85. .D2 Unit Load Instruction D2S Format

Table 3-134. Addressing Mode Encodings for Load and Store instructions

ADDR MODE	Load Instructions	Scalar Store Instructions	Vector Store Instructions
000	baseR[ucst5]	baseR[ucst5]	baseR
001	baseR(scst32)	baseR(scst32)	baseR(scst32)
010	baseR[offsetR32]	baseR[offsetR32]	baseR[offsetR32]
011	Reserved	Reserved	Reserved
100	baseR++[ucst5]	baseR++[ucst5]	baseR++
101	baseR++(scst32)	baseR++(scst32)	baseR++(scst32)
110	baseR++[offsetR32]	baseR++[offsetR32]	baseR++[offsetR32]
111	Reserved	Reserved	Reserved

Table 3-135. D2S Format SRC1 (OffsetR) Encoding

OffsetR (SRC1) Encoding	
00000-00011	SA0-SA3
00100-00111	Reserved
01000-01111	A8-A15
10000-11111	D0-D15

Table 3-136. D2S Format SRC2 (BaseR) Encoding

BaseR (SRC2) Encoding	
00000-01101	A0-A13
011110	ECSP
011111	PCE1
10000-11111	D0-D15

Table 3-137. D2S Format Destination Opcode Encoding (DST)

DST Opcodes	.D1 Unit	.D2 Unit
000000-001111	A0-A15	A0-A15
010000-011111	VB0-VB15	VB0-VB15
100000-101111	D0-D15	D0-D15
110000-110011	AL0-AL3	AL0-AL3
110100-110111	VBL0-VBL3	VBL0-VBL3
111000-111011	AM0-AM3	AM0-AM3
111100-111111	VBM0-VBM3	VBM0-VBM3

3.20.9.3 Store Instructions ST2S Format

The CPU 32-bit opcodes used in the .D unit for load instructions are mapped in Figure 3-86 and Figure 3-87.

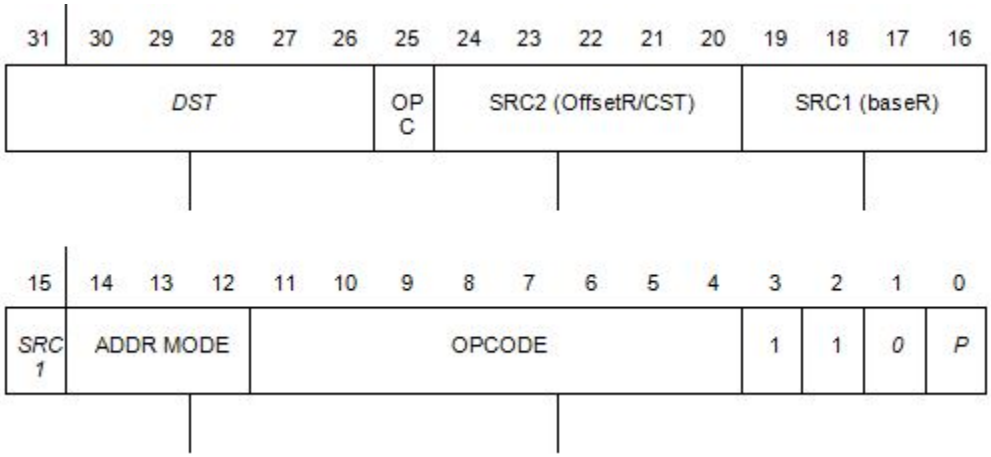


Figure 3-86. .D1 Unit ST2S Instruction Format

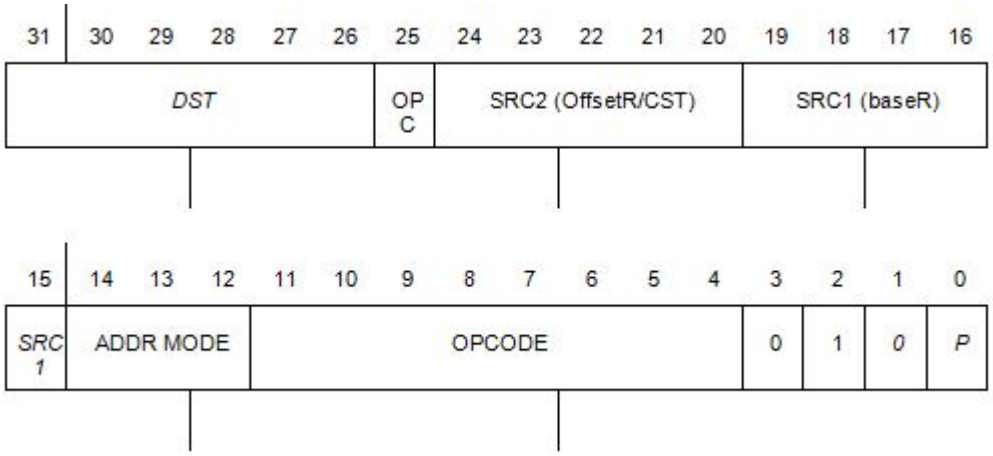


Figure 3-87. .D2 Unit ST2S Instruction Format

The SRC1, SRC2, and ADDR MODE fields of the ST2S have the same encodings as the load D2S format. The SRC3 field contains the store data register address to be read out.

Table 3-138. ST2S Format SRC3 Opcode Encoding

SRC3 Opcodes	.D1 Unit	.D2 Unit
000000-001111	A0-A15	A0-A15 (via crosspath)
010000-011111	B0-B15 (via crosspath)	VB0-VB15
100000-101111	D0-D15	Reserved
110000-111101	Reserved	Reserved
111110	Constant of zeros	Constant of zeros
111111	32-bit constant from 27-bit constant extension, signed extend to 32-bit	32-bit constant from 27-bit constant extension, signed extend to 32-bit

3.20.10 Branch Instructions Opcode Map

3.20.10.1 BRK Format

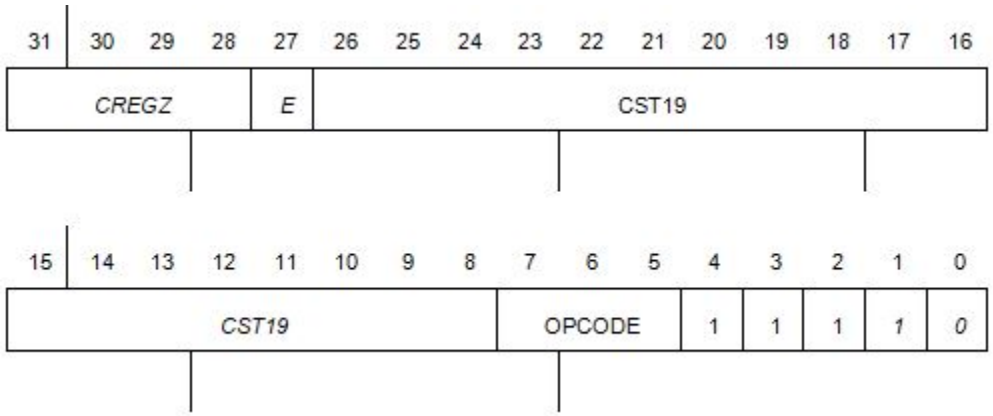


Figure 3-88. Branch BRK Format

Table 3-139. Registers That Can Be Tested by CREGZ field

Specified Conditional Register	creg				z
	Bit:	31	30	29	28
Unconditional		0	0	0	0
Reserved		0	0	0	1
A0		0	0	1	z
A1		0	1	0	z
A2		0	1	1	z
A3		1	0	0	z
A4		1	0	1	z
A5		1	1	0	z
Reserved		1	1	1	x(1)

(1) x can be any value.

3.20.10.2 BRK-NC (Non-Conditional) Format

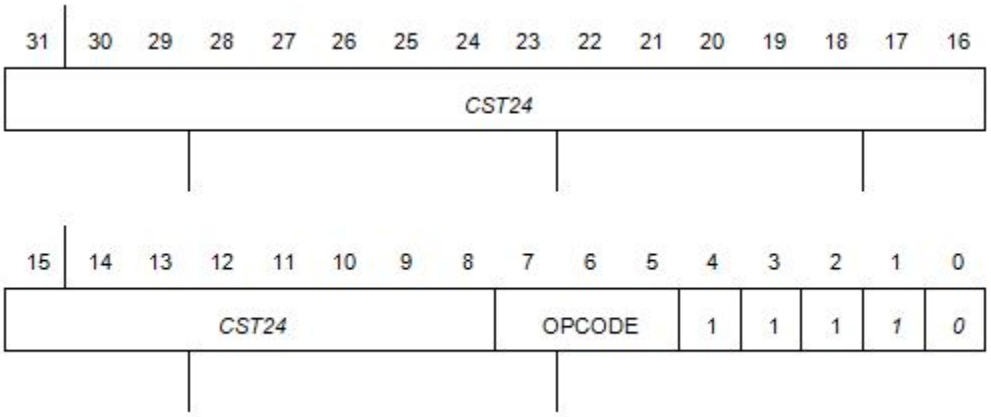


Figure 3-89. Branch BRK-NC Format

3.20.10.3 BRO Format

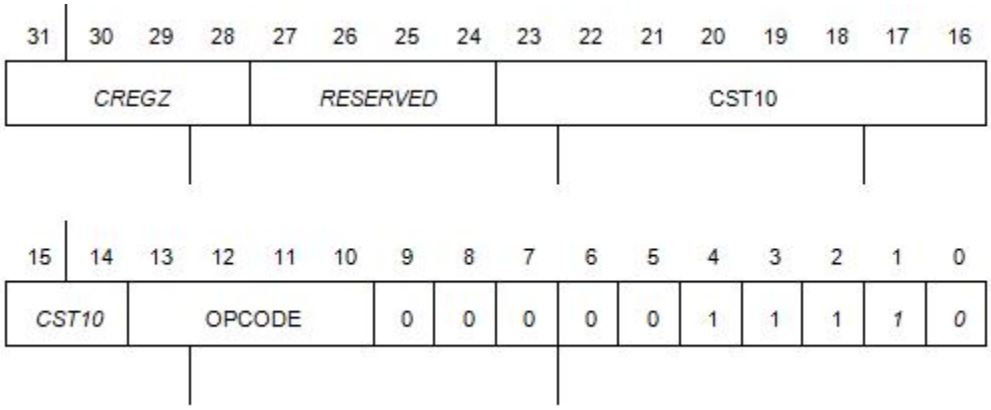


Figure 3-90. Branch BRO Format

3.20.11 Move Instructions Opcode Map

3.20.11.1 MV

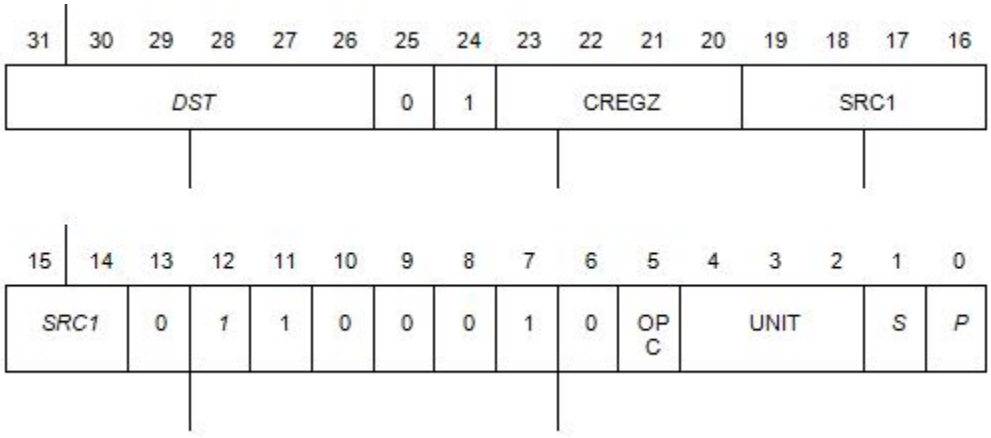


Figure 3-91. Data MV Instruction Formats (L2S_MV, S2S_MV, C2S_MV, M2S_MV)

3.20.11.2 Constant Move Instruction Formats (L2S_MVK, S2S_MVK, M2S_MVK)

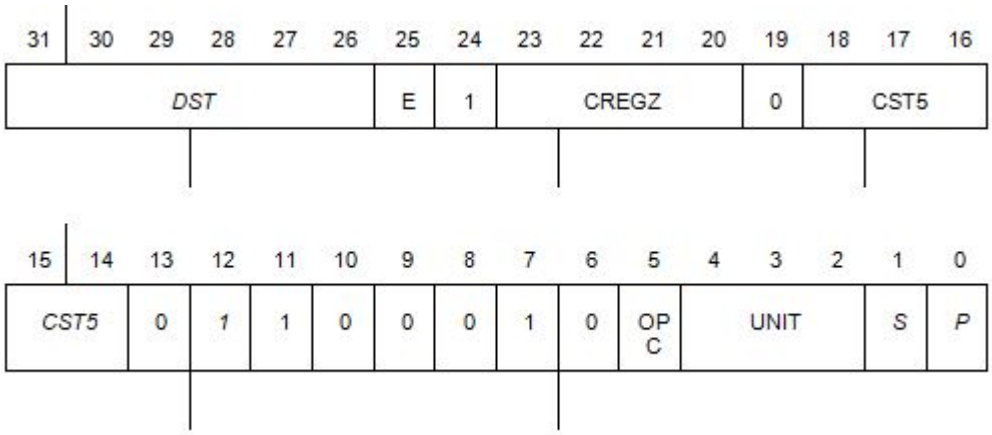


Figure 3-92. Constant Move Instruction Formats

3.20.11.3 Unconditional Constant Move Formats (L2S_MVKNC, S2S_MVKNC, M2S_MVKNC)

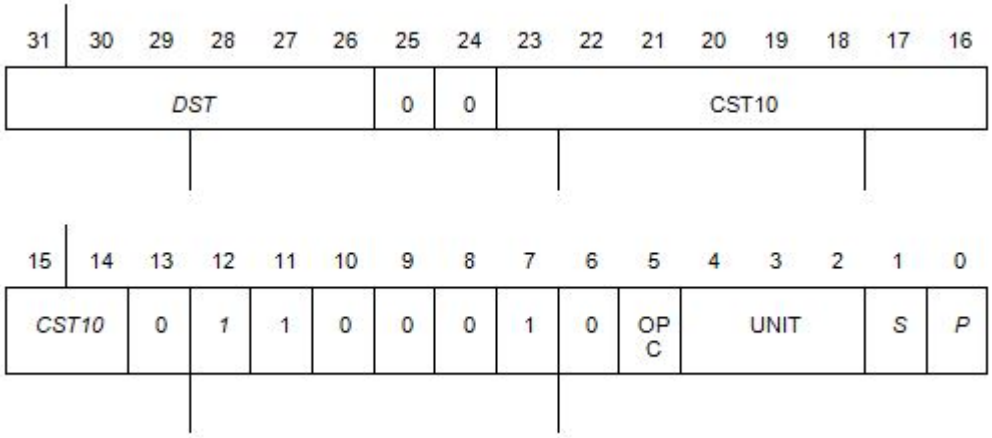


Figure 3-93. Unconditional Constant Move Instruction Formats

3.20.11.4 Unconditional 64-bit Constant Move Formats (L2S_MVKNC_S64, S2S_MVKNC_S64)

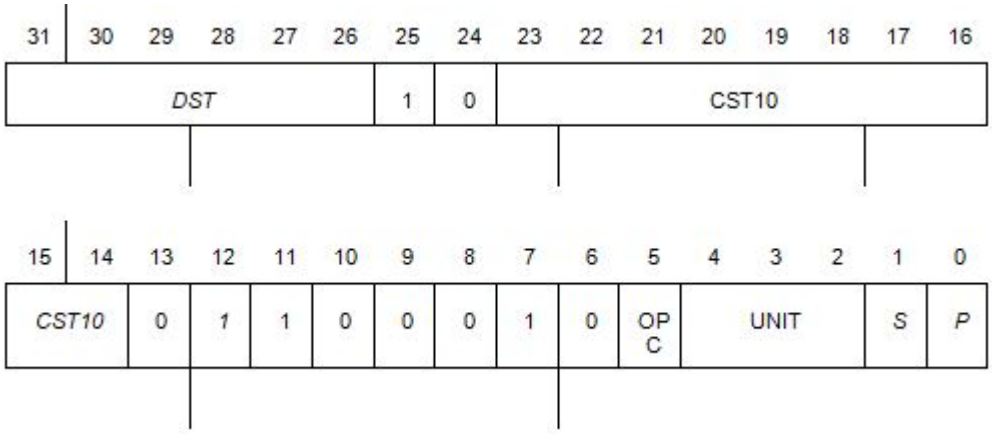


Figure 3-94. Unconditional 64-bit Constant Move Instruction Formats

3.20.12 Unitless Opcode Map

The unitless instruction opcodes are mapped in Figure 3-95.

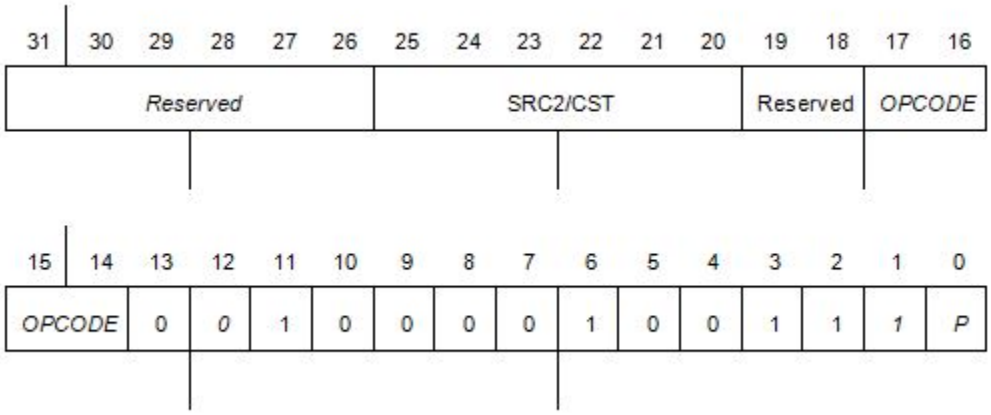


Figure 3-95. Unitless Instruction Format

3.20.13 Other Instruction Formats

3.20.13.1 VSEL Format

This format is used for VSEL instruction.

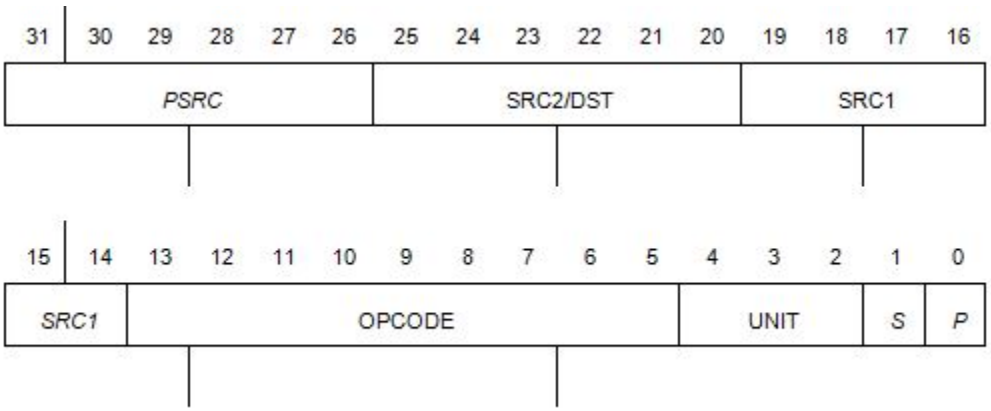


Figure 3-96. VSEL Instruction Format

3.20.13.2 VMINP/VMAXP Instruction Format

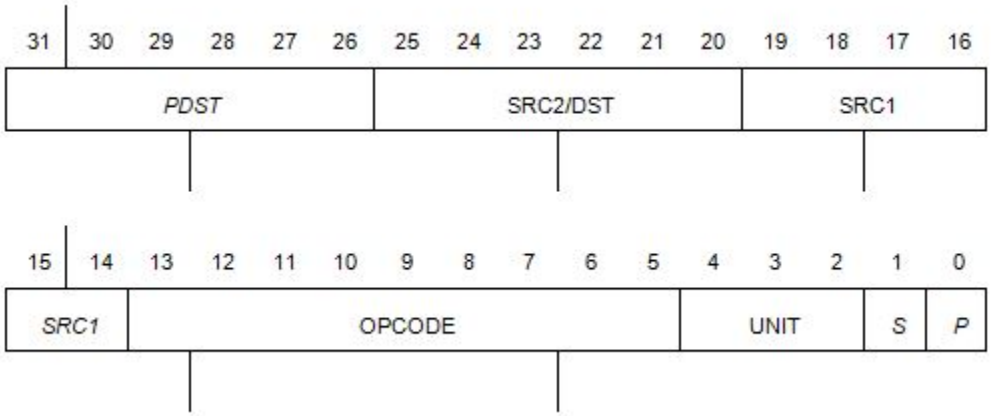


Figure 3-97. VMINP/VMAXP Instruction Format

3.20.13.3 VGATHER Instruction Format

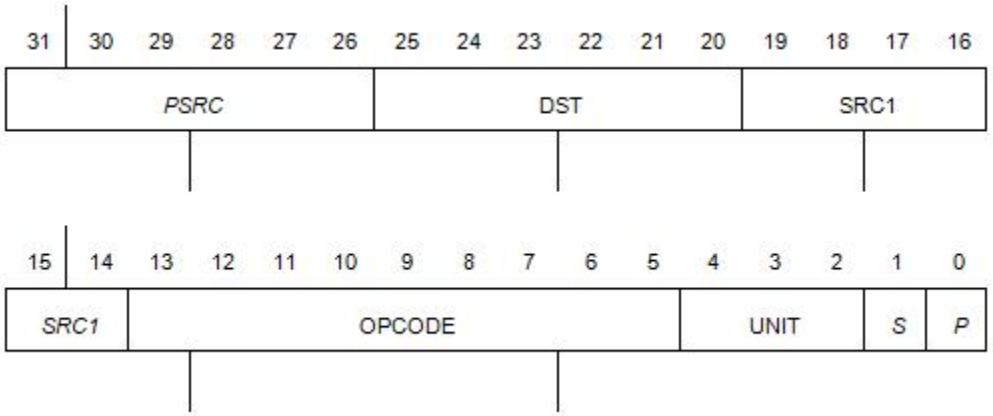


Figure 3-98. VGATHER Instruction Format

3.20.13.4 EXT/EXTU Instruction Format

TBD

3.20.13.5 REPLACE/VREPLACE Instruction Format

TBD

3.20.13.6 VPUTB/H/W/D Instruction Format

TBD

3.20.14 Instruction Orthogonality Summary

Table 3-140 summarizes the element size supported by different operations.

Table 3-140. Element Size

Operations	B	H	W	D	S	U	US/SU
ABS	Y	Y	Y	Y	Y	N	N
ADD	Y	Y	Y	Y	Y	N	N
SADD	Y	Y	Y	N	Y	Y	Y
ADDC	Y	Y	Y	Y	Y	N	N
ADDA	Y	Y	Y	Y	Y	N	N

Table 3-140. Element Size (continued)

Operations	B	H	W	D	S	U	US/SU
APYS	Y	Y	Y	Y	Y	N	N
AVG	Y	Y	Y	Y	Y	Y	N
AVGNR	Y	Y	Y	Y	Y	Y	N
BINLOG	N	N	Y	N	N	N	N
BITCNT	Y	Y	Y	Y	n/a	n/a	n/a
BITR	N	N	Y	Y	n/a	n/a	n/a
AND	N	N	Y	Y	n/a	n/a	n/a
ANDN	N	N	Y	Y	n/a	n/a	n/a
NAND	N	N	Y	Y	n/a	n/a	n/a
NOR	N	N	Y	Y	n/a	n/a	n/a
OR	N	N	Y	Y	n/a	n/a	n/a
ORN	N	N	Y	Y	n/a	n/a	n/a
XNOR	N	N	Y	Y	n/a	n/a	n/a
XOR	N	N	Y	Y	n/a	n/a	n/a
LAND	N	N	N	Y	n/a	n/a	n/a
LOR	N	N	N	Y	n/a	n/a	n/a
CMAX	Y	Y	N	N	Y	N	N
CMPEQ	Y	Y	Y	Y	Y	N	N
CMPGT	Y	Y	Y	Y	Y	Y	N
CMPGE	Y	Y	Y	Y	Y	Y	N
CROT90	N	Y	Y	N	Y	N	N
CROT270	N	Y	Y	N	Y	N	N
DEAL2	Y	Y	Y	N	n/a	n/a	n/a
DEAL4	Y	Y	N	N	n/a	n/a	n/a
DECIMATE	Y	Y	Y	Y	n/a	n/a	n/a
DIV	N	N	Y	Y	Y	Y	N
MOD	N	N	Y	Y	Y	Y	N
DUP	Y	Y	Y	Y	n/a	n/a	n/a
DUP2	Y	Y	Y	Y	n/a	n/a	n/a
DUP4	Y	Y	Y	Y	n/a	n/a	n/a
DUP8	Y	Y	Y	N	n/a	n/a	n/a
EXT	N	N	Y	N	Y	Y	n/a
FIR4HW	N	Y	N	N	Y	Y	Y
FIR4DS2HW	N	Y	N	N	Y	Y	Y
FIR8HD	N	Y	N	N	Y	Y	Y
FIR8DS2HD	N	Y	N	N	Y	Y	Y
FIR8DS4HD	N	Y	N	N	Y	Y	Y
FIR8HW	N	Y	N	N	Y	Y	Y
FIR8DS2HW	N	Y	N	N	Y	Y	Y
FIR8DS4HW	N	Y	N	N	Y	Y	Y
GATHER	Y	N	N	N	n/a	n/a	n/a
SCATTER	Y	N	N	N	n/a	n/a	n/a
GET	Y	Y	Y	Y	Y	Y	n/a
GETBIT	N	N	Y	N	n/a	n/a	n/a
GETDUP	Y	N	N	N	n/a	n/a	n/a
GMPY	Y	N	Y	N	n/a	n/a	n/a
LMBD	Y	Y	Y	Y	n/a	n/a	n/a

Table 3-140. Element Size (continued)

Operations	B	H	W	D	S	U	US/SU
NORM	Y	Y	Y	Y	n/a	n/a	n/a
MASK	Y	Y	Y	Y	n/a	n/a	n/a
MAX	Y	Y	Y	Y	Y	Y	n/a
MIN	Y	Y	Y	Y	Y	Y	n/a
MPY (same precision)	Y	Y	Y	Y	Y	Y	Y
MPY (output precision one higher)	Y	Y	Y	N	Y	Y	Y
SMPY (same precision)	N	N	Y	N	Y	N	N
SMPY (output precision one higher)	N	Y	N	N	Y	N	N
MATMPYHW	N	Y	N	N	Y	Y	Y
MATMPYHD	N	Y	N	N	Y	Y	Y
CMATMPYHW	N	Y	N	N	Y	N	N
CCMATMPYHW	N	Y	N	N	Y	N	N
NORM2U	N	Y	Y	N	N	Y	N
PRMBD	Y	Y	Y	Y	n/a	n/a	n/a
ROTL	Y	Y	Y	Y	n/a	n/a	n/a
SAT	Y	Y	Y	Y	Y	N	N
GSAT	Y	Y	Y	Y	Y	Y	Y
SHFL2	Y	Y	Y	Y	n/a	n/a	n/a
SHFL4	Y	Y	Y	Y	n/a	n/a	n/a
SHL	N	Y	Y	Y	n/a	n/a	n/a
SHR	Y	Y	Y	Y	Y	Y	n/a
SHRR	Y	Y	Y	N	Y	Y	N
SHVL	N	N	Y	N	Y	Y	N
SSHL	N	Y	Y	N	Y	Y	Y
SSHLR	N	Y	Y	Y	Y	Y	Y
SUB	Y	Y	Y	Y	Y	N	N
SUBA	Y	Y	Y	Y	Y	N	N
SUBR	Y	Y	Y	Y	Y	N	N
SSUB	Y	Y	Y	N	Y	N	N
SUBABS	Y	Y	Y	Y	Y	N	N
SUBC	N	N	Y	N	Y	N	N
SWAP	Y	Y	Y	Y	n/a	n/a	n/a
XORMPY	N	N	Y	N	n/a	n/a	n/a

3.21 Appendix – Overview of IEEE Floating Point Standard

Floating-point operands are classified as single-precision (SP) and double-precision (DP). Single-precision floating-point values are 32-bit values stored in a single register. Double-precision floating-point values are 64-bit values stored in a register pair. The register pair consists of consecutive even and odd registers from the same register file. The 32 least-significant-bits are loaded into the even register; the 32 most-significant-bits containing the sign bit and exponent are loaded into the next register (that is always the odd register). The register pair syntax places the odd register first, followed by a colon, then the even register (that is, A1:A0, B1:B0, A3:A2, B3:B2, and so forth).

Instructions that use DP sources fall in two categories: instructions that read the upper and lower 32-bit words on separate cycles, and instructions that read both 32-bit words on the same cycle. All instructions that produce a double-precision result write the low 32-bit word one cycle before writing the high 32-bit word. If an instruction that writes a DP result is followed by an instruction that uses the result as its DP source and it reads the upper and lower words on separate cycles, then the second instruction can be scheduled on the same cycle that the high 32-bit word of the result is written. The lower result is written on the previous cycle. This is because the second instruction reads the low word of the DP source one cycle before the high word of the DP source.

IEEE floating-point numbers consist of normal numbers, denormalized numbers, NaNs (not a number), and infinity numbers. Denormalized numbers are nonzero numbers that are smaller than the smallest nonzero normal number. Infinity is a value that represents an infinite floating-point number. NaN values represent results for invalid operations, such as (+infinity + (-infinity)).

Normal single-precision values are always accurate to at least six decimal places, sometimes up to nine decimal places. Normal double-precision values are always accurate to at least 15 decimal places, sometimes up to 17 decimal places.

Table 3-141 shows notations used in discussing floating-point numbers.

Table 3-141. IEEE Floating-Point Notations

Symbol	Meaning
s	Sign bit
e	Exponent field
f	Fraction (mantissa) field
x	Can have value of 0 or 1 (don't care)
NaN	Not-a-Number (SNaN or QNaN)
SNaN	Signal NaN
QNaN	Quiet NaN
NaN_out	QNaN with all bits in the f field = 1
Inf	Infinity
LFPN	Largest floating-point number
SFPN	Smallest floating-point number
LDFPN	Largest denormalized floating-point number
SDFPN	Smallest denormalized floating-point number
signed Inf	+infinity or -infinity
signed NaN_out	NaN_out with s = 0 or 1

3.21.1 Single-Precision Formats

Figure 3-99 shows the fields of a single-precision floating-point number represented within a 32-bit register.

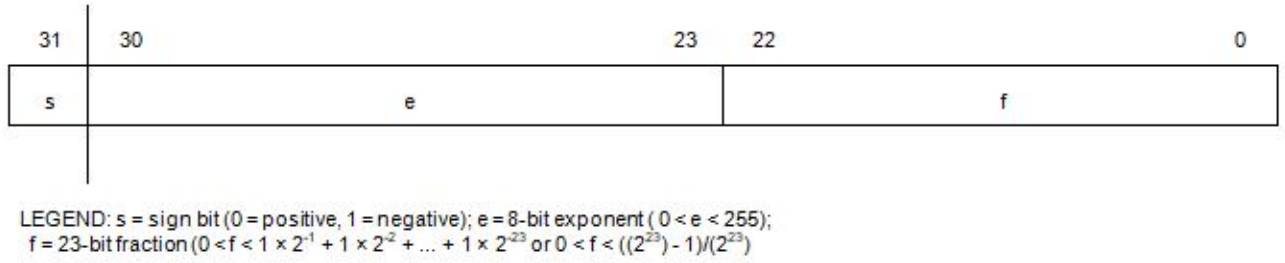


Figure 3-99. Single-Precision Floating-Point Fields

The floating-point fields represent floating-point numbers within two ranges: normalized (e is between 0 and 255) and denormalized (e is 0). The following formulas define how to translate the s, e, and f fields into a single-precision floating-point number.

Normalized: $-1^{s} \times 2^{(e - 127)} \times 1.f$; $0 < e < 255$

Denormalized (Subnormal): $-1^{s} \times 2^{-126} \times 0.f$; e = 0; f is nonzero

Table 3-142 shows the s, e, and f values for special single-precision floating-point numbers.

Table 3-142. Special Single-Precision Values

Symbol	Sign (s)	Exponent (e)	Fraction (f)
+0	0	0	0
-0	1	0	0
+Inf	0	255	0
-Inf	1	255	0
NaN	x	255	nonzero
QNaN	x	255	1xx..x
SNaN	x	255	0xx..x and nonzero

Table 3-143 shows hexadecimal and decimal values for some single-precision floating-point numbers.

Table 3-143. Hexadecimal and Decimal Representation for Selected Single-Precision Values

Symbol	Hex Value	Decimal Value
NaN_out	7FFF FFFF	QNaN
0	0000 0000	0.0
-0	8000 0000	-0.0
1	3F80 0000	1.0
2	4000 0000	2.0
LFPN	7F7F FFFF	3.40282347e+38
SFPN	0080 0000	1.17549435e-38
LDFPN	007F FFFF	1.17549421e-38
SDFPN	0000 0001	1.40129846e-45

3.21.2 Double-Precision Formats

Figure 3-100 shows the fields of a double-precision floating-point number represented within a pair of 32-bit registers.

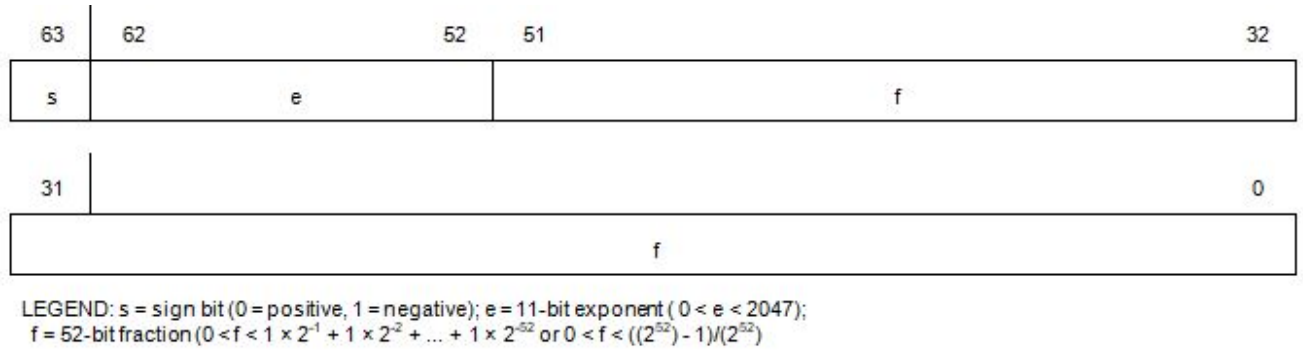


Figure 3-100. Double-Precision Floating-Point Fields

The floating-point fields represent floating-point numbers within two ranges: normalized (e is between 0 and 2047) and denormalized (e is 0). The following formulas define how to translate the s, e, and f fields into a double-precision floating-point number.

Normalized: $-1^{s} \times 2^{(e - 1023)} \times 1.f$; $0 < e < 2047$

Denormalized (Subnormal): $-1^{s} \times 2^{-1022} \times 0.f$; e = 0; f is nonzero

Table 3-144 shows the s, e, and f values for special double-precision floating-point numbers.

Table 3-144. Special Double-Precision Values

Symbol	Sign (s)	Exponent (e)	Fraction (f)
+0	0	0	0
-0	1	0	0
+Inf	0	2047	0
-Inf	1	2047	0
NaN	x	2047	nonzero
QNaN	x	2047	1xx..x
SNaN	x	2047	0xx..x and nonzero

Table 3-145 shows hexadecimal and decimal values for some double-precision floating-point numbers.

Table 3-145. Hexadecimal and Decimal Representation for Selected Double-Precision Values

Symbol	Hex Value	Decimal Value
NaN_out	7FFF FFFF FFFF FFFF	QNaN
0	0000 0000 0000 0000	0.0
-0	8000 0000 0000 0000	-0.0
1	3FF0 0000 0000 0000	1.0
2	4000 0000 0000 0000	2.0
LFPN	7FEF FFFF FFFF FFFF	1.7976931348623157e+308
SFPN	0010 0000 0000 0000	2.2250738585072014e-308
LDFPN	000F FFFF FFFF FFFF	2.2250738585072009e-308
SDFPN	0000 0000 0000 0001	4.9406564584124654e-324

3.21.3 Half-Precision Format

Figure 3-101 shows the fields of a half-precision floating-point number represented within a 16-bit register.

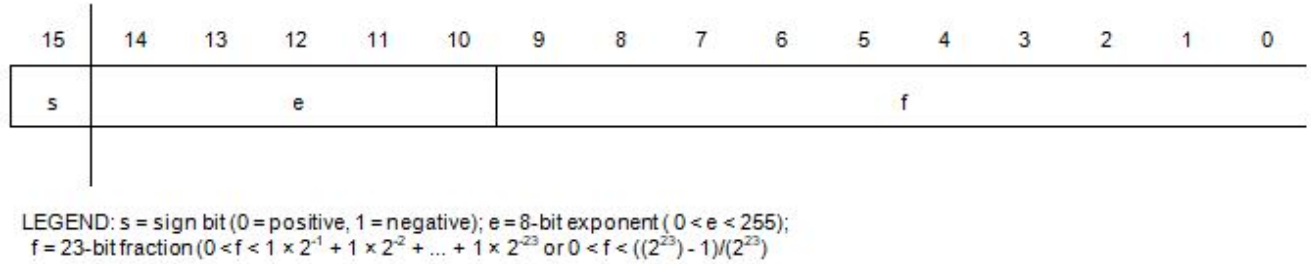


Figure 3-101. Half-Precision Floating-Point Fields

The floating-point fields represent floating-point numbers within two ranges: normalized (e is between 0 and 31) and denormalized (e is 0). The following formulas define how to translate the s, e, and f fields into a single-precision floating-point number.

Normalized: $-1^{s} \times 2^{(e - 15)} \times 1.f$; $0 < e < 31$

Denormalized (Subnormal): $-1^{s} \times 2^{-14} \times 0.f$; $e = 0$; f is nonzero

Table 3-146 shows the s, e, and f values for special half-precision floating-point numbers.

Table 3-146. Special Single-Precision Values

Symbol	Sign (s)	Exponent (e)	Fraction (f)
+0	0	0	0
-0	1	0	0
+Inf	0	31	0
-Inf	1	31	0

Table 3-146. Special Single-Precision Values (continued)

Symbol	Sign (s)	Exponent (e)	Fraction (f)
NaN	x	31	nonzero
QNaN	x	31	1xx..x
SNaN	x	31	0xx..x and nonzero

Table 3-147 shows hexadecimal and decimal values for some half-precision floating-point numbers.

Table 3-147. Hexadecimal and Decimal Representation for Selected Half-Precision Values

Symbol	Hex Value	Decimal Value
NaN_out	7FFF	QNaN
0	0000	0.0
-0	8000	-0.0
1	3C00	1.0
2	4000	2.0
LFPN	7BFF	6.5504e4
SFPN	0400	6.1035e-5
LDFPN	03FF	6.0976e-5
SDFPN	0001	5.9605e-8

3.22 Streaming Engine

3.22.1 Feature Summary

3.22.1.1 Features Supported

The streaming engine provides a flexible, high bandwidth mechanism for reading large quantities of data into the DSP CPU through two independent, programmable streams. The streaming engine consists of the following:

- Two stream address generators, supporting two simultaneous streams
  - Provides 6-level loop nest
    - Independent trip counts
    - Early exiting from 1 to 6 levels
  - Provides 5 programmable array dimensions
  - Capable of generating multiple request types:
    - Aligned 512-bit read requests
    - Aligned 1024-bit read requests
    - Cache maintenance operations (clean, invalidate, and clean-invalidate)
    - Cache preload requests
  - Multi-dimensional circular addressing support
    - Up to two separate circular block sizes per stream
    - Each dimension individually selectable between linear, circular block 0, or circular block 1.
  - Capable of generating two new requests per generator per cycle
  - Includes one µTLB per generator for virtual-to-physical address translation
- Two independent read-only memory interfaces
  - 512 bits/cycle at CLK/1
  - Connects directly to L2 controller
  - Credit-based protocol
  - Both interfaces usable by both streams
  - Fully coherent with all writes made before opening stream
- Two independent data buffers, one for each stream, organized as follows:
  - 32 slots, tagged by virtual address
  - 64 bytes/slot in 8 banks of 8 bytes
  - Capable of 4 accesses (2 reads, 2 writes) per cycle
  - Precise slot lifetime tracking
- Two data formatting networks supporting the following:

  - Extract non-aligned data from aligned slots
  - Endianness conversion
  - Complex-number format conversion (RI/IR swap)
  - Flexible integer type promotion
    - Integer sign/zero extension
    - Promotion from 8-bit, 16-bit, or 32-bit types
    - Promotion to 16-bit, 32-bit, or 64-bit types from any smaller type
  - Limited decimation support when also promoting data type
  - Lane-order reversal, to support convolution
- Matrix transpose on 8, 16, 32, 64, 128, and 256-bit boundaries
  - Support for 8-bit and 16-bit transpose within a 32-bit boundary when 4x decimation and minimum 4x promote, and 2x decimation and minimum 2x promote is enabled, respectively
- Two 512-bit CPU fetch paths, one for each stream
  - Up to 512 bits per cycle per stream
  - CPU references decoupled from stream advance
- CPU data store fencing logic with L1D to hold off on L2 fetches until fence clears
- Two-dimensional “data strip mining” feature to define the actual width and height of a data frame using DECDIM1_WIDTH and DECDIM2_WIDTH as the actual dimensions – in both linear and transpose streams
- \ Secondary decrement feature for DECDIM1 and DECDIM2 for more complicated MMA patterns; see DECDIM1SD and DECDIM2SD flags
- Error suppression to not report any error types within the address range that is masked out by DECDIMx and DECDIMxSD memory regions
- LEZR feature to return N number of zero vectors to CPU after selected dimension ends
- Full interrupt and exception support
  - Context save/restore supports multitasking
- Deferred exceptions: Triggers exception only if CPU consumes the data
- ECR-accessible registers for debug per stream
- Debugger support, including:
  - Loop counter visibility
  - Data buffer address tag visibility
  - Address tag hit/miss events
- Error detection and correction support
  - Full SECDED support on incoming data
  - 1-bit parity per 32-bit value in data buffers and tags
  - Semi-automatic recovery for all single-bit data and tag errors
- Safety
  - Safety error injection and detection for 1-bit parity per 32-bit data in storage
  - Aligns to Keystone 3 safety architecture provided testing and mechanism for ECC detection logic
  - Aligns to Keystone 3 safety architecture for reporting of detected parity and ECC faults from both error injection and functional access
- Hardware watch point (HWWP) and command interest (cinterest) matching compare for HWWP criteria’s, and interest address matching generation
- CPU initiated stream open command Interest generation

With the above features, the streaming engine can sustain 1024-bits/cycle total bandwidth (512 bits/cycle on each of two streams) to the DSP CPU. When combined with its 512-bit/cycle vector load unit and 512-bit/cycle store unit, the DSP CPU can access 2048 bits/cycle of data.

3.22.1.2 Features Not Supported

The streaming engine does not support the following:

- More than two simultaneous streams
- Running two streams concurrently from two different execution contexts (page table bases and/or privilege levels)
- Writing data through the streaming engine
- Coherence between program writes and already-open streams
- Type conversion between integer and floating point types

- Floating point promotion

3.22.2 Interface Description

See the Streaming Engine Micro Arch Spec in TBD.

3.22.3 Stream Concepts

3.22.3.1 Stream Properties

A stream consists of a sequence of elements of a particular type. Programs that operate on streams read the data sequentially, operating on each element in turn. Every stream has the following basic properties:

- Well-defined element lifetime. Each element within a stream has a well-defined lifetime within that stream.
- Read-only while active. Programs should not write to elements of a stream during their active lifetime.
- Homogeneous. Fixed element size and type throughout the entire stream.
- Fixed element sequence. Programs cannot seek randomly within the stream.

3.22.3.1.1 Well-Defined Element Lifetime

Programs start streams by opening them and end streams by closing them. While the stream remains open, the streaming engine presents elements to the CPU in a particular order, sometimes presenting the same element more than once, as dictated by the stream's parameters.

The presentation order defines the lifetime of each element within the stream, with distinct periods before, during, and after the element's lifetime.

The period before an element's lifetime ends when the program opens the stream.

The period after an element's lifetime begins when the program reads the last reference to that element from the streaming engine, or when the program closes the stream, whichever comes earlier. Because a given stream may refer to the same element more than once, only the last reference affects the lifetime of the element.

Figure 3-102 illustrates element lifetimes with a simple example. In this stream, element A gets referenced once, element B gets referenced twice, and element C does not get referenced before the program closes the stream. The period between before and after defines the period during the element's lifetime.

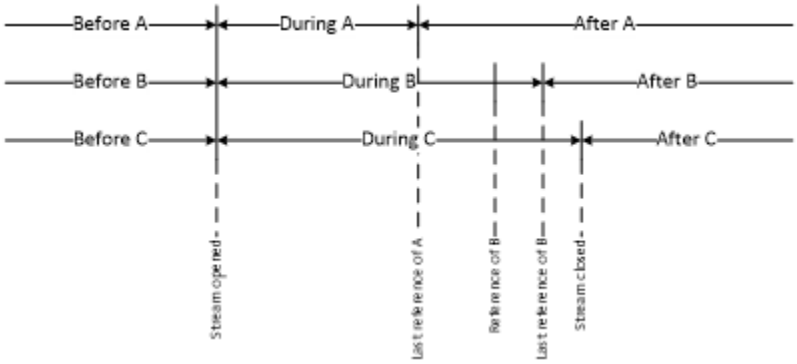


Figure 3-102. Element Lifetimes

In this example, the lifetime of each element begins when the program opens the stream. The lifetime of A and B ends when the program reads the last reference of each. The lifetime of C ends when the program closes the stream.

3.22.3.1.2 Read-Only While Active

The streaming engine may read a given element from memory at any point in its lifetime, possibly more than once, to present its value to the DSP CPU. This implies, for each element in the stream:

- Streams see all writes to an element that occur before that element's lifetime.
- Streams see no writes to an element that occur after that element's lifetime.
- Streams see an indeterminate subset of writes to an element that occur during that element's lifetime.

To see deterministic behavior, programs must not write to an element during that element's lifetime. Programs may write to a given element before its lifetime in a stream or after, but not during.

Note

The streaming engine may incidentally read an element outside of its lifetime while fetching other elements nearby. The streaming engine does not present these additional references to the CPU. The streaming engine guarantees these additional incidental references do not violate the properties stated above.

3.22.3.1.3 Homogeneous

All elements in a given stream share a common size and data type. Each element comprises the same number of bytes. This means the elements pack in a regular structure in memory.

Type refers to the following:

- Integer versus floating point
- Signed versus unsigned (if integer)
- Real versus complex

These attributes apply to all elements of a stream. This makes the stream definition compact, though rigid.

Streams contain real or complex elements. Real elements contain a single value in each element. Complex elements consist of a pair of sub-elements, each consisting of a single value. Element refers to the minimum fetch granule for a stream, while sub-element refers to each individual value in the stream. All sub-elements in a stream share the same numeric type, whether integer or floating point, and all elements share the same structure, whether real or complex.

Note

The designations real and complex only define whether an element consists of one or two sub-elements. The underlying data need not actually be real-valued or complex-valued data in the mathematical sense.

Because the elements in the stream all share the same structure and type, the stream definition can also compactly specify transformations that apply to all stream elements. Typical transformations include type promotion and exchanging the sub-elements of complex numbers in a stream.

3.22.3.1.4 Fixed Element Sequence

The stream parameters define a fixed order of elements in memory. The stream parameters entirely determine the stream layout. No data in the stream—for example, linked list pointers—modify its layout.

This property allows the streaming engine to read quite far ahead of the DSP CPU's need for the data. It also allows the streaming engine to discard data as soon as it hands the data to the DSP CPU, if necessary.

3.22.3.2 Defining a Stream

Stream elements typically reside in normal memory. The memory itself imposes no particular structure. Programs define streams—and therefore impose structure—by specifying the following stream attributes:

- Location of the first element of the stream
- Address sequence associated with the stream
- Size and type of the elements in the stream
- Formatting for data in the stream—for example, type promotion

Figure 3-103 shows a high-level representation of these concepts.

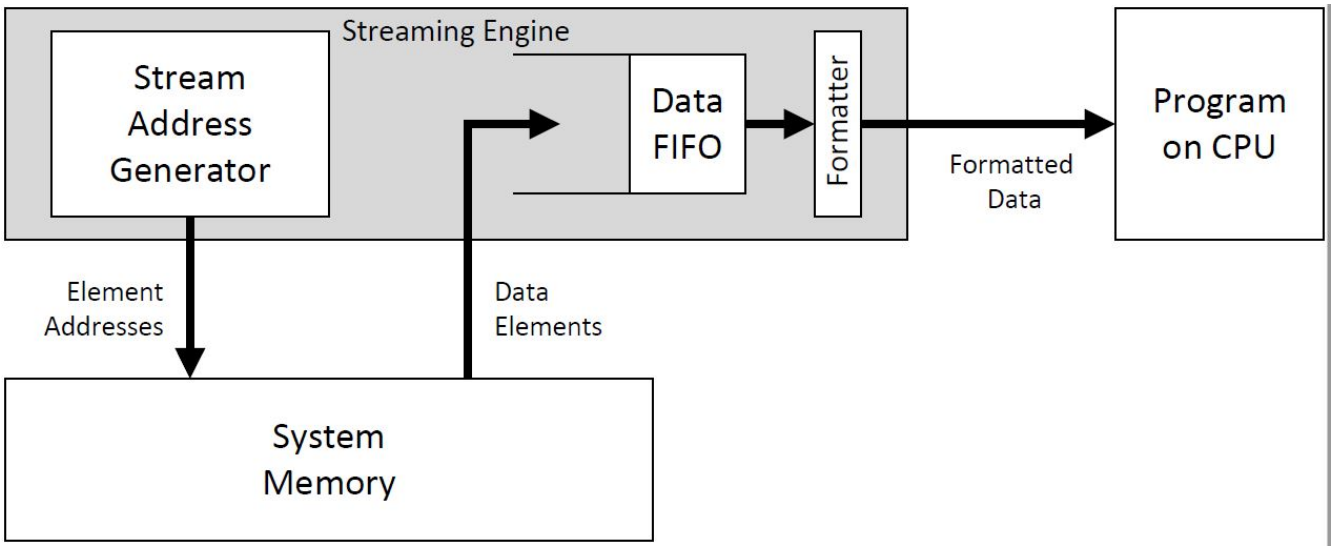


Figure 3-103. High-Level Conceptual View of a Streaming Engine

The streaming process begins with addressing, flows through multiple data formatting steps, and ends at the CPU. Figure 3-104 shows the overall flow between these steps.

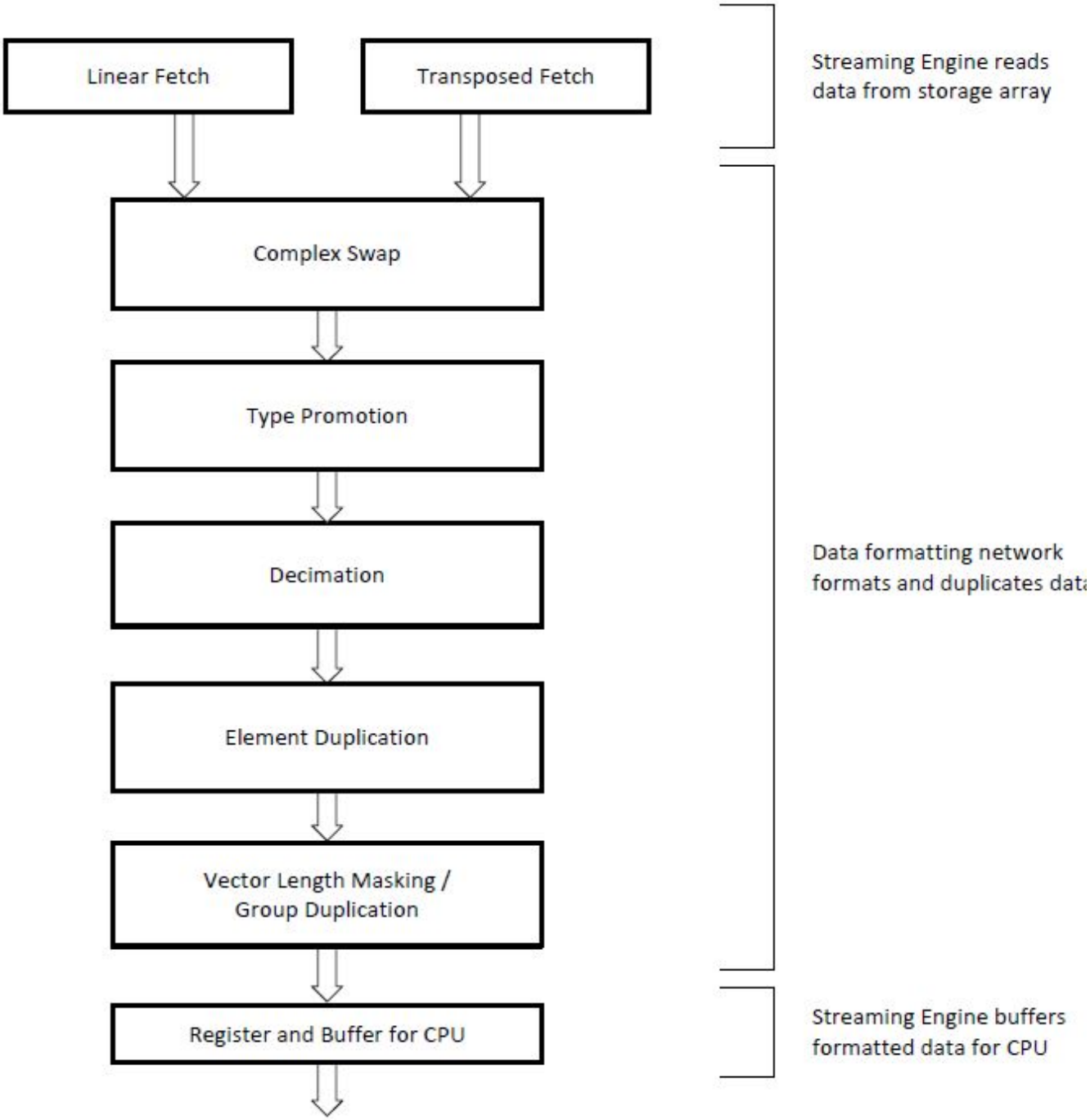


Figure 3-104. Data Formatting Flow

The following sections describes each of these steps in detail.

3.22.3.2.1 Stream Addressing

This streaming engine defines an address sequence for elements of the stream in terms of a pointer walking through memory. A multiple-level loop nest controls the path the pointer takes. The iteration count for a loop level indicates the number of times that level repeats. The dimension gives the distance between pointer positions for consecutive iterations of that loop level.

3.22.3.2.1.1 Linear Stream (Forward)

For a basic, linear stream, the innermost loop always consumes physically contiguous elements from memory. Its implicit dimension is 1 element. The pointer itself moves from element to element in consecutive, increasing

order. In each level outside the inner loop, the loop moves the pointer to a new location based on the size of that loop level's dimension.

The following code illustrates the basic algorithm for a 6-level loop nest. In the conceptual stream model, each call to fetch() in the example below copies an element from memory into the stream data FIFO. This defines the order in which the streaming engine presents elements to the CPU.

Linear stream addressing algorithm:

```
// ptr is a byte pointer.
ptr = BASE_ADDRESS
for (i5 = 0; i5 < ICNT5; i5++)
{
    ptr5 = ptr; // save current position before entering next level
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        ptr4 = ptr; // save current position before entering next level
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            ptr3 = ptr; // save current position before entering next level
            for (i2 = 0; i2 < ICNT2; i2++)
            {
                ptr2 = ptr; // save current position before entering next level
                for (i1 = 0; i1 < ICNT1; i1++)
                {
                    ptr1 = ptr; // save current position before entering next level
                    for (i0 = 0; i0 < ICNT0; i0++)
                    {
                        fetch( ptr, ELEM_BYTES );
                        ptr = ptr + ELEM_BYTES;
                    }
                    // Update based on saved pointer for this level
                    ptr = ptr1 + DIM1;
                }
                // Update based on saved pointer for this level
                ptr = ptr2 + DIM2;
            }
            // Update based on saved pointer for this level
            ptr = ptr3 + DIM3;
        }
        ptr = ptr4 + DIM4;
    }
    ptr = ptr5 + DIM5;
}
```

This form of addressing allows programs to specify regular paths through memory in a small number of parameters. Table 3-148 defines these parameters more explicitly.

Table 3-148. Addressing Parameters for a Basic Stream

Parameter	Definition
ELEM_BYTES	Size of each element in bytes.
ICNT0	Number of iterations for the innermost loop dimension, loop level 0. At this level, all elements are physically contiguous. That is, DIM0 = ELEM_BYTES.
ICNT1	Number of iterations for the first level above the innermost loop, loop level 1.
DIM1	Number of elements between starting points for consecutive iterations of loop level 1.
ICNT2	Number of iterations for loop level 2.
DIM2	Number of elements between starting points for consecutive iterations of loop level 2.
ICNT3	Number of iterations for loop level 3.
DIM3	Number of elements between starting points for consecutive iterations of loop level 3.
ICNT4	Number of iterations for loop level 4.
DIM4	Number of elements between starting points for consecutive iterations of loop level 4.
ICNT5	Number of iterations for loop level 5
DIM5	Number of elementsbetween starting points for consecutive iterations of loop level 5.

3.22.3.2.1.2 Linear Stream (Reverse)

The definition above maps consecutive elements of the stream to increasing addresses in memory. This works well for most algorithms, but not all.

For instance, discrete convolution computes vector dot-products as per Equation 1.

$$(f * g)[t] = \sum_{x=-\infty}^{\infty} f[x]g[t-x] \tag{1}$$

In most DSP code, f[] and g[] represent arrays in memory. For each output, the algorithm reads f[] in the forward direction, but g[] in the reverse direction. Also, practical filters limit the range of indices for [x] and [t - x] to a finite number elements.

To support this and related patterns, the streaming engine supports reading elements in decreasing address order. This effectively reverses the direction of dimension 0, as shown below.

Linear stream addressing, reversed:

```
// ptr is a byte pointer.
ptr = BASE_ADDRESS
for (i5 = 0; i5 < ICNT5; i5++)
{
    ptr5 = ptr; // save current position before entering next level
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        ptr4 = ptr; // save current position before entering next level
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            ptr3 = ptr; // save current position before entering next level
            for (i2 = 0; i2 < ICNT2; i2++)
            {
                ptr2 = ptr; // save current position before entering next level
                for (i1 = 0; i1 < ICNT1; i1++)
                {
                    ptr1 = ptr; // save current position before entering next level
                    for (i0 = 0; i0 < ICNT0; i0++)
                    {
                        ptr = ptr - ELEM_BYTES;
                        fetch( ptr, ELEM_BYTES );
                    }
                    // Update based on saved pointer for this level
                    ptr = ptr1 + DIM1;
                }
                // Update based on saved pointer for this level
                ptr = ptr2 + DIM2;
            }
            // Update based on saved pointer for this level
            ptr = ptr3 + DIM3;
        }
        // Update based on saved pointer for this level
        ptr = ptr4 + DIM4;
    }
    // Update based on saved pointer for this level
    ptr = ptr5 + DIM5;
}
```

The algorithm changes in two ways: The inner loop decrements the element size from the pointer rather than incrementing; and the decrement occurs before the fetch, not after.

3.22.3.2.1.3 Transposed Streams

Linear streams work for large classes of algorithms, but not all. For example, matrix multiplication presents a unique problem for the streaming engine: each element in the matrix product contains the result of a vector dot

product between a row from the first matrix and a column from the second, as illustrated in this example C++ code.

```
Generic Matrix Multiplication in C++
template <typename T, int r1, int c1r2, int c2>
void matrix_multiplication
(
    const         T A[restrict   r1][c1r2],
    const         T B[restrict c1r2][c2  ],      // c1 == r2
          T C[restrict   r1][c2  ]
)
{
    // Multiply each row A[i][:] with each column B[:][j],
    // writing the product to C[j][i].
    for (int i = 0; i < r1; i++)
        for (int j = 0; j < c2; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < c1r2; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}
```

Programs typically store matrices all in row-major or column-major order. Row-major order stores all the elements of a single row contiguously in memory. C and C++ store arrays in row-major order. Column-major order stores all elements of a single column contiguously in memory. FORTRAN stores arrays in column-major order. Whichever the language, matrices typically get stored in the same order as the default array order for the language.

As a result, only one of the two matrices in a matrix multiplication map on to the streaming engine's 2-dimensional stream definition. One can see this in the example above: the index k steps through columns on array A, but rows on array B. This problem is not unique to the streaming engine. In fact, matrix multiplication's access pattern fits poorly with most general-purpose memory hierarchies. Some software libraries attack this problem by directly transposing one of the two matrices, so that both get accessed row-wise (or column-wise) during multiplication. Others use a technique described in the next section.

With the streaming engine, programs need not resort to that extreme. The streaming engine supports implicit matrix transposition with a notion of transposed streams. Transposed streams avoid the cost of explicitly transforming the data in memory. Instead of accessing data in strictly consecutive-element order, the streaming engine effectively interchanges the inner two loop dimensions in its traversal order, fetching elements along the second dimension into contiguous vector lanes.

To a first order, transposed addressing looks like this example of a transposed stream addressing algorithm, baseline:

```
// ptr is a byte pointer.
ptr = BASE_ADDRESS
for (i5 = 0; i5 < ICNT5; i5++)
{
    ptr5 = ptr; // save current position before entering next level
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        ptr4 = ptr; // save current position before entering next level
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            ptr3 = ptr; // save current position before entering next level
            for (i2 = 0; i2 < ICNT2; i2++)
            {
                ptr2 = ptr; // save current position before entering next level
                for (i0 = 0; i0 < ICNT0; i0++)
                {
                        ptr0 = ptr; // save curr pos'n before entering next level
                        for (i1 = 0; i1 < ICNT1; i1++)
                        {
                            fetch( ptr, ELEM_BYTES );
                            ptr = ptr + DIM1;
                        }
                        // Update based on saved pointer for this level
                        ptr = ptr0 + ELEM_BYTES;
                }
```

```
                // Update based on saved pointer for this level
                ptr = ptr2 + DIM2;
            }
            // Update based on saved pointer for this level
            ptr = ptr3 + DIM3;
        }
        // Update based on saved pointer for this level
        ptr = ptr4 + DIM4;
    }
    // Update based on saved pointer for this level
    ptr = ptr5 + DIM5;
}
```

The above algorithm works, but is impractical to implement for small element sizes. Also, some algorithms wish to work on matrix tiles—that is, multiple columns and rows together.

Therefore, the streaming engine defines a separate transposition granularity. The hardware imposes a minimum granularity. The transposition granularity must be at least as large as the element size.

The transposition granularity causes the streaming engine to fetch one or more consecutive elements from dimension 0 before moving along dimension 1. When the granularity equals the element size, this results in fetching a single column from a row-major array. Otherwise, the granularity specifies fetching 2, 4 or more columns at a time from a row-major array. (For column-major layout, exchange row and column in the preceding description.)

The transposition granularity modifies the fetching algorithm as shown below. GRANULE indicates the transposition granularity in bytes.

Transposed stream addressing, with transpose granule:

```
// ptr is a byte pointer.
ptr = BASE_ADDRESS
for (i5 = 0; i5 < ICNT5; i5++)
{
    ptr5 = ptr; // save current position before entering next level
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        ptr4 = ptr; // save current position before entering next level
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            ptr3 = ptr; // save current position before entering next level
            for (i2 = 0; i2 < ICNT2; i2++)
            {
                ptr2 = ptr; // save current position before entering next level
                for (i0 = 0; i0 < ICNT0; i0 += GRANULE / ELEM_BYTES)
                {
                    ptr0 = ptr; // save current position before entering next level
                    for (i1 = 0; i1 < ICNT1; i1++)
                    {
                        // Fetch consecutive elements up to the transpose GRANULE
                        for (t = 0; t < GRANULE; t += ELEM_BYTES)
                            fetch( ptr + t, ELEM_BYTES );
                        ptr = ptr + DIM1;
                    }
                    // Update based on saved pointer for this level
                    ptr = ptr0 + GRANULE;
                }
                // Update based on saved pointer for this level
                ptr = ptr2 + DIM2;
            }
            // Update based on saved pointer for this level
            ptr = ptr3 + DIM3;
        }
        // Update based on saved pointer for this level
        ptr = ptr4 + DIM4;
    }
    // Update based on saved pointer for this level
    ptr = ptr5 + DIM5;
}
```

The above algorithm assumes a forward direction stream. Reverse direction streams visit dimension 0 in decreasing-address order.

Reversed, transposed stream addressing, with transpose granule:

```
// ptr is a byte pointer.
ptr = BASE_ADDRESS
for (i5 = 0; i5 < ICNT5; i5++)
{
    ptr5 = ptr; // save current position before entering next level
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        ptr4 = ptr; // save current position before entering next level
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            ptr3 = ptr; // save current position before entering next level
            for (i2 = 0; i2 < ICNT2; i2++)
            {
                ptr2 = ptr; // save current position before entering next level
                for (i0 = 0; i0 < ICNT0; i0 += GRANULE / ELEM_BYTES)
                {
                    ptr0 = ptr; // save current position before entering next level
                    for (i1 = 0; i1 < ICNT1; i1++)
                    {
                        // Fetch consecutive elements up to the transpose GRANULE
                        for (t = ELEM_BYTES; t <= GRANULE ; t += ELEM_BYTES)
                            fetch( ptr - t, ELEM_BYTES );
                        ptr = ptr + DIM1;
                    }
                    // Update based on saved pointer for this level
                    ptr = ptr0 - GRANULE;
                }
                // Update based on saved pointer for this level
                ptr = ptr2 + DIM2;
            }
            // Update based on saved pointer for this level
            ptr = ptr3 + DIM3;
        }
        // Update based on saved pointer for this level
        ptr = ptr4 + DIM4;
    }
    // Update based on saved pointer for this level
    ptr = ptr5 + DIM5;
}
```

3.22.3.2.1.4 Circular Addressing

Circular addressing modifies how the streaming engine performs address arithmetic so that addresses remain within a power-of-2 sized window. Circular addressing works by holding upper address bits constant during an address update, while allowing the lower address bits to vary. This contrasts with the default—linear addressing—which allows all of the address bits to vary.

The streaming engine provides address mode selection for each level of loop nest. Each level can select between linear addressing or circular addressing with one of two circular block sizes.

Conceptually, selectable addressing mode support replaces all address arithmetic with a function similar to the following.

Address arithmetic with selectable linear / circular addressing modes:

```
enum addr_mode_t
{
    LINEAR,                // Linear addressing mode
    CIRC0,                // Circular addressing with block size 0
    CIRC1                // Circular addressing with block size 1
};
uint64_t circ_mask_0;                        // Circular addressing mask for block size 0
uint64_t circ_mask_1;                        // Circular addressing mask for block size 1
uint64_t address_add( uint64_t base, int32_t offset, addr_mode_t mode )
{
    uint64_t new_addr = base + offset;
    if ( mode == LINEAR )
        return new_addr;
    // Look up address mask based on selected circular buffer size.
    // Mask contains 1s for bits that remain fixed, and 0s elsewhere.
    uint64_t addr_mask = mode == CIRC0 ? circ_mask_0 : circ_mask_1;
```

```
    return ( base & addr_mask ) | ( new_addr & ~addr_mask );
}
```

The address_add primitive then takes the place of plain addition for all stream address computations. The following example illustrates address_add in the context of a simple forward stream.

Linear stream addressing algorithm:

```
// ptr is a byte pointer.
ptr = BASE_ADDRESS
for (i5 = 0; i5 < ICNT5; i5++)
{
    ptr5 = ptr; // save current position before entering next level
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        ptr4 = ptr; // save current position before entering next level
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            ptr3 = ptr; // save current position before entering next level
            for (i2 = 0; i2 < ICNT2; i2++)
            {
                ptr2 = ptr; // save current position before entering next level
                for (i1 = 0; i1 < ICNT1; i1++)
                {
                    ptr1 = ptr; // save current position before entering next level
                    for (i0 = 0; i0 < ICNT0; i0++)
                    {
                        fetch( ptr, ELEM_BYTES );
                        ptr = address_add( ptr, ELEM_BYTES, addr_mode_0 );
                    }
                    // Update based on saved pointer for this level
                    ptr = address_add( ptr1, DIM1, addr_mode_1 );
                }
                // Update based on saved pointer for this level
                ptr = address_add( ptr2, DIM2, addr_mode_2 );
            }
            // Update based on saved pointer for this level
            ptr = address_add( ptr3, DIM3, addr_mode_3 );
        }
        ptr = address_add( ptr4, DIM4, addr_mode_4 );
    }
    ptr = address_add( ptr5, DIM5, addr_mode_5 );
}
```

To reduce clutter, this specification omits address_add calls in most stream addressing algorithm examples. However, each address computation should be understood to use address_add for its address arithmetic.

In transposed streams, addr_mode_0 applies to dimension 0 and addr_mode_1 applies to dimension 1, despite the fact that the streaming engine interchanges these two loop levels.

3.22.3.2.1.5 Early Exits (Breaks) in Streams

By default, the streaming engine loops through all elements defined by the stream's parameters. Programs may need to skip some elements in the stream.

The streaming engine provides a simple mechanism to exit early—that is, break—from one or more levels of loop nest between vectors of elements. In other words, stream breaks always happen at vector boundaries. A stream break skips the remaining elements in each loop level broken out of, resuming at the next iteration of the containing loop.

In the following algorithm, the variables break0, break1, break2, break3, and break4 indicate whether the streaming engine should break out of level 0, 1, 2, 3, or 4 of the loop nest. In hardware, the program requests stream breaks with dedicated instructions that set these flags outside this code.

Forward stream addressing algorithm, with break tests:

```
// ptr is a byte pointer.
ptr = BASE_ADDRESS
break5 = false;
for (i5 = 0; i5 < ICNT5; i++)
{
    ptr5 = ptr; // save current position before entering next level
```

```
    break4 = false;
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        ptr4 = ptr; // save current position before entering next level
        break3 = false;
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            ptr3 = ptr; // save current position before entering next level
            break2 = false;
            for (i2 = 0; i2 < ICNT2; i2++)
            {
                ptr2 = ptr; // save current position before entering next level
                break1 = false;
                for (i1 = 0; i1 < ICNT1; i1++)
                {
                    ptr1 = ptr; // save current position before entering next level
                    break0 = false;
                    for (i0 = 0; i0 < ICNT0; i0++)
                    {
                        fetch( ptr, ELEM_BYTES );
                        ptr = ptr + ELEM_BYTES;

                        if (break0 || break1 || break2 || break3 || break4 || break5)
                            break;
                    }
                    // Update based on saved pointer for this level
                    ptr = ptr1 + DIM1;

                    if (break1 || break2 || break3 || break4 || break5)
                        break;
                }
                // Update based on saved pointer for this level
                ptr = ptr2 + DIM2;

                if (break2 || break3 || break4 || break5)
                    break;
            }
            // Update based on saved pointer for this level
            ptr = ptr3 + DIM3;

            if (break3 || break4 || break5)
                break;
        }
        // Update based on saved pointer for this level
        ptr = ptr4 + DIM4;

        if (break4 || break5)
            break;
    }
    // Update based on saved pointer for this level
    ptr = ptr5 + DIM5;
    if (break5)
        break;
}
```

Breaks always count from the innermost dimension outward, as defined by the type of stream. For example, transposed streams make dimension 1 the innermost dimension, as opposed to dimension 0. The following algorithm example illustrates how breaks apply to transposed streams.

Transposed stream addressing, with transpose granule and break tests:

```
// ptr is a byte pointer.
ptr = BASE_ADDRESS
break5 = false;
for (i5 = 0; i5 < ICNT5; i++)
{
    ptr5 = ptr; // save current position before entering next level
    break4 = false;
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        ptr4 = ptr; // save current position before entering next level
        break3 = false;
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            ptr3 = ptr; // save current position before entering next level
            break2 = false;
            for (i2 = 0; i2 < ICNT2; i2++)
```

```
            {
                ptr2 = ptr; // save current position before entering next level
                break1 = false;
                for (i0 = 0; i0 < ICNT0; i0 += GRANULE / ELEM_BYTES)
                {
                    ptr0 = ptr; // save current position before entering next level
                    break0 = false;
                    for (i1 = 0; i1 < ICNT1; i1++)
                    {
                        // Fetch consecutive elements up to the transpose GRANULE
                        for (t = 0; t < GRANULE; t += ELEM_BYTES)
                            fetch( ptr + t, ELEM_BYTES );
                        ptr = ptr + DIM1;
                        if (break0 || break1 || break2 || break3 || break4 || break5)
                            break;
                    }
                    // Update based on saved pointer for this level
                    ptr = ptr0 + GRANULE;
                    if (break1 || break2 || break3 || break4 || break5)
                        break;
                }
                // Update based on saved pointer for this level
                ptr = ptr2 + DIM2;
                if (break2 || break3 || break4 || break5)
                    break;
            }
            // Update based on saved pointer for this level
            ptr = ptr3 + DIM3;

            if (break3 || break4 || break5)
                break;
        }
        // Update based on saved pointer for this level
        ptr = ptr4 + DIM4;

        if (break4 || break5)
            break;
    }
    // Update based on saved pointer for this level
    ptr = ptr5 + DIM5;
    if (break5)
        break;
}
```

Programs request stream breaks with a dedicated instruction; see Section 3.22.4.5.2 for the instructions that implement stream breaks.

3.22.3.2.2 Data Formatting

The streaming engine formats data after reading it from memory, to streamline data processing. A stream definition may specify type promotion, sub-element swap for complex data, and element duplication.

Data formatting is completely orthogonal to stream addressing. The streaming engine supports the full range of data formatting options for both linear and transposed streams, in both forward and reverse directions.

3.22.3.2.2.1 Type Promotion

Algorithms generally prefer to work with high precision values, but higher precision values require more storage and bandwidth than lower precision values. Commonly, programs store data in memory at low precision, and promote those values to a higher precision before operating on them. Then, at the end, they then might demote the values to lower precision.

The streaming engine supports this directly by allowing algorithms to specify integer promotion from 8, 16, and 32 bit to any larger integer type, up to 64 bits. Every sub-element gets promoted to the target size, with either sign or zero extension.

3.22.3.2.2.2 2:1 and 4:1 Decimation

Many algorithms operate on every second element or every fourth element of a larger structure. To support this, the streaming engine supports decimating data by a factor of 2 or by a factor of 4. Decimation 2:1 drops every odd-numbered element from the stream, returning only the even-numbered elements. Decimation 4:1 drops three elements from the stream, returning every other fourth element.

Due to limitations in the formatting network, the streaming engine only supports decimation 2:1 when also promoting the integer type by at least a factor of 2.

Likewise, the streaming engine only supports decimation 4:1 when also promoting the integer type by at least a factor of 4.

The streaming engine always drops whole elements, not sub-elements. Promotion does not directly affect decimation. The only relationship between promotion and decimation is that promotion must be enabled to enable decimation.

Because decimation considers elements in pairs of two for 2:1 and considers elements in groups of 4 for 4:1, ICNT0 must be multiple of 2 when decimating by 2:1, or multiple of 4 when decimating by 4:1. This is true for both linear streams and transposed streams.

In transposed streams, decimation always removes columns, and never removes rows. To ensure this is the case, the transpose granule must be at least twice the element size for 2:1 decimation and four times the element size for 4:1 decimation —i.e. GRANULE ≥ 2 × ELEM_BYTES, or GRANULE ≥ 4× ELEM_BYTES respectively.

Decimation occurs before element and group duplication.

3.22.3.2.2.3 Complex Data Sub-element Swap

As stated earlier, the streaming engine defines a complex element as a single element with two sub-elements. These sub-elements may represent the real and imaginary (rectangular) or magnitude and angle (polar) portions of the complex number.

Not all programs or peripherals agree what order these sub-elements should appear in memory. Therefore, the streaming engine offers the ability to swap the two sub-elements of a complex number with no cost.

3.22.3.2.2.4 Element Duplication

Some algorithms use each element more than once. To accommodate these algorithms, the streaming engine can duplicate each element a power-of-2 number of times. Element duplication resembles an additional level of loop nest, repeating the call to fetch multiple times.

Element duplication always duplicates individual elements. It never duplicates sub-elements, for example. It also ignores the transposition granularity in transposed streams. The following example illustrates the basic linear fetch algorithm, modified to include element duplication. The variable ELDUP corresponds to the duplication factor.

Linear stream addressing algorithm, with element duplication:

```
// ptr is a byte pointer.
ptr = BASE_ADDRESS
for (i5 = 0; i5 < ICNT5; i5++)
{
    ptr5 = ptr; // save current position before entering next level
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        ptr4 = ptr; // save current position before entering next level
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            ptr3 = ptr; // save current position before entering next level
            for (i2 = 0; i2 < ICNT2; i2++)
            {
                ptr2 = ptr; // save current position before entering next level
                for (i1 = 0; i1 < ICNT1; i1++)
                {
                    ptr1 = ptr; // save current position before entering next level
                    for (i0 = 0; i0 < ICNT0; i0++)
                    {
                        for (d = 0; d < ELDUP; d++)
                            fetch( ptr, ELEM_BYTES );
                        ptr = ptr + ELEM_BYTES;
                    }
                    // Update based on saved pointer for this level
                    ptr = ptr1 + DIM1;
                }
                // Update based on saved pointer for this level
```

```
                ptr = ptr2 + DIM2;
            }
            // Update based on saved pointer for this level
            ptr = ptr3 + DIM3;
        }
        // Update based on saved pointer for this level
        ptr = ptr4 + DIM4;
    }
    // Update based on saved pointer for this level
    ptr = ptr5 + DIM5;
}
```

The next example illustrates transposed stream addressing, modified to include element duplication. Duplication occurs at the element level, regardless of the transposition granularity.

Transposed stream addressing, with element duplication:

```
// ptr is a byte pointer.
ptr = BASE_ADDRESS
for (i5 = 0; i5 < ICNT5; i5++)
{
    ptr5 = ptr; // save current position before entering next level
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        ptr4 = ptr; // save current position before entering next level
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            ptr3 = ptr; // save current position before entering next level
            for (i2 = 0; i2 < ICNT2; i2++)
            {
                ptr2 = ptr; // save current position before entering next level
                for (i0 = 0; i0 < ICNT0; i0 += GRANULE / ELEM_BYTES)
                {
                    ptr0 = ptr; // save current position before entering next level
                    for (i1 = 0; i1 < ICNT1; i1++)
                    {
                        // Fetch consecutive elements up to the transpose GRANULE
                        for (t = 0; t < GRANULE; t += ELEM_BYTES)
                            for (d = 0; d < ELDUP; d++)
                                fetch( ptr + t, ELEM_BYTES );
                        ptr = ptr + DIM1;
                    }
                    // Update based on saved pointer for this level
                    ptr = ptr0 + GRANULE;
                }
                // Update based on saved pointer for this level
                ptr = ptr2 + DIM2;
            }
            // Update based on saved pointer for this level
            ptr = ptr3 + DIM3;
}
        // Update based on saved pointer for this level
        ptr = ptr4 + DIM4;
    }
    // Update based on saved pointer for this level
    ptr = ptr5 + DIM5;
}
```

3.22.3.2.3 Mapping Streams to Vectors

While the streaming engine defines a stream as a discrete sequence of elements, the DSP CPU consumes elements packed contiguously in vectors.

These vectors resemble streams in as much as they contain multiple homogeneous elements, with some implicit sequence. Because the streaming engine reads streams, but the DSP CPU consumes vectors, the streaming engine must map streams onto vectors consistently.

Vectors consist of equal-sized lanes, each lane containing a sub-element. The DSP CPU designates the rightmost lane of the vector as lane 0, regardless of device's current endian mode. Lane numbers increase right-to-left. The actual number of lanes within a vector varies depending on the length of the vector and the size of the sub-element type for the stream. For example, a stream with a vector length of 4 elements and an element size of 2-bytes, occupies 8 byte lanes when presented to the CPU with no data promotion or element duplication.

The streaming packs elements into lanes in increasing lane order. It never subdivides elements, even if they consist of multiple sub-elements.

3.22.3.2.3.1 Stream Vector Length versus CPU Vector Length

The DSP CPU has a single native vector width. In the current generation DSP, this is fixed at 512 bits; however, variants of the CPU have been proposed with narrower vector widths. Future family members may have larger vector lengths.

Separately of the DSP vector length, streams define the length of vector they pack into, starting with a vector length of 1 element, and increasing by powers of 2, up to a max of 64 elements. The streaming engine presents the stream to the CPU in groups of this size. The streaming engine supports stream vector lengths up to the size of the CPU's vector. See Section 3.22.4.3.10.

3.22.3.2.3.2 Stream Dimensions and Vectors

The streaming engine groups elements from the innermost stream dimension into stream vector length-sized groups of elements. Earlier elements always go in lower numbered lanes than later elements within the same vector.

This is true regardless of whether this particular stream goes in increasing or decreasing address order. Whatever order the stream defines, the streaming engine deposits elements in vectors in increasing-lane order. For non-complex data, it places the first element in lane 0 of the first vector the CPU fetches, the second in lane 1, and so on.

For complex data, it places the first element in lanes 0 and 1, second in lanes 2 and 3, and so on. Sub-elements within an element retain the same relative ordering regardless of the stream direction. For non-swapped complex elements, this places the sub-elements with the lower address of each pair in the even numbered lanes, and the sub-elements with the higher address of each pair in the odd numbered lanes. Swapped complex elements reverse this mapping.

The streaming engine fills each vector the CPU fetches with as many elements as it can from the innermost stream dimension. If the innermost dimension is not a multiple of the CPU's native vector length, the streaming engine pads that dimension out to a multiple of the vector length with zeros.

Thus, for higher-dimension streams, the first element from each iteration of an outer dimension arrives in lane 0 of a vector.

The streaming engine always maps the innermost dimension to consecutive lanes in a vector. For transposed streams, the innermost dimension consists of groups of sub-elements along dimension 1, not dimension 0, as transposition exchanges these two dimensions.

3.22.3.2.3.3 Stream Padding, Valid and Invalid Lanes

When mapping stream elements to CPU vectors, one or more vector lanes may not contain an actual element from the stream. For example, the stream vector length may be less than the CPU vector length. Or, the stream's innermost dimension may not be a multiple of the stream vector length.

In these situations, the streaming engine fills the extra lanes with zeros—specifically, all bits zero—and marks the lanes as invalid. The streaming engine marks lanes with actual data as valid.

The streaming engine exposes each lane's validity to the CPU by way of the CPU's vector predicate registers. Section 3.22.4.6.2 defines this mapping.

As of spec version 0.80, the "Stream Vector Predicates" have been de-speced and CPU does not support vector predicates. The streaming engine drives all the byte predicates to zero. This feature may be supported in next generation of the CPU.

3.22.3.2.3.4 Example Stream to Vector Mappings

Each of the following examples examine how the streaming engine maps a one dimensional stream onto vectors, across different stream vector lengths and CPU vector sizes. The stream consists of 29 elements, each 64 bits.

Table 3-149 illustrates how the example stream maps onto bits within 512-bit CPU vectors, given a stream vector length of 512-bits (8-elements), with no data formatting.

Table 3-149. Mapping Elements to Bits Within Vectors; Stream Vector Length of 512 bits

Vector	Bits 511 - 448	Bits 447 - 384	Bits 383 - 320	Bits 319 - 256	Bits 255 - 192	Bits 191 - 128	Bits 127 - 64	Bits 63 - 0
1	Element 7	Element 6	Element 5	Element 4	Element 3	Element 2	Element 1	Element 0
2	Element 15	Element 14	Element 13	Element 12	Element 11	Element 10	Element 9	Element 8
3	Element 23	Element 22	Element 21	Element 20	Element 19	Element 18	Element 17	Element 16
4	Ø	Ø	Ø	Element 28	Element 27	Element 26	Element 25	Element 24

Table 3-150 and Table 3-151 show the same example as above, except with a stream vector length of 256 bits (4-elements) or 128 bits (2-elements). They illustrate how the streaming engine fills unused lanes with zeros.

Table 3-150. Mapping Elements to Bits Within Single Vector Pairs; Stream Vector Length of 256 bits

Vector	Bits 511 - 448	Bits 447 - 384	Bits 383 - 320	Bits 319 - 256	Bits 255 - 192	Bits 191 - 128	Bits 127 - 64	Bits 63 - 0
1	Ø	Ø	Ø	Ø	Element 3	Element 2	Element 1	Element 0
2	Ø	Ø	Ø	Ø	Element 7	Element 6	Element 5	Element 4
3	Ø	Ø	Ø	Ø	Element 11	Element 10	Element 9	Element 8
4	Ø	Ø	Ø	Ø	Element 15	Element 14	Element 13	Element 12
5	Ø	Ø	Ø	Ø	Element 19	Element 18	Element 17	Element 16
6	Ø	Ø	Ø	Ø	Element 23	Element 22	Element 21	Element 20
7	Ø	Ø	Ø	Ø	Element 27	Element 26	Element 25	Element 24
8	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Element 28

Table 3-151. Mapping Elements to Bits Within Vectors; Stream Vector Length of 128 bits

Vector	Bits 511 - 448	Bits 447 - 384	Bits 383 - 320	Bits 319 - 256	Bits 255 - 192	Bits 191 - 128	Bits 127 - 64	Bits 63 - 0
1	Ø	Ø	Ø	Ø	Ø	Ø	Element 1	Element 0
2	Ø	Ø	Ø	Ø	Ø	Ø	Element 3	Element 2
3	Ø	Ø	Ø	Ø	Ø	Ø	Element 5	Element 4
4	Ø	Ø	Ø	Ø	Ø	Ø	Element 7	Element 6
5	Ø	Ø	Ø	Ø	Ø	Ø	Element 9	Element 8
6	Ø	Ø	Ø	Ø	Ø	Ø	Element 11	Element 10
7	Ø	Ø	Ø	Ø	Ø	Ø	Element 13	Element 12
8	Ø	Ø	Ø	Ø	Ø	Ø	Element 15	Element 14
9	Ø	Ø	Ø	Ø	Ø	Ø	Element 17	Element 16
10	Ø	Ø	Ø	Ø	Ø	Ø	Element 19	Element 18
11	Ø	Ø	Ø	Ø	Ø	Ø	Element 21	Element 20
12	Ø	Ø	Ø	Ø	Ø	Ø	Element 23	Element 22
13	Ø	Ø	Ø	Ø	Ø	Ø	Element 25	Element 24
14	Ø	Ø	Ø	Ø	Ø	Ø	Element 27	Element 26
15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Element 28

3.22.3.2.3.5 Group Duplication

The streaming engine provides the ability to duplicate a group of elements to fill the machine's vector width. The stream vector length defines the length of the group to replicate.

When the stream vector length is less than the CPU's vector length, and the stream enables group duplication, the streaming engine fills the extra lanes with additional copies of the stream vector. Because stream vector lengths and CPU vector lengths are always powers of two, group duplication always produces a power of two number of duplicate copies.

Table 3-152 and Table 3-153 illustrate the effect of group duplication on the examples in Table 3-150 and Table 3-151.

Table 3-152. Stream Vector Length of 256 bits, and Group Duplication

Vector	Bits 511 - 448	Bits 447 - 384	Bits 383 - 320	Bits 319 - 256	Bits 255 - 192	Bits 191 - 128	Bits 127 - 64	Bits 63 - 0
1	Element 3	Element 2	Element 1	Element 0	Element 3	Element 2	Element 1	Element 0
2	Element 7	Element 6	Element 5	Element 4	Element 7	Element 6	Element 5	Element 4
3	Element 11	Element 10	Element 9	Element 8	Element 11	Element 10	Element 9	Element 8

Table 3-152. Stream Vector Length of 256 bits, and Group Duplication (continued)

Vector	Bits 511 - 448	Bits 447 - 384	Bits 383 - 320	Bits 319 - 256	Bits 255 - 192	Bits 191 - 128	Bits 127 - 64	Bits 63 - 0
4	Element 15	Element 14	Element 13	Element 12	Element 15	Element 14	Element 13	Element 12
5	Element 19	Element 18	Element 17	Element 16	Element 19	Element 18	Element 17	Element 16
6	Element 23	Element 22	Element 21	Element 20	Element 23	Element 22	Element 21	Element 20
7	Element 27	Element 26	Element 25	Element 24	Element 27	Element 26	Element 25	Element 24
8	Ø	Ø	Ø	Element 28	Ø	Ø	Ø	Element 28

Table 3-153. Stream Vector Length of 128 bits, and Group Duplication

Vector	Bits 511 - 448	Bits 447 - 384	Bits 383 - 320	Bits 319 - 256	Bits 255 - 192	Bits 191 - 128	Bits 127 - 64	Bits 63 - 0
1	Element 1	Element 0	Element 1	Element 0	Element 1	Element 0	Element 1	Element 0
2	Element 3	Element 2	Element 3	Element 2	Element 3	Element 2	Element 3	Element 2
3	Element 5	Element 4	Element 5	Element 4	Element 5	Element 4	Element 5	Element 4
4	Element 7	Element 6	Element 7	Element 6	Element 7	Element 6	Element 7	Element 6
5	Element 9	Element 8	Element 9	Element 8	Element 9	Element 8	Element 9	Element 8
6	Element 11	Element 10	Element 11	Element 10	Element 11	Element 10	Element 11	Element 10
7	Element 13	Element 12	Element 13	Element 12	Element 13	Element 12	Element 13	Element 12
8	Element 15	Element 14	Element 15	Element 14	Element 15	Element 14	Element 15	Element 14
9	Element 17	Element 16	Element 17	Element 16	Element 17	Element 16	Element 17	Element 16
10	Element 19	Element 18	Element 19	Element 18	Element 19	Element 18	Element 19	Element 18
11	Element 21	Element 20	Element 21	Element 20	Element 21	Element 20	Element 21	Element 20
12	Element 23	Element 22	Element 23	Element 22	Element 23	Element 22	Element 23	Element 22
13	Element 25	Element 24	Element 25	Element 24	Element 25	Element 24	Element 25	Element 24
14	Element 27	Element 26	Element 27	Element 26	Element 27	Element 26	Element 27	Element 26
15	Ø	Element 28	Ø	Element 28	Ø	Element 28	Ø	Element 28

3.22.3.2.3.6 Reading Beyond the End of Stream

The streaming engine allows programs to read beyond the end of stream. If a program reads beyond the end of stream, the streaming engine fills all subsequent vectors with zeros, and marks all lanes invalid.

3.22.3.3 Conceptual Stream Examples

The following sections give some simple stream examples, to make the concepts more concrete. They also illustrate how streams map elements in memory to elements in vectors.

3.22.3.3.1 One Dimensional Stream

Consider a simple stream consisting of a block of consecutive elements—a “one dimensional” stream. Such streams require just two parameters from the above set:

- The element size (ELEM_BYTES)
- The iteration count for loop level 0 (ICNT0)

The remaining loop levels do not iterate, so should be ‘1’ for this stream. Because levels 1 and above do not iterate, the streaming engine ignores those dimensions.

This first example illustrates a one dimensional stream mapped onto memory. In this example the stream has the following parameters:

- Starting address 0x1010
- ELEM_BYTES = 4 bytes
- ICNT0 = 1021
- ICNT1 through ICNT5 = 1
- DIM1 through DIM5 values do not matter

Figure 3-105 shows how elements 0 through 1020 (1021 total elements) map onto memory locations 0x1010 through 0x2003. Memory addresses read in increasing order from left to right and top to bottom. The first element resides at 0x1010 - 0x1013, and the last element resides at 0x2000 - 0x2003.

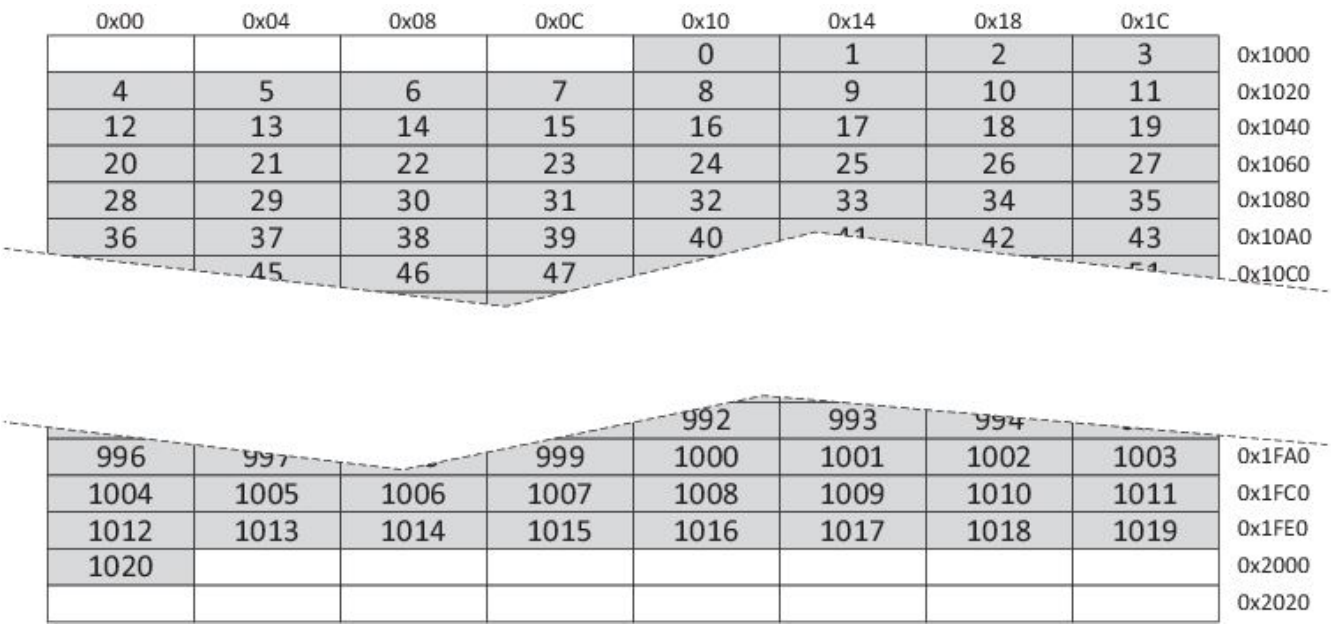


Figure 3-105. One Dimensional Stream Example

As shown in this example, the stream resides in 4080 consecutive bytes of RAM, starting at address 0x1010 and ending at address 0x2003. The outer loop dimensions play no role in mapping this stream to memory.

Table 3-154 shows how elements in memory map to lanes in 512-bit vectors.

Table 3-154. Mapping Elements to Vector Lanes for 1-D Stream

Fetch	Lanes															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1[1]	104C[2]	1048	1044	1040	103C	1038	1034	1030	102C	1028	1024	1020	101C	1018	1014	1010
2	108C	1088	1084	1080	107C	1078	1074	1070	106C	1068	1064	1060	105C	1058	1054	1050
3	10CC	10C8	10C4	10C0	10BC	10B8	10B4	10B0	10AC	10A8	10A4	10A0	109C	1098	1094	1090
4	110C	1108	1104	1100	10FC	10F8	10F4	10F0	10EC	10E8	10E4	10E0	10DC	10D8	10D4	10D0
...	...															
62	1F8C	1F88	1F84	1F80	1F7C	1F78	1F74	1F70	1F6C	1F68	1F64	1F60	1F5C	1F58	1F54	1F50
63	1FCC	1FC8	1FC4	1FC0	1FBC	1FB8	1FB4	1FB0	1FAC	1FA8	1FA4	1FA0	1F9C	1F98	1F94	1F90
64	Ø[3]	Ø	Ø	2000	1FFC	1FF8	1FF4	1FF0	1FEC	1FE8	1FE4	1FE0	1FDC	1FD8	1FD4	1FD0

(1) The CPU fetch sequence here assumes no data formatting is enabled. When data formatting is enabled, additional fetches are required to consume all the data.

(2) All addresses in hexadecimal. Each table element with an address indicates a word sourced from the indicated address.

(3) Ø indicates a lane filled with zeros and marked invalid.

In the above stream, the CPU processes elements in increasing-address order. This is the most natural order for accessing most streams. As stated before, algorithms such as convolution naturally try to access a stream in reverse order.

Figure 3-106 illustrates the access sequence for the same set of elements, but with the following parameters:

- Starting address 0x2004
- Reverse direction
- ELEM_BYTES = 4 bytes
- ICNT0 = 1021
- ICNT1through ICNT5 = 1
- DIM1 through DIM5 values do not matter

The above parameters result in the element sequence shown in Figure 3-106.

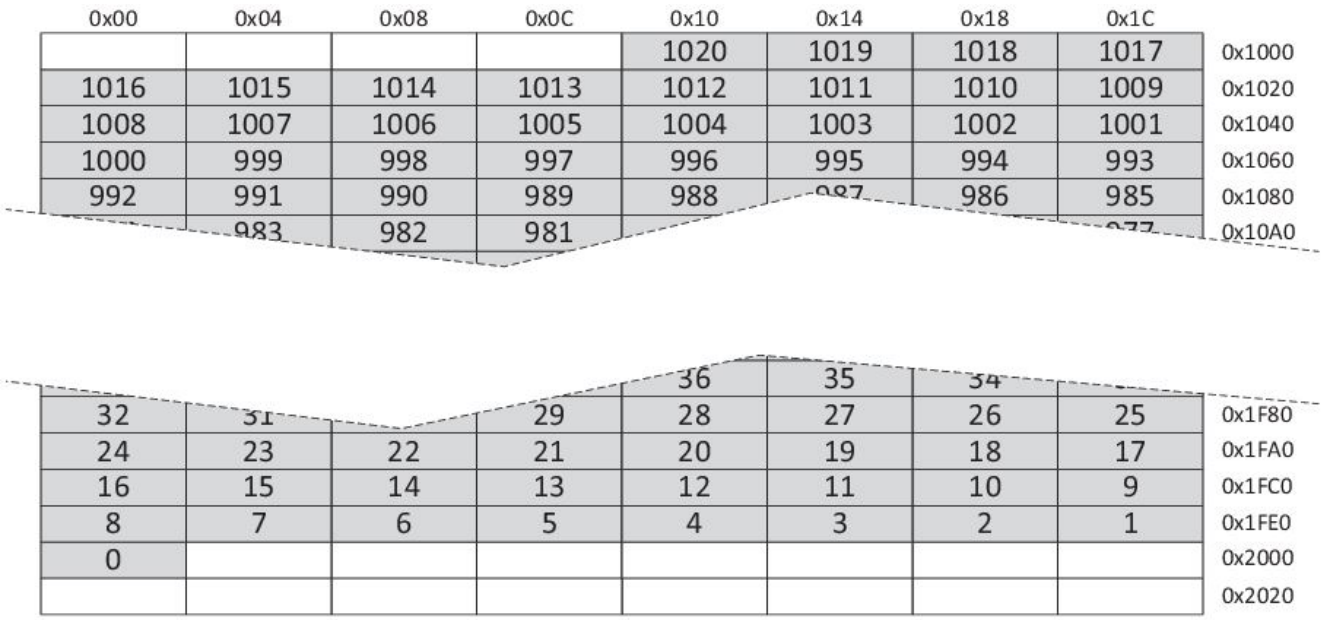


Figure 3-106. Reversed 1-D Stream

This modified stream accesses the same elements, in reverse order. The first element comes from location 0x2000, and the last element comes from location 0x1010.

Note

The parameters for this stream give the starting address as 0x2004, even though the first element resides at 0x2000. That is because the algorithm for reversed streams decrements the address before it fetches from memory.

Table 3-155 illustrates how addresses map to lanes for the example above.

Table 3-155. Mapping Elements to Vector Lanes for 1-D Stream, Reversed

Fetch	Lanes															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1(1)	1FC4(2)	1FC8	1FCC	1FD0	1FD4	1FD8	1FDC	1FE0	1FE4	1FE8	1FEC	1FF0	1FF4	1FF8	1FFC	2000
2	1F84	1F88	1F8C	1F90	1F94	1F98	1F9C	1FA0	1FA4	1FA8	1FAC	1FB0	1FB4	1FB8	1FBC	1FC0
3	1F44	1F48	1F4C	1F50	1F54	1F58	1F5C	1F60	1F64	1F68	1F6C	1F70	1F74	1F78	1F7C	1F80
4	1F04	1F08	1F0C	1F10	1F14	1F18	1F1C	1F20	1F24	1F28	1F2C	1F30	1F34	1F38	1F3C	1F40
...	...															
62	1084	1088	108C	1090	1094	1098	109C	10A0	10A4	10A8	10AC	10B0	10B4	10B8	10BC	10C0
63	1044	1048	104C	1050	1054	1058	105C	1060	1064	1068	106C	1070	1074	1078	107C	1080
64	Ø(3)	Ø	Ø	1010	1014	1018	101C	1020	1024	1028	102C	1030	1034	1038	103C	1040

(1) The CPU fetch sequence here assumes no data formatting is enabled. When data formatting is enabled, additional fetches are required to consume all the data.

(2) All addresses in hexadecimal. Each table element with an address indicates a word sourced from the indicated address.

(3) Ø indicates a lane filled with zeros and marked invalid.

3.22.3.3.2 Two Dimensional Streams

Two dimensional streams exhibit great variety as compared to one dimensional streams. The following subsections explore a few important varieties.

- Basic 2-D stream: Extracts a smaller rectangle from a larger rectangle.
- Transposed 2-D stream: Reads a rectangle column-wise instead of row-wise.
- Looping streams, where second dimension overlaps first:
  - FIR filter taps which ‘loop’ repeatedly

– FIR filter samples which provide a sliding window of input samples

Streams can represent each of these patterns. The following sections illustrate these patterns.

3.22.3.3.2.1 Basic 2-D Stream

Consider a basic two dimensional stream. The inner two dimensions, represented by ELEM_BYTES, ICNT0, DIM1, and ICNT1 give sufficient flexibility to describe extracting a smaller rectangle from a larger rectangle, as shown in Figure 3-107.

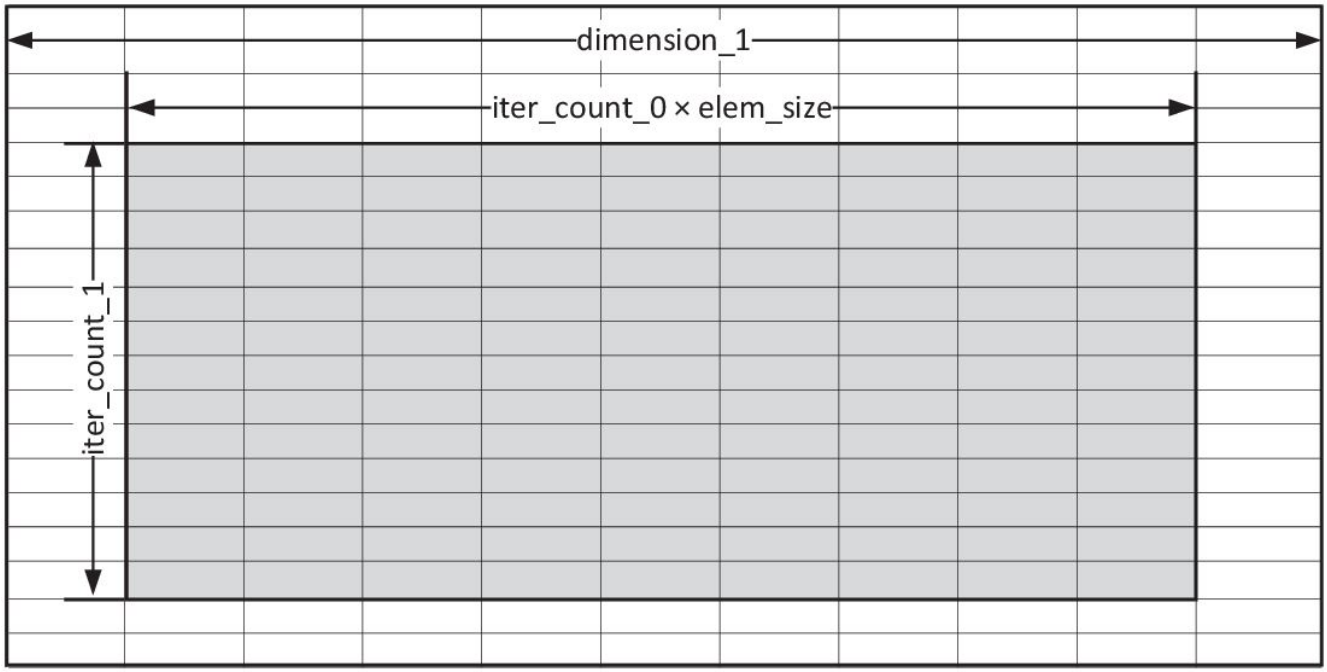


Figure 3-107. Two Dimensional Addressing Example

This example extracts a 9 by 13 rectangle of 64-bit values from a larger 11 by 19 rectangle. The following stream parameters define this stream:

- ICNT0 = 9
- ELEM_BYTES = 8
- ICNT1 = 13
- DIM1 = 11 (11 × ELEM_BYTES)

The ELEM_BYTES parameter only scales the innermost dimension. The first dimension has ICNT0 elements of size ELEM_BYTES. The stream address generator also scale the outer dimensions. Therefore, DIM1 = 11, which is 11 elements scaled by 8 bytes per element.

Figure 3-108 illustrates the order of elements within the stream. The first 9 elements come from the first row of the rectangle, left-to-right. The 10th through 18th elements comes from the second row, and so on.

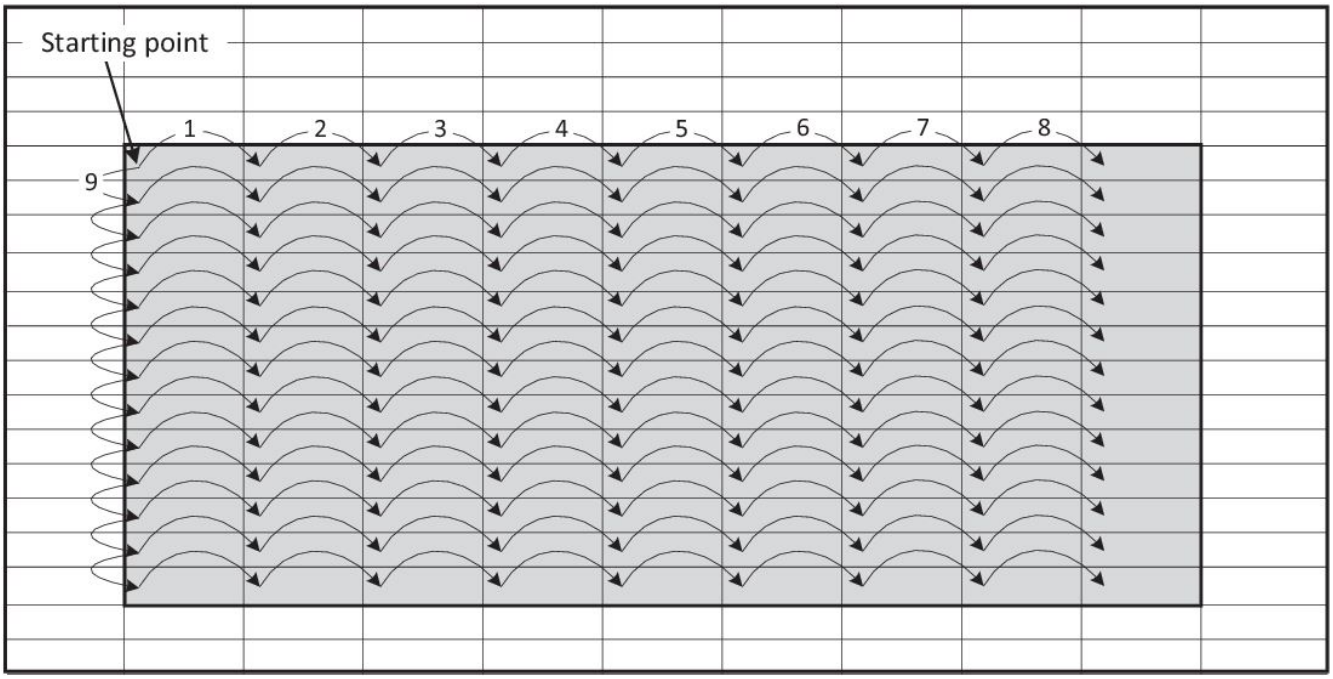


Figure 3-108. Element Sequence in a Two Dimensional Stream

When the stream moves from the 9th element to the 10th element (hop number 9 in Figure 3-108), the streaming engine computes the new location based on the pointer's position at the start of the inner loop, not where the pointer ended up at the end of the first dimension.

This makes DIM1 independent of ELEM_BYTES and ICNT0. It always represents the distance between the first bytes of each consecutive row.

3.22.3.3.2.2 Transposed 2-D Streams

Transposed streams access along dimension 1 before dimension 0, as described in Section 3.22.3.2.1.3. The following examples illustrate some transposed streams, varying the transposition granularity.

Consider a 12 × 8 array of 16-bit elements, as shown in Figure 3-109.

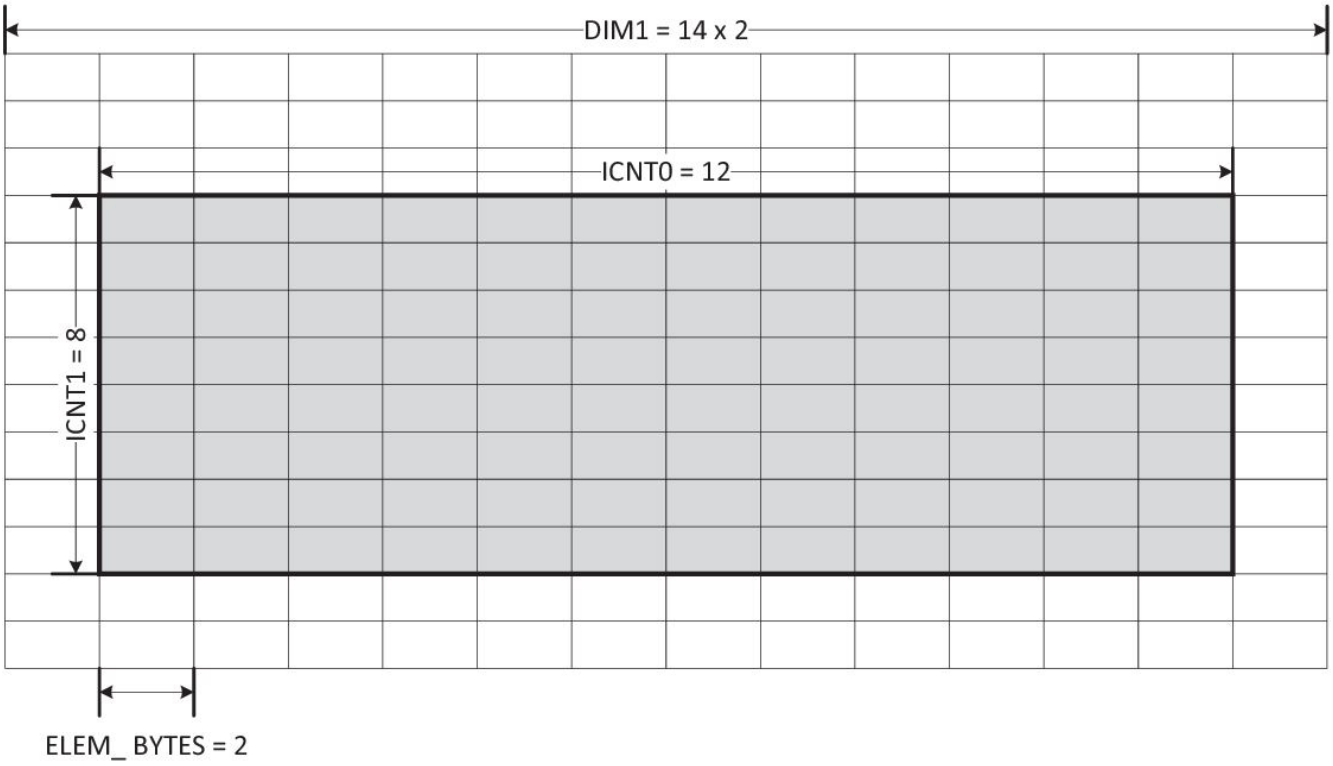


Figure 3-109. 2-D Array

Figure 3-110 illustrates how the streaming engine would fetch this stream with a transposition granularity of 4 bytes. It fetches pairs of elements from each row, but otherwise moves down the columns. When it reaches the bottom of a pair of columns, it repeats this pattern with the next pair of columns.

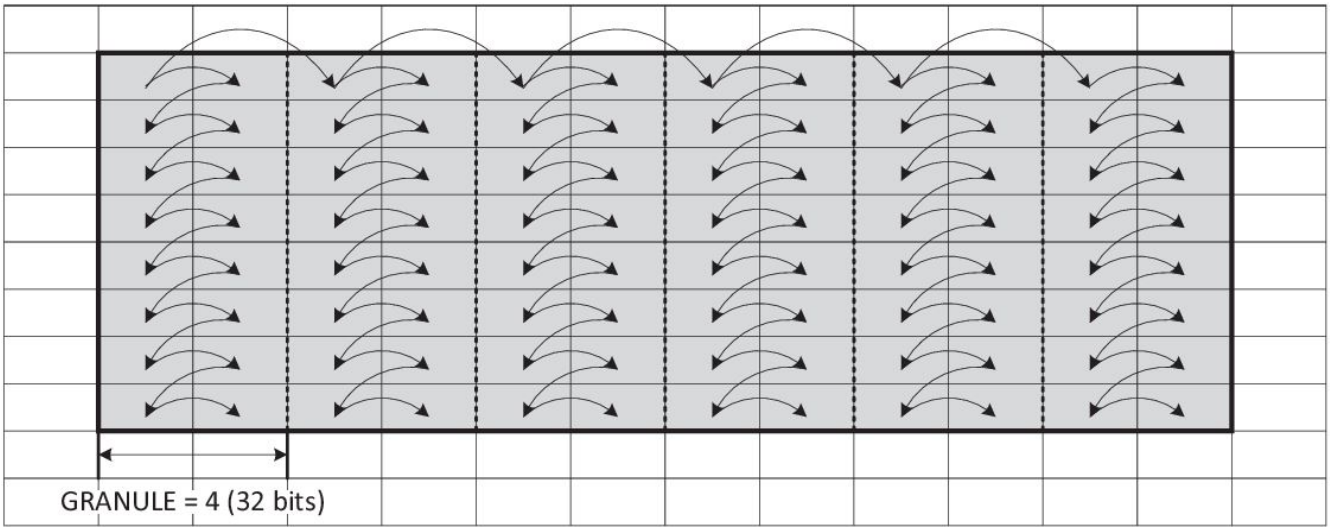


Figure 3-110. Transposed Stream with Granule = 4

Figure 3-111 illustrates the same stream, only transposing on 8 byte (64-bit) boundaries. The overall structure remains the same; however, the streaming engine fetches 4 elements from each row before moving to the next row in the column. Additional examples of Transpose patterns can be found in Section 3.22.4.3.2.

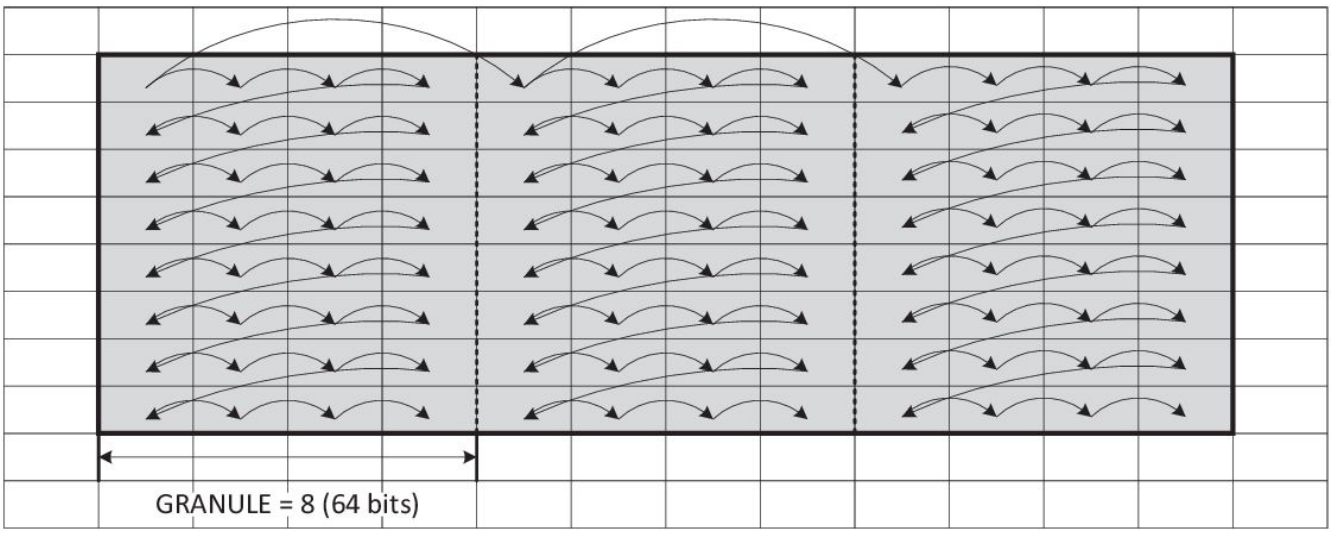


Figure 3-111. Transposed Stream with Granule = 8

3.22.4 Programmer's Model

The previous section defined high-level stream concepts. This section defines the programmer's model for the actual streaming engine, including the specific capabilities of the streaming engine, the instructions that manipulate it, and so forth.

3.22.4.1 Streaming Engine Architecture: Block Overview

Figure 3-112 highlights the major blocks within the streaming engine and how they connect. This structure defines the streaming engine's capabilities. The following sections describe the major blocks, describing their role and capabilities.

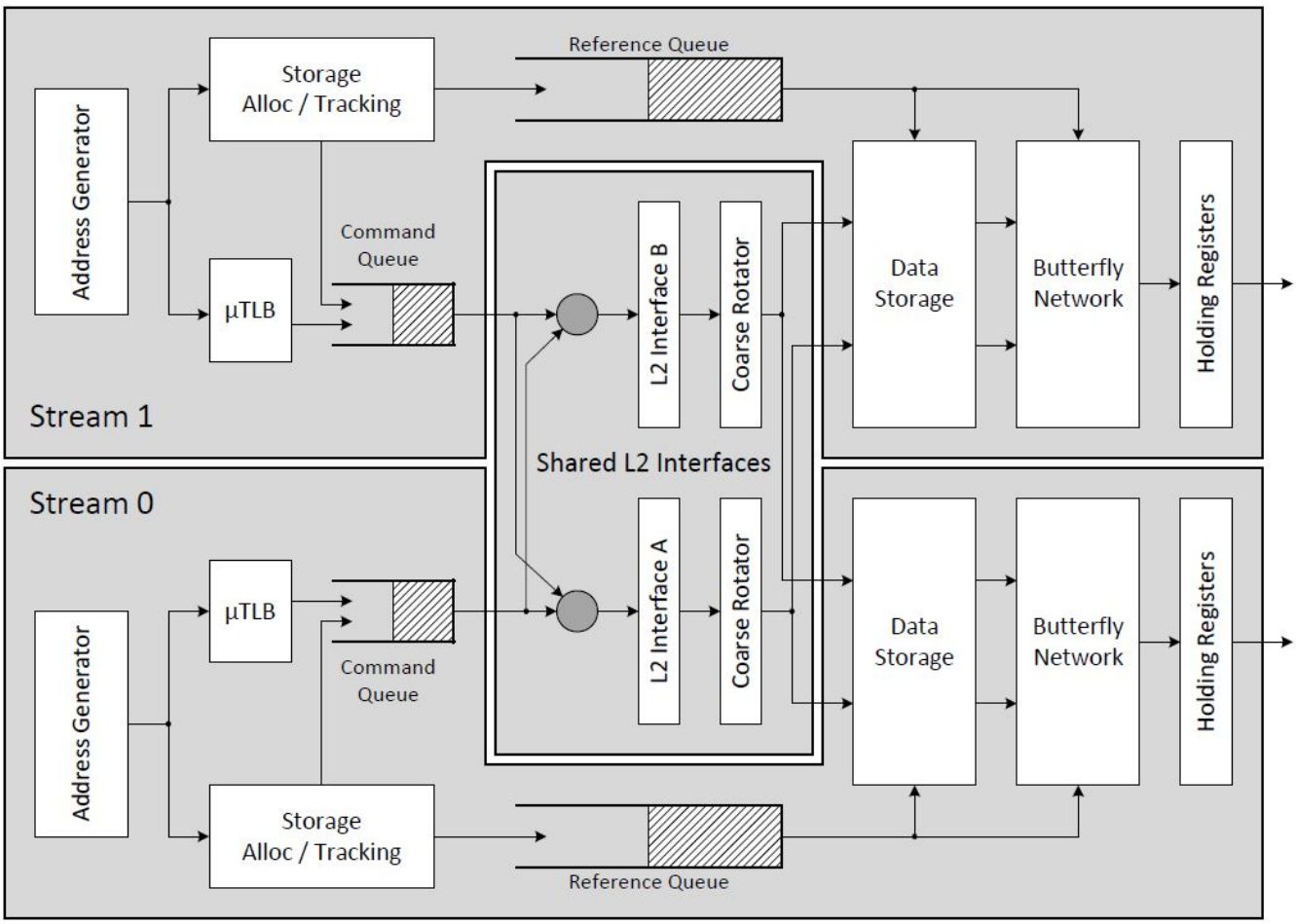


Figure 3-112. Architectural Block Diagram

The streaming engine contains three major sections:

- Stream 0
- Stream 1
- Shared L2 Interfaces

Stream 0 and Stream 1 both contain identical hardware that operates in parallel. Stream 0 and Stream 1 both share two L2 interfaces.

Each stream provides the CPU with up to 512 bits/cycle, every cycle. The streaming engine architecture enables this through its dedicated stream paths and shared dual L2 interfaces.

The following sections describe the blocks above at an architectural level and how they work to achieve the streaming engine's goals. describes the streaming engine's performance targets.

3.22.4.1.1 Address Generator

Each streaming engine includes a dedicated 6-dimensional stream address generator that can generate one new non-aligned request (two 512-bit aligned virtual addresses) per cycle.

The address generator outputs 512-bit aligned addresses that overlap the elements in the sequence defined by the algorithm in Section 3.22.3.2.1. Section 3.22.4.3 below describes the stream parameters and how they translate to address generator inputs.

3.22.4.1.2 µTLB

Each address generator connects to a dedicated µTLB. The µTLB converts a single 49-bit virtual address to a 40-bit physical address each cycle (for a page hit; otherwise table walk returns a physical address). Each µTLB has 8 entries, covering a minimum of 32kB with 4kB pages.

The streaming engine supports all page sizes returned from the µTLB, which can range from 4kB page to 16 GB page sizes. These page sizes are used by the streaming engine to determine a hit or miss if the address falls within the same page size.

The address generator generates 2 addresses per cycle; however, the µTLB only translates 1 address per cycle. To maintain throughput, the streaming engine takes advantage of the fact that most stream references will be within the same page, and the fact that address translation does not modify lower bits of the address. Therefore, if both addresses reside in the same pagesize window, then the streaming engine only asks the µTLB to translate one address and then applies the translation to the upper bits of both. For 4kB page size, the lower 0 - 11 bits are not modified, and for 16 GB page size, the lower 0 - 33 bits are unchanged.

Translated addresses are queued in the command queue, where they are aligned with information from the Storage Allocation and Tracking block.

The streaming engine does not explicitly manage the µTLB. Rather, it relies on the CMMU to invalidate the µTLB as necessary during context switches. Otherwise, the µTLB contents remain stable across streams.

The µTLB also returns the permissions associated with each virtual address. This includes whether the page is readable or writable, the type of memory (normal or device), and whether the page is secure or non-secure. The streaming engine performs the following checks:

- Data streams must access readable, normal memory. Translations that do not meet this criteria result in a page fault.
- Preload streams must access readable, normal, cacheable memory. The streaming engine drops preload requests for pages that do not meet this criteria, as described in Section 3.22.4.4.2.
- Cache maintenance streams must access normal, cacheable memory. Read versus write permission depends on the specific cache maintenance operation, as described in Section 3.22.4.4.1.
- For all streams, the security of the page must be compatible with the CPU's current execution mode (CPRIV). Streaming engine forces all requests to non-secure pages to the non-secure privilege level, regardless of CPRIV.

3.22.4.1.3 Storage Allocation and Tracking

This block manages the stream's internal storage, discovering data reuse and tracking the lifetime of each piece of data. The block accepts 2 virtual addresses per cycle, and binds those addresses to slots in the stream's data storage.

The streaming engine organizes its data store as an array of slots. It maintains the following metadata to track the contents and lifetime of the data in each slot.

Table 3-156. Data Storage Metadata

Address	49-bit virtual address associated with the slot
Valid	Single bit indicating whether the tag address is valid
Ready	Single bit indicating the data has arrived for this address
Active	Single bit indicating whether there are any references outstanding to this data
Last Reference	Value indicating the most recent reference to this slot in the reference queue

The valid, ready, and active bits can take the combination of states listed in Table 3-157.

Table 3-157. Valid, Ready, and Active State Bits

Valid	Ready	Active	Interpretation	Available for Allocation
0	-	-	Address is invalid.	Yes
1	0	0	Invalid: Impossible to have data pending without reference in flight	-
1	0	1	Request sent for slot; data pending	No
1	1	0	No active references in flight; data available	Yes
1	1	1	Reference in flight; data available	No

With this metadata, the storage allocation and tracking logic can identify data reuse opportunities in the stream.

3.22.4.1.3.1 Tag Lookup

The allocation logic performs the following steps for each address:

- Compares the address against the relevant tags in its tag array.
  - If one tag matches (a hit):
    - Cancel the outgoing command associated with this address.
    - Remember the slot number that matches the address.
  - If no tags match (a miss)
    - Allocate a free slot.
    - Set Valid = 1, Ready = 0 on that slot.
    - Update the outgoing command to direct its data to this slot.
  - In either case, this results in a slot number associated with the address.
- Inserts the reference in the reference queue.
- Updates slot specific metadata for the slot.
  - Sets Active = 1, indicating an active reference in the reference queue.
  - Updates the Last Reference for this slot to the position of the reference in the reference queue—that is, the value of the reference queue's insertion pointer at the time of insertion.

This process converts the generated addresses into the slot numbers that represent the data, along with commands to fetch the data if necessary. From this point forward, the streaming engine does not need to track addresses directly.

3.22.4.1.3.2 Allocation Policy

To maximize reuse and minimize stalls, the streaming engine allocates slots in the following order:

1. Pick the slot one after the most recent allocation if available (FIFO order), else:
2. Pick the lowest number available slot, if any, else:
3. If no slot is available, stall and iterate the above two steps until allocation succeeds.

This allocates slots in FIFO order, but avoids stalling if a particular reuse pattern works against FIFO order.

3.22.4.1.3.3 Reference Queue

The reference queue stores the sequence of references generated by the address generator. This information drives the data formatting network so that it can present data to the CPU in the correct order.

Each entry in the reference queue contains the information necessary to read data out of the data store and align it for the CPU. It maintains the information listed in Table 3-158 in each slot.

Table 3-158. Reference Queue Entry

Data Slot Low	Slot number for the "lower half" of the data (ie. that associated with aout0)
Data Slot High	Slot number for the "upper half" of the data (ie. that associated with aout1)
Rotation	The number of bytes to rotate the data by to align the next element to lane 0
Length	Total number of valid bytes in this reference
Expansions	Number times to expand this entry (say, due to type promotion, element duplication)

The allocation and tracking logic inserts references in this queue as the address generator generates new addresses. It removes references from the queue when the data becomes available and there is room in the stream holding registers.

As the streaming engine removes slot references from the queue and formats data, it checks whether the references represent the last reference to the corresponding slots. It compares the reference queue's removal pointer against the slot's recorded Last Reference. If they match, then the tracking logic marks the slot inactive when it's done with the data.

Note

Implementation note: The stream could generate a new reference to the same slot in the same cycle as it's clearing the oldest reference. In this circumstance, the slot must remain marked Active.

3.22.4.1.4 Data Storage

Architecturally, the streaming engine stores an arbitrary number of elements. Deep buffering allows the streaming engine to fetch far ahead in the stream, hiding memory system latency. Too much buffering adds unnecessary area and leakage power, among other deleterious effects. The right amount of buffering may vary across implementations. Therefore, the architecture purposefully leaves the size of this storage unspecified.

The current streaming engine implementation dedicates 32 slots of 64 bytes to each stream. Future implementations may change this amount without changing the semantics of a stream.

3.22.4.1.5 Data Formatting Network

The data formatting network aligns and formats data for the CPU. It performs the following data formatting operations:

- Rotating the next element down to byte lane 0.
- Lane order flip for reverse-direction streams
- Endianness compensation for current processor endian mode
- Sub-element swap within complex elements
- Integer data type promotion, with sign or zero extension
- Element decimation (that is, discarding elements)
- Individual element duplication
- Vector length masking / group duplication

This corresponds to the conceptual sequence described in ‘Section 3.22.3.2.2, with some additional steps at the beginning to compensate for data alignment, order and endianness.

The user directly specifies element size, type promotion, element decimation, real/imaginary swap, element duplication, and group duplication as part of the stream’s parameters. The streaming engine infers alignment and lane order flip from other stream parameters. Endianness comes from the CPU’s current operating mode. Section 3.22.4.3 describes the stream parameters more fully.

3.22.4.1.6 Holding Registers

The streaming engine attempts to fetch and format data ahead of the CPU’s demand for it, so that it can maintain full throughput. These holding registers provide a small amount of elastic buffering so that the process remains fully pipelined.

Like the rest of the streaming engine pipeline, these holding registers are not directly architecturally visible, except for the fact that the streaming engine provides full throughput.

3.22.4.1.7 Shared L2 Interfaces

The two streams share a pair of independent L2 interfaces: L2 Interface A (IFA) and L2 Interface B (IFB). Each L2 interface provides 512 bits/cycle throughput direct to the L2 controller, for an aggregate bandwidth of 1024 bits/cycle for the entire streaming engine.

The streaming engine applies a straightforward round-robin scheme to map stream requests to L2 interfaces. This scheme forces each stream to spread its requests across both interfaces. This maximizes the bandwidth available to both streams, regardless of irregularities in either stream’s access pattern.

Each L2 interface can pull one new command from either stream every cycle. Each stream, however, can produce up to two new commands per cycle. Furthermore, each stream command can request up to two data-phases of data. The L2 interfaces apply a simple algorithm to smooth these imbalances and provide peak bandwidth to both streams.

Both L2 interfaces examine the next two requests from both streams, along with both streams’ available credits. The streaming engine also maintains a stream-to-interface preference bit, that breaks ties between streams at each interface.

Figure 3-113 illustrates the connections between streams and arbiters. In the following diagram, Request N refers to the oldest command from a stream not yet sent to L2, and Request N + 1 refers to the next oldest such command.

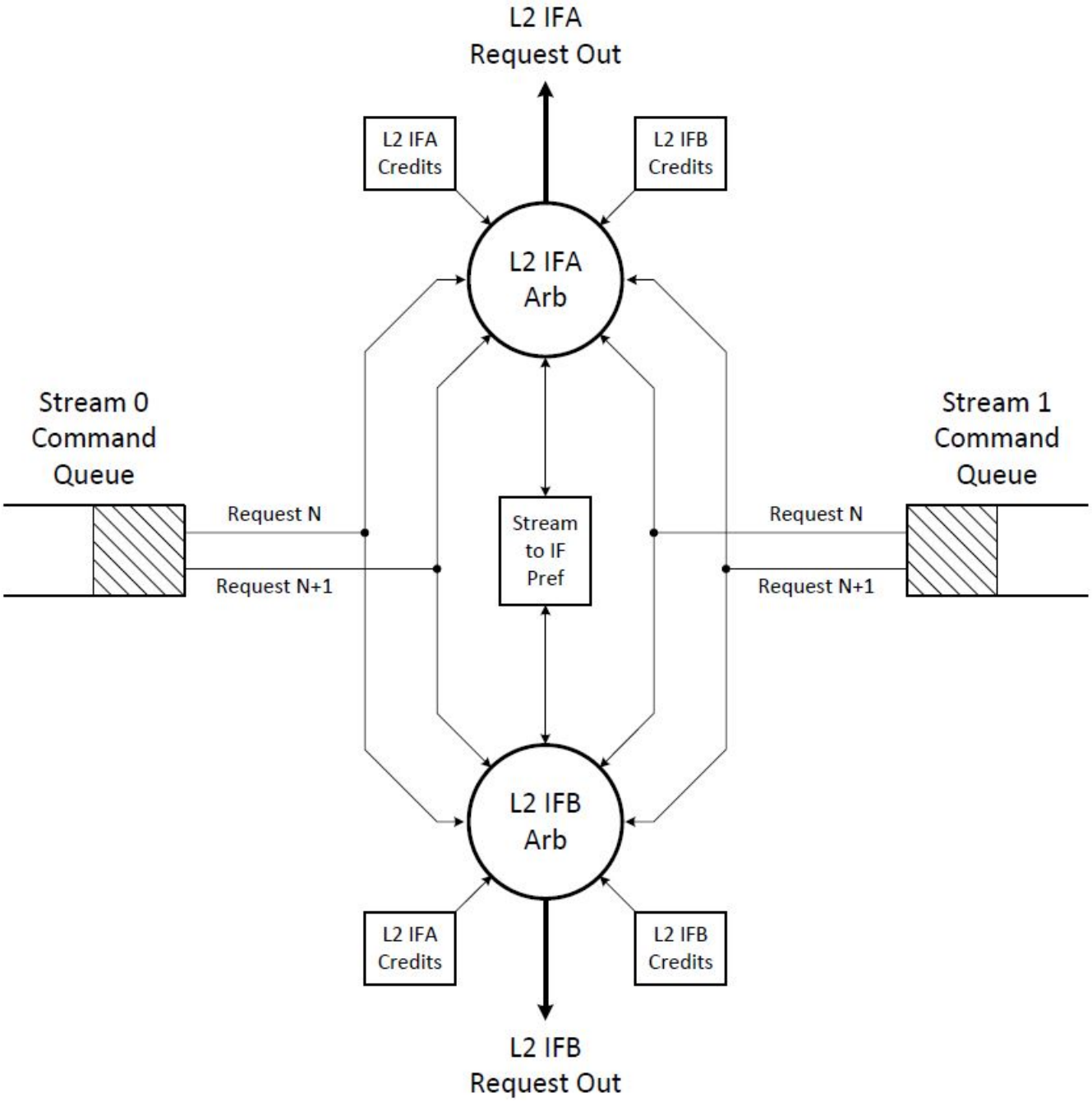
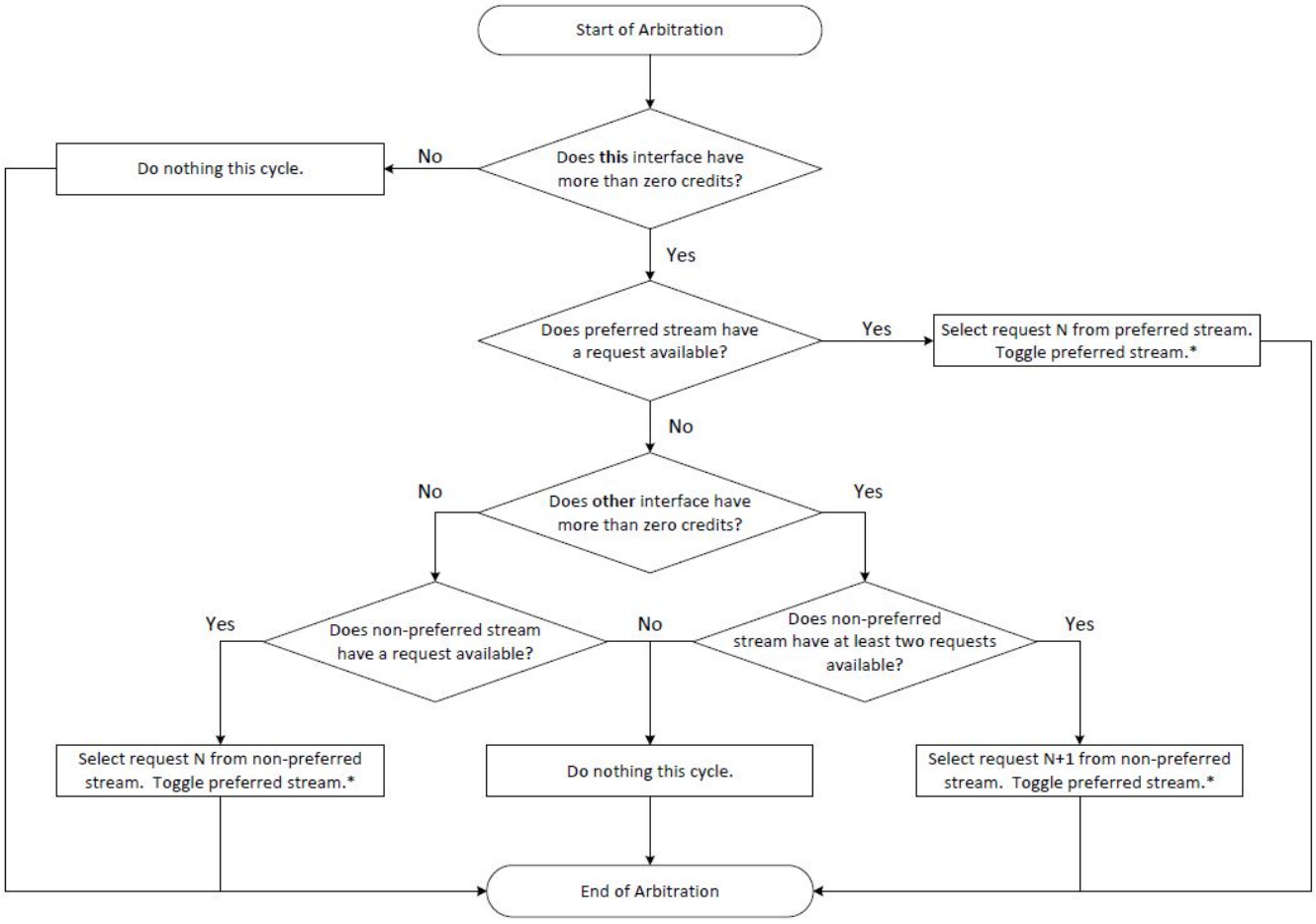


Figure 3-113. Stream to L2 Interface Connections

Each interface tries to send a command on every cycle it has a credit available. It chooses commands from its preferred stream over commands from the non-preferred stream when it can. If the preferred stream does not have a command available, the interface attempts to pull a command from the non-preferred stream.

If either interface manages to send a command on a given cycle, the streaming engine toggles the preferred stream to interface mapping. This forces both streams to spread their commands over both interfaces.

Figure 3-114 illustrates the decision process.



*If *either* arbiter toggles preferred stream on a given cycle, the preferred stream for *both* arbiters toggles for the next cycle.

Figure 3-114. L2 Interface Arbiter Decision Tree

This flow ensures that a single stream with a large number of non-aligned fetches can achieve full bandwidth across both interfaces, and more arbitrary combinations of streams manage to spread their bandwidth evenly across both interfaces.

3.22.4.1.8 Coarse Grain Rotator

The coarse-grain rotator enables the streaming engine to support a transposed matrix addressing mode. In this mode, the streaming engine interchanges the two innermost dimensions of its multidimensional loop.

This accesses an array column-wise rather than row-wise from a C program perspective, and vice-versa from a FORTRAN perspective.

The rotator itself is not architecturally visible, except as enabling this transposed access mode.

3.22.4.1.9 Hardware Watch Point (HWWP) and Command Interest Generation

The streaming engine supports the C7x Debug Controller Hardware Watch Point and command interest criteria matching monitoring. The SE matches on any and all enabled criteria's such as PROC, VMID, DCTXT, and ADDR criteria, along with their respective mask values. The cinterest and triggered logic generation are also generated when matching criteria and enabled by the HWWP. The cinterest is sent to µTLB and L2 when matching occurs, and the triggered output is sent back to HWWP.

3.22.4.1.10 CPU Stream Open Command Interest Generation

The CPU can open each stream independently with a command interest flag. When a command interest stream is opened, the SE marks all stream addresses sent to the µTLB and L2 with the cinterest flag.

3.22.4.1.11 Safety and Functional Testing for EDC logic

The SE maintains parity bits for all of its storage data. There is a 1-bit parity for each 32-bits of data in the storage per stream. The SE aligns to the Keystone 3 safety architecture provided testing mechanisms for the EDC detection and correction logic. In addition, the SE aligns to the Keystone 3 safety architecture for reporting of detected parity EDC faults from both the EDC controller initiated error injection, or functional access errors.

3.22.4.2 Stream States

Each stream may be in one of three states:

- Inactive
- Active
- Frozen

After reset, both streams begin in the inactive state. Opening a stream moves that stream to the active state. Closing the stream returns it to the inactive state. In the absence of interrupts and exceptions, streams ordinarily do not make other state transitions.

Note

The remainder of this section uses the word interrupt to refer to both interrupts and exceptions. The streaming engine makes the same state transitions for both interrupts and exceptions. Likewise, all references to ITSR should be understood to refer to NTSR when working with exceptions.

Note

As of spec version 0.80, the ITSR and NTSR registers inside CPU have been deprecated. However, the TSR register inside CPU is still present and thus any references to ITSR/NTSR should be understood to refer to TSR.

To account for interrupts, the streaming engine adds a third state: frozen. The frozen state represents an interrupted active stream.

Figure 3-115 summarizes the valid state transitions between inactive, active and frozen. The remainder of the section defines the meaning of these transitions.

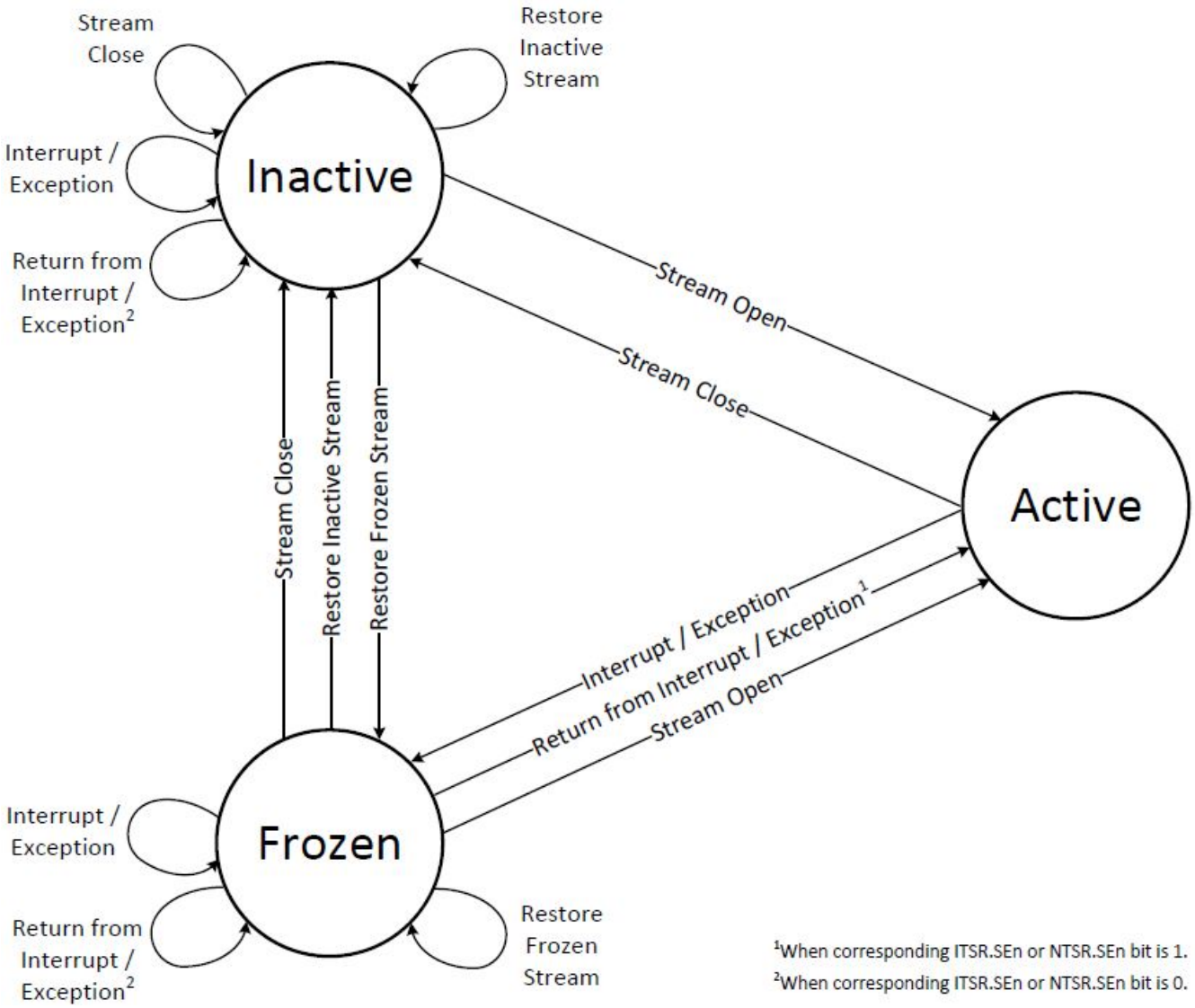


Figure 3-115. Valid Stream State Transitions

Architecturally, a total of four bits, two bits per stream, defines the state of both streams. One bit per stream resides within the streaming engine, and the other bit resides within the CPU. The streaming engine internally tracks whether each stream holds a parameter set associated with an active stream. This bit distinguishes an inactive stream from a not-inactive stream. The DSP CPU separately tracks whether it considers each stream active with a dedicated bit per stream in the Task State Register (TSR)—TSR.SE0 for stream 0, and TSR.SE1 for stream 1. These bits distinguish between active and not-active streams.

Table 3-159 gives the resulting state mapping.

Table 3-159. Stream State Mapping

TSR Stream Active Bit	Internal Stream Active Bit(1)	Stream State
0	0	Inactive
0	1	Frozen
1	0	Invalid(2)
1	1	Active

(1) The internal stream active bit is notionally included in the state saved and restored by SESAVE and SERSTR. See Section 3.22.4.5.3.

(2) Any attempt to read data from the streaming engine while in this state triggers an exception. This might occur if an interrupt handler incorrectly restores stream state before returning from interrupt. See Section 3.22.4.2.3.

3.22.4.2.1 Opening and Closing Streams

Opening a stream always moves it to the active state. Closing a stream always moves it to the inactive state.

If a program opens a new stream over a frozen stream, the new stream replaces the old stream. The streaming engine discards the contents of the previous stream. The streaming engine now supports opening a new stream on an already active stream. The streaming engine discards the contents of the previous stream, flushes it's pipeline, and starts fetching data for the new opened stream. Data to CPU is asserted when the data have returned.

If a program closes an already closed stream, nothing happens. If it closes an open or frozen stream, the streaming engine discards all state related to the stream. It clears its internal stream-active bit, and clears its counter, tag and address registers. Closing a stream serves two purposes:

- Closing an active stream allows a program to specifically state it no longer needs the stream and the resources associated with it.
- Closing a frozen stream allows context switching code to clear the state of the frozen stream, so that other tasks do not see it.

For the specific instructions that open and close streams, see Section 3.22.4.5.1.

Note

The streaming engine has a handshake mechanism with the L1D data cache for any current CPU stores issued to L1D. This handshake acts as a fencing to ensure that all stores have landed in the system before the streaming starts sending commands into the system. See Section 3.22.4.2.2.

3.22.4.2.2 Stream Open CPU Store Fencing

The streaming engine by default provides a "fencing" mechanism between outstanding CPU stores to L1D data cache, which prevents the streaming engine from sending requests into the system until the outstanding stores from CPU have landed. This is accomplished through internal signaling between the CPU, L1D, and streaming engine. This fencing check is done on any new opened streams issued by CPU. Upon receiving a stream open, the streaming engine starts its address generation and μTLB lookups, and prepares the request to go out. When the requests reaches the arbitration unit, the commands are stalled until the fence is complete. This is accomplished by monitoring the IDLE signals from L1D indicating the pertinent stores have completed. This prevents streaming engine from receiving old or stale data. See Section 3.22.3.1.1. The fencing is also done when the SE comes back from context switching.

3.22.4.2.3 Context Switching: Saving and Restoring Streams

Multitasking operating environments must be able to context switch the entire state of the DSP CPU to switch between tasks. This includes the state of both streams.

The streaming engine allows saving and restoring a given stream's state when it is either inactive or frozen. This, when coupled with the CPU's normal behavior when entering and leaving interrupts, provides a convenient mechanism for context switching streams in a preemptive multitasking environment.

Note

If a program attempts a save or restore action on a stream in the active state, the CPU generates an exception. To context switch an active stream, the CPU must first take an interrupt or exception. This transitions the active stream to the frozen state.

At a high level, the CPU and streaming engine respond to interrupts as follows:

- The CPU copies the current TSR to ITSR.
- The CPU marks active streams inactive, by clearing TSR.SE0 and TSR.SE1.
- The streaming engine responds by freezing active streams. To freeze a stream, the streaming engine takes the following steps:
  - Snapshots stream parameters, along with current address and loop counts. The streaming engine captures sufficient state to allow restoring the stream later.
  - Flushes the internal stream reference queue.

  - Discards any commands not yet sent to µTLB. Commands which have reached ARB block go out to L2 but are marked and discarded on return.
  - Marks all slots in internal storage as invalid.
  - Arranges to discard return data for any in-flight commands.
- If the interrupt handler decides to context switch the stream, it can save the stream state for both streams on arrival to the interrupt handler.

Thus, interrupts cause the streaming engine to discard the data it currently holds, but preserve sufficient state to restore the stream to the same point in the future. The interrupt handler can then safely manipulate the state of the streaming engine to perform a context switch if needed.

When returning from an interrupt, the interrupt handler, CPU and streaming engine perform the following actions.

- The interrupt handler restores the appropriate context to both streams, if needed.
- The interrupt handler restores the appropriate context to ITSR.
  - If a stream was active in the interrupted context, the corresponding ITSR.SEn bit is 1.
  - Restoring ITSR does not immediately affect the streaming engine, regardless of the contents of ITSR.SEn.
- The interrupt handler executes a Return From Interrupt instruction.
  - If either TSR.SE0 = 1 or TSR.SE1 = 1, the CPU generates and takes an exception immediately, rather than returning from the interrupt.
- The CPU copies ISTR to TSR.
  - At this point, TSR.SEn will become 1 if ITSR.SEn was 1. This action triggers the streaming engine to reactivate frozen streams.
  - If TSR.SEn becomes 1, but the corresponding stream was in the inactive state (rather than frozen), the streaming engine signals an exception to the CPU.

Thus, on return from interrupt, the streaming engine reactivates frozen streams. The streaming engine fetches the required data to restore the stream to its previous state.

For the specific instructions that save and restore streams, see Section 3.22.4.5.3.

3.22.4.2.4 Nested Interrupts

To support nested interrupts on the DSP CPU, the interrupt handler must save ITSR and the interrupt return pointer before re-enabling interrupts, and must restore ITSR and the interrupt return pointer before returning from interrupts. This action preserves the state of ITSR.SEn, so that TSR.SEn gets restored correctly when returning to the interrupted code.

To support nested interrupts with the streaming engine, the interrupt service routine must save and restore the streaming engine state only if the interrupt handler itself uses the streaming engine. Otherwise, the streaming engine does not require special handling when nesting interrupts.

3.22.4.3 Stream Definition Template

The stream definition template defines a stream of data for the streaming engine to fetch from memory. The template contains the loop counts, array dimensions and other flags which tell the streaming engine what to fetch and how to format it.

Note

The streaming engine supports a small number of auxiliary functions with a special dataless stream. These do not use the stream definition template. See Section 3.22.4.4.

The basic stream definition template provides all the parameters necessary to describe a 6 dimensional stream. The template itself fits into 512 bits.

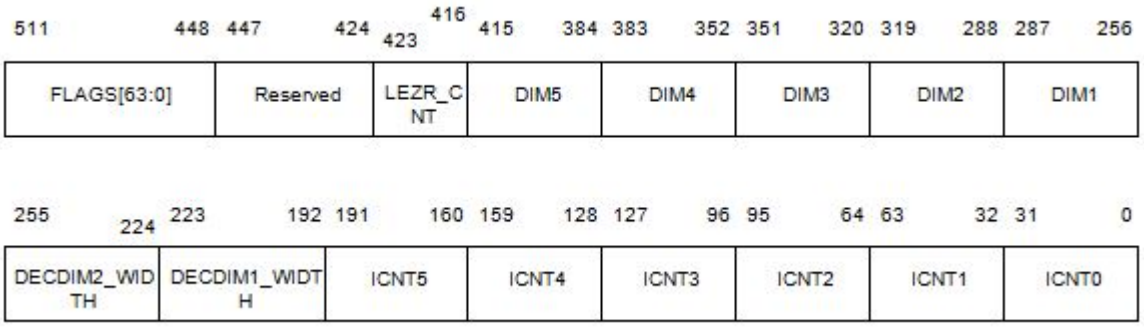


Figure 3-116. Basic Stream Parameter Template, 6-D

The streaming engine defines a six-level loop nest for addressing elements within the stream, using the algorithms described in Section 3.22.3.2. Most of the fields in the stream template map directly to the parameters in those algorithms.

Table 3-160. Stream Fields

Field Name	Description	Size
ICNT0(1)	Total loop iteration count for level 0 (innermost)	32 bits
ICNT1	Total loop iteration count for level 1	32 bits
ICNT2	Total loop iteration count for level 2	32 bits
ICNT3	Total loop iteration count for level 3	32 bits
ICNT4	Total loop iteration count for level 4	32 bits
ICNT5	Total loop iteration count for level 5 (outermost)	32 bits
DECDIM1_WIDTH(2)	Total Actual Width count when in DECDIM1 mode. See Section 3.22.4.3.2.	32 bits
DECDIM2_WIDTH(3)	Total Actual Width count when in DECDIM2 mode. See Section 3.22.4.3.2.	32 bits
DIM1	Signed dimension for loop level 1	32 bits
DIM2	Signed dimension for loop level 2	32 bits
DIM3	Signed dimension for loop level 3	32 bits
DIM4	Signed dimension for loop level 4	32 bits
DIM5	Signed dimension for loop level 5	32 bits
LEZR_CNT(4)	Total iteration count for number of Zero Padded Vector Reads to CPU following the end of selected loop dimension count as set in LEZR Flags Field.	8 bits
FLAGS	Stream modifier flags. See Figure 3-117.	64 bits

(1) ICNT0 defines the loop iteration for the innermost dimension, implied DIM0, in terms of number of ELEM_BYTES. In Transpose mode, it defines the horizontal dimension because ICNT0 and ICNT1 are interchanged in transpose.

(2) DECDIM1_WIDTH defines the total actual width when in DECDIM1or DECDIM1SDmode in terms of number of ELEM_BYTES.

(3) DECDIM2_WIDTH defines the total actual width when in DECDIM2 or DECDIM2SD mode in terms of number of ELEM_BYTES.

(4) This iteration count instructs SE to send back the specified number of zero padded vector to CPU, after the selected loop dimension count ends, as set by the LEZR Flags field. These extra zero padded vector reads to CPU are done as an extension of the current loop dimensions, and each vector reads return 64-bytes of zero data.

In this template, each loop count is limited to 32 bits, and most dimensions are also limited to 32 bits. All loop counts are unsigned integers. All dimensions are signed integers. The DECDIM actual width count is an unsigned integer and used when in DECDIM mode.

The 32-bit fields in the stream definition template limit the total range of loop counts and dimensions the template can express. An iteration count of 0 at any level (ICNT0, ICNT1, ICNT2, ICNT3, ICNT4, or ICNT5, or DECDIM_WIDTH) specifies an empty stream. A stream with more than zero elements must specify a minimum iteration count of 1 in all loop levels. The streaming engine detects if any iteration count of zero is programmed for the selected dimensions (see Section 3.22.4.3.6), and would not start the stream, and returns zero data to the CPU. The template above fully specifies the type of elements, length, and dimensions of the stream. It does not, however specify starting address for the stream. The stream instructions provide that as a separate

argument. This allows a program to open multiple streams at different addresses using the same template. Table 3-161 details the contents of the 64-bit FLAGS field. These fields define many important aspects of the stream.

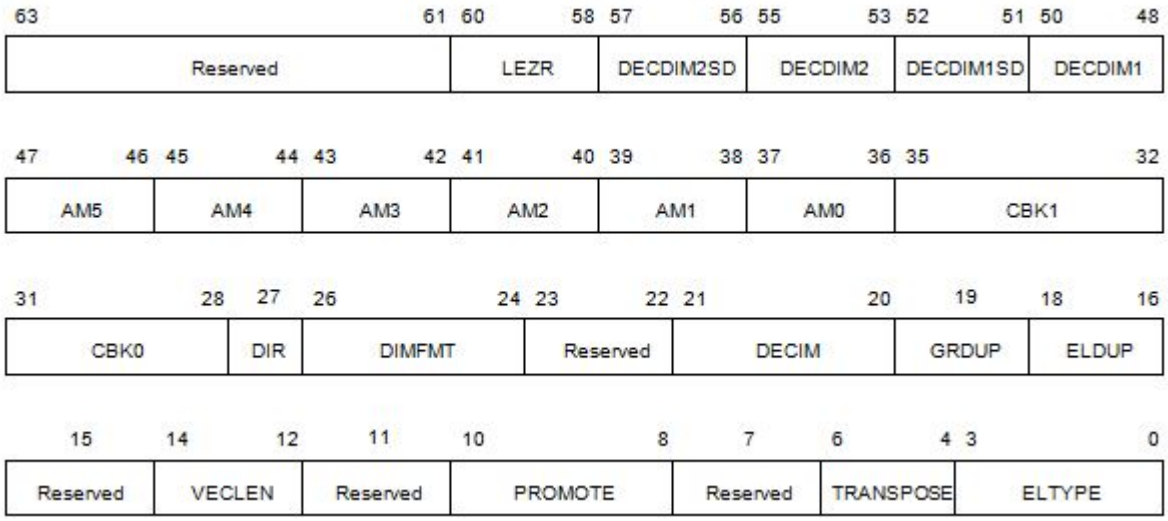


Figure 3-117. Stream Parameter Template FLAGS Field

Table 3-161. FLAGS Field Descriptions

Bit	Field	Description	Reference
63:61	Reserved	Reserved for future expansion of VECLEN. Set to 0 to ensure future compatibility	
60:58	LEZR	Selected Loop Dimension Count End, and Start Sending Zero Padded Vectors to CPU. The number of zero padded vectors sent to CPU is defined by the LEZR_CNT in template. 000b = Normal Operating Mode 001b = Start sending Zero Padded Vectors to CPU at end of ICNT0 010b = Start sending Zero Padded Vectors to CPU at end of ICNT1 011b = Start sending Zero Padded Vectors to CPU at end of ICNT2 100b = Start sending Zero Padded Vectors to CPU at end of ICNT3 101b = Start sending Zero Padded Vectors to CPU at end of ICNT4 110b = Start sending Zero Padded Vectors to CPU at end of ICNT5 - end of stream by default sends back zero vectors to CPU indefinitely. 111b = Reserved	Section 3.22.4.3.1
57:56	DECDIM2SD	Secondary Data Strip Mining Decrement Dimension for DECDIM2_WIDTH count(1) 00b = Normal Operating Mode 01b = Strip Mining on DIM1 (DECDIM2_WIDTH decrements when entering DIM1) (2) 10b = Strip Mining on DIM2 (DECDIM2_WIDTH decrements when entering DIM2) 11b = Strip Mining on DIM3 (DECDIM2_WIDTH decrements when entering DIM3)	Section 3.22.4.3.2
55:53	DECDIM2	Data Strip Mining Decrement Dimension for DECDIM2_WIDTH count(3) 000b = Normal Operating Mode 001b = Strip Mining on DIM1 (DECDIM2_WIDTH decrements when entering DIM1)(4) 010b = Strip Mining on DIM2 (DECDIM2_WIDTH decrements when entering DIM2) 011b = Strip Mining on DIM3 (DECDIM2_WIDTH decrements when entering DIM3) 100b = Strip Mining on DIM4 (DECDIM2_WIDTH decrements when entering DIM4) 101b = Strip Mining on DIM5 (DECDIM2_WIDTH decrements when entering DIM5) 110b = Reserved 111b = Reserved	Section 3.22.4.3.2

Table 3-161. FLAGS Field Descriptions (continued)

Bit	Field	Description	Reference
52:51	DECDIM1SD	Secondary Data Strip Mining Decrement Dimension for DECDIM1_WIDTH count[5] 00b = Normal Operating Mode 01b = Strip Mining on DIM1 (DECDIM1_WIDTH decrements when entering DIM1)[6] 10b = Strip Mining on DIM2 (DECDIM1_WIDTH decrements when entering DIM2) 11b = Strip Mining on DIM3 (DECDIM1_WIDTH decrements when entering DIM3)	Section 3.22.4.3.2
50:48	DECDIM1	Data Strip Mining Decrement Dimension for DECDIM1_WIDTH count[7] 000b = Normal Operating Mode 001b = Strip Mining on DIM1 (DECDIM1_WIDTH decrements when entering DIM1)[8] 010b = Strip Mining on DIM2 (DECDIM1_WIDTH decrements when entering DIM2) 011b = Strip Mining on DIM3 (DECDIM1_WIDTH decrements when entering DIM3) 100b = Strip Mining on DIM4 (DECDIM1_WIDTH decrements when entering DIM4) 101b = Strip Mining on DIM5 (DECDIM1_WIDTH decrements when entering DIM5) 110b = Reserved 111b = Reserved	Section 3.22.4.3.2
47 - 46	AM5	Addressing mode selection for dimensions 0 through 5. 00b = Linear addressing for this dimension (circular addressing disabled) 01b = Circular addressing using CBK0 10b = Circular addressing using CBK0 + CBK1 + 1 11b = Reserved	Section 3.22.4.3.3
45 - 44	AM4		
43 - 42	AM3		
41 - 40	AM2		
39 - 38	AM1		
37 - 36	AM0		
35 - 32	CBK1	Circular addressing buffer size. Refer to Table 1-18 “Circular Block Size Decoding” on page 1-65.	Section 3.22.4.3.4
31 - 28	CBK0		
27	DIR	Stream direction selection 0b = Forward: increasing address order 1b = Reverse: decreasing address order	Section 3.22.4.3.5
26 - 24	DIMFMT	Stream Dimensions 000b = 1-dimensional stream 001b = 2-dimensional stream 010b = 3-dimensional stream 011b = 4-dimensional stream 100b = 5-dimensional stream 101b = 6-dimensional stream 110b = Reserved 111b = Reserved	Section 3.22.4.3.6
23 - 22	Reserved	Reserved for future expansion of VECLEN. Set to 0 to ensure future compatibility	
21:20	DECIM	Element decimation[9] 00b = No decimation 01b = Decimate 2:1 10b = Decimate 4:1 11b = Reserved	Section 3.22.4.3.8
19	GRDUP	Group duplication 0b = Group duplication disabled 1b = Group duplication enabled	Section 3.22.4.3.11
18 - 16	ELDUP	Element duplication 000b = No duplication 001b = Duplicate 2× 010b = Duplicate 4× 011b = Duplicate 8× 100b = Duplicate 16× 101b = Duplicate 32× 110b = Duplicate 64× 111b = Reserved	Section 3.22.4.3.9
15	Reserved	Reserved for future expansion of VECLEN. Set to 0 to ensure future compatibility	

Table 3-161. FLAGS Field Descriptions (continued)

Bit	Field	Description	Reference
14 - 12	VECLEN	Stream Vector length 000b = 1 element 001b = 2 elements 010b = 4 elements 011b = 8 elements 100b = 16 elements 101b = 32 elements 110b = 64 elements 111b = Reserved	Section 3.22.4.3.10
11	Reserved	Reserved for future expansion of PROMOTE. Set to 0 to ensure future compatibility.	
10 - 8	PROMOTE	Type promotion mode. Valid encodings depend on element type. Refer to Table 3-175. 000b = No promotion 001b = Integer promote 2× with zero-extension 010b = Integer promote 4× with zero-extension 011b = integer promote 8× with zero-extension 100b = Reserved 101b = Integer promote 2× with sign-extension 110b = Integer promote 4× with sign-extension 111b = integer promote 8× with sign-extension	Section 3.22.4.3.12
7	Reserved	Reserved for future expansion of TRANSPOSE. Set to 0 to ensure future compatibility.	
6 - 4	TRANSPOSE	2-D transposition mode 000b = Transposition disabled (Linear Stream Mode) 001b = Transpose on 8-bit boundaries(10) 010b = Transpose on 16-bit boundaries(11) 011b = Transpose on 32-bit boundaries 100b = Transpose on 64-bit boundaries 101b = Transpose on 128-bit boundaries 110b = Transpose on 256-bit boundaries 111b = Reserved	Section 3.22.4.3.13
3 - 0	ELTYPE	Type of element in stream. Refer to Table 1-31 “ELTYPE Encodings” on page 1-76.	Section 3.22.4.3.14

(1) The selected DECDIM2D dimension (DIM1 - DIM3) must be an unsigned value (that is, the DIMx bit 31 is zero). All other dimensions can be signed or unsigned as defined.

(2) DECDIM2D = DIM1 is not valid when in Transpose Mode Streams - setting this option in DECDIM and Transpose results in error returned to CPU.

(3) The selected DECDIM2 dimension (DIM1 - DIM5) must be an unsigned value (that is, the DIMx bit 31 is zero). All other dimensions can be signed or unsigned as defined.

(4) DECDIM2 = DIM1 is not valid when in Transpose Mode Streams - setting this option in DECDIM and Transpose results in error returned to CPU.

(5) The selected DECDIM1D dimension (DIM1 - DIM3) must be an unsigned value (that is, the DIMx bit 31 is zero). All other dimensions can be signed or unsigned as defined.

(6) DECDIM1D = DIM1 is not valid when in Transpose Mode Streams - setting this option in DECDIM and Transpose results in error returned to CPU.

(7) The selected DECDIM1 dimension (DIM1 - DIM5) must be an unsigned value (that is, the DIMx bit 31 is zero). All other dimensions can be signed or unsigned as defined.

(8) DECDIM1 = DIM1 is not valid when in Transpose Mode Streams - setting this option in DECDIM and Transpose results in error returned to CPU.

(9) DECIM 2:1 requires at least promote 2x, and DECIM 4:1 requires at least promote 4x.

(10) Transpose on 8-bit boundaries has the following restrictions due to hardware storage structure:

- The first element in each dimension must start on a 4 byte (32-bit) boundaries (that is, the same restrictions as transposing on 32-bit boundaries - TRANSPOSE = 011b).
- Decimate 4x must be enabled.
- Promotion by 4x must be enabled.
- ELTYPE size must be 1-byte, ELTYPE = 0000b

(11) Transpose on 16-bit boundaries has the following restrictions due to hardware storage structure:

- The first element in each dimension must start on a 4 byte (32-bit) boundaries (that is, the same restrictions as transposing on 32-bit boundaries - TRANSPOSE = 011b).
- Decimate 2x must be enabled.
- Promotion by 2x must be enabled.
- ELTYPE size must be 2-bytes, ELTYPE = 0001b or ELTYPE = 1000b.

The following sections expand on each of these fields.

3.22.4.3.1 LEZR: End of Loop Dimension Zero Vectors

The LEZR feature allows programmers to write code such that when the selected LEZR loop count ends, the SE starts to send zero padded vectors back to the CPU. The number of CPU-fetched zero padded vectors to send are defined in the template as an 8-bit unsigned value called “LEZR_CNT”. The SE sends back a full 64-byte vector on each CPU fetch, thus any data formatting scaling and replay done during the normal dimensional loop is not adhered to.

3.22.4.3.1.1 LEZR Restrictions

LEZR is supported in linear reverse and forward streams. For transpose mode, all GRANULE are supported in both forward and reverse streams. For 8-bit and 16-bit GRANULE, the LEZR setting for ICNT1 is not supported (all other ICNTx are supported). LEZR is not applicable and thus not supported in cache maintenance and cache pre-warm streams.

3.22.4.3.2 DECDIM: Data Strip Mining Decrement Dimension for DECDIM_WIDTH Count

The DECDIM feature encompasses two sets of FLAGS (DECDIMx and DECDIMxSD), and along with their respective DECDIM_WIDTH, lets the programmer define 4 mask values on selected dimensions to zero out parts of the stream data. Any combination of settings for DECDIM is supported, with exception of the restrictions described below. The description that follows pertains to all DECDIM FLAGS.

The DECDIM FLAGS field supports a data strip mining feature that allows the programmer to provide a “Total Actual Width” size of an image using the DECDIM_WIDTH count to provide this max actual width. In this mode, the DECDIM_WIDTH is decremented by the selected DECDIM dimension (DIM1 - DIM5) when the loop enters that dimension. For example, if DECDIM = 010b, then every time the loop enters dimension 2, the current value of DECDIM_WIDTH is decremented by the DIM2 value. When the DECDIM_WIDTH becomes less than the ICNT0 (the “Tile Width” in DECDIM mode), the SE pads the unused elements with zero data. This is referred to as “data strip mining”.

The DECDIM_WIDTH count value only reloads to the programmed value again when the selected DECDIM dimension loop count expires. In the above example, the width would reload when dimension 2 (that is, ICNT2) expires, and dimension 3 is entered. In other words, the width reloads when any dimension higher than the selected DECDIM dimension is entered. Thus, it is possible to program the width smaller than the loop iterations count for the selected dimension, which could cause the width count to underflow. This is explained further below.

There are also DECDIMxSD FLAGS. These FLAGS allow a “secondary decrement” count mask on top of its respective DECDIMx FLAG for the respective DECDIM_WIDTH. In other words, the respective DECDIM_WIDTH is decremented or reloaded using the settings for both their respective DECDIMx and DECDIMxSD when the selected dimension is entered or ends. Examples for these FLAGS are shown later in this section.

3.22.4.3.2.1 DECDIM Restrictions

In linear mode, the DECDIM selection can be set for any DIM1 - DIM5. However, in transpose mode, DIM1 selection is not supported and only DIM2 - DIM5 is supported. In addition, the selected DECDIM dimension must be an unsigned value (that is, the DIMx bit 31 is zero). All other dimensions can be unsigned or signed as defined. For example, if DECDIM = 010b, then DIM2 must be an unsigned value. DIM1, DIM3, DIM4, and DIM5 can be unsigned or signed. Further restrictions apply to the selected DECDIM dimension and the DECDIM_WIDTH when decimation 2x or 4x is enabled. See Section 3.22.4.3.8 and Section 3.22.4.7.1.

Note

In hardware, the innermost loop count is always scaled in terms of total number of bytes. In other words, the innermost I0 count is equal (ICNT0 × ELEM_BYTES), and in the case of DECDIM mode, the DECDIM_WIDTH count is (DECDIM_WIDTH × ELEM_BYTES). During CPU fetches, the innermost loop count expires when all elements have been consumed before accounting for any data formatting (element duplication, promotion, decimation). Thus, with data formatting enabled, multiple CPU fetches could occur before the innermost loop count expires.

The value of DECDIM_WIDTH count is saturated at zero in the condition the loop iteration count parameters for the selected DECDIM (that is, ICNT1 - ICNT5) are incorrectly programmed causing DECDIM_WIDTH count to underflow when decremented with the selected DIMx value. When saturation occurs, the streaming engine holds DECDIM_WIDTH count at zero and subsequent data phases to the CPU are also zero.

Table 3-162 shows how this feature is implemented. The figure examples show what normal stream patterns would look like, and what data strip mining patterns would look like. The DECDIM_WIDTH value in the inserted tables represent the count of the entire frame, which gets decremented by the selected DIMx value as each DECDIM dimension is entered. The examples show data moving left to right, thus the LSByte is leftmost. Also, the first set of example figures shown below do not enable any ‘data formatting’. If data formatting is enabled, additional CPU fetches would be required to consume each of the “Tile” width, that is, innermost ICNT0. All DECDIM with data formatting examples are shown in figures indicating ELDUP.

Table 3-162. DECDIM Data Strip Mining Operation

DECDIM(1) (2)	Meaning
000b	Normal operating mode
001b(3)	DECDIM_WIDTH count is decremented by DIM1 value when entering DIM1. Reloaded when DIM1, or higher DIMs expire.
010b	DECDIM_WIDTH count is decremented by DIM2 value when entering DIM2. Reloaded when DIM2, or higher DIMs expire.
011b	DECDIM_WIDTH count is decremented by DIM3 value when entering DIM3. Reloaded when DIM3, or higher DIMs expire.
100b	DECDIM_WIDTH count is decremented by DIM4 value when entering DIM4. Reloaded when DIM4, or higher DIMs expire.
101b	DECDIM_WIDTH count is decremented by DIM5 value when entering DIM5. Reloaded when DIM5 expires (end of stream).
110b	Reserved(4)
111b	Reserved

(1) The selected DECDIM dimension (DIM1 - DIM5) must be an unsigned value (that is, the DIMx bit 31 is zero). All other dimensions can be signed or unsigned as defined.

(2) DECDIMx uses all 3-bits for encoding and DECDIMxSD uses the 2 LSBit for encoding (see FLAGS).

(3) DECDIM = DIM1 is not valid when in Transpose Mode Streams - setting this option in DECDIM and Transpose results in error returned to CPU.

(4) The reserved encoding triggers an error.

In all DECDIM cases, the SE does not send any µTLB or L2 fetches for regions which have been fully masked and zeroed out (DECDIM_WIDTH saturated), and thus programmers need not allocate these regions in memory even though the stream addressing patterns cover these regions. The SE does this by keeping track of the overall DECDIM_WIDTH mask count and applying the DECDIMx and DECDIMxSD settings, and when this count saturates to zero or ends on a 64-byte line boundary, all subsequent L2 fetches are killed. By doing this, the SE suppresses any errors or faults associated with subsequent un-allocated regions since no µTLB or L2 fetches were sent. This feature is part of the DECDIM error suppression.

Figure 3-118. Linear Stream Normal Mode: Example 1

Figure 3-119. Linear Stream Data Strip Mining: Example 2, DECDIM on DIM2

Figure 3-120. Linear Stream Data Strip Mining: Example 3, DECDIM on DIM2 with DECDIM_WIDTH Saturation

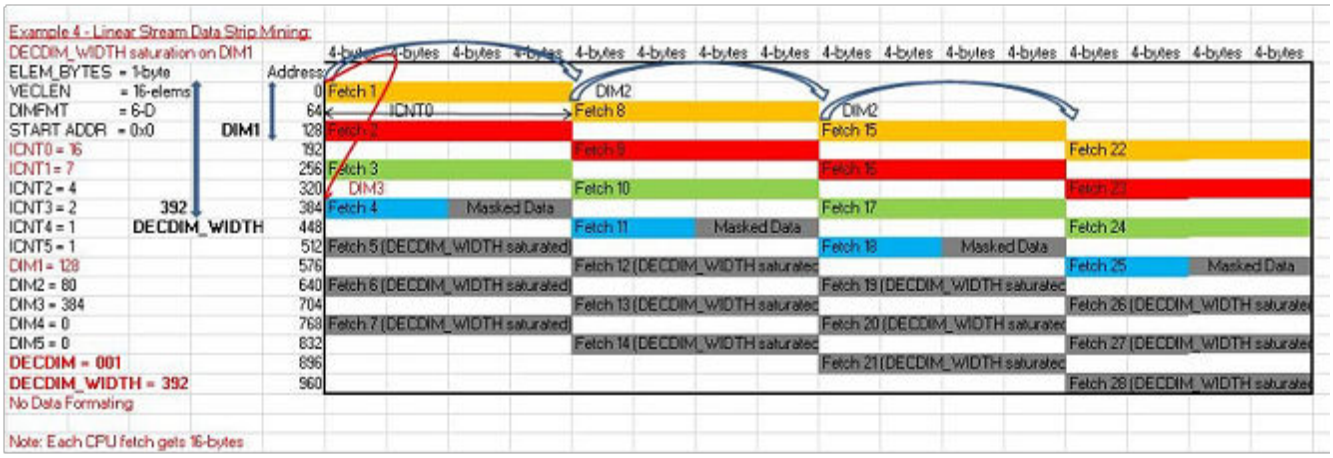


Figure 3-121. Linear Stream Data Strip Mining: Example 4, DECDIM on DIM1 with DECDIM_WIDTH Saturation

Figure 3-122. Linear Stream Data Strip Mining: Example 5, DECDIM on DIM2 with ELDUP 2x and Promote 2x

Figure 3-123. Linear Stream Data Strip Mining: Example 6, DECDIM on DIM2 with ELDUP 4x and Promote 4x

Transpose data strip mining is similar to linear data strip mining, except the data moves in a transpose pattern. DECDIM FLAGS provides the dimension for when the DECDIM_WIDTH count decrement occurs. This allows the data strip mask to apply to the remaining elements in the GRANULE. In DECDIM mode and transpose, the setting for DECDIM = DIM1 is not supported as mentioned in previous tables. The following figures show normal and data strip mining patterns for transpose. For regular transpose patterns, when the ICNT0 is not a multiple of the GRANULE, gappy data occurs and is filled with zeros. This is still valid in DECDIM transpose mode when the ICNT0 is not a multiple of the GRANULE, in addition to the data strip masking that is applied using DECDIM_WIDTH.

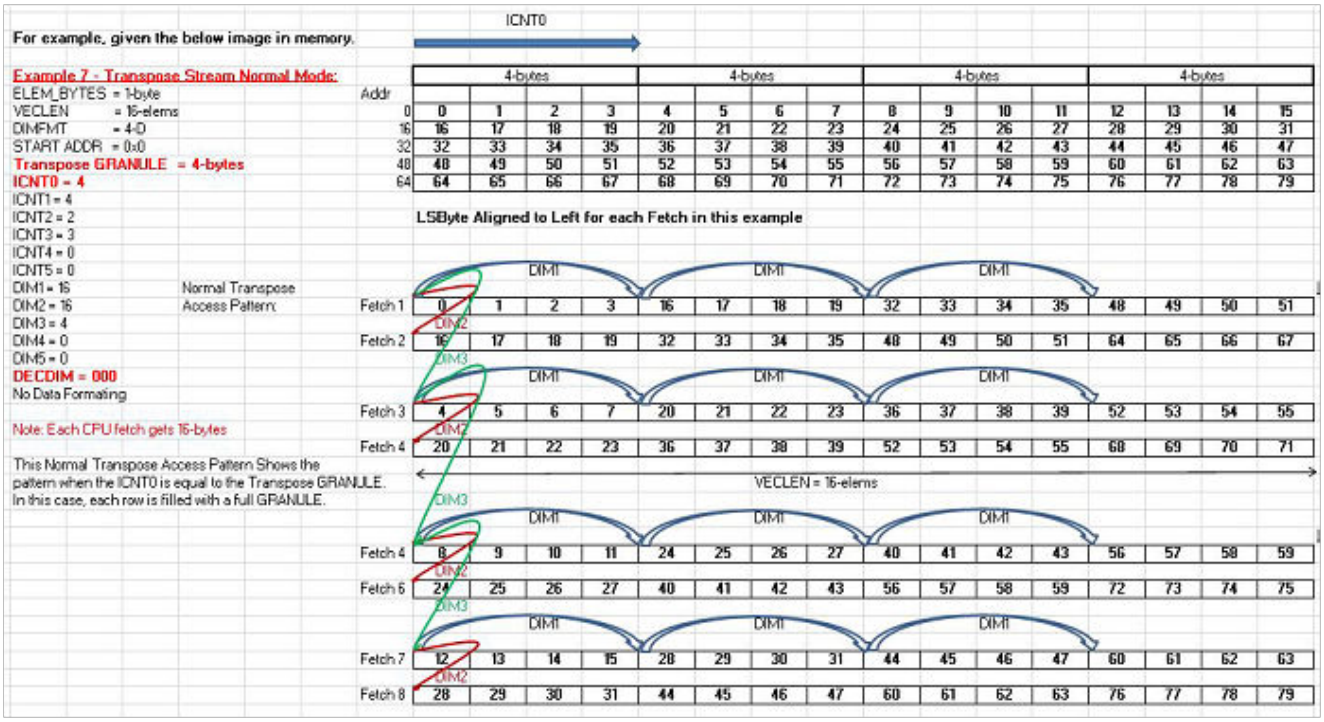


Figure 3-124. Transpose Stream Normal Mode: Example 7, ICNT0 Equal GRANULE

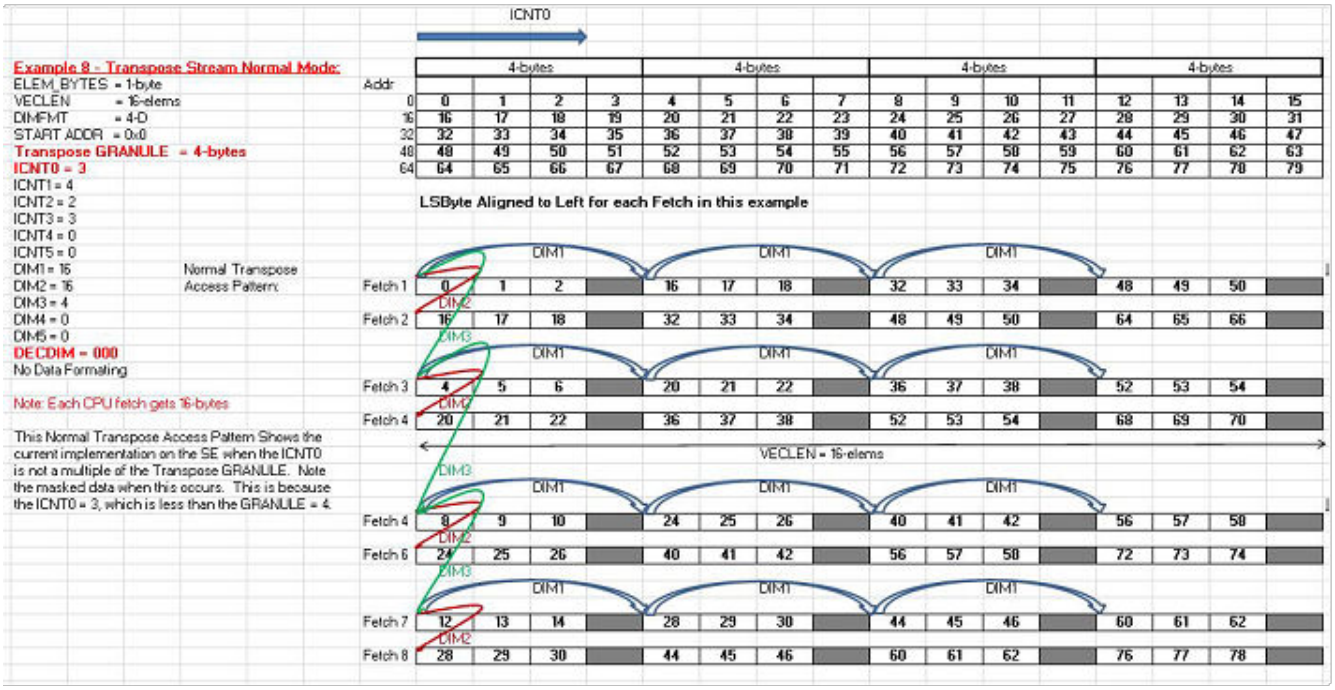


Figure 3-125. Transpose Stream Normal Mode: Example 8, ICNT0 Less Than GRANULE

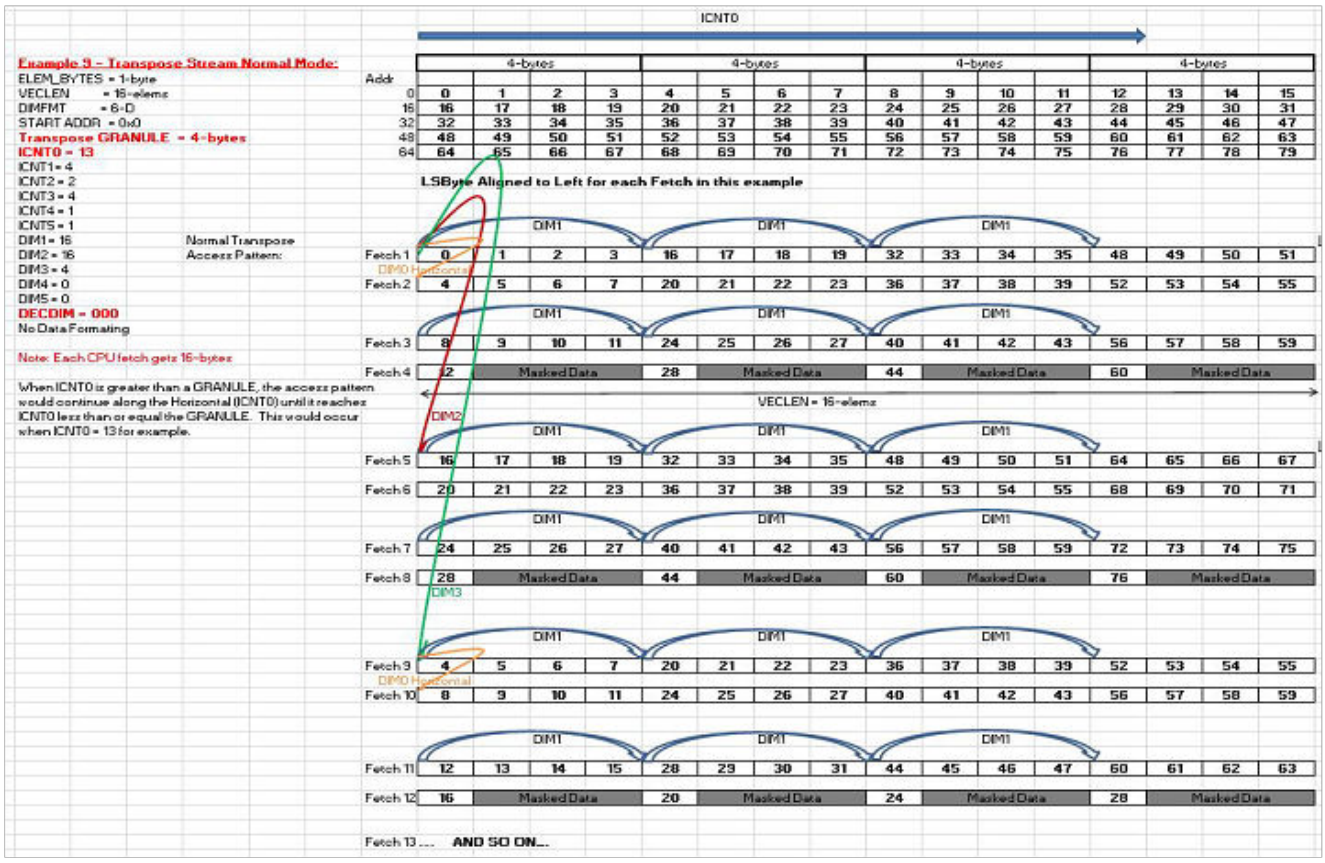


Figure 3-126. Transpose Stream Normal Mode: Example 9, ICNT0 Greater Than GRANULE

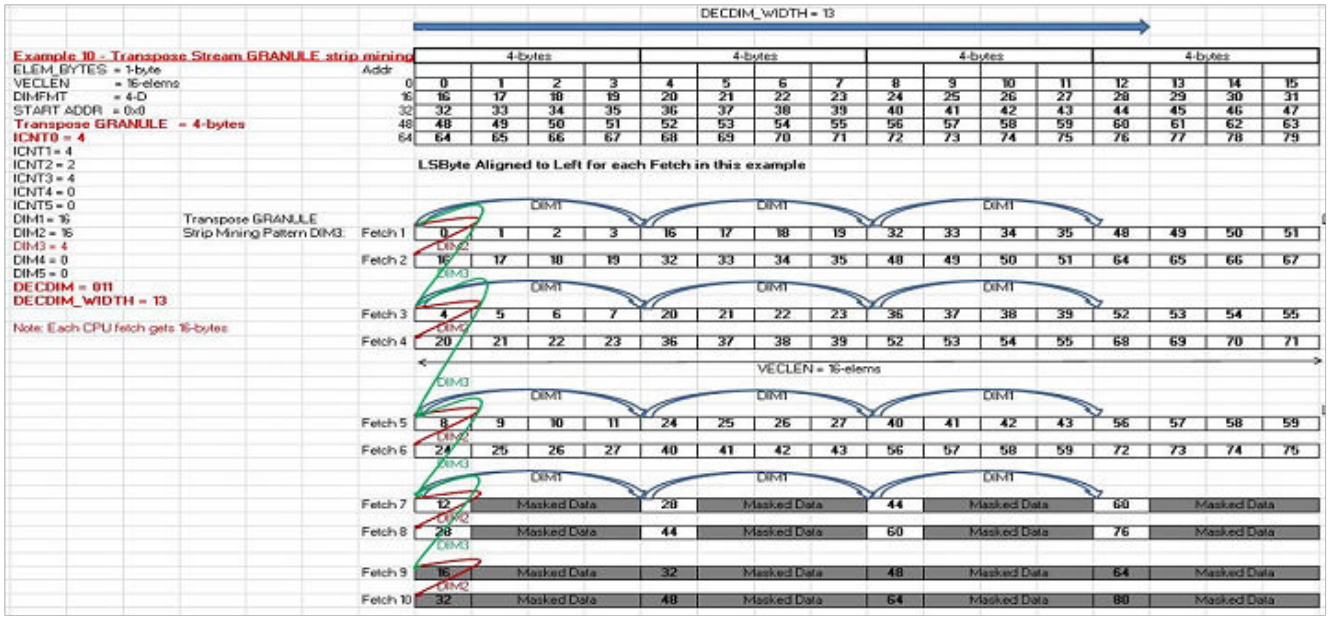


Figure 3-127. Transpose Data Strip Mining: Example 10, DECDIM on DIM3

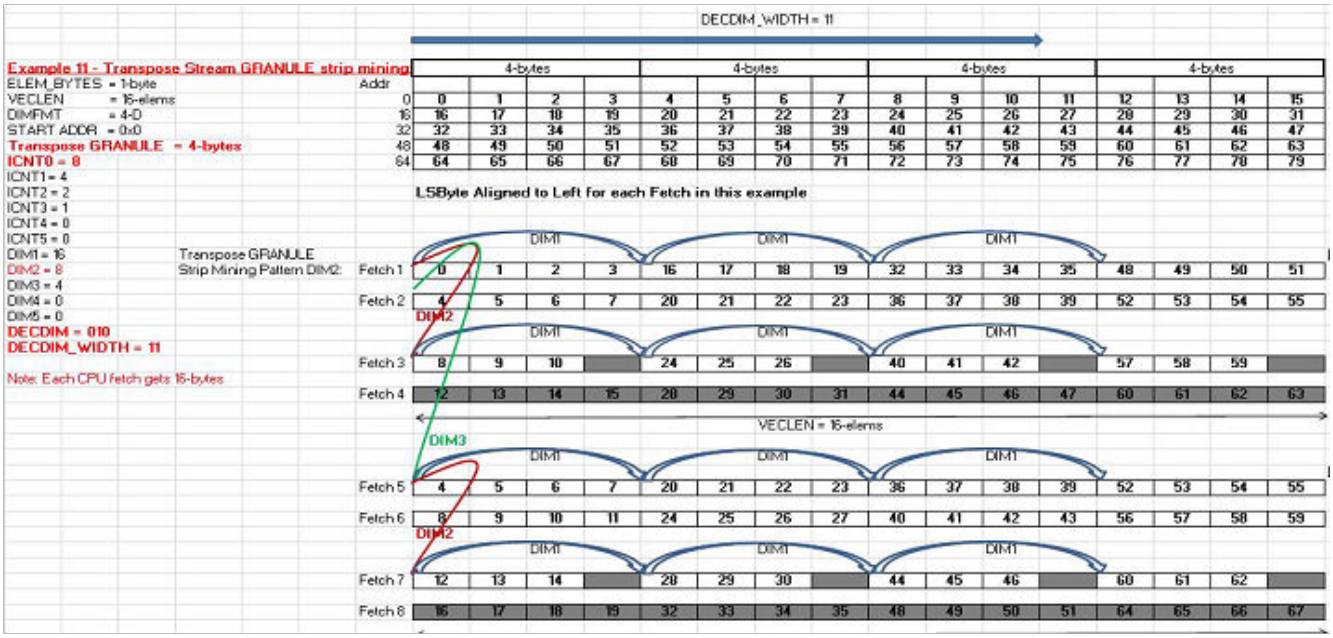


Figure 3-128. Transpose Data Strip Mining: Example 11, DECDIM on DIM2

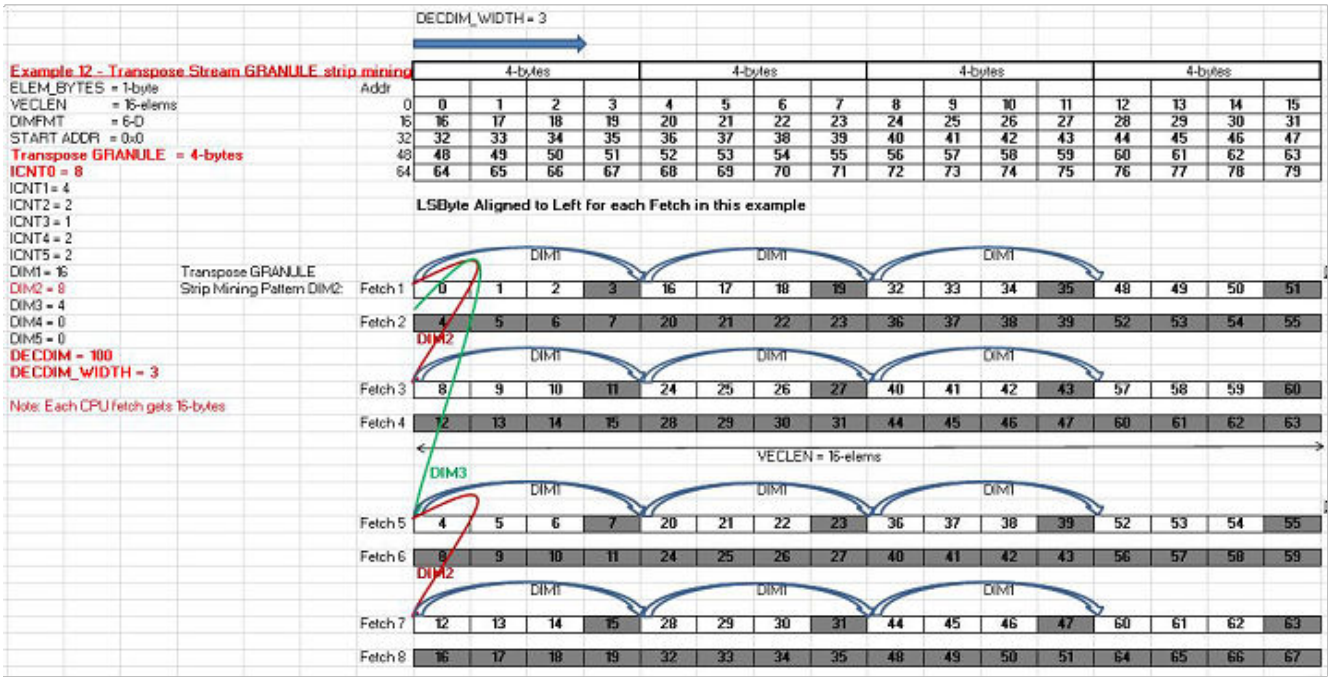


Figure 3-129. Transpose Data Strip Mining: Example 12, DECDIM on DIM4

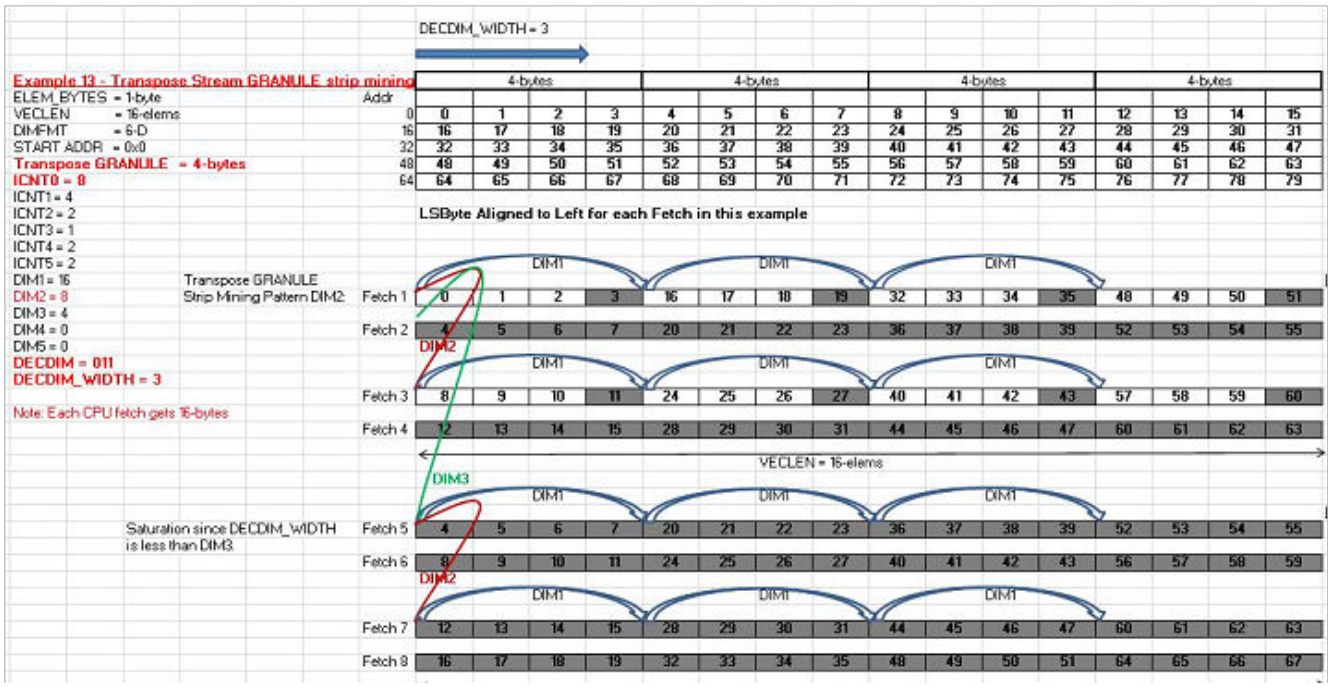


Figure 3-130. Transpose Data Strip Mining: Example 13, DECDIM on DIM3

The previous example all uses one set of DECDIM FLAGS (either DECDIM1 or DECDIM2). In the cases where two sets of DECDIM FLAGS are used, the following examples show how the masking is performed.

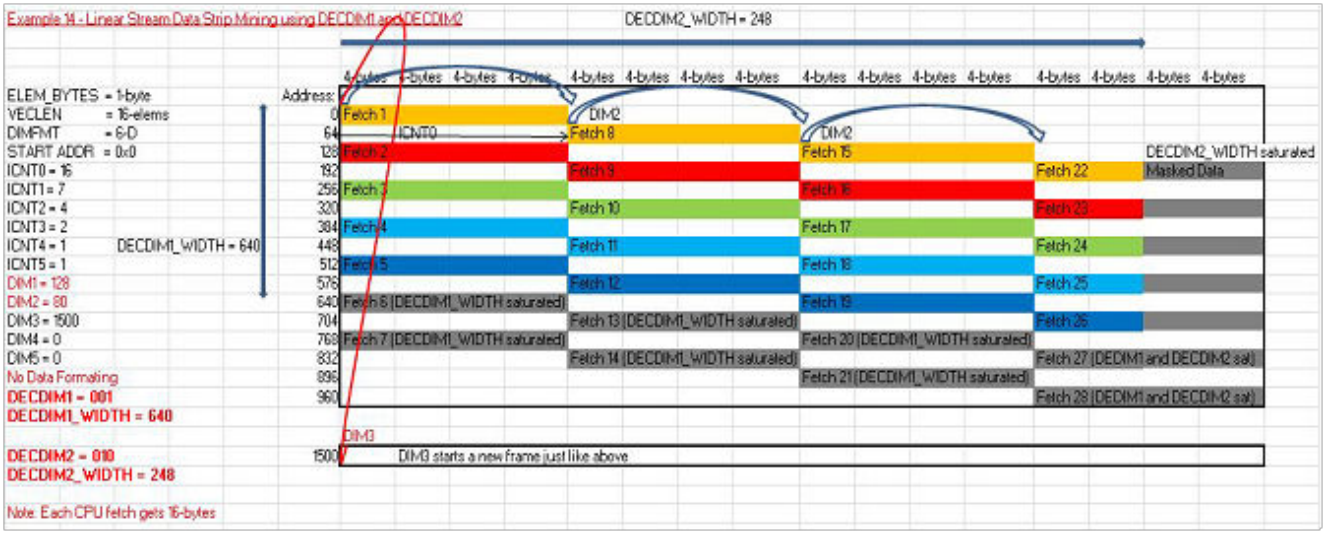


Figure 3-131. Linear Stream Data Strip Mining: Example 14, DECDIM1on DIM1 and DECDIM2 on DIM2

Figure 3-132. Transpose Data Strip Mining: Example 15, DECDIM1 on DIM2 and DECDIM2 on DIM3, ICNT0 equal GRANULE

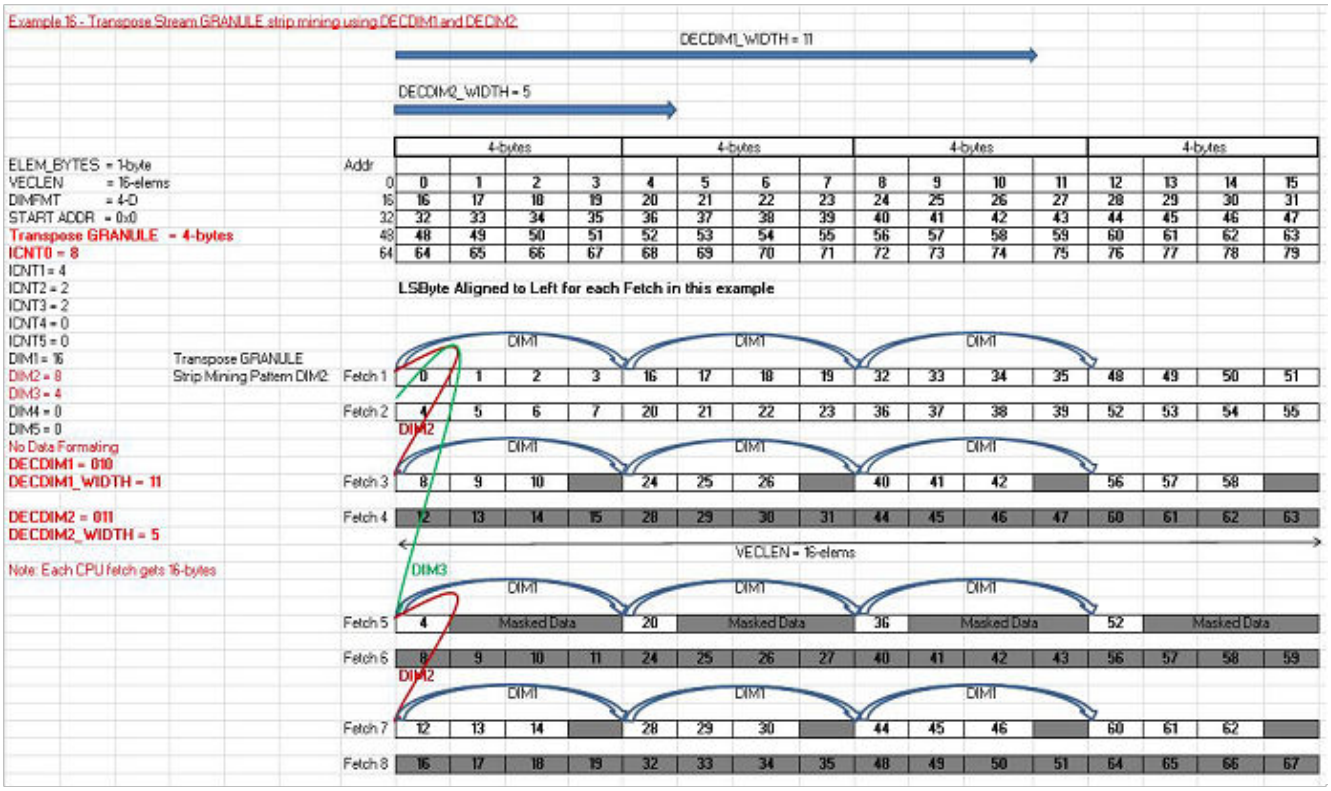


Figure 3-133. Transpose Data Strip Mining: Example 16, DECDIM1 on DIM2 and DECDIM2 on DIM3, ICNT0 greater than GRANULE

The next set of figures illustrate all DECDIMx and DECDIMxSD interactions. The examples shown are for linear streams. Transpose streams follow a similar scheme, but in transpose fashion on the GRANULE, and have the same requirements and restrictions as discussed earlier.

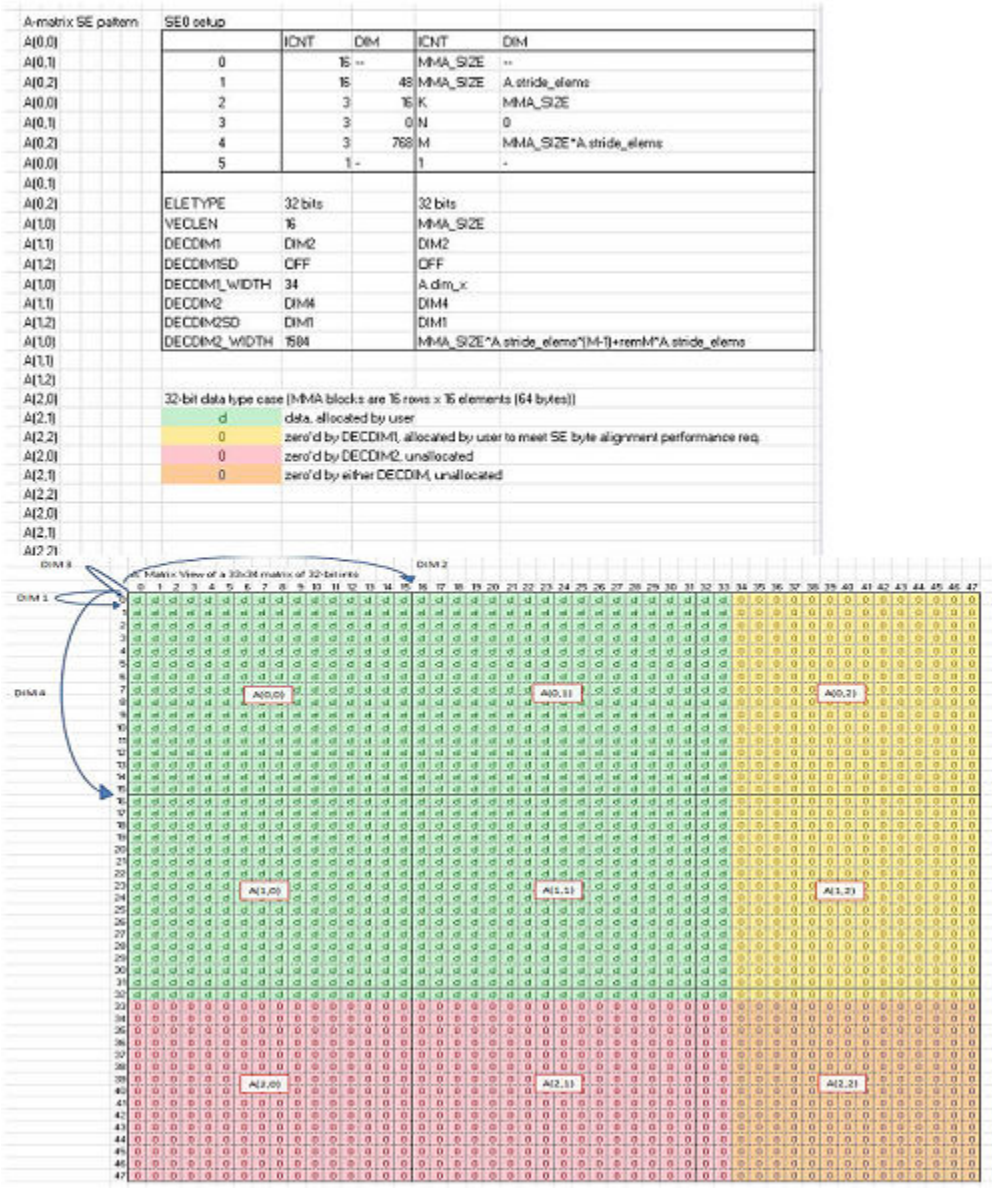


Figure 3-134. Linear Stream Data Strip Mining: Example 17, DECDIMx and DECDIMxSD

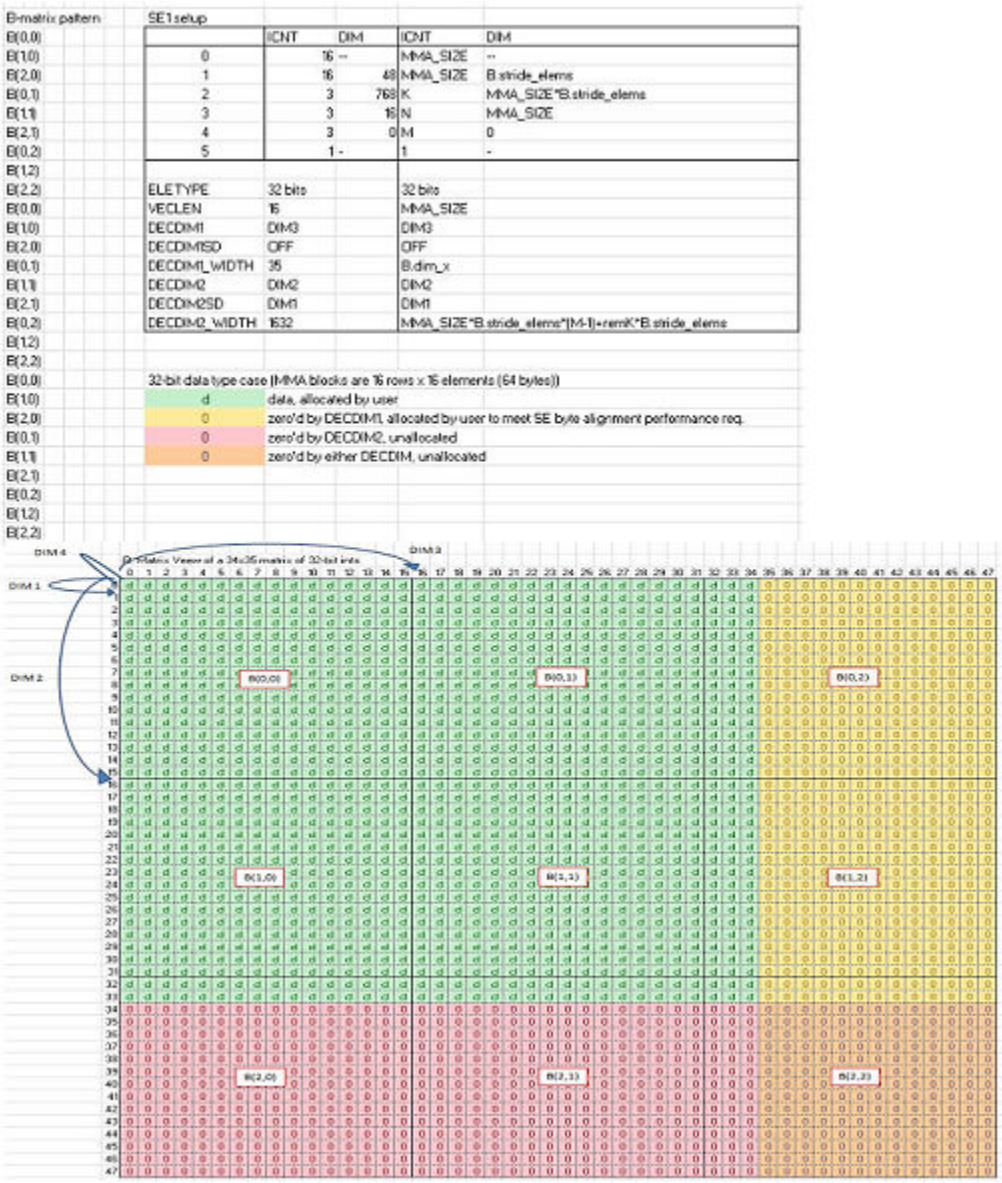


Figure 3-135. Linear Stream Data Strip Mining: Example 18, DECDIMx and DECDIMxSD

3.22.4.3.3 AMn: Addressing Mode Selection

The fields AM0 through AM5 control the addressing modes for each of the 6 dimensions. The streaming engine supports the addressing modes listed in Table 3-163.

Table 3-163. Addressing Modes Selected by AM0 Through AM5

AMx	Meaning
00b	Linear addressing mode
01b	Circular addressing using circular buffer size in CBK0
10b	Circular addressing using circular buffer size determined by CBK0 + CBK1 + 1
11b	Reserved(1)

(1) The reserved encoding triggers an error.

Each dimension has its own addressing mode, independent of the other dimensions. This permits any mixture of linear and circular addressing among the 6 dimensions.

When a given dimension selects linear addressing, the streaming engine uses unmodified 2s complement arithmetic to update addresses for that loop level.

When a given dimension selects circular addressing through CBK0 or CBK1, the streaming engine uses the circular address arithmetic described in Section 3.22.3.2.1.4. The next section describes the circular addressing mask computations for CBK0 and CBK1.

3.22.4.3.4 CBK0 and CBK1: Circular Block Size Selection

The CBK0 and CBK1 fields set the circular block sizes for circular addressing. CBK0 directly determines the circular block size for block 0. CBK1 combines with CBK0 to determine the circular block size for block 1. Specifically, the streaming engine sets block 1's size according to CBK0 + CBK1 + 1. Therefore, circular block 1 is always larger than circular block 0.

Table 3-164 illustrates the resulting valid block sizes for circular block 0 and 1. For circular block 0, use the value of CBK0 directly. Circular block 0 supports block sizes ranging from 512 bytes to 16M bytes. For circular block 1, use the value of CBK0 + CBK1 + 1. Circular block 1 supports sizes ranging from 1K bytes to 64G bytes.

Table 3-164. Circular Block Size Decoding

Encoded Block Size	Decoded Block Size(1)	Encoded Block Size	Decoded Block Size	Encoded Block Size	Decoded Block Size	Encoded Block Size	Decoded Block Size
0	512	8	128K	16	32M	24	8GB
1	1K	9	256K	17	64M	25	16GB
2	2K	10	512K	18	128M	26	32GB
3	4K	11	1M	19	256M	27	64GB
4	8K	12	2M	20	512M	28	Reserved
5	16K	13	4M	21	1G	29	Reserved
6	32K	14	8M	22	2G	30	Reserved
7	64K	15	16M	23	4G	31	Reserved

(1) All decoded block sizes are in bytes, irrespective of element size.

The following example code shows one way to compute the masks referred to by the address_add function described in Section 3.22.3.2.1.4.

Circular address mask computation:

```
// Circular addressing masks.
// 1 bits in the mask indicate address bits that remain fixed.
// 0 bits in the mask indicate address bits that are allowed to vary.
uint64_t circ_mask_0;                       // mask for circular addressing block 0
uint64_t circ_mask_1;                       // mask for circular addressing block 1
bool compute_circular_address_masks( int cbk0, int cbk1 )
{
    int block_size_0 = cbk0 + 9;                                        // power-of-2 of block size
in bytes
    int block_size_1 = cbk0 + cbk1 + 10;                                        // power-of-2 of
block size in bytes
    if ( block_size_1 > 35 )
        return false;                                   // invalid block size detected
    circ_mask_0 = (~0ULL) << block_size_0;
    circ_mask_1 = (~0ULL) << block_size_1;
    return true;                // both block sizes are valid
}
```

3.22.4.3.5 DIR: Stream Direction Flag

The DIR field sets the stream direction. When DIR = 0, the streaming engine reads all elements along dimension 0 in increasing address order. When DIR = 1, the streaming engine reads all elements along dimension 0 in decreasing address order.

3.22.4.3.6 DIMFMT: Stream Dimensions versus Linear, Transpose, and SEBRK

The streaming engine can support up to 6 dimensions. Programs set up the DIMFMT field according to how many dimensions are desired. The ICNTs for the selected DIMFMT should not be programmed to zero because that implies an empty stream, and the streaming engine would return zero data back to the CPU on subsequent reads. Any ICNTs which are not active for the selected DIMFMT are internally not used inside the streaming engine, thus programs are not required to program those ICNTs. All ICNTs are unsigned integer values.

In transposed streams, the two innermost loops are interchanged. See Section 3.22.4.3.6. Table 3-165 and Table 3-166 illustrate the valid ICNT settings for the selected dimension.

3.22.4.3.6.1 DIMFMT versus Loop Levels

Table 3-165. Stream Dimension Loop Levels, Linear Streams

DIMFMT	ICNT5	ICNT4	ICNT3	ICNT2	ICNT1	ICNT0
000b	NA	NA	NA	NA	NA(1)	>= 1(2)
001b	NA	NA	NA	NA	> 1	>= 1
010b	NA	NA	NA	> 1	>= 1	>= 1
011b	NA	NA	> 1	>= 1	>= 1	>= 1
100b	NA	> 1	>= 1	>= 1	>= 1	>= 1
101b	> 1	>= 1	>= 1	>= 1	>= 1	>= 1
110b - 111b	Reserved					

(1) The streaming engine does not internally use the non-active ICNTs for the selected DIMFMT. Programs are not required to program these ICNTs.

(2) ICNT0 - ICNT5 cannot be programmed as zero. They must be equal to 1, or greater than 1 for the selected DIMFMT. See Section 3.22.4.3.6.

Table 3-166. Stream Dimension Loop Levels, Transpose Streams

DIMFMT	ICNT5	ICNT4	ICNT3	ICNT2	ICNT1(1)	ICNT0(2)
000b(3)	NA	NA	NA	NA(4)	>= 1, <= 16	NA
001b	NA	NA	NA	NA	>= 1, <= 16	>= 1(5)
010b	NA	NA	NA	> 1	>= 1, <= 16	>= 1
011b	NA	NA	> 1	>= 1	>= 1, <= 16	>= 1
100b	NA	> 1	>= 1	>= 1	>= 1, <= 16	>= 1
101b	> 1	>= 1	>= 1	>= 1	>= 1, <= 16	>= 1
110b - 111b	Reserved					

(1) In transpose streams, the minimum vertical height of the column is 1, and the maximum vertical height of the column is currently limited to 16. The two innermost loops in transpose are interchanged, which means ICNT1/DIM1 is the first dimension and ICNT0 is the second dimension.

(2) In transpose streams, if the ICNT0 is not a multiple of the GRANULE size, zero padding is inserted for missing elements within each GRANULE.

(3) Conceptually, this is a 1-dimensional vertical column stream with 1 element and zero padding to fill GRANULE. Technically, a 1-D transpose is not valid because transpose requires at least a 2-D (DIMFMT = 001b).

(4) The streaming engine does not internally use the non-active ICNTs for the selected DIMFMT. Programs are not required to program these ICNTs.

(5) ICNT0 - ICNT5 should not be programmed as zero. They must be equal to 1, or greater than 1 for the selected DIMFMT. See Section 3.22.4.3.6.

3.22.4.3.6.2 DIMFMT versus DIMs

Each dimension stride, DIM1 - DIM5, is tied directly to it's respective ICNT1 - ICNT5 loop levels. All DIMs are signed integer values.

Table 3-167. Stream Dimension DIMs, Linear Streams

DIMFMT	DIM5	DIM4	DIM3	DIM2	DIM1
000b	NA	NA	NA	NA	NA(1)
001b	NA	NA	NA	NA	Signed Value
010b	NA	NA	NA	Signed Value	Signed Value
011b	NA	NA	Signed Value	Signed Value	Signed Value
100b	NA	Signed Value	Signed Value	Signed Value	Signed Value
101b	Signed Value	Signed Value	Signed Value	Signed Value	Signed Value

Table 3-167. Stream Dimension DIMs, Linear Streams (continued)

DIMFMT	DIM5	DIM4	DIM3	DIM2	DIM1
110b - 111b	Reserved				

(1) The streaming engine does not internally use the non-active DIMs for the selected DIMFMT. Programs are not required to program these DIMs.

Table 3-168. Stream Dimension DIMs, Transpose Streams

DIMFMT	DIM5	DIM4	DIM3	DIM2	DIM1[1]
000b[2]	NA	NA	NA	NA[3]	Signed Value
001b	NA	NA	NA	NA	Signed Value
010b	NA	NA	NA	Signed Value	Signed Value
011b	NA	NA	Signed Value	Signed Value	Signed Value
100b	NA	Signed Value	Signed Value	Signed Value	Signed Value
101b	Signed Value	Signed Value	Signed Value	Signed Value	Signed Value
110b - 111b	Reserved				

(1) In transpose mode, the two innermost loops are interchanged, which means ICNT1/DIM1 is the first dimension and ICNT0 is the second dimension.

(2) Conceptually, this is a 1-dimensional vertical column stream with 1 element and zero padding to fill GRANULE. Technically, a 1-D transpose is not valid because transpose requires at least a 2-D (DIMFMT = 001b).

(3) The streaming engine does not internally use the non-active DIMs for the selected DIMFMT. Programs are not required to program these DIMs.

3.22.4.3.6.3 DIMFMT versus Transpose Streams Loop Interchange

Transpose mode interchanges the two innermost loop levels. That is, in transpose mode, the two innermost loops ICNT0 and ICNT1 are interchanged. ICNT1 determines the number of rows in each column. A column is defined as a GRANULE size. ICNT0 is the second dimension in a transpose stream and defines the horizontal width (which may or may not be a multiple of the GRANULE). In this generation of the streaming engine, the maximum row height, ICNT1, must be at least 1 and less than or equal 16. There is no restrictions on the ICNT0 in transpose. However, if the ICNT0 is not a multiple of the GRANULE size, the streaming engine pads zeros in the missing elements of each GRANULE. Table 3-169 shows the interchange of the loop levels in transpose mode. See the section on transpose mode for further details, Section 3.22.4.3.13.

Table 3-169. Loop Levels Interchanged by Transposed Streams

DIMFMT	Levels Interchanged
000b	ICNT0 and ICNT1
001b	ICNT0 and ICNT1
010b	ICNT0 and ICNT1
011b	ICNT0 and ICNT1
100b	ICNT0 and ICNT1
101b	ICNT0 and ICNT1
110b - 110b	Reserved

3.22.4.3.6.4 DIMFMT versus Stream Break

A stream break is used to exit early, or break out, of any loop level in the stream. Depending on which dimension is active, a SEBRK on a higher dimension could end the stream. When a SEBRK causes a stream to end, the streaming engine starts to return zero data to the CPU for each subsequent read, as if the CPU read through to the end of the stream. See the section on stream breaks for further details, Section 3.22.3.2.1.5.

Table 3-170. Loop Level Broken Out of by SEBRK, Linear Streams

DIMFMT	SEBRK5[1]	SEBRK 4	SEBRK 3	SEBRK 2	SEBRK 1	SEBRK 0
000b	ICNT5	ICNT4	ICNT3	ICNT2	ICNT1	ICNT0[2]
001b	ICNT5	ICNT4	ICNT3	ICNT2	ICNT1[3]	ICNT0
010b	ICNT5	ICNT4	ICNT3	ICNT2[4]	ICNT1	ICNT0

Table 3-170. Loop Level Broken Out of by SEBRK, Linear Streams (continued)

DIMFMT	SEBRK5(1)	SEBRK 4	SEBRK 3	SEBRK 2	SEBRK 1	SEBRK 0
011b	ICNT5	ICNT4	ICNT3(5)	ICNT2	ICNT1	ICNT0
100b	ICNT5	ICNT4(6)	ICNT3	ICNT2	ICNT1	ICNT0
101b	ICNT5(7)	ICNT4	ICNT3	ICNT2	ICNT1	ICNT0
110b - 111b	Reserved					

(1) SEBRK5 always ends the stream.
(2) In a 1-dimensional stream, SEBRK0 - SENRK5 ends the stream.
(3) In a 2-dimensional stream, SEBRK1 - SEBRK5 ends the stream.
(4) In a 3-dimensional stream, SEBRK2 - SEBRK5 ends the stream.
(5) In a 4-dimensional stream, SEBRK3 - SEBRK5 ends the stream.
(6) In a 5-dimensional stream, SEBRK4 - SEBRK5 ends the stream.
(7) In a 6-dimensional stream, SEBRK5 ends the stream.

SEBRK behaves slightly differently in transposed streams, as its notion of loop level refers to the transposed loop nest as described in ‘Section 3.22.3.2.1.5. Specifically, in transposed streams, the two innermost loop levels ICNT0 and ICNT1 are interchanged. Table 3-171 illustrates how SEBRK for transposed streams work.

Table 3-171. Loop Level Broken Out of by SEBRK, Transposed Streams

DIMFMT	SEBRK5(1)	SEBRK 4	SEBRK 3	SEBRK 2	SEBRK 1	SEBRK 0
000b(2)	ICNT5	ICNT4	ICNT3	ICNT2	ICNT0	ICNT1(3)
001b	ICNT5	ICNT4	ICNT3	ICNT2	ICNT0(4)	ICNT1
010b	ICNT5	ICNT4	ICNT3	ICNT2(5)	ICNT0	ICNT1
011b	ICNT5	ICNT4	ICNT3(6)	ICNT2	ICNT0	ICNT1
100b	ICNT5	ICNT4(7)	ICNT3	ICNT2	ICNT0	ICNT1
101b	ICNT5(8)	ICNT4	ICNT3	ICNT2	ICNT0	ICNT1
110b - 111b	Reserved					

(1) SEBRK5 always ends the stream.
(2) Conceptually, this is a 1-dimensional vertical column stream with 1 element and zero padding to fill GRANULE. Technically, a 1-D transpose is not valid because transpose requires at least a 2-D (DIMFMT = 001b).
(3) In a 1-dimensional stream, SEBRK0 - SENRK5 ends the stream.
(4) In a 2-dimensional stream, SEBRK1 - SEBRK5 ends the stream.
(5) In a 3-dimensional stream, SEBRK2 - SEBRK5 ends the stream.
(6) In a 4-dimensional stream, SEBRK3 - SEBRK5 ends the stream.
(7) In a 5-dimensional stream, SEBRK4 - SEBRK5 ends the stream.
(8) In a 6-dimensional stream, SEBRK5 ends the stream.

3.22.4.3.7 THROTTLE: Fetch Ahead Throttling (Stretch Goal)

This feature has been deferred to next generation.

The THROTTLE field controls how aggressively the streaming engine fetches ahead of the CPU.

Table 3-172. THROTTLE Encodings

THROTTLE	Description
00b	Minimum throttling / maximum fetch-ahead
01b	Less throttling / more fetch-ahead
10b	More throttling / less fetch-ahead
11b	Maximum throttling / minimum fetch-ahead

THROTTLE does not change the meaning of the stream, and serves only as a hint. The streaming engine may ignore this field. Crucially, programs must not rely on the specific throttle behavior for program correctness, as the architecture does not specify the precise throttle behavior.

THROTTLE allows programmers to provide hints to the hardware about the program’s own behavior. By default, the streaming engine attempts to get as far ahead of the CPU as it can to hide as much latency as possible,

while providing full stream throughput to the CPU. While several key applications need this level of throughput, it can lead to bad system level behavior for others.

For instance, the streaming engine discards all fetched data across context switches. Therefore, aggressive fetch-ahead can lead to wasted bandwidth in a system with large numbers of context switches. Aggressive fetch-ahead only makes sense in those systems if the CPU consumes data quickly.

3.22.4.3.8 DECIM: Element Decimation Flag

The DECIM field determines whether the streaming engine decimates the incoming stream. DECIM = 00b disables decimation. DECIM = 01b enables 2:1 decimation. DECIM = 10b enables 4:1 decimation. Decimation always removes whole elements, not sub-elements. When DECIM = 01b, the streaming engine removes every odd-numbered element from the incoming stream, and when DECIM = 10b, the streaming engine drops three elements from the stream, returning every other 4th element as described in Section 3.22.3.2.2.2.

When DECIM = 01b, the following restrictions apply:
- Data promotion by at least 2x must be enabled. That is, PROMOTE ≠ 000b.
- ICNT0 must be a multiple of 2.
- DECDIM_WIDTH and the selected DECDIM dimension must be multiple of 2.
- The total vector length (VECLEN) must be large enough to hold a single promoted, duplicated element, as per Table 3-174.

In transposed streams (TRANSPOSE ≠ 0), the transpose granule must be at least twice the element size in bytes before promotion, as shown in Table 3-176.

When DECIM = 10b, the following restrictions apply:
- Data promotion by at least 4x must be enabled. That is, PROMOTE ≠ 000b, 001b, and 101b.
- ICNT0 must be a multiple of 4.
- DECDIM_WIDTH and the selected DECDIM dimension must be multiple of 4.
- The total vector length (VECLEN) must be large enough to hold a single promoted, duplicated element, as per Table 3-174.

In transposed streams (TRANSPOSE ≠ 0), the transpose granule must be at least four times the element size in bytes before promotion, as shown in Table 3-176.

3.22.4.3.9 ELDUP: Element Duplication

The ELDUP field specifies the number of times to duplicate each element. The element size multiplied with the element duplication amount must not exceed the 64 bytes. Table 3-173 illustrates the ELDUP encoding and maximum permitted element size.

Table 3-173. ELDUP Encodings

ELDUP	Duplication Factor	Maximum Bytes per Promoted Element
000b	No duplication	16(1)
001b	Duplicate 2×	16
010b	Duplicate 4×	16
011b	Duplicate 8×	8
100b	Duplicate 16×	4
101b	Duplicate 32×	2
110b	Duplicate 64×	1
111b	Reserved	

(1) Largest promoted element size currently defined is 16 bytes/element. See Section 3.22.4.3.14.

3.22.4.3.10 VECLEN: Stream Vector Length

The VECLEN field defines the stream vector length for this stream, in terms of elements. The streaming engine breaks the stream into groups of elements that are VECLEN elements long, which can include duplicated elements when element duplication is enabled. The product of the VECLEN, and element size, and any promotion and duplication factor must be shorter than the CPU's native vector width (which is currently 64 bytes).

The streaming engine presents the stream to the CPU as either a sequence of pairs of single vectors or a sequence of double vectors. When VECLEN elements is combined with data formatting option is shorter than the CPU's native vector width, the streaming engine pads out the extra lanes with zeros in the vector it provides to the CPU. The GRDUP field (next section) determines the type of padding the streaming engine uses.

Element duplication expands elements to fill more bytes within the stream. The duplicated element must fit entirely within the VECLEN, and any combinations of data formatting must fit within the CPU's native vector width of 64 bytes. This renders some combinations of ELDUP and VECLEN invalid, as shown in Table 3-174, which shows valid combinations for an element size of 1-byte, as defined in ELTYPE flags field.

Table 3-174. VECLEN Encoding

VECLEN	Vector Length in Elements	Maximum Element Duplication for Each VECLEN Setting(1)						
		No ELDUP	ELDUP 2x	ELDUP 4x	ELDUP 8x	ELDUP 16x	ELDUP 32x	ELDUP 64x
000b	1 element	Valid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
001b	2 elements	Valid	Valid	Invalid	Invalid	Invalid	Invalid	Invalid
010b	4 elements	Valid	Valid	Valid	Invalid	Invalid	Invalid	Invalid
011b	8 elements	Valid	Valid	Valid	Valid	Invalid	Invalid	Invalid
100b	16 elements	Valid	Valid	Valid	Valid	Valid	Invalid	Invalid
101b	32 elements	Valid	Valid	Valid	Valid	Valid	Valid	Invalid
110b	64 elements	Valid	Valid	Valid	Valid	Valid	Valid	Valid
111b	Reserved							

(1) For element size greater than 1-byte, defined in ELTYPE flags field, if the combinations of (VECLEN x Element size x ELDUP) exceeds 64-bytes, then some "Valid" rows would be "Invalid". This table is only conveying the maximum ELDUP setting that is valid for selected VECLEN. Any type Promotions in addition to this that causes the vector width to exceed 64-bytes would also make some rows invalid.

From a compiler perspective, the method for using VECLEN in compiler code would look similar to the following examples.

Short32 example setup and usage:

```
Setup:
VECLEN = __SE_VECLEN_32ELEMS; /* 32 shorts */
ELTYPE = __SE_ELTYPE_16BIT;
PROMOTE = __SE_PROMOTE_OFF;
Usage:
short32 Sum;
Sum = __SE0ADV(short32);
```

In the above example, the type being returned has the number of elements that user expects the SE to return per CPU fetch - "short32" returns 32 shorts. Thus, the VECLEN flag should be set to 32 elements and the ELTYPE to a 2-byte element size. Also, ELTYPE is equal to the "short" portion because type Promotion is OFF. If vectors of "short16" is desired, then the VECLEN should be set to 16ELEMS and the SE returns 16 shorts of data for each CPU fetch.

The next example shows how to promote "short" to "int" using Promotion.

int16 example setup and usage:

```
Setup:
VECLEN = __SE_VECLEN_16ELEMS; /* 32 shorts */
ELTYPE = __SE_ELTYPE_16BIT;
PROMOTE = __SE_PROMOTE_2X_ZEROEXT; /* promote from short to int by zero-extending */
Usage:
int16 Sum;
Sum = __SE0ADV(int16);
```

The next example enables element duplication following the rules specified in Table 3-174.

short32 with element duplication:

```
Setup:
VECLEN = __SE_VECLEN_16ELEMS; /* 32 shorts */
ELTYPE = __SE_ELTYPE_16BIT;
PROMOTE = __SE_PROMOTE_OFF;
ELDUP = __SE_ELDUP_2x;
Usage:
short32 Sum;
Sum = __SE0ADV(short32);
```

In the above example, the VECELN is set to 16ELEMS because element duplication of 2X is enabled. The SE provides 32 shorts per CPU fetch.

Lastly, with group duplication, the SE duplicates the VECELEN elements in groups over the full CPU native vector width of 64 bytes. The following example illustrates this.

short32 with group duplication:

```
Setup:
VECLEN = __SE_VECLEN_4ELEMS; /* we want groups of 4 elements */
ELTYPE = __SE_ELTYPE_16BIT;
PROMOTE = __SE_PROMOTE_OFF;
ELDUP = __SE_ELDUP_OFF;
GRDUP = __GRPDUP_ON;
Usage:
short32 Sum;
Sum = __SE0ADV(short32);
```

In the above example, if only ‘short16’ are needed and group duplication is still desired, the same setup still suffices because the upper vector lanes would be ignored by the compiler.

Note

In summary, the VECLEN should be the total number of elements that is returned after promotion has been factored in, and except in the group duplication case, should exactly match the type that is placed on the “__SE0ADV(<type>)" function which accesses it.

3.22.4.3.11 GRDUP: Group Duplication

The GRDUP field specifies how the stream engine pads stream vectors out to the length of CPU vectors. When GRDUP = 0, the streaming engine fills the extra lanes with zero, and marks them invalid. When GRDUP = 1, the streaming engine fills the extra lanes with copies of the group of elements in each stream vector, as illustrated in Section 3.22.3.2.3.5.

GRDUP = 1 has no effect when VECLEN is set to the CPU’s native vector width. However, if that same program runs on a machine with a larger native vector width, then GRDUP’s setting becomes meaningful. Therefore, some code may find it useful to set GRDUP = 1 even when VECLEN equals the CPU’s native vector width, to correctly adapt to larger CPU vector widths in the future.

3.22.4.3.12 PROMOTE: Element Promotion

The PROMOTE field controls whether the streaming engine promotes sub-elements in the stream, and what type to promotion to apply. When enabled, the streaming engine promotes types by a single power-of-2 size.

Table 3-175. PROMOTE Encodings and Resulting Sub-element Sizes

PROMOTE	Promotion Factor	Promotion Type	Resulting Sub-element Sizes for Each Initial Size			
			8-bit	16-bit	32-bit	64-bit
000b	1×	N/A	8-bit	16-bit	32-bit	64-bit
001b	2×	Zero Extend	16-bit	32-bit	64-bit	Invalid
010b	4×	Zero Extend	32-bit	64-bit	Invalid	Invalid
011b	8×	Zero Extend	64-bit	Invalid	Invalid	Invalid
100b	Reserved					
101b	2×	Sign Extend	16-bit	32-bit	64-bit	Invalid

Table 3-175. PROMOTE Encodings and Resulting Sub-element Sizes (continued)

PROMOTE	Promotion Factor	Promotion Type	Resulting Sub-element Sizes for Each Initial Size			
			8-bit	16-bit	32-bit	64-bit
110b	4×	Sign Extend	32-bit	64-bit	Invalid	Invalid
111b	8×	Sign Extend	64-bit	Invalid	Invalid	Invalid

When the stream specifies no promotion, each sub-element occupies a vector lane equal in width to the size specified by ELTYPE. Otherwise, each sub-element occupies a vector lane 2×, 4×, or 8× as large. Therefore, when PROMOTE ≠ 000b, the promoted element size is larger than the original element size. Table 3-175 illustrates how sub-elements grow according to the promotion type. For complex data types, the element size is twice the sub-element size.

Promotion modes 001b through 011b perform zero-extension. The streaming engine fills the bytes added to the upper part of each expanded sub-element with zeros. Zero extension is useful for promoting unsigned integer types.

Note

Promotion modes 101b through 111b perform sign extension. Instead of filling the upper part of each expanded sub-element with zeros, the streaming engine fills those bytes with 0x00 or 0xFF, based on the most significant bit of the original sub-element. If that bit was zero, the streaming engine fills with 0x00; otherwise it fills with 0xFF. Sign extension is useful for promoting signed integer types.Promotion mode 100b is reserved. One proposed use for this encoding is floating point promotion: promote short-float to float, or float to double. Currently off-the-table.

3.22.4.3.13 TRANSPOSE: Transposed Streams

The TRANSPOSE field determines whether the streaming engine accesses the stream in a transposed order—that is, as if it exchanged the inner two addressing levels—and if so, at what granularity it transposes the stream.

Table 3-176. TRANSPOSE Encodings

TRANSPOSE	Transposition Granule	Maximum Initial Element Size(1)			Minimum Dimension Alignment(2)
		DECIM = 00b	DECIM = 01b(3)	DECIM = 10b(4)	
0000b	None. Transpose disabled.	N/A	N/A	N/A	1 byte
0001b	1 byte(5)	Not Allowed(6)	Not Allowed	1 byte	4 bytes(7)
0010b	2 bytes(8)	Not Allowed	2 bytes(9)	Not Allowed	4 bytes(10)
0011b	4 bytes	4 bytes	2 bytes	1 byte	4 bytes
0100b	8 bytes	8 bytes	4 bytes	2 bytes	4 bytes
0101b	16 bytes	16 bytes	8 bytes	4 bytes	8 bytes
0110b	32 bytes	16 bytes(11)	16 bytes	8 bytes	16 bytes
0111b	Reserved				

(1) Element size before promotion, element duplication, or group duplication. Transposed fetch occurs before promotion, element duplication, and group duplication.
(2) This applies to the starting address of the stream, as well as all active DIMn fields.
(3) For Transpose Granule of 4 bytes to 32 bytes, DECIM = 01b requires the transpose granule to be at least twice the element size in bytes.
(4) For Transpose Granule of 4 bytes to 32 bytes, DECIM = 10b requires the transpose granule to be at least four times the element size in bytes.
(5) 1 byte transpose is only supported when DECIM = 4x and at least Promote = 4x with minimum dimension alignment of 4 bytes.
(6) For 1 byte and 2 byte transpose, both decimation and promotion needs to be enabled, and will be flagged as error when not enabled.
(7) Alignment requirement allows reading individual 1-byte transpose columns out of the 4 columns within the 4 byte word alignment.
(8) 2 byte transpose is only supported when DECIM = 2x and at least Promote = 2x with minimum dimension alignment of 4 bytes.
(9) Due to hardware limitations, the element size should be set to 2-bytes in 2 bytes transpose: ELTYPE = 0001b or 1000b.
(10) Alignment requirement allows reading individual 2-byte transpose columns out of the 2 columns within the 4 byte word alignment.
(11) Largest non-promoted element size currently defined is 16 bytes/element. See Table 3-173.

The streaming engine transposes at a different granularity than the element size, allowing programs to fetch multiple columns of elements from each row. The transposition granularity must be no smaller than the element size (see Note below).

Due to hardware limitations, in the case of transpose by 2 bytes (TRANSPOSE = 0010b), the element size should be set to 2-bytes: ELTYPE = 0001b or 1000b. The streaming engine also requires that each granule fits entirely within VECLEN after promotion and element duplication. That is, the product of the granule size, promotion factor, decimation factor, and element duplication factor must be less than or equal to the total VECLEN size in bytes.

Currently, the streaming engine only supports transposed streams whose second active dimension is greater than or equal to 1, and less than or equal to 16. The second active dimension is ICNT1 in Transpose Mode. See Table 3-166.

The streaming engine does not place additional limitations on the iteration counts for other dimensions for transposed streams.

3.22.4.3.13.1 Transpose Dimension Alignment Requirements

The streaming engine does require that the first element in each dimension of a transposed stream start on a 4-byte (32-bit) boundary. For transpose granules larger than 8 bytes, the streaming engine requires the first element in each dimension of a transposed stream start on GRANULE / 2 boundaries. Refer to the Minimum Dimension Alignment in Table 3-176.

3.22.4.3.14 ELTYPE: Element Type Selection

The ELTYPE field encodes the type of elements in the stream. This includes element size, sub-element size, and whether the stream swaps sub-elements. Table 3-177 illustrates the ELTYPE encoding, as well as the resulting element size before type promotion. To determine sub-element size after promotion, refer to Table 3-175.

Table 3-177. ELTYPE Encodings

ELTYPE	Real / Complex(1)	Initial Sub-Element Size	Initial Element Size
0000b	Real	1 bytes	1 byte
0001b		2 bytes	2 bytes
0010b		4 bytes	4 bytes
0011b		8 bytes	8 bytes
0100b	Reserved		
0101b			
0110b			
0111b			
1000b	Complex, Not swapped	1 bytes	2 bytes
1001b		2 bytes	4 bytes
1010b		4 bytes	8 bytes
1011b		8 bytes	16 bytes
1100b	Complex, Swapped	1 bytes	2 bytes
1101b		2 bytes	4 bytes
1110b		4 bytes	8 bytes
1111b		8 bytes	16 bytes

(1) Real versus complex merely denotes whether an element has one or two sub-elements.

Sub-Element Size determines the type for purposes of type promotion and vector lane width. For example, 16-bit sub-elements get promoted to 32-bit or 64-bit sub-elements when a stream requests type promotion. The vector lane width matters when the DSP CPU operates in big endian mode, as it always lays out vectors in little endian order.

Total Element Size determines the minimal granularity of the stream. This corresponds to ELEM_BYTES in the conceptual stream definition. In the stream addressing model, it determines the number of bytes the stream

fetches for each iteration of the innermost loop. Streams always read whole elements, either in increasing or decreasing order. Therefore, the dimension 0 of a stream spans ICNT0 × Total Element Size bytes in memory.

Complex Type determines whether the streaming engine treats each element as a complex numbers, and whether to swap the halves of those complex number. Complex types have a total element size that is twice their sub-element size. Otherwise, sub-element size equals total element size.

3.22.4.3.15 Reserved Fields

The stream template and its FLAGS field contain several reserved fields. Programs must set these fields to 0 to ensure future compatibility. The streaming engine checks reserve fields and if non-zero, then issues a bad status back to CPU on initial read.

Note

Many of the reserved fields are placed strategically, to allow other, adjacent fields to grow, should future architectures require it. Future architecture extensions should avoid using these reserved fields for other uses unless it's strictly necessary.

3.22.4.4 Auxiliary Functions: Dataless Streams

In addition to its core data stream support, the streaming engine supports a range of special dataless streams. Programs manage these streams through special mechanisms, and so they do not fit within the standard stream template given above.

The streaming engine supports two types of dataless streams:

- Block-oriented cache maintenance operations
- Block-oriented cache preload

Both stream types move data between various levels of cache and memory, without bringing data to the CPU.

Section 3.22.4.4.1 and Section 3.22.4.4.2 detail the individual dataless stream instructions.

Dataless streams reuse the hardware for stream number 0 to perform their actions. Therefore, stream 0 must be inactive when issuing a dataless stream instruction.

Note

The streaming engine can use both stream number 0 and stream number 1 to perform dataless stream operations, but due to CPU limitations, it can only issue dataless streams to stream number 0.

Starting a dataless stream raises TSR.SE0. The streaming engine asserts to the CPU a "coherence active" signal which CPU can monitor to indicate an active coherence is in progress. The streaming engine deasserts this signal when all coherence commands associated with the dataless streams sent have returned. TSR.SE0 remains high until the program issues a SECLOSE for stream 0. If the program issues a SECLOSE before the dataless stream completes, the streaming engine stops issuing requests to the memory system.

Synchronizing with dataless streams: Programs can synchronize with a dataless stream by reading from it with a reference to *SE0 or *SE0++. The streaming engine begins returning empty data-phases to the CPU when it has issued all of the commands associated with the stream, and those commands have all completed. The streaming engine reports any errors encountered in the fault status associated with these data-phases. See Section 3.22.4.7.

TI recommends the following program sequence to ensure a dataless stream runs to completion:

1. Open a dataless stream. The streaming engine begins issuing requests to the memory system. TSR.SE0 goes to 1 indicating that stream 0 is active.
2. Read a single dataphase from the stream. This stalls the CPU until the streaming engine finishes issuing the commands for the dataless stream, and the memory system completes all of the commands.
3. The streaming engine returns an empty data phase with error status if any. The CPU can check error status bus and fault status register, and decide how to proceed if error received. The CPU can then save and restore the dataless stream after it has corrected the errors.
4. Issue SECLOSE for stream 0. TSR.SE0 goes to 0, indicating TSR.SE0 is inactive.

5. Optionally, programs can issue a SEOPEN without a SECLOSE for a new stream while a dataless stream is active. The CPU would stall any new commands inside the CPU until the dataless stream is done, as indicated by the streaming engine through the “coherence active” signal.

These steps can be spread out in a program, allowing other computation to occur in parallel with the dataless stream. In particular, programs may do significant other work between steps 1 and 2.

3.22.4.4.1 Block Cache Maintenance Operations

Cache maintenance operations (CMOs) move cache lines out of cache levels nearer to the CPU to cache levels that are further from the CPU. This allows the programmer to manually manage when data leaves the CPU caches.

In a cache coherent environment—that is, all data pages are marked shared and all masters are coherent masters—cache maintenance operations are usually a performance optimization. In a non-coherent environment—that is, some pages are not marked shared, or some masters are not fully coherent masters—cache maintenance operations are critical to the correctness of a program.

The streaming engine provides the following block-oriented cache maintenance operations on both shareable and non-shareable memory:

- Data cache clean to point of unification
- Data cache clean-invalidate to point of unification
- Data cache invalidate to point of unification
- Data cache clean to point of coherence
- Data cache clean-invalidate to point of coherence
- Data cache invalidate to point of coherence

Table 3-178 defines the cache maintenance nomenclature.

Table 3-178. Cache Maintenance Nomenclature

Clean	The cache writes back any updates to the line that have not yet been sent to the system.
Invalidate	The cache marks the line invalid, removing it from the cache.
Clean-Invalidate	The cache writes back any updates, and then removes the line from the cache
Point of Unification	The level in the cache hierarchy where instruction and data fetch streams meet.
Point of Coherence	The level in the cache hierarchy where all masters can see the updated data.
Shareable Memory	Memory for which the hardware enforces coherence among all masters accessing it
Non-shareable Memory	Memory for which the hardware does not enforce coherence, unless it is non-cacheable.

3.22.4.4.1.1 Privilege Checks

The streaming engine treats Clean and Clean-Invalidate operations as equivalent to reads. That is, the Clean or Clean-Invalidate command may proceed as long as the address range is readable by the current privilege level.

The streaming engine treats Invalidate operations as equivalent to writes. The Invalidate commands may proceed only if the address range is writable by the current privilege level.

If the streaming engine detects a privilege violation, it signals this fault as described in Section 3.22.4.7.2, and halts the block CMO.

Note

To prevent data leakage between tasks, operating systems should perform a Data Cache Clean to PoC on any newly allocated page to commit the page’s contents before handing the page to a user-level task. Otherwise, the user-level task could use Data Cache Invalidate to expose the page’s previous contents.

3.22.4.4.1.2 Scope

CMOs to the Point of Unification (PoU) are useful for synchronizing the data and instruction caches for the CPU. Typically, instructions caches are not fully coherent with the rest of the cache hierarchy. A block CMO to PoU in conjunction with an instruction cache invalidation ensures the CPU sees updates to instruction memory.

CMOs to the Point of Coherence (PoC) are useful for synchronizing different masters that are not fully coherent. PoC ensures that all masters see the updated data.

3.22.4.4.1.3 Interaction with Memory Types

Not all memory types support cache maintenance operations. If programs issue cache maintenance operations on unsupported memory types, the streaming engine may either discard the CMOs or signal an exception similar to a privilege violation. For example, any CMOs to memory type marked as “System / Device” memory type as returned by the µTLB would be an error and the streaming engine would drop the command from being sent into the memory system, and return an error to CPU on the read status packet.

Note

As of version 0.96 of this spec, all CMOs from the SE are mandatory regardless of memory type. In other words, CMOs always go out to L2 without checking memory type attributes.

3.22.4.4.1.4 MFENCE and CMOs

The streaming engine asserts a separate coherence active signal to the CPU indicating whether any CMO stream commands are still active in the system. It asserts this signal regardless of whether the CMO stream that generated those commands is active, frozen, or inactive.

The CPU’s MFENCE instruction waits for this signal to go low, indicating that all outstanding CMO operations have completed. This may be necessary when closing a block CMO stream early, or during context switches.

3.22.4.4.2 Block Preload Instructions

The streaming engine support four varieties of block preload.

- Preload to L3 Cache for Read
- Preload to L3 Cache for Write
- Preload to L2 Cache for Read
- Preload to L2 Cache for Write

These actions are hints to the memory system, suggesting how the data will be used subsequently. The memory system is free to ignore the hints.

3.22.4.4.2.1 Preload to L3 versus L2

Preload to L3 Cache attempts to bring data to a cache level outside the CorePac module. The goal is to ensure data is on-chip, even if it is not yet in CorePac. This is potentially useful for large data sets that do not fit within the CorePac L2 memory.

Preload to L2 Cache attempts to bring data to the L2 cache within CorePac. This is potentially useful for data sets that do fit within L2 cache.

3.22.4.4.2.2 Preload for Read versus Write

Preload for Read indicates that the CPU intends to read the data, but not necessarily write it. The streaming engine requests shared access to the data, allowing other caches to retain copies of the data if they already have copies.

Preload for Write indicates that the CPU intends to write to the preloaded block. The streaming engine requests exclusive access to the data, with the intent of removing copies from other caches. This reduces the cost of subsequent CPU writes to those lines.

3.22.4.4.2.3 Privilege Checks

The streaming engine treats all preload requests as reads, including Preload for Write. If a privilege check fails, the streaming engine silently ignores it. It drops the faulted preload command and continues with the block preload operation.

3.22.4.4.2.4 Interaction with Memory Types

The streaming engine only sends preload requests for normal, cacheable memory. It silently drops preload requests for other memory types.

3.22.4.5 Streaming Engine Instruction Set

The DSP CPU exposes the streaming engine to programs through a small number of instructions and specialized registers. This section defines the stream management instructions. The next section defines the stream reference registers, and the relationship of streams to predication registers.

Dedicated stream control instructions define the start and end of streams and allow saving/restoring the context of active streams. Additional instructions invoke the streaming engine's auxiliary functions, as described previously in Section 3.22.4.4-.

The DSP CPU provides the following groups of instructions to manage the streaming engine. Within each group, the CPU provides one or more distinct instructions.

Table 3-179. Stream Instruction Groups

Instruction	Description
SEOPEN	Opens a new stream on one of the two streaming engines, replacing any previous stream on that streaming engine
SECLOSE	Closes a stream on one or both of the streaming engines
SEBRK	Breaks (early exits) from one or more levels of stream loop nest
SESAVE	Captures sufficient state for a stream to restart it in the future
SERSTR	Restores a previously saved stream
BLKCMO	Block cache maintenance operation
BLKPLD	Block preload

This section defines the overall behavior of these instructions. Refer to the CPU ISA chapters for precise definitions of each instruction's encoding and pipeline behavior.

3.22.4.5.1 Opening and Closing Streams: SEOPEN / SECLOSE

Programs start and end streams with SEOPEN and SECLOSE. SEOPEN opens a new stream. The stream remains open until terminated explicitly by SECLOSE or replaced by a new stream with SEOPEN.

Note

Opening a new SEOPEN on an already "active stream" without first closing the current "active stream" is not allowed and results in an error detected in the CPU. Opening a new SEOPEN can only be executed when a stream is not active, TSR.SEn = 0, or the stream is in a "frozen" state.

Note

As of version 0.85 of this spec, multiple back to back SEOPENs can be executed on an already "active stream" without first closing the stream.

3.22.4.5.1.1 SEOPEN

SEOPEN opens a new stream, potentially replacing an already active stream. The canonical form of SEOPEN takes the following arguments:

```
SEOPEN              address_reg, stream_number, stream_template_reg
```

Table 3-180. SEOPEN Arguments

Argument	Description
address_reg	Scalar 64-bit register containing the stream's starting address
stream_number	Which stream number to open (0 or 1)
stream_template_reg	Vector register containing the stream definition

SEOPEN replaces any stream currently executing on that stream number within the streaming engine, as if the programmer issued a SECLOSE first on that stream. SEOPEN on one stream has no effect on the other stream.

When opening a new stream over an existing stream, the streaming engine discards all data from the previous stream. This provides a seamless transition to the new stream, while adhering to the memory ordering guarantees laid out in Section 3.22.3.1.1.

The DSP CPU and streaming engine support a small set of short-cut instructions for common stream types, as shown in Table 3-181. These stream short-cuts do not require a separate stream template vector register. Each short-cut stream sets ICNT0 to 0xFFFFFFFF, and all other dimensions to 0.

```
SEOPENxxx           stream_number, address_reg
```

Table 3-181. Stream Open Short-Cut Instructions

Short-cut Instruction	Element Size	Promotion
SEOPENB	1 byte	None
SEOPENBH	1 byte	Sign extend 2×
SEOPENBUH	1 byte	Zero extend 2×
SEOPENH	2 bytes	None
SEOPENHW	2 bytes	Sign extend 2×
SEOPENHUW	2 bytes	Zero extend 2×
SEOPENW	4 bytes	None
SEOPENWD	4 bytes	Sign extend 2×
SEOPENWUD	4 bytes	Zero extend 2×
SEOPEND	8 bytes	None

Concretely, each of the above instructions sends one of the templates listed in Table 3-182 to open the stream.

Table 3-182. Explicit Template Encodings for Stream Short Cut Instructions

Instruction	Bits [511:448](1)	Bits[447:32](2)	Bits[31:0](3)
SEOPENB	0x00000000_00006000	All zeros	0xFFFFFFFF
SEOPENBH	0x00000000_00005500	All zeros	0xFFFFFFFF
SEOPENBUH	0x00000000_00005100	All zeros	0xFFFFFFFF
SEOPENH	0x00000000_00005001	All zeros	0xFFFFFFFF
SEOPENHW	0x00000000_00004501	All zeros	0xFFFFFFFF
SEOPENHUW	0x00000000_00004101	All zeros	0xFFFFFFFF
SEOPENW	0x00000000_00004002	All zeros	0xFFFFFFFF
SEOPENWD	0x00000000_00003502	All zeros	0xFFFFFFFF
SEOPENWUD	0x00000000_00003102	All zeros	0xFFFFFFFF
SEOPEND	0x00000000_00003003	All zeros	0xFFFFFFFF

(1) Corresponds to FLAGS[63:0] - setting up a 1-dimensional stream.
(2) Corresponds to DIM5 down to DIM1, and ICNT5 down to ICNT1 (where non-active ICNT's are not used).
(3) Corresponds to ICNT0.

3.22.4.5.1.2 SECLOSE

SECLOSE explicitly marks a stream inactive, flushing any outstanding activity. Any further references to the stream trigger exceptions. SECLOSE also allows a program to prematurely terminate one or both streams.

```
SECLOSE                 stream_number
```

Table 3-183. SECLOSE Arguments

Argument	Description
stream_number	Which stream number to close (0 or 1)

SECLOSE marks the stream inactive, flushing any buffered contents and zeroes out stored addresses, counters, and tags. It stops sending new requests into the system. SECLOSE ends both normal streams and dataless streams immediately.

SECLOSE does not stall the DSP CPU, and does not wait for any stream to finish sending requests. To synchronize the DSP CPU with a no-data stream, programs should issue an appropriate MFENCE operation.

The CPU and streaming engine permit SECLOSE on an inactive or frozen stream. This forces the streaming engine into the inactive state if it is currently frozen. This allows cleanup code to blindly issue SECLOSE for both streams to put the CPU in a known state.

3.22.4.5.2 Exiting Early with Stream Breaks: SEBRK

Stream breaks allow exiting early from 1 to 6 levels of loop nest within a stream. SEBRK takes the arguments listed in Table 3-184.

```
SEBRK                level_to_break_from, stream_number
```

Table 3-184. SEBRK Arguments

Argument	Description
stream_number	Which stream to trigger a break on (0 or 1)
levels_to_break	Level of loop to break out of (0, 1, 2, 3, 4, or 5)

Issuing a SEBRK causes the stream engine to skip all remaining elements for the corresponding loop levels.

The exact behavior depends on which loop levels are currently active. Refer to Table 3-170 and Table 3-169.

3.22.4.5.3 Saving and Restoring Streams: SESAVE / SERSTR

The SESAVE and SERSTR instructions give software the ability to context switch a stream. At a high level, SESAVE captures enough stream state from the streaming engine to allow restoring the stream to its current position in the future. Conversely, SERSTR copies the saved stream state back into the streaming engine, so that it can resume where it left off.

Programs can also use SESAVE to inspect the state of a stream that triggered an exception. This provides a clean mechanism for servicing page faults.

```
SESAVE               segment_number, stream_number, stream_save_register
    SERSTR               segment_number, stream_number, stream_save_register
```

Table 3-185. SESAVE / SERSTR Arguments

Argument	Description
segment_number	Which state segment to operate on (0, 1, 2, or 3)
stream_number	Which stream to operate on (0 or 1)
stream_save_reg	Vector to save stream state to or restore stream state from

SESAVE copies the current state of a frozen or inactive stream into the CPU registers. This includes whether the stream was frozen or inactive, all of the stream parameters, and all of the current loop counts and pointers associated with the stream. SERSTR copies this state back into a stream. They provide a context switch mechanism for the streaming engine.

WARNING

The stream engine reuses and captures each SERSTR segment into the same ‘Segment 0’ hardware vector register each time a SERSTR command is issued, before moving them to internal registers where they belong. Thus, because the ‘Segment 0’ register is being updated and overridden after each SERSTR with data from Segment 1, then 2, and then 3, it is important for software to make sure that the last SERSTR command is issued to ‘Segment 0’, so that it is the last vector data captured in register. In other words, the ordering for a SERSTR command once any SERSTR has started, should be to restore Segment 3, 2, 1 in any order, and then the last SERSTR command should be to restore Segment 0. This MUST be done once any SERSTR command(s) have started on any other Segment, in order to restore Segment 0. There is, however, no ordering requirements for SESAVE.

Note

When the stream engine is in the “frozen” state, it can restart from its previous position in the stream before it was frozen, when it moves to the “active” state (TSR.SEn = 1), even in the absence of an explicit SESAVE and SERSTR. In this scenario, hardware simply copies all the current states and counts and restarts the stream from where the last CPU fetch left of.

The streaming engine architecture defines the save/restore record as 4 vector-sized segments. The architecture does not, however, define the exact format or contents of these vectors. The exact contents of the save/restore record may vary from generation to generation.

The architecture ensures that saving the save/restore record saves sufficient context to allow restoring the stream engine state at some arbitrary point in the future. This enables an operating system to context-switch the streaming engine between multiple tasks.

Figure 3-136 provides the layout for the current generation. This definition may change in future architecture revisions. Programs that wish to operate across device generations—such as programs that do not wish to identify individual versions of the streaming engine by its SEn_PID—should save and restore all four segments, despite the fact the present generation does not use the fourth segment.

As mentioned in the WARNING above, it is important that the last SERSTR command should always be issued to Segment 0, when any attempts at restoring Segment 3, 2, or 1 have been issued. Restoring Segments 3, 2, or 1 can be done in any order, but Segment 0 should always be the last vector to be restored when any SERSTR commands have started to any other Segment.

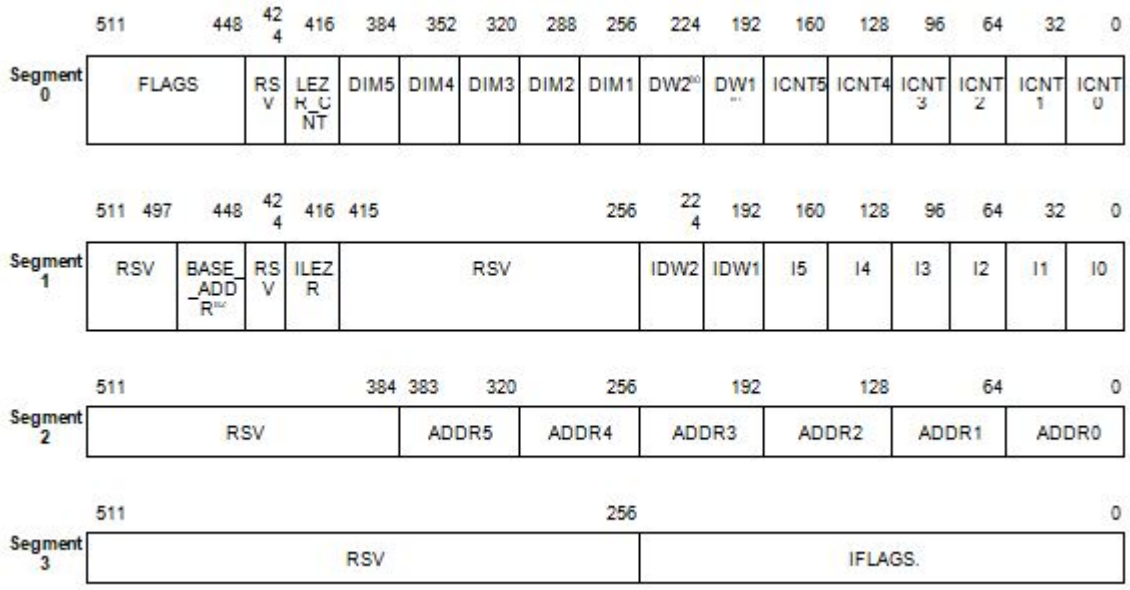


Figure 3-136. Stream Save and Restore Format

Table 3-186 describes all the save/restore fields in the current generation. The first segment shares its definition with the stream template, whereas the streaming engine's internal state and microarchitectural details define the second and third segment.

Table 3-186. Stream Save/Restore Field Definitions

Field Name	Description
Segment 0	Saved stream template for this stream(1)
I0	Remaining iterations for loop level 0 (innermost)
I1	Remaining iterations for loop level 1
I2	Remaining iterations for loop level 2
I3	Remaining iterations for loop level 3
I4	Remaining iterations for loop level 4
I5	Remaining iterations for loop level 5 (outermost)
IDW1	Remaining iterations for DECDIM1_WIDTH
IDW2	Remaining iterations for DECDIM2_WIDTH
ILEZR	Remaining iterations for LEZR_CNT
BASE_ADDR	Starting 49-bit Virtual Base Address captured during SEOPEN
ADDR0	Saved address for loop level 0(2)
ADDR1	Saved address for loop level 1(2)
ADDR2	Saved address for loop level 2(2)
ADDR3	Saved address for loop level 3(2)
ADDR4	Saved address for loop level 4(2)
ADDR5	Saved address for loop level 5(2)
IFLAGS	Internal stream engine flags and state(3)

(1) For dataless streams, this takes on an internal format. For short-cut streams, it should resemble the reference short-cut templates in Table 3-182.

(2) To be consistent with the CPU's address handling, ADDR0 through ADDR5 are sign-extended to 64 bits. That is, bits [63:49] are the same as bit [48] of the address. Internally, the steaming engine only uses address bits [48:0]. Also see Table 3-196, for bit [48] address rollover checks.

(3) IFLAGS are internal stream engine register states which are returned to the CPU during a SESAVE. Upon return from a context switch, the CPU should restore, through SERSTR, the IFLAGS as they are with no modifications. This ensures the stream engine can switch to its original state and continue the stream after coming back from any context switch.

3.22.4.5.4 Block Cache Maintenance Operation: BLKCMO

The DSP CPU architecture defines a set of block cache maintenance instructions that operate on a contiguous block of addresses. Block cache maintenance operations take the following general form:

```
BLKCMO                base_address_reg, optype, byte_count_reg
```

Table 3-187. BLKCMO Instruction Arguments

Argument	Description		
base_address_reg	Starting virtual address for the block coherence operation (64 bits)		
optype	Block cache maintenance operation type. Valid operation types include:		
	DCCU	S	Data Cache Clean to Point of Unification, Shareable
	DCCIU	S	Data Cache Clean and Invalidate to Point of Unification, Shareable
	DCIU	S	Data Cache Invalidate to Point of Unification, Shareable
	DCCC	S	Data Cache Clean to Point of Coherence, Shareable
	DCCIC	S	Data Cache Clean and Invalidate to Point of Coherence, Shareable
	DCIC	S	Data Cache Invalidate to Point of Coherence, Shareable
	DCCUN	S	Data Cache Clean to Point of Unification, Non-Shareable
	DCCIU	NS	Data Cache Clean and Invalidate to Point of Unification, Non-Shareable
	DCIU	NS	Data Cache Invalidate to Point of Unification, Non-Shareable
	DCCC	NS	Data Cache Clean to Point of Coherence, Non-Shareable
	DCCIC	NS	Data Cache Clean and Invalidate to Point of Coherence, Non-Shareable
	DCIC	NS	Data Cache Invalidate to Point of Coherence, Non-Shareable
byte_count_reg	Total number of bytes to operate on (32 bits)		

Block cache maintenance operations move lines out of one or more caches, leaving the lines either clean or invalid.

Note

As of version 0.96 of this spec, only DCCICS and DCICS are supported by CPU. The SE fully supports all operations, however.

3.22.4.5.5 Block Cache Preload: BLKPLD

The DSP CPU architecture defines a set of block cache preload instructions that operate on a contiguous block of addresses. Block cache preload operations take the following general form:

```
BLKPLD                base_address_reg, optype, byte_count_reg
```

Table 3-188. BLKPLD Instruction Arguments

Argument	Description	
base_address_reg	Starting virtual address for the block coherence operation (64 bits)	
optype	Block cache preload type. Valid operation types include:	
	L2R	Preload to L2 for reading
	L2W	Preload to L2 for writing
	L3R	Preload to L3 for reading
	L3W	Preload to L3 for writing
byte_count_reg	Total number of bytes to operate on (32 bits)	

Preload instructions serve as a hint to the caches. Caches may ignore this.

3.22.4.6 Streaming Engine Register Interfaces

Programs communicate with the streaming engine through three register-oriented interfaces:

- Stream Vector Data registers. These registers convey data from the streams to the program.
- Vector Predicate registers. These registers indicate which lanes contain valid data and which lanes do not.

Note

The Vector Predicate registers inside the CPU are not supported in this version of the device. The stream engine, however, supports full vector predicates, but they are tied off to zero until the CPU supports the feature.

- Extended Control registers (ECRs). ECRs provide a low-bandwidth mechanism to examine stream engine internal state and status.

The following sections define each of these interfaces.

3.22.4.6.1 Stream Vector Data Registers: SE0 and SE1

The streaming engine conveys the data for both streams through two special 512-bit vector registers, SE0 and SE1.

Programs access streams through these special registers. To access the contents of the stream, programs substitute one of these special register names in place of a CPU register.

Table 3-189 defines these registers.

Table 3-189. Stream Vector Data Registers

Stream Number	Stream Vector Register Name	Contents
0	SE0	Data at the head of stream 0
1	SE1	Data at the head of stream 1

Any number of instructions can include a vector reference in a given execute packet. The CPU places no restrictions on the number of stream vector references that occur in parallel.

The streaming engine vector data registers may be used in place of vector registers as input to nearly any instruction, with the following restrictions:

- It may not be used as a destination argument for an instruction, including combined source/destination arguments.
- It may not be used as a source argument for a store instruction.
- It may not be used as an argument for any instruction on the A-side of the machine.

Other than those restrictions, the SE0 and SE1 registers behave as any other vector register.

SE0 and SE1 may even be used with scalar instructions on the B-side. Scalar instructions see the lowest 64 bits of the stream data.

Note

Certain DSP CPU instructions in the VFIR and VMATMPY family of instructions place additional restrictions on SE0 and SE1. See the description of these instructions in the CPU instruction set for further details.

3.22.4.6.1.1 Advancing the Stream

The special registers listed above always refer to the vector at the top of the stream. Programs can re-read the same top of stream data as many times as necessary. By default, reading these special registers does not advance the stream.

To advance a stream, the streaming engine provides an alternate stream reference encoding that tells the streaming engine to advance to the next 512-bit bundle. The assembly syntax represents stream advance with a post-increment suffix, as shown in Table 3-190.

Table 3-190. Stream Vector Data Registers with Stream Advance

Syntax	Meaning
SE0++	Returns the same data as SE0, and then advances stream to the next 512 bit bundle..
SE1++	Returns the same data as SE1, and then advances stream to the next 512 bit bundle..

Stream advances always advance by 512 bits, regardless of which register name the program uses to signal the advance.

At most one stream reference per execute packet can request a stream advance on a given stream. If multiple parallel instructions attempt to advance the same stream in the same cycle, the stream only advances by one vector. Stream references to different streams in the same execute packet can advance those streams in parallel.

3.22.4.6.2 Stream Vector Predicates

Note

The Vector Predicate registers inside the CPU are not supported in this version of the silicon. The stream engine, however, supports full vector predicates but they are tied off to zero until the CPU supports the feature.

As described in Section 3.22.3.2.3, the streaming engine marks each vector lane as valid or invalid, depending on whether that lane contains a valid element from the current stream.

The CPU provides a set of 8 vector predicate registers, P0 through P7. Each vector predicate register is 64 bits wide, and each bit corresponds to a single byte within a vector. The CPU also allows pairing vector predicates when applying vector predicates to a double vector.

The CPU's vector predicates provide a mechanism for ignoring portions of a vector in certain operations, such as vector stores. This naturally fits with the notion of valid and invalid lanes maintained by the streaming engine.

Therefore, the streaming engine converts the lane valid/invalid status it maintains internally to byte valid/invalid status to present to the CPU. A zero bit indicates a byte in an invalid lane, and a one bit indicates a byte in a valid lane. The streaming engine generates its vector predicate value in parallel with formatting data for the CPU. When the CPU reads data from the streaming engine, it copies the vector predicate values into one or more vector predicate registers.

The streaming engine defines a fixed 1:1 mapping between the bytes it presents through SEn and the vector predicate values it generates. In general, byte N within a vector register maps directly to bit N within the corresponding predicate register.

Table 3-191 defines the mapping between stream vector data registers and the CPU's vector predicate registers. References to SE0 or SE1 copy the predicate bits associated with that data to the corresponding vector predicate register. Vector predicates update at the end of the cycle that holds the stream reference that triggers the update.

Table 3-191. Mapping Stream Reference to Vector Predicates

Stream Reference	Corresponding Vector Predicate
SE0	P0
SE1	P1

The following examples illustrate the vector predicates in action. The examples are written for clarity, not speed.

In the following example, the copy loop uses VMV to read data from the streaming engine. This step also copies the corresponding vector predicates to P0. The subsequent VSTP64B instruction stores the data to memory; however, the vector predicate bits in P0 prevent VSTP64B from writing bytes corresponding to invalid lanes in the data from the stream.

Streaming engine based memory copy loop:

```
copy_loop:          VMV          SE0++,       VB0              ; Data to VB0, predicate to P0
          VSTP64B           P0,       VB0,       *D0++       ; store data, controlled by pred.
          BDEC          copy_loop
```

3.22.4.6.3 Extended Control Register (ECR) Functional Interface

The streaming engine provides a small number of Extended Control registers (ECRs) for functional program access. These ECRs exist primarily for error recording and reporting purposes.

The streaming engine provides the following ECRs. The ECR offsets are relative to the streaming engine's address range in the ECR address map.

Table 3-192. Streaming Engine ECRs

Offset	Name	Indexed	Permissions	Privilege Levels	Description	Reference
0x00	SE0_PID	No	Read Only	All	Streaming Engine Peripheral ID	Section 3.22.4.6.3.1
0x01	SE0_FAR	No	Read/Write	Supv, Debug	Stream 0 fault virtual address	Section 3.22.4.6.3.2
0x02	SE0_FSR	No	Read/Write	Supv, Debug	Stream 0 fault details	
0x03 - 0x07	Reserved	No	No Access	None		
0x08	SE0_TAG	Yes	Read Only	Debug	Stream 0 tag metadata	'Refer to Section 3.22.4.7 for further details on how the streaming engine reports various types of fault.Section 3.22.4.6.3.3
0x09	SE0_ICNT	Yes	Read Only	Debug	Stream 0 iteration count registers	Section 3.22.4.6.3.4
0x0A	SE0_DIM	Yes	Read Only	Debug	Stream 0 dimension registers	Section 3.22.4.6.3.5
0x0B	SE0_ADDR	Yes	Read Only	Debug	Stream 0 intermediate addresses	Section 3.22.4.6.3.6
0x0C	SE0_STAT	No	Read Only	Debug	Stream 0 status and flags	"Section 3.22.4.6.3.7
0x0D - 0x0F	Reserved	No	No Access	None		
0x10	SE1_PID	No	Read Only	All	Streaming Engine Peripheral ID	Section 3.22.4.6.3.1
0x11	SE1_FAR	No	Read/Write	Supv, Debug	Stream 1 fault virtual address	Section 3.22.4.6.3.2
0x12	SE1_FSR	No	Read/Write	Supv, Debug	Stream 1 fault details	
0x13 - 0x17	Reserved	No	No Access	None		
0x18	SE1_TAG	Yes	Read Only	Debug	Stream 1 tag metadata	'Refer to Section 3.22.4.7 for further details on how the streaming engine reports various types of fault.Section 3.22.4.6.3.3
0x19	SE1_ICNT	Yes	Read Only	Debug	Stream 1 iteration count registers	Section 3.22.4.6.3.4
0x1A	SE1_DIM	Yes	Read Only	Debug	Stream 1 dimension registers	Section 3.22.4.6.3.5
0x1B	SE1_ADDR	Yes	Read Only	Debug	Stream 1 intermediate addresses	Section 3.22.4.6.3.6

Table 3-192. Streaming Engine ECRs (continued)

Offset	Name	Indexed	Permissions	Privilege Levels	Description	Reference
0x1C	SE1_STAT	No	Read Only	Debug	Stream 1 status and flags	Section 3.22.4.6.3.7
0x1D - 0x1F	Reserved	No	No Access	None		
0x20	SE2_PID	No	Read Only	All	Reserved for Stream 2; Reads as 0	Section 3.22.4.6.3.1
0x21 - 0x2F	Reserved	No	No Access	None		
0x30	SE3_PID	No	Read Only	All	Reserved for Stream 3; Reads as 0.	Section 3.22.4.6.3.1
0x31 - 0x3F	Reserved	No	No Access	None		

The ECRs fall into two categories: scalar and indexed. Scalar registers hold a single value that is up to 64 bits in size. Indexed ECRs hold multiple values that are selected by a separate 16-bit control register index. The indexing facility simplifies mapping large portions of state into the ECR space.

The following sections define each of the registers summarized in Table 3-192.

3.22.4.6.3.1 SEn_PID: Peripheral Identification

The SEn_PID registers indicate the presence of the streaming engine, as well as the version of streaming engine that is present. This specification defines four SEn_PID registers, allowing for up to four hardware streams. The current DSP CPU only supports two streams. The SEn_PID registers are scalar, not indexed.

The SEn_PID register for streams 0 and 1 returns the following.

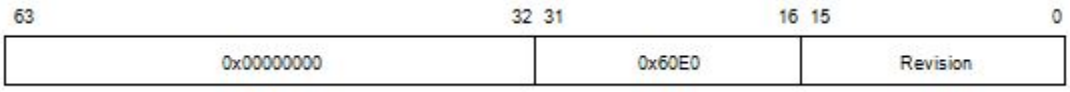


Figure 3-137. SEn_PID Register Layout

The exact revision information in the PID depends on the specific instance of the streaming engine.

SE2_PID and SE3_PID read as 0, indicating the absence of hardware streams 2 and 3.

3.22.4.6.3.2 SEn_FAR, SEn_FSR: Fault Reporting Registers

The SEn_FAR and SEn_FSR record the address and details of faults encountered by the streaming engine. The streaming engine attempts to record the earliest fault encountered on a stream. The SEn_FAR and SEn_FSR registers are both scalar, not indexed.

The SEn_FAR register captures the virtual address associated with the fault, if it is feasible to do so. In the event the streaming engine cannot capture the virtual address associated with a fault, it fills this register with 0s. The SEn_FAR register is laid out as follows. The lower 7 bits of the SEn_FAR are read as zeros because the SE tag stores addresses on a 128-byte line size.

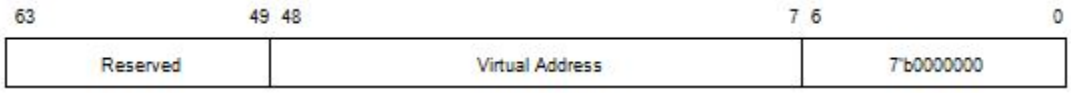


Figure 3-138. SEn_FAR Layout

The SEn_FSR register captures the fault details. The SEn_FSR register is laid out as shown in Figure 3-139.

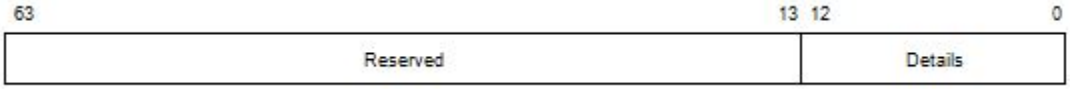


Figure 3-139. SEn_FSR Layout

Typically, upon encountering a fault, the streaming engine records the details in SEn_FAR and SEn_FSR, and passes the fault condition on to the CPU. If the CPU takes an interrupt or exception in response to the fault, it can read the details from SEn_FAR and SEn_FSR. If the CPU does not take an interrupt or exception, which could happen if the CPU closes the stream before seeing the error, then the update to SEn_FAR and SEn_FSR will likely go unnoticed.

The values in SEn_FAR and SEn_FSR remain unchanged if no software opens a stream on the given streaming engine, and no software writes to SEn_FAR or SEn_FSR. When software opens a stream on a given streaming engine, the streaming engine is allowed to overwrite SEn_FAR and SEn_FSR, even if the error condition it records does not result in a CPU interrupt or exception. Furthermore, if the given stream is closed, both SEn_FAR and SEn_FSR would be cleared. Refer to Section 3.22.4.7 for further details on how the streaming engine reports various types of faults.

3.22.4.6.3.3 SEn_TAG: Stream Tag Data and Metadata

The SEn_TAG registers hold both virtual address tag information and additional metadata for elements stored in the streaming engine's storage array. SEn_TAG aids debugging programs that use the streaming engine by showing what addresses the streaming engine currently holds.

SEn_TAG is only readable by the debugger. If software tries to read this register from any privilege level, it receives a privilege exception.

Each SEn_TAG entry is structured as shown in Figure 3-140.

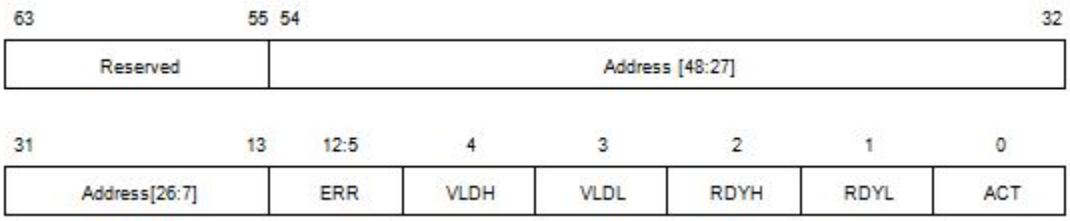


Figure 3-140. SEn_TAG: Stream Engine Tag

SEn_TAG is an indexed control register. The control register index determines which tag gets read by the CPU. Indices in the range 0x00 to 0x0F read tags 0 through 15. Indices above 0x0F return 0.

SEn_TAG values are only meaningful during an active stream. SECLOSE clears the address information in the tags on closing a stream. SERSTR has no effect on the SEn_TAG contents.

The SEn_TAG values may change dramatically while single-stepping code. The CPU may flush the streaming engine's pipeline between steps, which may cause the streaming engine to refetch all of its contents. Thus, the association of address to tag position may change from step-to-step. The SEn_TAG information therefore serves primarily as a hint to the programmer.

The exact format of the SEn_TAG field and the number of SEn_TAG registers is tied directly to the underlying implementation of the streaming engine. Future versions of the streaming engine may have a larger or smaller number of tags. Use SEn_PID to determine the streaming engine version. Refer to Section 3.22.4.6.3.1 for more details on SEn_PID.

3.22.4.6.3.4 SEn_ICNT: Iteration Count Registers

The SEn_ICNT registers hold the internal iteration counts for the streaming engine address generator. The purpose of SEn_ICNT is to allow the programmer to determine where the streaming engine is within a given stream while debugging a program.

Internally, the streaming engine keeps two copies of these counts: an early copy which indicates where the streaming engine is currently fetching, and a late copy, which indicates the last iteration the CPU has committed. In most cases, the late copy gives the clearest indication of the current state of the program. The early copy indicates how far the streaming engine has fetched ahead of the CPU.

SEn_ICNT is only readable by the debugger. If software tries to read this register from any privilege level, it receives a privilege exception.

SEn_ICNT is an indexed control register. The control register index determines which count to access, as per Table 3-193. Each register is 32 bits, with exception of LEZR count, which is 8 bits. Bits 63:32 (for LEZR, 63:8) read as 0.

Table 3-193. SEn_ICNT Index Map

Index	Description	Notes
0x00	Early I0	Early Remaining Iteration Count for Loop Level 0 (innermost loop)
0x01	Early I1	Early Remaining Iteration Count for Loop Level 1
0x02	Early I2	Early Remaining Iteration Count for Loop Level 2
0x03	Early I3	Early Remaining Iteration Count for Loop Level 3
0x04	Early I4	Early Remaining Iteration Count for Loop Level 4
0x05	Early I5	Early Remaining Iteration Count for Loop Level 5 (outermost loop)
0x06	Early DW1	Early Remaining Iteration Count for DECDIM1_WIDTH
0x07	Early DW2	Early Remaining Iteration Count for DECDIM2_WIDTH
0x08	Early LEZR	Early Remaining Iteration Count for LEZR_CNT
0x09	Reserved	Reads as zero
0x0A	Late I0	Late Remaining Iteration Count for Loop Level 0 (innermost loop)
0x0B	Late I1	Late Remaining Iteration Count for Loop Level 1
0x0C	Late I2	Late Remaining Iteration Count for Loop Level 2
0x0D	Late I3	Late Remaining Iteration Count for Loop Level 3
0x0E	Late I4	Late Remaining Iteration Count for Loop Level 4
0x0F	Late I5	Late Remaining Iteration Count for Loop Level 5 (outermost loop)
0x10	Late DW1	Late Remaining Iteration Count for DECDIM1_WIDTH
0x11	Late DW2	Late Remaining Iteration Count for DECDIM2_WIDTH
0x12	Late LEZR	Late Remaining Iteration Count for LEZR_CNT
0x13	Reserved	Reads as zero
0x14	ICNT0	From template[31:0]
0x15	ICNT1	From template[63:32]
0x16	ICNT2	From template[95:64]
0x17	ICNT3	From template[127:96]
0x18	ICNT4	From template[159:128]
0x19	ICNT5	From template[191:160]
0x1A	DECDIM1_WIDTH	From template[223:192]
0x1B	DECDIM2_WIDTH	From template[255:224]
0x1C	LEZR_CNT	From template[423:416]
0x1D - 0xFFFF	Reserved	Reads as zero

The exact interpretation of each counter depends on the DIMFMT for the currently active stream. DIMFMT is visible in the SEn_STAT register. Refer to Section 3.22.4.3 for the definition of DIMFMT. The current value of DIMFMT is visible in SEn_STAT. Refer to Section 3.22.4.6.3.7 for more details on SEn_STAT.

The I0, I1, I2, I3, I4, and I5 registers indicate the current loop iteration. On this generation stream engine, these counters actually count down to 0, rather than counting up. The ICNT0, ICNT1, ICNT2, ICNT3, ICNT4, and ICNT5 registers provide the reload values for these counters. ICNT0, ICNT1, ICNT2, ICNT3, ICNT4, and ICNT5 come directly from the streaming engine template for the active stream. In DECDIM mode, the Late DECDIM_WIDTH indicates the current loop count, and the template DECDIM_WIDTH provide the reload value. ICNT0 represents the innermost loop for the tile width in DECDIM mode.

The layout and interpretation of SEn_ICNT may change with future revisions of the streaming engine. Use SEn_PID to determine the streaming engine version. Refer to Section 3.22.4.6.3.1 for more details on SEn_PID.

3.22.4.6.3.5 SEn_DIM: Stream Dimension Registers

The SEn_DIM register reports the dimensions associated with the currently active stream. The purpose of SEn_DIM is to allow the programmer to see what dimensions were provided when the program opened the stream, to aid debugging the program.

SEn_DIM is only readable by the debugger. If software tries to read this register from any privilege level, it receives a privilege exception.

SEn_DIM is an indexed control register. The control register index determines which count to access, as per Table 3-194. Each register is 32 bits. Bits 63:32 read as 0.

Table 3-194. SEn_DIM Index Map

Index	Description	Notes
0x00	DIM1	From template[287:256]
0x01	DIM2	From template[319:288]
0x02	DIM3	From template[351:320]
0x03	DIM4	From template[383:352]
0x04	DIM5	From template[415:384]
0x05 - 0xFFFF	Reserved	Reads as zero

The DIMFMT of the currently active stream determines how fields in DIM1 through DIM5 map to dimensions of the stream. Refer to Section 3.22.4.3 for the definition of DIMFMT. The current value of DIMFMT is visible in SEn_STAT. Refer to Section 3.22.4.6.3.7 for more details on SEn_STAT.

The layout and interpretation of SEn_DIM may change with future revisions of the streaming engine. Use SEn_PID to determine the streaming engine version. Refer to Section 3.22.4.6.3.1 for more details on SEn_PID.

3.22.4.6.3.6 SEn_ADDR: Stream Address Registers

The SEn_ADDR registers hold the current saved address for each dimension of the currently active stream. The purpose of SEn_ADDR is to allow the programmer to see the span of addresses the streaming engine is fetching from while debugging a program.

Internally, the streaming engine keeps two copies of these addresses: an early copy which indicates where the streaming engine is currently fetching, and a late copy, which indicates the last iteration the CPU has committed. In most cases, the late copy gives the clearest indication of the current state of the program. The early copy indicates how far the streaming engine has fetched ahead of the CPU.

SEn_ADDR is only readable by the debugger. If software tries to read this register from any privilege level, it receives a privilege exception.

SEn_ADDR is an indexed control register. The control register index determines which count to access, as per Table 3-195.

Table 3-195. SEn_ADDR Index Map

Index	Description	Notes
0x00	Early ADDR0	Address for hardware dimension 0.
0x01	Early ADDR1	Address for hardware dimension 1.
0x02	Early ADDR2	Address for hardware dimension 2.
0x03	Early ADDR3	Address for hardware dimension 3.
0x04	Early ADDR4	Address for hardware dimension 4.
0x05	Early ADDR5	Address for hardware dimension 5.
0x06 - 0x07	Reserved	Reads as zero
0x08	Late ADDR0	Address for hardware dimension 0.
0x09	Late ADDR1	Address for hardware dimension 1.
0x0A	Late ADDR2	Address for hardware dimension 2.
0x0B	Late ADDR3	Address for hardware dimension 3.
0x0C	Late ADDR4	Address for hardware dimension 4.

Table 3-195. SEn_ADDR Index Map (continued)

Index	Description	Notes
0x0D	Late ADDR5	Address for hardware dimension 5.
0x0E - 0xFFFF	Reserved	Reads as zero

Each of the addresses is 49 bits. For consistency with the CPU's address handling, the streaming engine sign-extends these addresses to 64 bits. That is, bits 63:49 show the same value as bit 48.

The layout and interpretation of SEn_ADDR may change with future revisions of the streaming engine. Use SEn_PID to determine the streaming engine version. Refer to Section 3.22.4.6.3.1 for more details on SEn_PID.

3.22.4.6.3.7 SEn_STAT: Streaming Engine Status and Flags Register

Currently, the SEn_STAT register bits 63:0 are a direct copy of the FLAGS field from the template, and only visible to debug. The format may change with different versions of the SE.

3.22.4.7 Fault Handling and Reporting

The streaming engine reports a range of faults:

- Programming errors
- Memory translation faults
- Bit errors detected in the storage array
- Bus errors reported by the system
- Functional failures

Faults fall into two broad categories: synchronous faults and asynchronous errors. Synchronous errors align to program execution, while asynchronous errors do not. In this generation, the streaming engine only reports errors synchronously.

Synchronous faults: the streaming engine reports faults synchronously, by marking data from the streaming engine with a flag indicating a fault. This flagged data triggers an internal CPU exception event if and only if the program attempts to consume that data.

Synchronous in this context means synchronous with the instruction stream: The instruction that attempts to consume the data marked as errant is the instruction the CPU generates an internal exception event for. Thus, the relationship between errors recognized by the streaming engine and the instructions that experience an exception does not depend on factors such as stall timing. Internally, when the streaming engine detects a fault to report, it marks the affected data in its internal storage. In parallel, the streaming engine records the details of the error in SEn_FSR and SEn_FAR as appropriate. It may also disable further request generation for the stream.

Note

If the streaming engine detects multiple errors, it attempts to record the earliest error with respect to the stream's logical request order in SEn_FSR / SEn_FAR. There may be corner cases where the earliest fault is ambiguous or not easily determined. In those cases, the streaming engine reports one of the faults that occurred, whether or not it was actually earliest.

The fault status flows with the affected data through the streaming engine's pipeline and through the CPU's pipeline. The CPU triggers an internal exception event if and when the program actually consumes the data.

Due to this pipelined nature, many cycles may pass between when the streaming engine detects a fault, and when the CPU takes an exception. If the program never consumes the affected data, the CPU never generates an exception event.

This behavior treats streaming engine fetches as speculative until the program consumes the data. Consider the following example pseudo-code.

Streaming engine reads as speculative reads with respect to faults:

```
#    pragma              DATA_ALIGN(4096)
    uint8_t             mydata[16000];
uint32_t            process_mydata( void )
{
    uint32_t result = 0;
    SEOPENB( 0, mydata );                              // Open a byte stream to mydata. See 1.3.5.5.1
    for ( int i = 0; i < 16000 ; i += 64 )
        result = compute( result, SE0ADV );                                    // Fold data from
SE0 into result
    return result;
}
```

This example opens a short-cut stream of bytes reading the array mydata[]. The loop itself only reads the contents of mydata[]. The streaming engine, however, reads beyond the end of mydata[].

Suppose that none of the reads to mydata[] triggers a fault, but a read to a location after mydata[] does. Only that latter data is tagged as faulty. The CPU does not generate an internal exception event unless the program code tries to consume the faulty data. If the program code only reads bytes that were part of mydata[], then the CPU does not generate an exception event.

Note

If the compiler generates code that reads beyond the end of the array, then that code may trigger an internal exception event in this example. This can be avoided by using the full-syntax SEOPEN and specifying the exact array size, or by having the compiler generate more conservative code,

The CPU only takes the exception when the program attempts to consume the data associated with the fault. If the fault is recoverable, such as a page fault, then the stream resumes at the point of the data item that triggered the exception when the CPU returns from the exception event handler.

Asynchronous faults: The streaming engine does not report faults asynchronously. Future generations of the streaming engine may report some faults asynchronously, to signal functional failures that cannot be aligned with the program stream.

3.22.4.7.1 Programming Errors

Streaming engine programming errors can happen when opening a stream, or when saving and restoring a stream.

3.22.4.7.1.1 Errors When Opening a Stream

The streaming engine has the opportunity to detect and report an error only if the CPU lets the program open the stream.

At that point, the streaming engine detects invalid stream templates. Errors detected include, but are not necessarily limited to:

- Element size after promotion and element duplication must not exceed max vector length of 64 bytes.
- Transpose granule size after promotion, element duplication, and decimation must not exceed vector length of 64 bytes. Granules cannot be split across vectors after data formatting.
- Transpose 1-D stream not supported (DIMFMT = 000b).
- Decimation enabled while promotion is disabled
  - Decimate 2:1 requires at least Promote 2x
  - Decimate 4:1 requires at least Promote 4x
- Decimate 2:1 requires innermost loop count (ICNT0) to be a multiple of 2x element size.
- Decimate 2:1 requires DECDIM_WIDTH and selected DECDIM dimension (DIMx) to be a multiple of 2x element size when in DECDIM mode.
- Decimate 4:1 requires innermost loop count (ICNT0) to be a multiple of 4x element size.
- Decimate 4:1 requires DECDIM_WIDTH and selected DECDIM dimension (DIMx) to be a multiple of 4x element size when in DECDIM mode.
- Transposed row count (ICNT1) must be greater than equal 1 and less than or equal 16.
- Element size must be less than or equal transpose granule size.
- Incorrect aligned start address, and all DIM's for transpose streams

  - For transpose granule = 8-bit, must be 4-byte aligned
  - For transpose granule = 16-bit, must be 4-byte aligned
  - For transpose granule = 32-bit, must be 4-byte aligned
  - For transpose granule = 64-bit, must be 4-byte aligned
  - For transpose granule = 128-bit, must be 8-byte aligned
  - For transpose granule = 256-bit, must be 16-byte aligned
- Transpose settings for 8-bit and 16-bit TRANSPOSE not correctly set
  - 4:1 decimation and at least 4x promote for 8-bit transpose
  - 2:1 decimation and at least 2x promote for 16-bit transpose
  - Element size must be 1-byte for 8-bit transpose
  - Element size must be 2-bytes integer or 2-byte complex for 16-bit transpose
- Decimating in transpose mode
  - When Decimate 2:1, transpose granule must be >= 2x element size
  - When Decimate 4:1, transpose granule must be >= 4x element size
- Selected DECDIM in transpose mode cannot be set to DIM1.
- Selected DECDIM dimension must be an unsigned value.
- LEZR equal ICNT1 and transpose granule equal 8-bit or 16-bit not allowed.
- Reserved encoding used in any field.
- Non-zero values in reserved fields, or reserved FLAGS selected

When a program opens a stream with an invalid template, the streaming engine immediately records the fault in SEn_FSR. The streaming engine records the stream base address in SEn_FAR.

The streaming engine then feeds vectors filled with zeroed-out data and zeroed-out byte strobes to the CPU. Each vector is marked as faulty, so if the CPU attempts to consume any of these vectors, it takes an exception.

If a program opens a stream with an invalid template and then closes it without reading any data from it, the CPU does not take an exception.

3.22.4.7.1.2 Errors When Saving a Stream

The streaming engine does not support saving its state while the stream is active. It must be in the inactive or frozen state. If a program attempts to SESAVE an active stream, the CPU catches the error before the streaming engine ever sees it, and signals an exception.

3.22.4.7.1.3 Errors When Restoring a Stream

The streaming engine expects the state restored by SERSTR to have come from earlier calls to SESAVE. In a properly functioning system, this would be the case. In addition, the streaming engine also checks all the programming errors listed above after a SERSTR. It is also important to perform a SERSTR on Segment 0 as the last restore command, when any SERSTR have started on any other segments. See Section 3.22.4.5.3.

Note

When the stream engine is in the “frozen” state, it can restart from its previous position in the stream before it was frozen, when it moves to the “active” state (TSR.SEn = 1), even in the absence of an explicit SESAVE and SERSTR. In this scenario, hardware copies all the current states, and counts and restarts the stream from where the last CPU fetch stopped.

The streaming engine also expects its internal notion of stream-active to be consistent to the CPU’s notion of stream-active. These states are described in detail in Section 3.22.4.2. In particular, the streaming engine signals an error when the CPU expects the streaming engine to be active, but the streaming engine itself expects to be idle.

The streaming engine detects both classes of error when it sees the TSR.SEn transition from low to high (without a SEOPEN), as it would when returning from an interrupt, as described in Section 3.22.4.2.3.

On that transition, the streaming engine records a fault in SEn_FSR, similarly to the way it records a fault for an invalid stream template on SEOPEN. Because there is no specific address to capture for this fault, the streaming engine zeroes out SEn_FAR. The CPU takes an exception only when it attempts to read from a stream in this state.

3.22.4.7.2 Memory Translation Faults

Memory translation faults occur when the streaming engine requests a translation for a virtual address that the µTLB denies for one reason or another. Reasons include (but are not limited to) lack of privilege, lack of a valid translation entry, and an actual physical error encountered during translation. The streaming engine does not distinguish among the various possible translation faults. When the streaming engine encounters a translation fault, it records the details of the fault reported by the µTLB in SEn_FSR, and the virtual address associated with the fault in SEn_FAR.

Figure 3-142 describes the encoding. The streaming engine then reports the fault to the CPU by passing this fault status alongside data-phases requested by the CPU.

For regular, data-filled streams, the streaming engine marks the affected data as faulty, and halts further request generation. The CPU triggers an internal exception event when the program tries to read the faulty data.

For dataless CMO streams, the streaming engine halts generation of further CMO requests to the system. The CPU triggers an internal exception event when the program tries to read its synchronization data phase from the CMO stream.

For dataless PLD streams, the streaming engine silently drops the fault, and continues the stream.

3.22.4.7.3 Bit Errors Detected in the Storage Array

The streaming engine can detect bit errors in its internal storage. When it detects such an error, it flags the affected data as faulty. It records the faulty virtual address (if available) in SEn_FAR, and a corresponding error syndrome in SEn_FSR. The streaming engine records a 13-bit error status in the 13 LSBs of SEn_FSR, and provides this error status in parallel with each data-phase it provides to the CPU. The streaming engine encodes errors as follows, Section 3.22.4.7.6, and Section 3.22.4.7.7.

3.22.4.7.4 Bus Errors Reported by the System

The streaming engine could receive a bus error in response to one of its commands into the system. Its response depends on whether the command was issued for a normal or dataless stream.

For a normal stream, the streaming engine marks the affected data as faulty, as for bit errors and translation faults. The streaming engine records an appropriate error syndrome in SEn_FSR, and virtual address in SEn_FAR if the address is available. For dataless CMO streams, the streaming engine records an appropriate error syndrome in SEn_FSR, and the virtual address associated with the error in SEn_FAR. The streaming engine then signals a fault alongside the empty data phase it normally generates for CMO streams when CPU does a read. The streaming engine records a 13-bit error status in the 13 LSBs of SEn_FSR, and provides this error status in parallel with each data-phase it provides to the CPU. The streaming engine encodes errors as shown in Section 3.22.4.7.6 and Section 3.22.4.7.7.

For dataless PLD streams, the streaming engine silently drops the fault, and continues the stream.

3.22.4.7.5 Functional Failures

If the CPU requests data from the streaming engine when the streaming engine does not expect it—for example, no stream open, no SESAVE command active, and so forth—the streaming engine records an appropriate error syndrome in SEn_FSR and returns a single data-phase response to the CPU with the error syndrome. Such a request constitutes a violation of protocol between the streaming engine and the CPU. The streaming engine does not detect other violations of protocol arising from functional failures.

This is meant to capture the cases where the CPU violates protocol to the SE, and the SE ‘papers over it’ to prevent a system hang.

3.22.4.7.6 Error Syndrome Encoding

The streaming engine records a 13-bit error status in the 13 LSBs of SEn_FSR, and provides this error status in parallel with each data-phase it provides to the CPU. The streaming engine encodes errors as follows.

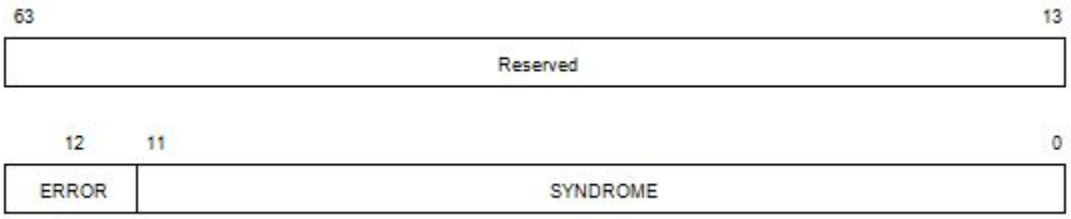


Figure 3-141. General Streaming Engine Error Syndrome Encoding

If the ERROR bit is zero, no error condition exists. All other bits of SEn_FSR must be zero. Otherwise, the error status falls into one of the following formats. For MMU related errors, see the MMU specification. For errors reported by the L2 memory controller, see the L2 memory controller specification.

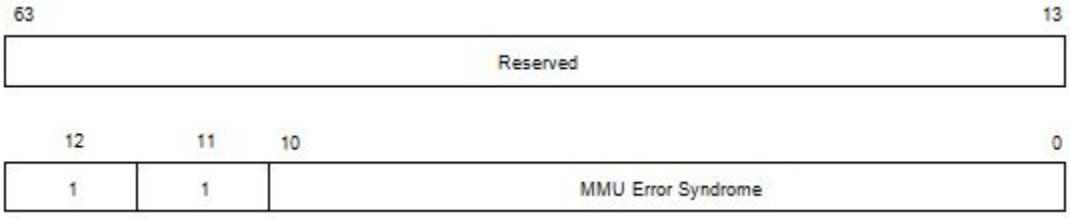


Figure 3-142. MMU Fault Syndrome Encoding

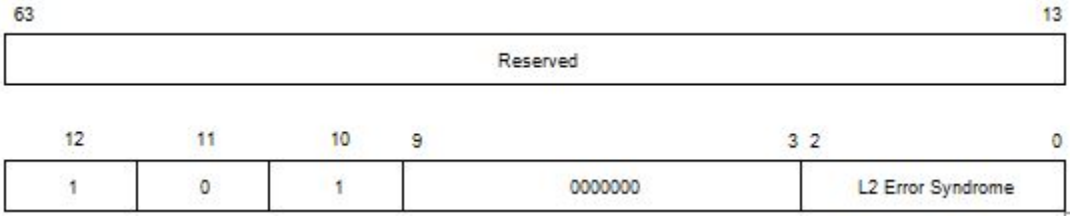


Figure 3-143. L2 Fault Syndrome Encoding

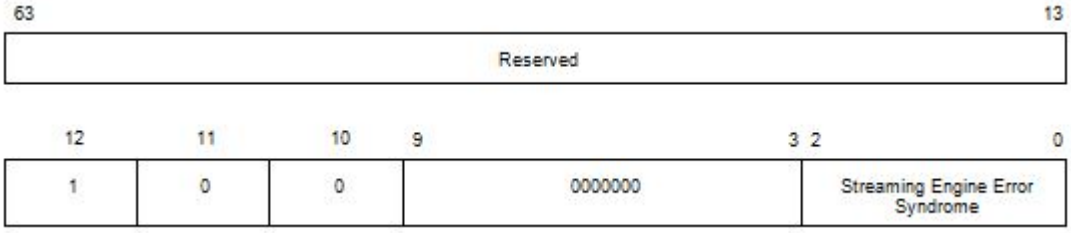


Figure 3-144. Streaming Engine Fault Syndrome Encoding

The MMU and L2 error syndromes are defined by the corresponding blocks. The streaming engine defines the following error syndromes for errors it detects.

Table 3-196. Streaming Engine Error Syndromes

Syndrome	Meaning
000b	No error. This encoding never appears when ERROR = 1
001b	Invalid stream template
010b	Stream active mismatch. See “1.3.5.2 Stream States” on page 1-42
011b	CPU spurious read while stream-active low
100b	Address bit 48 rollover. SE does not allow bit 48 of an address to change
101b	Internal Storage Parity Error
110b	L2 fragmented data return. SE does not support fragmented data return

Table 3-196. Streaming Engine Error Syndromes (continued)

Syndrome	Meaning
111b	Reserved

3.22.4.7.7 Fault Address Recording

When it can, the streaming engine records the 49-bit virtual address associated with the fault reported in SEn_FAR. The streaming engine sign extends this address to 64 bits when the CPU reads it.

The streaming engine is not always able to associate an address with every fault. In those circumstances, the streaming engine clears SEn_FAR to 0.

Certain special faults get recorded as follows (see previous sections for other faults):

- Invalid streams opened, or activating a frozen stream with invalid parameters (bad SERSTR) records the stream base address in SEn_FAR.
- Bit errors in the data storage for transpose streams record the storage location associated with the address at the top of the current transposed column.

Bus errors coming back from system for transpose streams records the storage location associated with the address at the top of the current transposed column. The streaming engine records the earliest fault associated with a stream. However, as the streaming engine can generate multiple addresses in parallel to satisfy a single data-phase destined for the CPU, it must choose one of those addresses to report if both addresses experience a fault. The streaming engine does not define an architectural order among these parallel addresses; rather, the address the streaming engine chooses to report the fault for when two faults occur in parallel is implementation defined.

3.22.4.8 Debugger Support

In this generation, the streaming engine only provides the debugger visibility into the streaming engine's internal state. It does not provide any sort of breakpoint or watchpoint capability.

Other debugger-related actions, such as flushing and restarting the streaming engine pipeline, are managed by the CPU and are largely transparent to the debugger.

3.22.4.8.1 Internal State Visibility

The streaming engine makes its tags, counters, and address registers visible to the debugger through an ECR interface. See Section 3.22.4.6.3 for more details.

3.22.4.8.2 Analysis Events

The streaming engine provides a limited number of analysis event outputs. These outputs allow analyzing the performance of the streaming engine. Analysis discrete events generated by the streaming engine.

Table 3-197. SE Debug Events and Stalls Per Stream

No.	Category	Event	Description	Generic / Interest Qualified	Level / Pulse
1		SEOPEN	SEOPEN on new stream	No	Pulse
2		SECLOSE	SECLOSE on a stream	No	Pulse
3		SEBRK	SEBRK on a stream	No	Pulse
4		Frozen	Stream is in a Frozen State	No	Level
5		TAG hit	Stream Address Tag Hit (Linear Stream Only)	No	Pulse
6		TAG miss	Stream Address Tag Miss (Linear Stream Only)	No	Pulse
7		SE stall	SE overall stall	No	Level
8		µTLB stall	SE µTLB stall (µTLB not ready for new command)	No	Level
9		64-byte command	SE command to L2 is 64-byte	No	Pulse
10		128-byte command	SE command to L2 is 128-bytes	No	Pulse
11		SE data stall	CPU ready for data but SE has no data	No	Level
12		CPU stall	SE has data ready but CPU is not reading data	No	Level
13		L2 Port 0 starve	SE L2 Port 0 has commands but no L2 credits	No	Level

Table 3-197. SE Debug Events and Stalls Per Stream (continued)

No.	Category	Event	Description	Generic / Interest Qualified	Level / Pulse
14		L2 Port 1 starve	SE L2 Port 1 has commands but no L2 credits	No	Level

4 C71x MMA

4.1 C7x MMA Architecture Specification

4.1.1 Introduction

The Matrix Multiply Accelerator (MMA) is a tightly coupled matrix multiplication coprocessor that extends the C7x processor architecture's scalar and vector facilities. Fed through the streaming engines (SEs), the MMA provides a large number of multiply-accumulate (MAC) operations, matrix-optimized storage, and matrix-optimized data movement that can be efficiently used for the dense linear algebra operations that dominate the computations found in vision (CNNs, SfM, filtering, and so forth), speech (xNNs), audio (convolutions), radar (FFTs) and controls (reinforcement learning, state updates) based applications.

The multiplication bandwidth of MMA might be attractive for some cryptographic applications that depend on bignum arithmetic. However, the MMA has not been hardened for these applications and using MMA for cryptography acceleration in products is not recommended.

4.1.1.1 Overview

The MMA key characteristics are summarized below.

4.1.1.1.1 Supported Data Types

Data traveling through MMA goes through several stages. At each stage, a set of data types is supported by the architecture.

The following input operand data types are supported:

- 4-bit unsigned integer (on A input only, transformed by LUT into 8-bit data)
- 8-bit unsigned integer
- 8-bit signed integer
- 16-bit unsigned integer
- 16-bit signed integer
- 32-bit unsigned integer
- 32-bit signed integer

The following result data types are supported for accumulation:

- 32-bit unsigned integer for 8-bit unsigned operands
- 32-bit signed integer for 8-bit signed or mixed signed/unsigned operands
- 64-bit unsigned integer for 16-bit unsigned operands
- 64-bit signed integer for 16-bit signed or mixed signed/unsigned operands
- 128-bit unsigned integer for 32-bit unsigned operands. Compiler support for this type is not expected.
- 128-bit signed integer for 32-bit signed or mixed signed/unsigned operands. Compiler support for this type is not expected.

The following result data types are supported on output:

- 8-bit unsigned integer (uint8_t)
- 8-bit signed integer (int8_t)
- 16-bit unsigned integer (uint16_t)
- 16-bit signed integer (int16_t)
- 32-bit unsigned integer (uint32_t)
- 32-bit signed integer (int32_t)
- 64-bit unsigned integer (uint64_t)
- 64-bit signed integer (int64_t)
- 128-bit unsigned integer (uint128_t). Compiler support for this type is not expected.
- 128-bit signed integer (int128_t). Compiler support for this type is not expected.

The hardware-supported conversions from accumulator to output types are listed in Table 4-1.

Table 4-1. Hardware-Supported Conversions

Accumulator Type	(u)int8_t out	(u)int16_t out	(u)int32_t out	(u)int64_t out	(u)int128_tout[1]
(u)int32_t	yes	yes	yes	no	no
(u)int64_t	yes	yes	yes	yes	no

Table 4-1. Hardware-Supported Conversions (continued)

Accumulator Type	(u)int8_t out	(u)int16_t out	(u)int32_t out	(u)int64_t out	(u)int128_tout(1)
(u)int128_t	yes	yes	yes	yes	yes

(1) Compiler support for these types is not expected.

These conversions include optional right shifting, 1/2 LSB rounding, and saturation.

4.1.1.1.2 Required Calculations

MMA computes the product of a row vector (A) with a square matrix (B). The row vector that results from this operation can be combined with a row vector in the square matrix C and written back to a row in C. Q point adjustment and non-linearity is available on final C matrix row contents that are moved to C7x registers.

4.1.1.1.3 Performance

Fundamentally, MMA computes the vector dot product of row-vector operand A with each the columns of matrix operand B and accumulates the dot products into a row of matrix C with a throughput of one clock cycle. These operations can be sustained indefinitely.

For 8-bit operands, there are 64 elements in row vector A and 4096 elements in square matrix B, resulting in a requirement for 4096 multiplications and the equivalent of 4096 summations per clock cycle. The intermediate dot product summations are accomplished without rounding or overflow.

For 16-bit operands, there are 32 elements in row vector A and 1024 elements in square matrix B, resulting in a requirement for 1024 multiplications and the equivalent of 1024 summations per clock cycle. The intermediate dot product summations are accomplished without rounding or overflow.

For 32-bit operands, there are 16 elements in row vector A and 256 elements in square matrix B, resulting in a requirement for 256 multiplications and the equivalent of 256 summations per clock cycle. The intermediate dot product summations are accomplished without rounding or overflow.

4.1.1.1.4 Safety

MMA includes these safety-related items:

- Self-checking FSMs
- Parity protection on configuration and status registers in MMA
- SECDED ECC on all SRAM data
- Detection and reporting several types of MMA programming errors

4.1.1.2 Theory of Operation

4.1.1.2.1 MMA Operands

For data processing instructions, the MMA unit reads up to two operands. The operands may be directed to the A vector, B matrix, C matrix storage, or to configuration registers. The operand width is always 512 bits and matches the width of the C7x vector registers. MMA data processing instructions may use vector registers or streaming engines as operands.

When a streaming engine is used for a MMA operand, it has the same behavior as it does for C7x instructions.

Results returned to the C7 CPU are 512 bits wide and are accompanied with status.

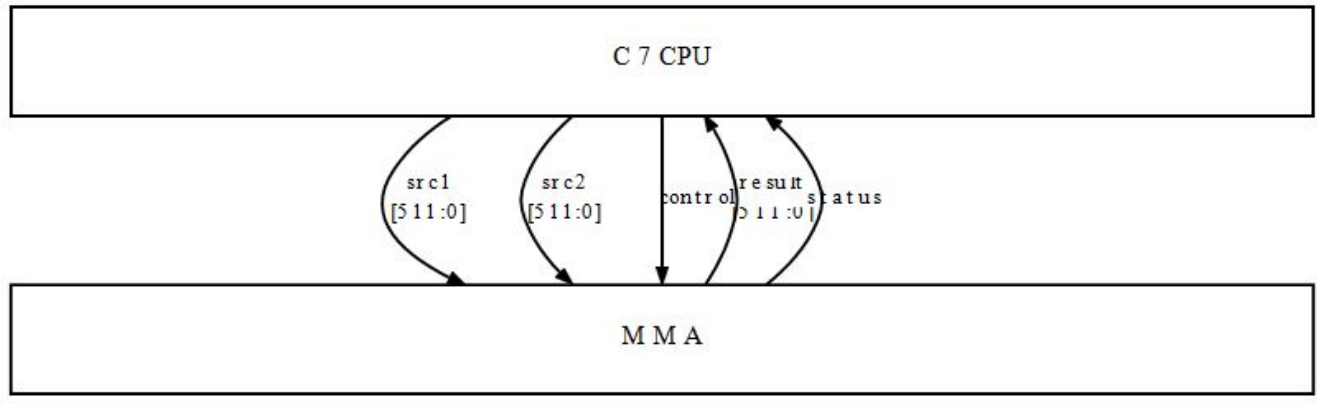


Figure 4-1. Title TBD

4.1.1.2.2 Operand Element Numbering Convention

Element numbering in the 512-bit MMA operands and results is consistent with the C7x endian convention, C7x vector operations, and streaming engine operations. Within a 512-bit operand, elements are numbered from right to left, starting with element 0. When looking at a 512-bit quantity, the element order is reversed from mathematical convention; however, in terms of data addresses in memory, the lowest-numbered elements are at the lowest- numbered address. The element mapping for the supported operand sizes in the 512-bit MMA operands is shown in Table 4-2.

Table 4-2. Element Mapping

(u)int8 Element 63 [511:504]	...	(u)int8 Element 1 [15:8]	(u)int8 Element 0 [7:0]
(u)int16 Element 31 [511:496]	...	(u)int16 Element 1 [31:16]	(u)int16 Element 0 [15:0]
(u)int32 Element 15 [511:480]	...	(u)int32 Element 1 [63:32]	(u)int32 Element 0 [31:0]

4.1.1.2.3 Row Vector A Storage

A single 512-bit array is provided for the A row vector. The A operand can be transformed by a lookup table to convert 4-bit data to 8 bits. No delay is required when loading the A storage and beginning a computation with the new value.

4.1.1.2.4 B Matrix Storage

Two memory arrays are provided to store the B matrix. Both arrays are limited no more than one read or write access per cycle and are intended to be used in a double buffered or ping-ponged fashion. The capacity available in B memory arrays varies with the element size in the B matrix. For 8-bit B matrix elements, a single 64×64 matrix fills one array. For 16-bit B matrix elements, a single 32×32 matrix fills one array. For 32-bit B matrix elements, a single 16×16 matrix fills one array.

In typical operation, a designated foreground B array, sometimes called Bf multiplied by a sequence of A vectors and in parallel, the background copy (Bb) receives incoming data for a subsequent matrix multiplication. The management of the foreground and background identity and swapping handled by two state machines: the B FSM and the C FSM, which are described later.

Several transformations are supported on the stream of operands destined for the B matrix storage.

4.1.1.2.5 C Matrix Storage

Two memory arrays are provided to store the C matrix. Both arrays support two accesses per cycle: one read and one write (this is different from the B storage described above). The arrays are expected to be used in a double buffered or ping-ponged fashion. Each of the C memory arrays contains 64 rows of 2048 bits. Unlike the B matrix, the storage capacity for the C matrix does not vary with the element size. For 8-bit multiply-accumulate

operations, a single 64×64 matrix with 32-bit accumulator elements fills one C memory array. For 16-bit multiply-accumulate operations, two 32×32 matrices with 64-bit accumulator elements fill one C memory array. For 32-bit multiply-accumulate operations, four 16×16 matrices with 128-bit accumulator elements fill one C memory array.

In typical operation, a designated foreground C array, sometimes called Cf is receiving rows of dot products from the multiply and add hardware. In parallel, the background copy (Cb) is being accessed by a state machine (the X FSM) that moves rows of C to the output transfer buffer.

A C storage array can be simultaneously accessed for write by the multiply and add hardware and for read by the X FSM.

There is no bypassing in the C memory arrays. A row in C cannot be accessed until 2 cycles after a prior write.

4.1.1.2.6 Reset

Resetting the MMA unit clears the HWA_CONFIG and HWA_OFFSET registers to avoid spurious parity errors and forces all the internal state machines to the initial state based on the cleared HWA_CONFIG. The values stored in the A vector, B matrix, and C matrix are unaffected by reset. After reset, a HWAOPEN instruction should be executed to prepare the MMA unit for useful work.

4.1.1.2.7 Pipeline Effects

The MMA pipeline through the data transfer operation appears to advance every clock cycle. Pipeline stages have corresponding valid bits that enable or suppress operations associated with instructions dispatched from the C7x CPU.

The fundamental vector-by-matrix calculation is deeply pipelined due to the high fanin of the dot product additions and the physical distance the intermediate products and sums must travel. While MMA computation operations can be issued without consideration for pipelining, results are visible in the C matrix storage and subsequently in the transfer buffer after fixed delays. These delays are measured in MMA clock cycles.

C matrix results are available to move into the transfer buffer 16 clock cycles after a computation instruction is issued.

Transfer buffer data are available to move into C7x CPU vector registers 4 clock cycles after an instruction that moves C matrix data into the transfer buffer.

4.1.1.2.8 Dataflow

Dataflow in MMA is fixed and proceeds from the operand inputs to operand storage. The matrix multiplication operands are next read and multiplied and added with an optional earlier value from the C matrix storage. The C matrix results are passed through an output processing block to an output buffer, which is accessed by the C7x. Figure 4-2 illustrates the data flow. Dimensions of the storage elements are described in later sections. Major control paths are indicated with dashed lines.

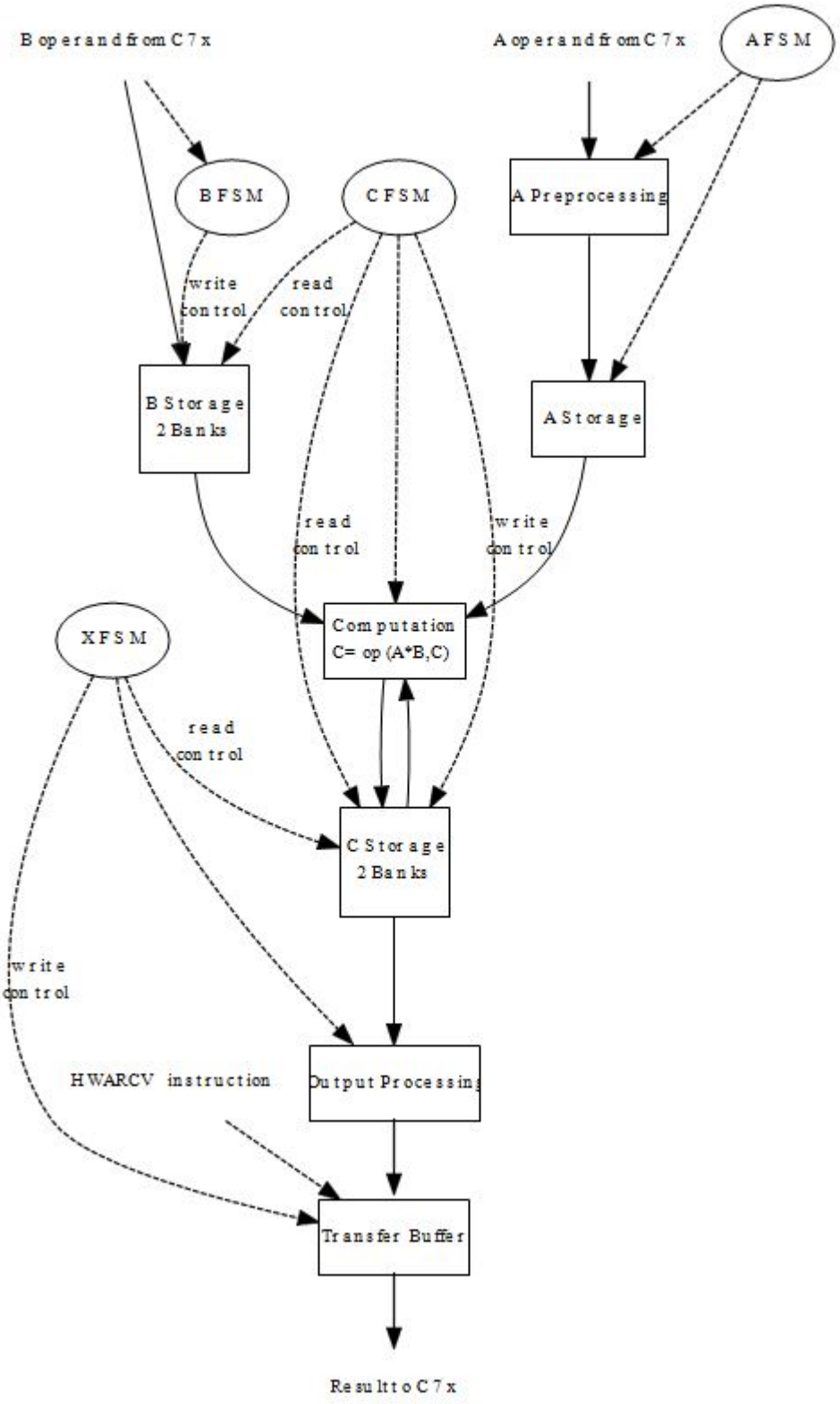


Figure 4-2. Data Flow

4.1.1.2.9 MMA Calculation Detail

The calculations supported by the MMA unit are:

$C_{f|b} = (A \times B_f)$

$C_{f|b} = -(A \times B_f)$

$C_{f|b} = C_f + (A \times B_f)$

$C_{f|b} = C_f - (A \times B_f)$

The matrix multiplication is always on the full size of the A vector and B matrix. 8-bit operations always multiply a 64-element vector by a 64×64 element matrix. 16-bit operations always multiply a 32-element vector by a 32×32 element matrix. 32-bit operations always multiply a 16-element vector by a 16×16 element matrix.

The A×B operation is performed with sufficient operand and result widths such that wrap-around or saturation cannot occur. The C matrix storage element size has sufficient width to support a number of accumulations without wrap- around or saturation. Long-running calculations can periodically store C matrix partial results for later processing in higher precision formats outside the MMA unit.

Table 4-3. Operand Types

B Operand Types	C Accumulator Width (bits)	Approximate Number of Accumulations Without Overflow
(u)int8_t	32	210
(u)int16_t	64	227
(u)int32_t	128	260

4.1.1.2.10 MMA Output Processing

Results transferred back to the C7x CPU are usually in lower precision than the accumulator format. The format conversion is programmable and can include a logical or arithmetic right shift, rounding, saturation, and ReLU non- linearity.

For operations that produce 128-bit C matrix elements, 128-bit vector elements can be specified as the result format. 128-bit integers are not a directly supported data format in the C7x architecture. 128-bit integers are not supported as a native type in the C7x code generation tools. Software may handle MMA-generated 128-bit integers by concatenating two or more of the supported integer widths, but two versions of such code may be required to run at full performance in both endian modes.

4.1.1.2.11 Transfer Buffer

The MMA transfer buffer is a FIFO of vector-width entries. Results going back to C7x are written into the transfer buffer after output processing from one of the C storage arrays. A C7x instruction can then read and implicitly pop the transfer buffer with relatively low latency due to the close physical location of the transfer buffer to the C7x CPU. The X FSM is responsible for pushing results from computations into the transfer buffer.

The transfer buffer is sized to accommodate 24 results. This is sufficient to sink all the MMA computations in-flight if there is some long delay in reading results. There is no backpressure mechanism to the C7x CPU and MMA transfers issued without a matching rate of transfer buffer reads will result in a FIFO overflow and an exception.

4.1.1.2.12 Controlling MMA from C7x

MMA hardware is visible through the C7x ISA as a set of HWA-prefixed hardware accelerator instructions. MMA is not visible in the C7x control register address space.

MMA instructions issued from the C7x pipeline always complete. There are no operating states in the MMA unit that can block instruction issue from the C7x CPU. There are no operating states in the MMA unit change the latency of operations inside the MMA unit. Instructions that can move data from MMA to C7x always deliver a result and always with the same latency. There are no conditions where the C7x pipeline stalls due to the an unexpected change in MMA result latency.

4.1.1.2.13 Internal MMA Control

Many of the activities that occur in the MMA module are a result of internal state machines that are responsive to specific groups of instructions. There is a strong similarity between the C7x streaming engine (SE) operation and MMA. In both cases, software configures sequencing logic (state machines) and the sequencing logic performs actions as a result of C7x machine code execution.

The four main state machines in MMA are named A FSM, B FSM, C FSM, and X FSM have similar control mechanisms. Each of the state machines has one or more period counters that perform specific actions when the period expires. These actions including swapping read or write banks on the B or C storage or re-initializing storage pointers.

The FSM configuration register is writable with the HWAOPEN instruction. The configuration register is available for read accesses in the transfer buffer register space.

4.1.1.2.14 Error Handling and Error Conditions

Several types of errors are detected in the MMA unit. These errors can be due to incorrect programming, transient soft errors (SEU), or persistent hardware failures. The error detection facilities are focused on control and status portions of MMA rather than the datapath and calculation hardware.

When errors are detected, an error indication and error code is sent to the C7x CPU with the next HWARCV instruction result. No other MMA instruction responds to the C7x CPU with an error indication.

Errors detected in the MMA unit are not reported in a way that identifies a specific error-causing instruction. Errors may be detected in activities that are not specific to a particular instruction. An error-causing instruction may have executed long before a HWARCV instruction is executed that makes the error condition visible to the C7x CPU. For example, a HWAOPEN instruction that configures the FSMs in a way that results in a B or C matrix storage bank collision can execute many clock cycles before the bank collision is detected.

When errors are detected, the MMA unit continues to accept instructions and deliver results, although the error condition may result in corrupted results. Error conditions are not physically destructive. Error conditions can persist for long periods of time in MMA without physical damage to the device. Software must read the C7x error cause register and take appropriate action to clear the error condition to ensure correct results are produced.

The error conditions that are detected in the MMA unit are:

- More than one read on the same C bank from C FSM and X FSM. This is a programming error.
- More than one write to the same C bank from C FSM due to a computation result and a direct C matrix load instruction. This is a programming error.
- Transfer buffer FIFO underflow and overflow conditions caused by programming errors.
- FSM state error. This is usually an internally detected soft error. Frequent errors of this type could indicate a hardware defect.
- Parity error on MMA static configuration register. This is usually an internally detected soft error. Frequent errors of this type could indicate a hardware defect.

These error conditions are visible in the C7x error code and HWA_STATUS register with the following encodings.

Table 4-4. Definition of Bus MMA_ERROR_CODE[5:0]

Field:	ErrorCode
Size:	6
Range:	5:0

ErroCode: Encoded error codes

Table 4-5. MMA_ERROR_CODE Description

Encoding	Name	Description
6'b000000	NO_ERROR	No error was detected
6'b000001	Reserved	Reserved encoding for future error detection
6'b000010	C_READ_ERROR	More than one read from a C matrix bank (programming error)
6'b000011	C_WRITE_ERROR	More than one write to a C matrix bank (programming error)
6'b000100	X_UNDERFLOW	Transfer buffer read was attempted but there was no valid data in the FIFO (programming error)
6'b000101	X_OVERFLOW	Transfer buffer read was attempted but an earlier HWA instruction caused the FIFO to overflow (programming error)
6'b000110	FSM_STATE_ERROR	FSM state transition error (hardware error)
6'b000111	OPERAND_ERROR	Unsupported on MMA 1.0. C7x does not source parity bits with HWA operands. Vector operand parity error from C7x (hardware error)

Table 4-5. MMA_ERROR_CODE Description (continued)

Encoding	Name	Description
6'b001000	CONTROL_ERROR	Unsupported on MMA 1.0. C7x does not source parity bits with HWA controls. Parity error on control from C7x (hardware error)
6'b001001	CONFIG_PARITY	Parity error on HWA_CONFIG (hardware error)
6'b001010	OFFSET_PARITY	Parity error on HWA_OFFSET (hardware error)

Multiple simultaneous errors should be rare. In the event of simultaneously detected errors, hardware errors are reported before programming errors. The priority of simultaneous errors within these cases is undefined; however, a valid error code consistent with one of the detected errors is reported.

Some errors related to the MMA unit are detected in the C7x CPU. The MMA unit does not record any information about these errors. These types of error include attempting to execute a MMA instruction when the C7x MMA_BUSY flag is set, or attempting to execute MMA instructions that are part of a malformed execute packet.

In C7x instances without the MMA unit, the outcome of executing MMA instructions is undefined. In these systems, the C7x CPU does not trap MMA instructions for emulation.

4.1.1.2.15 Computation Errors

There is no specific hardware in the MMA unit to detect errors that occur during a computation. For safety-critical applications, software must periodically run functional self-test code to detect hardware failures in the field. Due to the highly controllable and observable nature of logic in multipliers and adders, this functional self-test code has high fault coverage and compact code size. The MMA User's Guide includes the outline of MMA self-test code.

4.1.1.2.16 Debug Facilities

The MMA unit contains a large number of data and control state elements. Instructions exist to move operands into the data state elements and control state elements. The HWARCV instruction and transfer buffer register space are defined to non-destructively observe control and status state elements. To observe the A vector, B matrix storage arrays, and C matrix storage arrays, read access is provided through a debug register space (EDI) visible to software running on the C7x CPU.

State element reads through the debug register space do not advance the MMA pipeline. Pipeline effects may be visible to debug accesses. No bypassing or forwarding is provided on any debug mode read accesses. For example, the B matrix background storage may appear to be partially updated even though instructions that load an entire B matrix bank have completed.

State element reads through the debug register space may observe states or encodings that differ from the the values written from the MMA instructions for microarchitectural reasons. As an example, the B matrix storage in 32-bit mode may contain replicated values. The transformation of operands written by MMA instruction into the values observed by debug accesses is described in the MMA Microarchitecture Specification.

The values of state elements observed through the debug interface are undefined between the time a HWACLOSE instruction is issued and the time the next HWAOPEN instruction executes.

4.1.2 ISA

The MMA instruction set architecture has several types of objects that are covered in detail in the following sections. These objects are control and status registers, internal state machines, and C7x instructions.

4.1.2.1 Architectural Registers

Control and status registers in the MMA unit configure operations and provide visibility into the current operations. These registers are written by the HWAOPEN instruction, and are readable with the HWARCV instruction from the transfer buffer or through the EDI address space. There is no access to control and status registers in the MMA unit from the C7x MVC instruction.

4.1.2.1.1 HWA_CONFIG

The HWA_CONFIG register is the primary configuration register in the MMA unit. Operations sequenced by the A_FSM, B_FSM, C_FSM, and X_FSM use fields in HWA_CONFIG to define their operation.

HWA_CONFIG may be observed through the transfer buffer after executing the HWAXFER instruction.

The HWA_CONFIG is cleared by reset to avoid parity errors. The cleared value is not a useful configuration and software should update HWA_CONFIG before using the MMA unit.

4.1.2.1.1.1 Parity Control

The PARITYCTRL field of the HWA_CONFIG register provides controls that can be used to modify the operation of parity logic associated with the HWA_CONFIG and HWA_OFFSET registers. These controls are used to suppress parity checking to allow some functional testing on partially-good chips. More importantly, these controls are used for parity error injection for hardware design verification and software error handler verification.

The following steps are required for error injection and observation:

1. The HWAOPEN instruction is used to write HWA_CONFIG and HWA_OFFSET with parity computation enabled and parity checking enabled (PARITYCTRL = NORMAL).
2. The values in the vector registers are corrupted by software in the specific field to be tested.
3. The HWAOPEN instruction is again used to write HWA_CONFIG and HWA_OFFSET using the corrupted data with the parity computation disabled and parity checking enabled (PARITYCTRL = PNCMCK). In this mode, the originally computed and saved parity value is checked against the parity computed on use in the affected register.
4. The HWARCV instruction is executed to observe the error.

4.1.2.1.1.2 Register Definition

Table 4-6. Definition of Register HWA_CONFIG[511:0]

Field:	PARITYCTRL	RSVD	X_CONFIG	C_CONFIG	B_CONFIG	A_CONFIG
Size:	2	6	104	272	96	32
Range	:511:510	509:504	503:400	399:128	127:32	31:0

PARITYCTRL: Parity control, typically cleared in mission mode

Table 4-7. HWA_CONFIG Description

Encoding	Name	Description
2'b00	NORMAL	Parity is computed by hardware on write operations to HWA_CONFIG and HWA_OFFSET and checking is enabled.
2'b01	PNCM_CK	Parity is not computed by hardware on write operations to HWA_CONFIG and HWA_OFFSET (parity bits hold their prior state) and checking is enabled.
2'b10	PCM_NCK	Parity is computed by hardware on write operations to HWA_CONFIG and HWA_OFFSET and checking is disabled.
2'b11	PNCM_NCK	Parity is not computed by hardware on write operations to HWA_CONFIG and HWA_OFFSET (parity bits hold their prior state) and checking is disabled.

RSVD: Reserved field. Software should write zero to this field.

X_CONFIG: X FSM configuration field. See X_CONFIG bus definition.

C_CONFIG: C FSM configuration field. See C_CONFIG bus definition.

B_CONFIG: B FSM configuration field. See B_CONFIG bus definition.

A_CONFIG: A FSM configuration field. See A_CONFIG bus definition.

Table 4-8. Definition of Bus A_CONFIG[31:0]

Field:	RSVD	ALUTEN	RSVD	ATYPE
Size:	23	1	5	3
Range:	31:9	8:8	7:3	2:0

RSVD: Reserved field. Software should write zero to this field.

ALUTEN: A vector element lookup table expansion

Table 4-9. A_CONFIG[31:0] Description

Encoding	Name	Description
1'b1	LUT	Even nibbles (3:0, 15:8, etc.) are passed through a lookup table. The ATYPE field must be set to INT8 or UINT8. Lookup table values are located in the HWA_OFFSET register.
1'b0	NOLUT	Bypass the lookup table

RSVD: Reserved field. Software should write zero to this field.

ATYPE: A vector element type. Width must be consistent with the B FSM setting.

Table 4-10. A_CONFIG Description

Encoding	Name	Description
3'b111	INT32	Signed word (32-bit) data
3'b110	Reserved	Undefined B matrix element type
3'b101	INT16	Signed half word (16-bit) data
3'b100	INT8	Signed byte data
3'b011	UINT32	Unsigned word (32-bit) data
3'b010	Reserved	Undefined B matrix element type
3'b001	UINT16	Unsigned half word (16-bit) data
3'b000	UINT8	Unsigned byte (8-bit) data

Table 4-11. Definition of Bus B_CONFIG[95:48]

Field:	RSVD	BOFFSET	RSVD	BSTART	RSVD	ORDER
Size:	24	8	7	1	7	1
Range:	95:72	71:64	63:57	56:56	55:49	48:48

Table 4-12. Definition of Bus B_CONFIG[47:0]

Field:	RSVD	BTYPE	BRSTPER	BSWPER
Size:	6	2	8	32
Range:	47:42	41:40	39:32	31:0

RSVD: Reserved field. Software should write zero to this field.

BOFFSET: Global row or column offset

RSVD: Reserved field. Software should write zero to this field.

BSTART: Initial B bank selection for writing B matrix data

RSVD: Reserved field. Software should write zero to this field.

ORDER: Transpose control

Table 4-13. B_CONFIG Description

Encoding	Name	Description
1'b1	COL	Column-major order (transposed)
1'b0	ROW	Row-major order (non-transposed)

RSVD: Reserved field. Software should write zero to this field.

BTYPE: B matrix element type. Width must be consistent with the A FSM setting and C FSM setting.

Table 4-14. B_CONFIG Description

Encoding	Name	Description
2'b11	SIZE32	Signed or unsigned word (32-bit) data
2'b10	Reserved	Undefined B matrix element type
2'b01	SIZE16	Signed or unsigned half word (16- bit) data

Table 4-14. B_CONFIG Description (continued)

Encoding	Name	Description
2'b00	SIZE8	Signed or unsigned 8-bit data

BRSTPER: B offset reset period

BSWPER: B bank switch period

Table 4-15. Definition of Bus C_CONFIG[271:238]

Field:	CWRSTPER	CRRSTPER	CWSWPER
Size:	8	8	18
Range:	271:264	263:256	255:238

Table 4-16. Definition of Bus C_CONFIG[237:204]

Field:	CWSWPER	CRSWPER
Size:	14	20
Range:	237:224	223:204

Table 4-17. Definition of Bus C_CONFIG[203:170]

Field:	CRSWPER	BSWPER
Size:	12	22
Range:	203:192	191:170

Table 4-18. Definition of Bus C_CONFIG[169:136]

Field:	BSWPER	OP0PER
Size:	10	24
Range:	169:160	159:136

Table 4-19. Definition of Bus C_CONFIG[135:102]

Field:	OP0PER	OP1PER
Size:	8	26
Range:	135:128	127:102

Table 4-20. Definition of Bus C_CONFIG[101:68]

Field:	OP1PER	CLRSTPER	CLSWPER	RSVD	CLOFFSET	RSVD	CWOFFSET
Size:	6	8	8	2	6	2	2
Range:	101:96	95:88	87:80	79:78	77:72	71:70	69:68

Table 4-21. Definition of Bus C_CONFIG[67:34]

Fiel CdW	OFFSER	TSVD	CROFFSET	RSVD	CLSTARCT	WSTARC	TRSTARB	TSTARO	TPSTARR	TSVHD	WLDTYPR	ESVHD	WLDDS
Size	4	2	6	3	1	1	1	1	1	4	4	5	1
Range	67:64	63:62	61:56	55:53	52:52	51:51	50:50	49:49	48:48	47:44	43:40	39:35	34:34

Table 4-22. Definition of Bus C_CONFIG[33:0]

Field:	HWLDDST	RSVD	OPERATION1	RSVD	OPERATION0	RSVD	BTYPE	RSVD	ATYPE
Size:	2	6	2	6	2	5	3	7	1
Range	: 33:32	31:26	25:24	23:18	17:16	15:11	10:8	7:1	0:0

CWRSTPER: C write row offset reset period for computations

CRRSTPER: C read row offset reset period

CWSWPER: C bank switch period for computations writes

CRSWPER: C bank switch period for read instructions

BSWPER: B bank switch period

OP0PER: Operation0 period. See C FSM description for behavior when OP0PER equals zero.

OP1PER: Operation1 period. See C FSM description for behavior when OP0PER equals zero.

CLRSTPER: C write row offset reset period for HWALD* instructions

CLSWPER: C bank switch period for HWALD* instruction writes

RSVD: Reserved field. Software should write zero to this field.

CLOFFSET: C row write offset for HWALD* instructions. This value is scaled by 4x for HWLDDST X1 mode align with the row-granular address.

RSVD: Reserved field. Software should write zero to this field.

CWOFFSET: C row write offset for computations

RSVD: Reserved field. Software should write zero to this field.

CROFFSET: C row read offset

RSVD: Reserved field. Software should write zero to this field.

CLSTART: Initial C bank selection for writing operands from HWALD*

CWSTART: Initial C bank selection for writing computation results

CRSTART: Initial C bank selection for reading operands

BSTART: Initial B bank selection for reading B matrix data for the matrix computation

OPSTART: Initial C operation selection

RSVD: Reserved field. Software should write zero to this field.

HWLDTYPE: C matrix element type from a HWALD* instruction. In larger destination fields, signed types are sign extended and unsigned types are zero-extended.

Table 4-23. C_CONFIG Encodings

Encoding	Name	Description
4'b1111	Reserved	Undefined C operand element type
4'b1110	Reserved	Undefined C operand element type
4'b1101	Reserved	Undefined C operand element type
4'b1100	Reserved	Undefined C operand element type
4'b1011	INT32	Signed word (32-bit) data.
4'b1010	Reserved	Undefined C operand element type.
4'b1001	INT16	Signed half word (16-bit) data.
4'b1000	INT8	Signed byte (8-bit) data.
4'b0111	Reserved	Undefined C operand element type
4'b0110	Reserved	Undefined C operand element type
4'b0101	Reserved	Undefined C operand element type
4'b0100	Reserved	Undefined C operand element type
4'b0011	UINT32	Unsigned word (32-bit) data.
4'b0010	Reserved	Undefined C operand element type
4'b0001	UINT16	Unsigned half word (16-bit) data.
4'b0000	UINT8	Unsigned byte (8-bit) data.

RSVD: Reserved field. Software should write zero to this field.

HWLDDST: Element size written into C matrix storage and element scaling.

Table 4-24. C_CONFIG Encodings

Encoding	Name	Description
3'b111	X1	Input data is copied directly into C storage with no scaling
3'b110	Reserved	Undefined C storage element type
3'b101	Reserved	Undefined C storage element type
3'b100	Reserved	Undefined C storage element type
3'b011	X4_3	Input data is extended to 4 times the original bit width and shifted left by 3x the original bit width. Undefined for unsigned data.
3'b010	X4_2	Input data is extended to 4 times the original bit width and shifted left by 2x the original bit width. Undefined for unsigned data.
3'b001	X4	Input data is extended to 4 times the original bit width and shifted left by 1x the original bit width. Defined for signed and unsigned data.
3'b000	X4_0	Input data is extended to 4 times the original bit width and shifted left by 0x the original bit width. Undefined for unsigned data.

RSVD: Reserved field. Software should write zero to this field.

OPERATION1: Operation 1 calculation performed for HWA.*OP.* instructions. Operation 1 executes with a period defined in C_CONFIG.OP1PER.

Table 4-25. C_CONFIG Encodings

Encoding	Name	Description
2'b11	MULPLUS	Compute C+(A×B)
2'b10	MULMINUS	Compute C-(A×B)
2'b01	MULNEGATE	Compute -(A×B)
2'b00	MUL	Compute (A×B)

RSVD: Reserved field. Software should write zero to this field.

OPERATION0: Operation 0 calculation performed for HWA.*OP.* instructions. Operation 0 executes with a period defined in C_CONFIG.OP0PER.

Table 4-26. C_CONFIG Encodings

Encoding	Name	Description
2'b11	MULPLUS	Compute C+(A×B)
2'b10	MULMINUS	Compute C-(A×B)
2'b01	MULNEGATE	Compute -(A×B)
2'b00	MUL	Compute (A×B)

RSVD: Reserved field. Software should write zero to this field.

BTYPE: B matrix element type. This must be consistent with the B FSM setting.

Table 4-27. C_CONFIG Encodings

Encoding	Name	Description
3'b111	INT32	Signed word (32-bit) data. C element width will be 128 bits.
3'b110	Reserved	Undefined B matrix element type.
3'b101	INT16	Signed half word (16-bit) data. C element width will be 64 bits.
3'b100	INT8	Signed byte (8-bit) data. C element width will be 32 bits.
3'b011	UINT32	Unsigned word (32-bit) data. C element width will be 128 bits.
3'b010	Reserved	Undefined B matrix element type
3'b001	UINT16	Unsigned half word (16-bit) data. C element width will be 64 bits.
3'b000	UINT8	Unsigned byte (8-bit) data. C element width will be 32 bits.

RSVD: Reserved field. Software should write zero to this field.

ATYPE: A vector signed/unsigned control. This must be consistent with A FSM setting.

Table 4-28. C_CONFIG Encodings

Encoding	Name	Description
1'b1	SA	A vector elements are signed.
1'b0	UA	A vector elements are unsigned.

Table 4-29. Definition of Bus X_CONFIG[103:52]

Field:	RSVD	CSTART	COFFSET	CRRSTPER	CSWPER
Size:	7	1	8	8	28
Range:	103:97	96:96	95:88	87:80	79:52

Table 4-30. Definition of Bus X_CONFIG[51:0]

Field:	CSWPER	RSVD	CTYPE	RSVD	XTYPE	RSVD	SHIFT	RSVD	RE	RSVD	SAT	RSVD	ReLU
Size:	4	5	3	4	4	1	7	7	1	7	1	7	1
Range	: 51:48	47:43	42:40	39:36	35:32	31:31	30:24	23:17	16:16	15:9	8:8	7:1	0:0

RSVD: Reserved field. Software should write zero to this field.

CSTART: Initial C bank selection

COFFSET: C matrix row read address offset

CRRSTPER: C read row offset reset period

CSWPER: C read bank switch period

RSVD: Reserved field. Software should write zero to this field.

CTYPE: C matrix element type. This must be consistent with the B FSM setting.

Table 4-31. X_CONFIG Encodings

Encoding	Name	Description
3'b111	INT128	Signed 128-bit data
3'b110	Reserved	Undefined C matrix element type
3'b101	INT64	Signed 64-bit data
3'b100	INT32	Signed 32-bit data
3'b011	UINT128	Unsigned 128-bit data
3'b010	Reserved	Undefined C matrix element type
3'b001	UINT64	Unsigned 64-bit data
3'b000	UINT32	Unsigned 32-bit data

RSVD: Reserved field. Software should write zero to this field.

XTYPE: Transfer buffer element type. Not all combinations of CTYPE and XTYPE are supported.

Table 4-32. X_CONFIG Encodings

Encoding	Name	Description
4'b1111	Reserved	Undefined transfer buffer element type
4'b1110	Reserved	Undefined transfer buffer element type
4'b1101	INT128	Signed 128-bit data
4'b1100	INT64	Signed 64-bit data
4'b1011	INT32	Signed 32-bit data
4'b1010	Reserved	Undefined transfer buffer element type
4'b1001	INT16	Signed 16-bit data
4'b1000	INT8	Signed 8-bit data
4'b0111	Reserved	Undefined transfer buffer element type

Table 4-32. X_CONFIG Encodings (continued)

Encoding	Name	Description
4'b0110	Reserved	Undefined transfer buffer element type
4'b0101	UINT128	Unsigned 128-bit data
4'b0100	UINT64	Unsigned 64-bit data
4'b0011	UINT32	Unsigned 32-bit data
4'b0010	Reserved	Undefined transfer buffer element type
4'b0001	UINT16	Unsigned 16-bit data
4'b0000	UINT8	Unsigned 8-bit data

RSVD: Reserved field. Software should write zero to this field.

SHIFT: Right shift amount, signed or unsigned depending on CTYPE field. Right shift amounts greater than width of the type specified in the CTYPE field results in undefined behavior.

RSVD: Reserved field. Software should write zero to this field.

RE: Enable rounding via 1/2 LSB addition after shifting. Rounding is suppressed by hardware when the SHIFT field is 7'b0

RSVD: Reserved field. Software should write zero to this field.

SAT: Enable saturation in the transfer buffer element type after optional rounding. For predictable results ReLU should not be enabled with SAT.

RSVD: Reserved field. Software should write zero to this field.

ReLU: Enable Rectified Linear Units non-linearity after optional rounding. For predictable results, SAT should not be enabled with ReLU.

4.1.2.1.2 HWA_OFFSET

The HWA_OFFSET register is used by the A FSM for lookup table data values, and by the B FSM for data transfer operations that involve row or column offsets per input data element. The B FSM discussion describes how these offsets are used gives some examples.

The position in the offset register for specific rows or columns varies with the element size in the B matrix.

Lookup table fields and offset fields are interleaved to form a 32-bit pattern aligned on 32-bit boundaries. This simplifies HWA_OFFSET construction in different endian modes.

HWA_OFFSET may be observed through the transfer buffer after executing the HWAXFER instruction.

Table 4-33. Definition of Reg HWA_OFFSET[511:480]

Field:	A_LUT_VAL15	OFFSET63	OFFSET62	OFFSET61	OFFSET60
Size:	8	6	6	6	6
Range:	511:504	503:498	497:492	491:486	485:480

Table 4-34. Definition of Reg HWA_OFFSET[479:448]

Field:	A_LUT_VAL14	OFFSET59	OFFSET58	OFFSET57	OFFSET56
Size:	8	6	6	6	6
Range:	479:472	471:466	465:460	459:454	453:448

Table 4-35. Definition of Reg HWA_OFFSET[447:416]

Field:	A_LUT_VAL13	OFFSET55	OFFSET54	OFFSET53	OFFSET52
Size:	8	6	6	6	6
Range:	447:440	439:434	433:428	427:422	421:416

Table 4-36. Definition of Reg HWA_OFFSET[415:384]

Field:	A_LUT_VAL12	OFFSET51	OFFSET50	OFFSET49	OFFSET48
Size:	8	6	6	6	6

Table 4-36. Definition of Reg HWA_OFFSET[415:384] (continued)

Field:	A_LUT_VAL12	OFFSET51	OFFSET50	OFFSET49	OFFSET48
Range:	415:408	407:402	401:396	395:390	389:384

Table 4-37. Definition of Reg HWA_OFFSET[383:352]

Field:	A_LUT_VAL11	OFFSET47	OFFSET46	OFFSET45	OFFSET44
Size:	8	6	6	6	6
Range:	383:376	375:370	369:364	363:358	357:352

Table 4-38. Definition of Reg HWA_OFFSET[351:320]

Field:	A_LUT_VAL10	OFFSET43	OFFSET42	OFFSET41	OFFSET40
Size:	8	6	6	6	6
Range:	351:344	343:338	337:332	331:326	325:320

Table 4-39. Definition of Reg HWA_OFFSET[319:288]

Field:	A_LUT_VAL9	OFFSET39	OFFSET38	OFFSET37	OFFSET36
Size:	8	6	6	6	6
Range:	319:312	311:306	305:300	299:294	293:288

Table 4-40. Definition of Reg HWA_OFFSET[287:256]

Field:	A_LUT_VAL8	OFFSET35	OFFSET34	OFFSET33	OFFSET32
Size:	8	6	6	6	6
Range:	287:280	279:274	273:268	267:262	261:256

Table 4-41. Definition of Reg HWA_OFFSET[255:224]

Field:	A_LUT_VAL7	OFFSET31	OFFSET30	OFFSET29	OFFSET28
Size:	8	6	6	6	6
Range:	255:248	247:242	241:236	235:230	229:224

Table 4-42. Definition of Reg HWA_OFFSET[223:192]

Field:	A_LUT_VAL6	OFFSET27	OFFSET26	OFFSET25	OFFSET24
Size:	8	6	6	6	6
Range:	223:216	215:210	209:204	203:198	197:192

Table 4-43. Definition of Reg HWA_OFFSET[191:160]

Field:	A_LUT_VAL5	OFFSET23	OFFSET22	OFFSET21	OFFSET20
Size:	8	6	6	6	6
Range:	191:184	183:178	177:172	171:166	165:160

Table 4-44. Definition of Reg HWA_OFFSET[159:128]

Field:	A_LUT_VAL4	OFFSET19	OFFSET18	OFFSET17	OFFSET16
Size:	8	6	6	6	6
Range:	159:152	151:146	145:140	139:134	133:128

Table 4-45. Definition of Reg HWA_OFFSET[127:96]

Field:	A_LUT_VAL3	OFFSET15	OFFSET14	OFFSET13	OFFSET12
Size:	8	6	6	6	6
Range:	127:120	119:114	113:108	107:102	101:96

Table 4-46. Definition of Reg HWA_OFFSET[95:64]

Field:	A_LUT_VAL2	OFFSET11	OFFSET10	OFFSET9	OFFSET8
Size:	8	6	6	6	6
Range:	95:88	87:82	81:76	75:70	69:64

Table 4-47. Definition of Reg HWA_OFFSET[63:32]

Field:	A_LUT_VAL1	OFFSET7	OFFSET6	OFFSET5	OFFSET4
Size:	8	6	6	6	6
Range:	63:56	55:50	49:44	43:38	37:32

Table 4-48. Definition of Reg HWA_OFFSET[31:0]

Field:	A_LUT_VAL0	OFFSET3	OFFSET2	OFFSET1	OFFSET0
Size:	8	6	6	6	6
Range:	31:24	23:18	17:12	11:6	5:0

A_LUT_VAL15: A FSM LUT value for data input value 4'b1111

OFFSET63: Local offset 63 for (u)int8

OFFSET62: Local offset 62 for (u)int8, offset 31 for (u)int16

OFFSET61: Local offset 61 for (u)int8

OFFSET60: Local offset 60 for (u)int8, offset 30 for (u)int16, offset 15 for (u)int32

A_LUT_VAL14: A FSM LUT value for data input value 4'b1110

OFFSET59: Local offset 59 for (u)int8

OFFSET58: Local offset 58 for (u)int8, offset 29 for (u)int16

OFFSET57: Local offset 57 for (u)int8

OFFSET56: Local offset 56 for (u)int8, offset 28 for (u)int16, offset 14 for (u)int32

A_LUT_VAL13: A FSM LUT value for data input value 4'b1101

OFFSET55: Local offset 55 for (u)int8

OFFSET54: Local offset 54 for (u)int8, offset 27 for (u)int16

OFFSET53: Local offset 53 for (u)int8

OFFSET52: Local offset 52 for (u)int8, offset 26 for (u)int16, offset 13 for (u)int32

A_LUT_VAL12: A FSM LUT value for data input value 4'b1100

OFFSET51: Local offset 51 for (u)int8

OFFSET50: Local offset 50 for (u)int8, offset 25 for (u)int16

OFFSET49: Local offset 49 for (u)int8

OFFSET48: Local offset 48 for (u)int8, offset 24 for (u)int16, offset 12 for (u)int32

A_LUT_VAL11: A FSM LUT value for data input value 4'b1011

OFFSET47: Local offset 47 for (u)int8

OFFSET46: Local offset 46 for (u)int8, offset 23 for (u)int16

OFFSET45: Local offset 45 for (u)int8

OFFSET44: Local offset 44 for (u)int8, offset 22 for (u)int16, offset 11 for (u)int32

A_LUT_VAL10: A FSM LUT value for data input value 4'b1010

OFFSET43: Local offset 43 for (u)int8

OFFSET42: Local offset 42 for (u)int8, offset 21 for (u)int16

OFFSET41: Local offset 41 for (u)int8

OFFSET40: Local offset 40 for (u)int8, offset 20 for (u)int16, offset 10 for (u)int32

A_LUT_VAL9: A FSM LUT value for data input value 4'b1001

OFFSET39: Local offset 39 for (u)int8

OFFSET38: Local offset 38 for (u)int8, offset 19 for (u)int16

OFFSET37: Local offset 37 for (u)int8

OFFSET36: Local offset 36 for (u)int8, offset 18 for (u)int16, offset 9 for (u)int32

A_LUT_VAL8: A FSM LUT value for data input value 4'b1000

OFFSET35: Local offset 35 for (u)int8

OFFSET34: Local offset 34 for (u)int8, offset 17 for (u)int16

OFFSET33: Local offset 33 for (u)int8

OFFSET32: Local offset 32 for (u)int8, offset 16 for (u)int16, offset 8 for (u)int32

A_LUT_VAL7: A FSM LUT value for data input value 4'b0111

OFFSET31: Local offset 31 for (u)int8

OFFSET30: Local offset 30 for (u)int8, offset 15 for (u)int16

OFFSET29: Local offset 29 for (u)int8

OFFSET28: Local offset 28 for (u)int8, offset 14 for (u)int16, offset 7 for (u)int32

A_LUT_VAL6: A FSM LUT value for data input value 4'b0110

OFFSET27: Local offset 27 for (u)int8

OFFSET26: Local offset 26 for (u)int8, offset 13 for (u)int16

OFFSET25: Local offset 25 for (u)int8

OFFSET24: Local offset 24 for (u)int8, offset 12 for (u)int16, offset 6 for (u)int32

A_LUT_VAL5: A FSM LUT value for data input value 4'b0101

OFFSET23: Local offset 23 for (u)int8

OFFSET22: Local offset 22 for (u)int8, offset 11 for (u)int16

OFFSET21: Local offset 21 for (u)int8

OFFSET20: Local offset 20 for (u)int8, offset 10 for (u)int16, offset 5 for (u)int32

A_LUT_VAL4: A FSM LUT value for data input value 4'b0100

OFFSET19: Local offset 19 for (u)int8

OFFSET18: Local offset 18 for (u)int8, offset 9 for (u)int16

OFFSET17: Local offset 17 for (u)int8

OFFSET16: Local offset 16 for (u)int8, offset 8 for (u)int16, offset 4 for (u)int32

A_LUT_VAL3: A FSM LUT value for data input value 4'b0011

OFFSET15: Local offset 15 for (u)int8

OFFSET14: Local offset 14 for (u)int8, offset 7 for (u)int16

OFFSET13: Local offset 13 for (u)int8

OFFSET12: Local offset 12 for (u)int8, offset 6 for (u)int16, offset 3 for (u)int32

A_LUT_VAL2: A FSM LUT value for data input value 4'b0010

OFFSET11: Local offset 11 for (u)int8

OFFSET10: Local offset 10 for (u)int8, offset 5 for (u)int16

OFFSET9: Local offset 9 for (u)int8

OFFSET8: Local offset 8 for (u)int8, offset 4 for (u)int16, offset 2 for (u)int32

A_LUT_VAL1: A FSM LUT value for data input value 4'b0001

OFFSET7: Local offset 7 for (u)int8

OFFSET6: Local offset 6 for (u)int8, offset 3 for (u)int16

OFFSET5: Local offset 5 for (u)int8

OFFSET4: Local offset 4 for (u)int8, offset 2 for (u)int16, offset 1 for (u)int32

A_LUT_VAL0: A FSM LUT value for data input value 4'b0000

OFFSET3: Local offset 3 for (u)int8

OFFSET2: Local offset 2 for (u)int8, offset 1 for (u)int16

OFFSET1: Local offset 1 for (u)int8

OFFSET0: Local offset 0 for (u)int8, (u)int16, (u)int32

4.1.2.1.3 HWA_STATUS

HWA_STATUS is a read-only register that provides visibility into the current state of the control state machines in the MMA unit. HWA_STATUS may be observed through the transfer buffer after executing the HWAXFER instruction. There is a minimum of 2 clock cycle [subject to validation and change] delay between any FSM status updates and HWA_STATUS updates to reflect the current state.

The FirstErrorCode field of HWA_STATUS contains code that describes the earliest-observed error from by hardware in the MMA unit. The HWAOPEN instruction initializes this field to all zeros. When an error is detected, a non-zero value is written into the FirstErrorCode field. When the FirstErrorCode field contains a non-zero value, subsequent errors do not update the value: only the first-detected error cause is logged. HWA_STATUS also contains a LastErrorCode field, which captures the error code for any error condition that is observed. It is updated regardless of the contents of the FirstErrorCode field. The encoding of the FirstErrorCode and LastErrorCode fields is defined in the theory of operation section. Information in FirstErrorCode field is duplicated in the C7x exception reporting registers once a MMA exception is detected.

Table 4-49. Definition of Reg HWA_STATUS[511:256]

Field:	FirstErrorCode	LastErrorCode	RSVD	RSVD	X_STATUS	C_STATUS
Size:	6	6	6	110	80	48
Range:	511:506	505:500	499:494	493:384	383:304	303:256

Table 4-50. Definition of Reg HWA_STATUS[255:0]

Field:	C_STATUS	B_STATUS	A_STATUS
Size:	192	56	8
Range:	255:64	63:8	7:0

FirstErrorCode: Sticky error code.

LastErrorCode: Last-detected error code, same encoding as the FirstErrorCode field.

RSVD: Reserved field. Software should not assume a specific value will be returned by hardware when this field is read. In earlier specifications this field was X_OCCUPANCY.

RSVD: Reserved field. Software should not assume a specific value will be returned by hardware when this field is read.

X_STATUS: X FSM status field. See X_STATUS bus definition.

C_STATUS: C FSM status field. See C_STATUS bus definition.

B_STATUS: B FSM status field. See B_STATUS bus definition.

A_STATUS: A FSM status field. See A_STATUS bus definition.

Table 4-51. Definition of Bus A_STATUS[7:0]

Field:	RSVD	ERROR	HALTED
Size:	6	1	1
Range:	7:2	1:1	0:0

RSVD: Reserved field. Software should not assume a specific value will be returned by hardware when this field is read.

ERROR: set when the A FSM has detected an error condition

HALTED: set when the A FSM is halted for any reason

Table 4-52. Definition of Bus B_STATUS[55:0]

Field:	RSVD	BCNT	RSVD	BBANK	ERROR	HALTED	BRSTCNT	BSWCNT
Size:	2	6	5	1	1	1	8	32
Range:	55:54	53:48	47:43	42:42	41:41	40:40	39:32	31:0

RSVD: Reserved field. Software should not assume a specific value will be returned by hardware when this field is read.

BCNT: current B row/column storage counter

RSVD: Reserved field. Software should not assume a specific value will be returned by hardware when this field is read.

BBANK: current B bank

ERROR: set when the B FSM has detected an error condition

HALTED: set when the B FSM is halted for any reason

BRSTCNT: Remaining iterations in B offset period

BSWCNT: Remaining iterations in B bank swap period

Table 4-53. Definition of Bus C_STATUS[239:210]

Field	CLROW	RSVD	CWROW	RSVD	CRROW	RSVD	CLBANK	CWBANK	CRBANK	BBANK	OP
Size	8	2	6	2	6	1	1	1	1	1	1
Range	239:232	231:230	229:224	223:222	221:216	215:215	214:214	213:213	212:212	211:211	210:210

Table 4-54. Definition of Bus C_STATUS[209:180]

Field:	ERROR	HALTED	CLRSTCNT	CWRSTCNT
Size:	1	1	8	20
Range:	209:209	208:208	207:200	199:180

Table 4-55. Table 38: Definition of Bus C_STATUS[179:150]

Field:	CWRSTCNT	CRRSTCNT
Size:	12	18
Range:	179:168	167:150

Table 4-56. Definition of Bus C_STATUS[149:120]

Field:	CRRSTCNT	CLSWCNT	CWSWCNT
Size:	14	8	8
Range:	149:136	135:128	127:120

Table 4-57. Definition of Bus C_STATUS[119:90]

Field:	CWSWCNT	CRSWCNT
Size:	24	6

Table 4-57. Definition of Bus C_STATUS[119:90] (continued)

Field:	CWSWCNT	CRSWCNT
Range:	119:96	95:90

Table 4-58. Definition of Bus C_STATUS[89:60]

Field:	CRSWCNT	BSWCNT
Size:	26	4
Range:	89:64	63:60

Table 4-59. Definition of Bus C_STATUS[59:30]

Field:	BSWCNT	OPCNT
Size:	28	2
Range:	59:32	31:30

Table 4-60. Definition of Bus C_STATUS[29:0]

Field:	OPCNT
Size:	30
Range:	29:0

CLROW: Current C operand load (HWALD*) write row or sub-row depending on X1 or X4 scaling

RSVD: Reserved field. Software should not assume a specific value will be returned by hardware when this field is read.

CWROW: Current C computation write row

RSVD: Reserved field. Software should not assume a specific value will be returned by hardware when this field is read.

CRROW: Current C read row

RSVD: Reserved field. Software should not assume a specific value will be returned by hardware when this field is read.

CLBANK: Current C operand write bank

CWBANK: Current C computation write bank

CRBANK: Current C read bank

BBANK: Current B bank for the matrix calculation OP: Current operation selection

ERROR: Set when the C FSM has detected an error condition HALTED: Set when the C FSM is halted for any reason

CLRSTCNT: Remaining iterations in C operand load (HWALD*) write row offset reset period

CWRSTCNT: Remaining iterations in C computation write row offset reset period

CRRSTCNT: Remaining iterations in C computation read row offset reset period

CLSWCNT: Remaining iterations in C operand load (HWALD*) write bank swap period

CWSWCNT: Remaining iterations in C computation write bank swap period

CRSWCNT: Remaining iterations in C computation read bank swap period

BSWCNT: Remaining iterations in B bank swap period

OPCNT: Remaining iterations in current operation swap period

Table 4-61. Definition of Bus X_STATUS[79:0]

Field: C	RROW	RSVD	CBANK	ERROR	HALTED	RSVD	CRRSTCNT	CSWCNT
Size:	8	5	1	1	1	24	8	32

Table 4-61. Definition of Bus X_STATUS[79:0] (continued)

Field: C	RROW	RSVD	CBANK	ERROR	HALTED	RSVD	CRRSTCNT	CSWCNT
Range	: 79:72	71:67	66:66	65:65	64:64	63:40	39:32	31:0

CRROW: Current C read row

RSVD: Reserved field. Software should not assume a specific value will be returned by hardware when this field is read.

CBANK: Current C bank ERROR: Set when the X FSM has detected an error condition

HALTED: Set when the X FSM is halted for any reason

RSVD: Reserved field. Software should not assume a specific value will be returned by hardware when this field is read.

CRRSTCNT: Remaining iterations in C read row offset period

CSWCNT: Remaining iterations in C read bank swap period

4.1.2.1.4 HWA_BUSY

HWA_BUSY is a read-only register that may be observed through the transfer buffer with the HWAXFER instruction or EDI operations. The HWA_BUSY register contains a field named BUSY, which is a thermometer-encoded array of bits that represent the valid flags on the transfer buffer FIFO entries. The oldest entry is at the lowest-numbered bit position.

Table 4-62. Definition of Reg HWA_BUSY[511:0]

Field:	Reserved	BUSY
Size:	488	24
Range:	511:24	23:0

Reserved: Reserved field. Software should not expect any specific values in this field.

BUSY: Transfer buffer FIFO entry valid state bits. In a debug setting, the HWA_BUSY register can be used to determine which transfer buffer FIFO entries are holding results that have not be committed to C7x registers.

4.1.2.1.5 Transfer Buffer Registers

The MMA transfer buffer is part of a high-bandwidth read interface used to move MMA results, status, and control register state to C7x vector registers. There is no MMA register access with the MVC instruction.

The MMA transfer buffer is a hardware-managed FIFO used for transferring data from the C matrix and internal control and status registers. Each entry is 512 bits wide. The FIFO is non-blocking but can generate underflow and overflow exceptions. The number of FIFO elements currently occupied is visible in the X_OCCUPANCY field of the HWA_STATUS register. The FIFO is cleared when the HWACLOSE instruction or HWAOPEN full update instruction is executed.

Data is pushed into the transfer buffer FIFO by the HWAXFER and HWAOPXFER instructions.

4.1.2.1.6 EDI-Mapped Registers

Important MMA internal state registers are readable in the EDI address space. The MMA architecture does not support writes to MMA registers from the EDI address space; however, there is one writable EDI control register that modifies the EDI interface behavior.

Table 4-63. Definition of Bus MMA_EDI_ADDRESS[35:0]

Field:	Reserved	ROW	MMA_RANGE	MMA_ARRAY	Reserved	COLUMN
Size:	8	8	4	8	3	5
Range:	35:28	27:20	19:16	15:8	7:5	4:0

Reserved: Reserved field. Software should write zeros to this field for defined behavior.

ROW: MMA EDI subspace-specific row address

MMA_RANGE: MMA EDI address offset in C7x EDI address space

Table 4-64. MMA_EDI_ADDRESS Encodings

Encoding	Name	Description
4'b1001	MMA_EDI	EDI Control register subspace constant

MMA_ARRAY: MMA EDI address array-specific subspace

Table 4-65. MMA_EDI_ADDRESS Encodings

Encoding	Name	Description
8'b00000000	A_OPERAND	Multiplier A operand
8'b00000001	Reserved	Reserved encoding. For defined behavior, software should not use this encoding.
8'b00000010	B0_OPERAND	B bank 0 multiplier operand
8'b00000011	B1_OPERAND	B bank 1 multiplier operand
8'b00000100	C0_STORAGE	C bank 0 accumulator storage
8'b00000101	C1_STORAGE	C bank 1 accumulator storage
8'b00000110	X_STORAGE	Transfer fifo storage
8'b00000111	Reserved	Reserved encoding. For defined behavior, software should not use this encoding.
8'b00001000	CONTROL	EDI Control register subspace
8'b00001001	Reserved	Reserved encoding. For defined behavior, software should not use this encoding.
8'b00001010	Reserved	Reserved encoding. For defined behavior, software should not use this encoding.
8'b00001011	Reserved	Reserved encoding. For defined behavior, software should not use this encoding.
8'b00001100	Reserved	Reserved encoding. For defined behavior, software should not use this encoding.
8'b00001101	Reserved	Reserved encoding. For defined behavior, software should not use this encoding.
8'b00001110	Reserved	Reserved encoding. For defined behavior, software should not use this encoding.
8'b00001111	Reserved	Reserved encoding. For defined behavior, software should not use this encoding.

Reserved: Reserved field. Software should write zeros to this field for defined behavior.

COLUMN: MMA EDI subspace-specific 8-byte granular column address

Table 4-66. Definition of Bus MMA_EDI_X_STORAGE_SUBSPACE[13:0]

Field:	Reserved	XREG	COL
Size:	6	5	3
Range:	13:8	7:3	2:0

Reserved: Reserved field. Software should write zeros to this field for defined behavior.

XREG: Transfer Buffer address

Table 4-67. MMA_EDI_X_STORAGE_SUBSPACE Encodings

Encoding	Name	Description
5'b00000	X0	Transfer buffer element 0
5'b00001	X1	Transfer buffer element 1
5'b00010	X2	Transfer buffer element 2
5'b00011	X3	Transfer buffer element 3
5'b00100	X4	Transfer buffer element 4
5'b00101	X5	Transfer buffer element 5
5'b00110	X6	Transfer buffer element 6
5'b00111	X7	Transfer buffer element 7
5'b01000	X8	Transfer buffer element 8
5'b01001	X9	Transfer buffer element 9
5'b01010	X10	Transfer buffer element 10

Table 4-67. MMA_EDI_X_STORAGE_SUBSPACE Encodings (continued)

Encoding	Name	Description
5'b01011	X11	Transfer buffer element 11
5'b01100	X12	Transfer buffer element 12
5'b01101	X13	Transfer buffer element 13
5'b01110	X14	Transfer buffer element 14
5'b01111	X15	Transfer buffer element 15
5'b10000	X16	Transfer buffer element 16
5'b10001	X17	Transfer buffer element 17
5'b10010	X18	Transfer buffer element 18
5'b10011	X19	Transfer buffer element 19
5'b10100	X20	Transfer buffer element 20
5'b10101	X21	Transfer buffer element 21
5'b10110	X22	Transfer buffer element 22
5'b10111	X23	Transfer buffer element 23
5'b11000	HWA_CONFIG	HWA_CONFIG control register
5'b11001	HWA_OFFSET	HWA_OFFSET control register
5'b11010	HWA_STATUS	HWA_STATUS control register
5'b11011	HWA_BUSY	HWA_BUSY control register
5'b11100	RSVD0	Reserved address. Results are not defined when reading this address.
5'b11101	RSVD1	Reserved address. Results are not defined when reading this address.
5'b11110	RSVD2	Reserved address. Results are not defined when reading this address.
5'b11111	RSVD3	Reserved address. Results are not defined when reading this address.

COL: 64-bit field in the selected register

Table 4-68. Definition of Bus MMA_EDI_CONTROL_SUBSPACE[13:0]

Field:	Reserved	REGISTER
Size:	13	1
Range:	13:1	0:0

Reserved: Reserved field. Software should write zeros to this field for defined behavior.

REGISTER: Control register selection

4.1.2.1.6.1 EDI B Matrix Accesses

The mapping of B matrix data to B matrix storage elements distributed in the MMA multiplier array depends on the B element type and matrix loading order (row-major or column major).

B matrix bank 0 and 1 accesses through EDI are always performed as if the current matrix operation has 8-bit elements loaded in row-major order, regardless of the operand sizes specified by the HWA_CONFIG register. B matrix data is always returned as 8 packed 8-bit values in a 64-bit value.

Host software may re-assemble the matrix elements from the 8-bit view of the data into the types consistent with the current operation. The current operation's element sizes can be obtained by reading the HWA_CONFIG register through a preliminary EDI access.

4.1.2.1.6.2 EDI C Matrix Accesses

Unlike the A and B operand data, the C matrix read accesses are always performed on a contiguous 2048 value and return packed 64-bit values. Host software may re-assemble the matrix elements from the 2048-bit view of the data into the types consistent with the current operation.

4.1.2.1.6.3 EDI Control Register

In typical debug operations, there is no need to modify the behavior of the MMA EDI controller. The MMA EDI control register provides some mechanisms that are useful in special circumstances.

Table 4-69. Definition of Reg MMA_EDI_CONTROL_REGISTER[63:0]

Field:	Reserved	TC	ARB_MODE	TIMEOUT
Size:	50	1	1	12
Range:	63:14	13:13	12:12	11:0

Reserved: Reserved field. Software should write zeros to this field for defined behavior.

TC: Timeout control

Table 4-70. MMA_EDI_CONTROL_REGISTER Encodings

Encoding	Name	Description
1'b0	TIMEOUT_DISABLED	MMA EDI transaction timeouts are disabled
1'b1	TIMEOUT_ENABLED	MMA EDI transaction timeouts are enabled. This is the default value after reset.

ARB_MODE: Arbitration Mode

Table 4-71. MMA_EDI_CONTROL_REGISTER Encodings

Encoding	Name	Description
1'b0	GOOD	MMA EDI transactions arbitrate for access to state shared by mission- mode operations. This is the default value after reset.
1'b1	EVIL	MMA EDI transactions ignore mission mode activity and immediately access state shared by mission-mode operations. Results computed by mission-mode operations may be corrupted.

4.1.2.2 A Finite State Machine

4.1.2.2.1 A FSM Inputs

The A FSM advances to the next state when any MMA instruction is executed that loads A vector data. These instructions are any instruction matching the pattern HWALD(A|AB).*. In addition, the A finite state machine responds to the HWAOPEN instruction by initializing its state register and counters.

4.1.2.2.2 A FSM Outputs

The primary outputs of the A FSM are data pre-processing controls and the write enables for the memory that holds the A vector data.

4.1.2.2.3 A FSM Operation

The A FSM implements the state transitions, as shown in Figure 4-3.

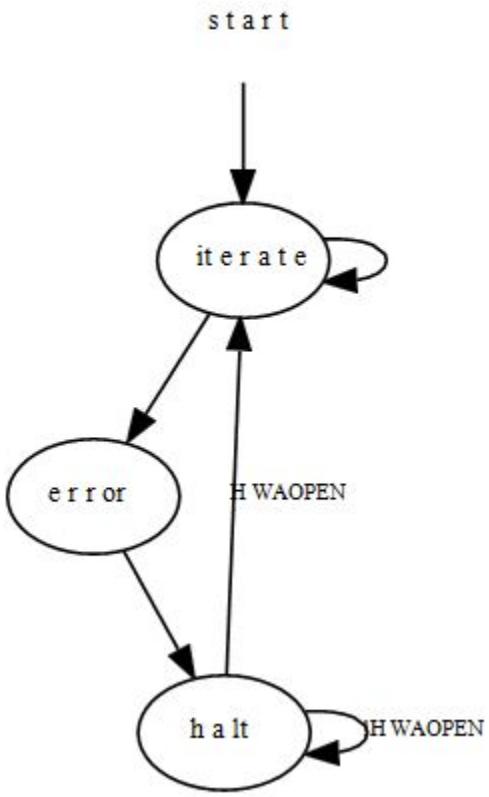


Figure 4-3. State Transitions

The operation of the A FSM is parametrized the A_CONFIG field of the HWA_CONFIG register. Software may update the state of the configuration register while the A FSM is iterating.

While the A FSM is in the halted or error states, any input data is ignored.

The following pseudocode summarizes the A FSM operation:

```
start:
while(1) {
wait_for_issue_cycle() ; // block until next A_FSM HWA instruction Move A operand into A storage
}
```

4.1.2.2.4 LUT Expansion

The HWA_CONFIG.A_CONFIG field contains a lookup table enable, and the HWA_OFFSET.A_LUT* fields contain lookup table data used for the optional expansion of 4-bit operands to 8-bit operands. When the lookup table is enabled, the least-significant 4-bit field of every 8-byte aligned field is passed through a lookup table, and the resulting 8-bit value replaces the entire 8-bit field. This is illustrated in Figure 4-4.

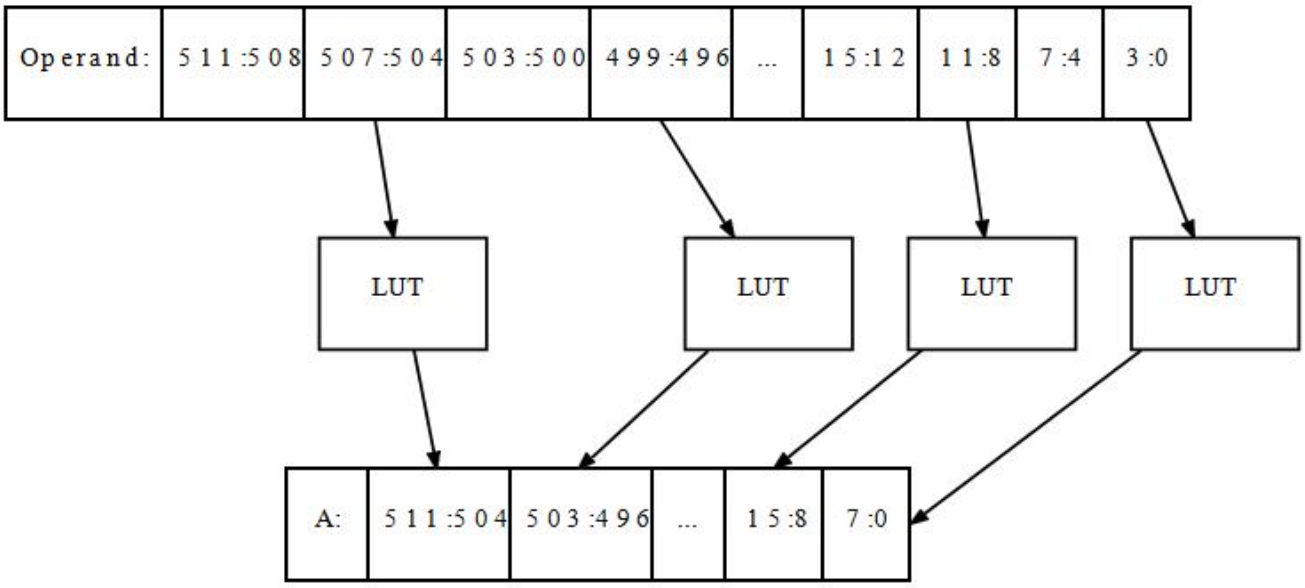


Figure 4-4. LUT Expansion

See also:

HWA_CONFIG register definition

HWA_OFFSET register definition

4.1.2.3 B Finite State Machine

4.1.2.3.1 B FSM Inputs

The B FSM advances to the next state when any MMA instruction is executed that loads B matrix data. These instructions are any instruction matching the pattern HWALD(B|AB|CB).* . In addition, the B FSM responds to the HWAOPEN instruction by initializing its state register and counters.

4.1.2.3.2 B FSM Outputs

The primary outputs of the B FSM are the write enables for the memory that holds the B matrix data.

4.1.2.3.3 B FSM Operation

The B FSM implements the state transitions, as shown in .Figure 4-5

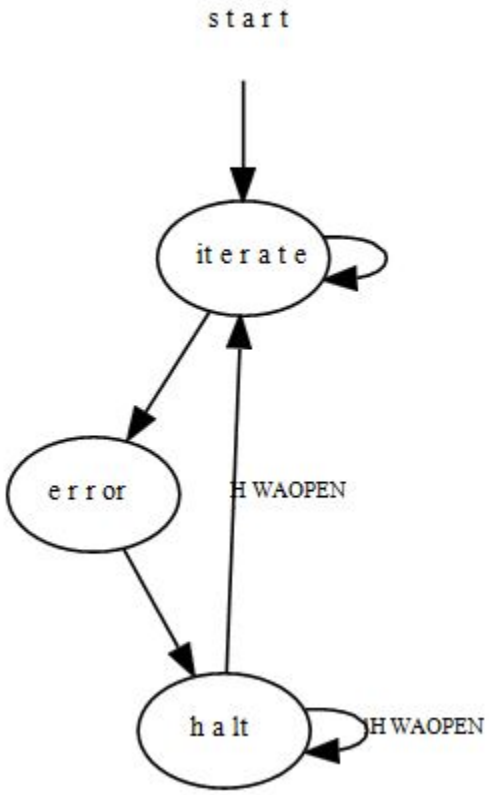


Figure 4-5. State Transitions

The operation of the B FSM is parametrized by the B_CONFIG field of the HWA_CONFIG register. Software may update the state of the configurations while the B FSM is iterating.

While the FSM is in the halted or error states, any input data is ignored.

The following pseudocode summarizes the B FSM operation:

```
// assume B storage is B[BANKS][ROWS][COLS] start:
HWA_STATUS.B_STATUS.BCNT    = HWA_CONFIG.B_CONFIG.BOFFSET HWA_STATUS.B_STATUS.BBANK
= HWA_CONFIG.B_CONFIG.BSTART HWA_STATUS.B_STATUS.BRSTCNT = HWA_CONFIG.B_CONFIG.BRSTPER
HWA_STATUS.B_STATUS.BSWPER    = HWA_CONFIG.B_CONFIG.BSWPPER
while(1) {
wait_for_issue_cycle() ; // block until next B_FSM HWA instruction
// illustrating only 8-bit element case if ( ORDER == ROW ) {
// row-major order (no-transpose) for ( c = 0 ; c < 64 ; r++ ) {
bc = (HWA_OFFSET[r]+HWA_STATUS.BCNT)%COLS B[HWA_STATUS.B_STATUS.BBANK][r][bc] = B_INPUT[c]
}
}
HWA_STATUS.B_STATUS.BCNT += 1 ; // update B storage counter if ( ! --HWA_STATUS.B_STATUS.BRSTCNT ) {

// reset storage offset
HWA_STATUS.B_STATUS.BCNT    = HWA_CONFIG.B_CONFIG.BOFFSET HWA_STATUS.B_STATUS.BRSTCNT =
HWA_CONFIG.B_CONFIG.BRSTPER
}
if ( ! --HWA_STATUS.B_STATUS.BSWPER ) { HWA_STATUS.B_STATUS.BBANK    += 1
HWA_STATUS.B_STATUS.BSWPER    = HWA_CONFIG.B_CONFIG.BSWPPER
}
}
```

4.1.2.3.4 B Storage Location

Compared to the A vector, C matrix, and transfer buffer, the B load addressing is more complicated. When B matrix row data is loaded into B matrix storage there is a set of address calculations that determine where the data is deposited. That calculation depends on the B storage bank, matrix loading order, the current storage offset, the element size, and the individual row or column offsets. The storage counter can be initialized

to an arbitrary value through the HWA_CONFIG.B_CONFIG.BOFFSET register. The B_STATUS field of the HWA_STATUS register provides observability to the current bank and storage counter computed by the FSM.

Row-major, 8-bit data: The HWA_OFFSET register fields specify per-column offsets that are added to the storage counter r that is incremented by the B FSM. For 8-bit element n, the matrix location where the element will be written is: B[BBANK][(r+HWA_OFFSETn)%64][n].

Row-major, 16-bit data: The HWA_OFFSET register fields specify per-column offsets that are added to the storage counter r that is incremented by the B FSM. For 16-bit element n, the matrix location where the element will be written is: B[BBANK][(r+HWA_OFFSET2*n)%32][n].

Row-major, 32-bit data: The HWA_OFFSET register fields specify per-column offsets that are added to the storage counter r that is incremented by the B FSM. For 32-bit element n, the matrix location where the element will be written is: B[BBANK][(r+HWA_OFFSET4*n)%16][n].

Column-major, 8-bit data: The HWA_OFFSET register fields specify per-column offsets that are added to the storage counter r that is incremented by the B FSM. For 8-bit element n, the matrix location where the element will be written is: B[BBANK][n][(r+HWA_OFFSETn)%64].

Column-major, 16-bit data: The HWA_OFFSET register fields specify per-column offsets that are added to the storage counter r that is incremented by the B FSM. For 16-bit element n, the matrix location where the element will be written is: B[BBANK][n][(r+HWA_OFFSET2*n)%32].

Column-major, 32-bit data: The HWA_OFFSET register fields specify per-column offsets that are added to the storage counter r that is incremented by the B FSM. For 32-bit element n, the matrix location where the element will be written is: B[BBANK][n][(r+HWA_OFFSET4*n)%16].

4.1.2.3.5 B Storage Location Examples

In the following examples, the displayed element order is not the in-memory element order, but is the order consistent with mathematical conventions. As discussed earlier, element zero is located at the lowest address, which is the LSB of 512-bit vectors in the C7x definition.

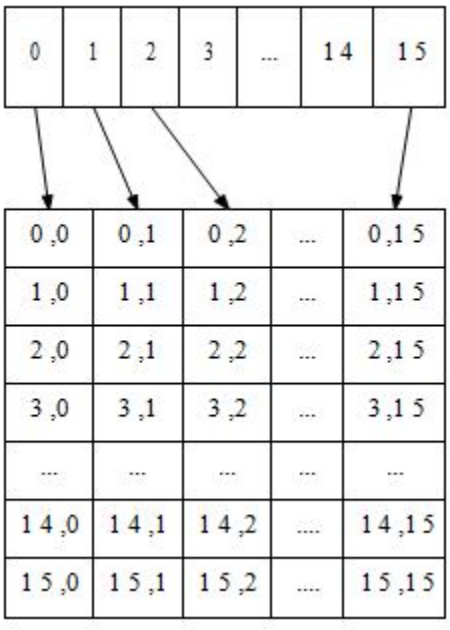


Figure 4-6. Row-Major with 32-bit Data and HWA_OFFSET Values Equal to [0,0,0...0], Initial Storage Offset Equal 0, First Iteration

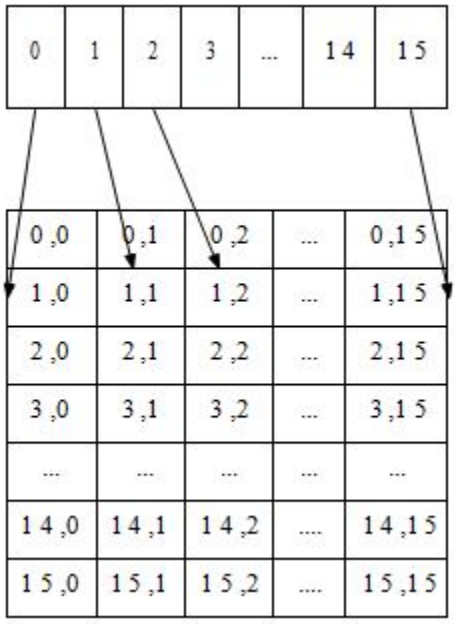


Figure 4-7. Row-Major with 32-bit Data and HWA_OFFSET Values Equal to [0,0,0...0], Initial Storage Offset Equal 0, Second Iteration

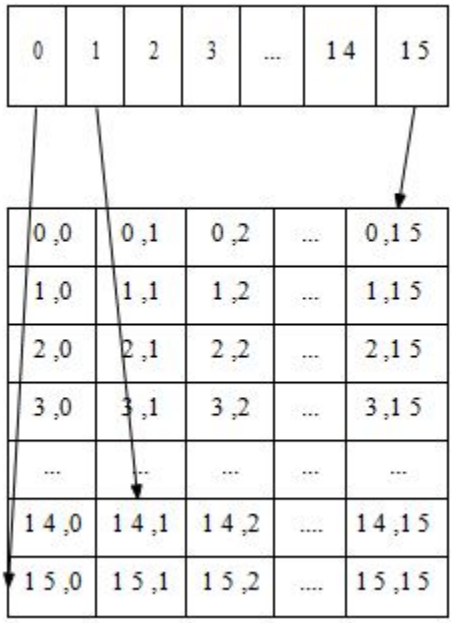


Figure 4-8. Row-Major with 32-bit Data and HWA_OFFSET Values Equal to [15,14,13...0], Initial Storage Offset Equal 0, First Iteration

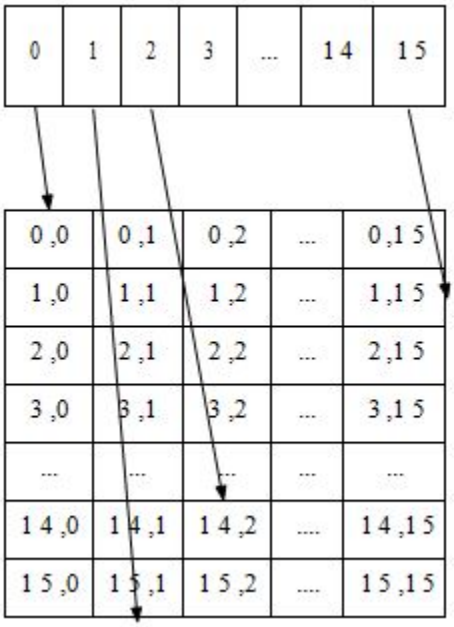


Figure 4-9. Row-Major with 32-bit Data and HWA_OFFSET Values Equal to [15,14,13...0], Initial Storage Offset Equal 0, Second Iteration

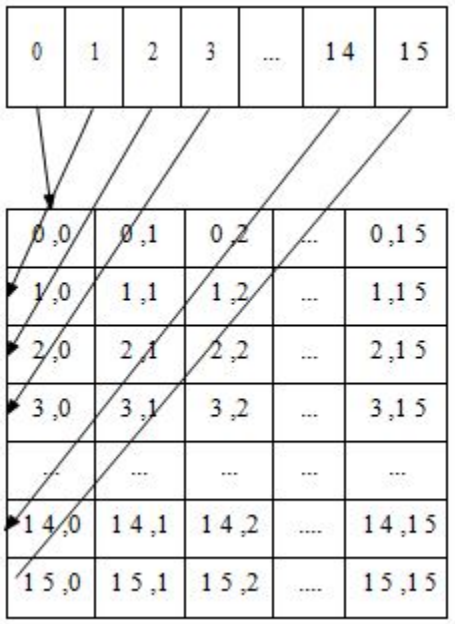


Figure 4-10. Column-Major with 32-bit Data and HWA_OFFSET Values Equal to [0,0,0...0], Initial Storage Offset Equal 0, First Iteration

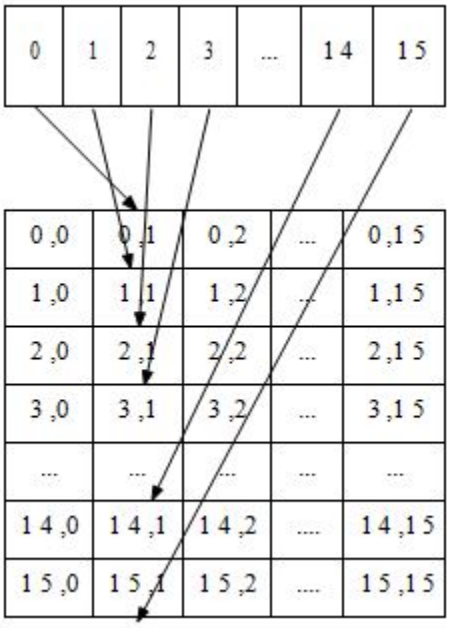


Figure 4-11. Column-Major with 32-bit Data and HWA_OFFSET Values Equal to [0,0,0...0], Initial Storage Offset Equal 0, Second Iteration

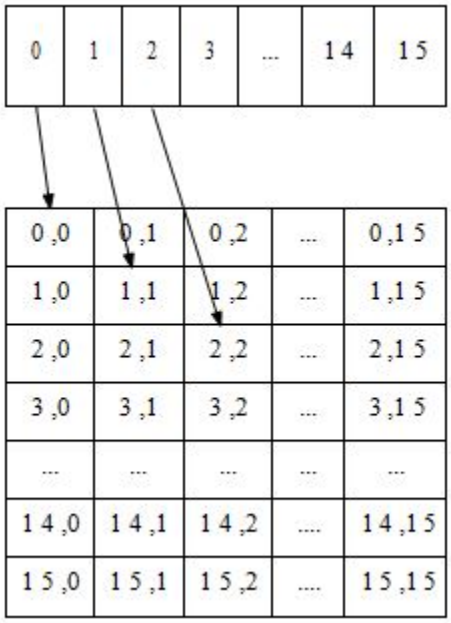


Figure 4-12. Column-Major with 32-bit Data and HWA_OFFSET Values Equal to [0,1,2,3..15], Initial Storage Offset Equal 0, First Iteration

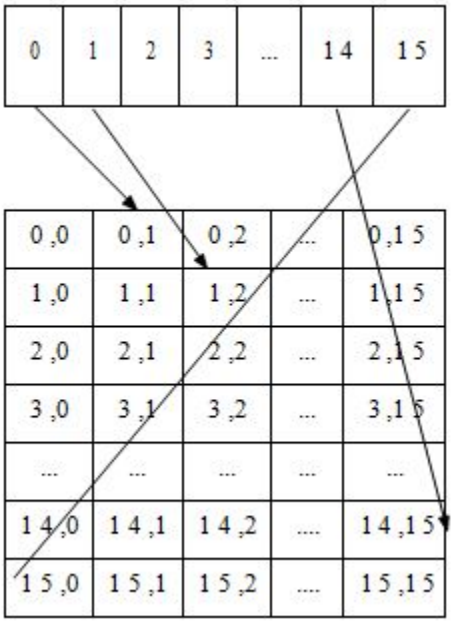


Figure 4-13. Column-Major with 32-bit Data and HWA_OFFSET Values Equal to [0,1,2,3..15], Initial Storage Offset Equal 0, Second Iteration

See also:

Theory of operation

HWA_CONFIG register definition

HWA_OFFSET register definition

4.1.2.4 C Finite State Machine

4.1.2.4.1 C Finite State Machine Inputs

The C FSM advances to the next state when any MMA instruction is executed that loads C matrix data. These instructions are any instruction matching the pattern HWALD(C|BC) of HWA.*(OP).*. In addition, the C finite state machine responds to the HWAOPEN instruction by initializing its state register and counters.

4.1.2.4.2 C FSM Outputs

The primary outputs of the C FSM are the bank selection for the memory that holds the B matrix data, the address and read enables for the memory that holds the C matrix data, the address and write enables for the memory that holds the C matrix data, and the controls that define the calculation to be performed.

4.1.2.4.3 C FSM Operation

The C FSM implements the state transitions, as shown in Figure 4-14.

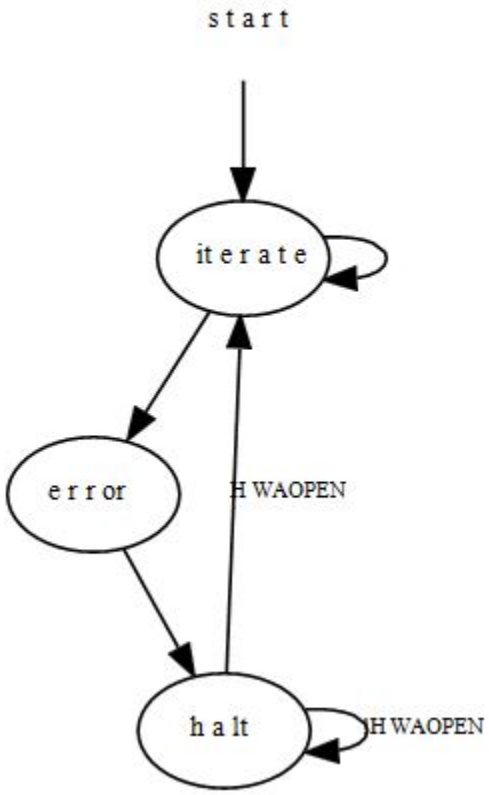


Figure 4-14. State Transitions

The operation of the C FSM is parametrized by the C_CONFIG field of the HWA_CONFIG register. Software may update the state of the configurations while the C FSM is iterating.

While the FSM is in the halted or error states, any input data is ignored.

The following pseudocode summarizes the C FSM operation:

```
// assume B storage is B[B BANKS][ROWS][COLS]
// assume C storage is C[C BANKS][ROWS][COLS] restart:
// initialize B bank, C banks, C read/write offsets, counters HWA_STATUS.C_STATUS.BBANK
= HWA_CONFIG.C_CONFIG.BSTART    ; HWA_STATUS.C_STATUS.OP    = HWA_CONFIG.C_CONFIG.OPSTART    ;
HWA_STATUS.C_STATUS.CRBANK    = HWA_CONFIG.C_CONFIG.CRSTART    ; HWA_STATUS.C_STATUS.CWBANK    =
HWA_CONFIG.C_CONFIG.CWSTART    ; HWA_STATUS.C_STATUS.CLBANK    = HWA_CONFIG.C_CONFIG.CLSTART    ;
HWA_STATUS.C_STATUS.CRROW    = HWA_CONFIG.C_CONFIG.CROFFSET ; HWA_STATUS.C_STATUS.CWROW
= HWA_CONFIG.C_CONFIG.CWOFFSET ; HWA_STATUS.C_STATUS.CLROW    = HWA_CONFIG.C_CONFIG.CLOFFSET
<<(4*(LDDST==X1));
HWA_STATUS.C_STATUS.BSWCNT    =    HWA_CONFIG.C_CONFIG.BSWPER    ;
HWA_STATUS.C_STATUS.OP0CNT    =    HWA_CONFIG.C_CONFIG.OP0PER    ;
HWA_STATUS.C_STATUS.OP1CNT    =    HWA_CONFIG.C_CONFIG.OP1PER    ;

HWA_STATUS.C_STATUS.CRSWCNT = HWA_CONFIG.C_CONFIG.CRRSTCNT ; HWA_STATUS.C_STATUS.CWSWCNT =
HWA_CONFIG.C_CONFIG.CWRSTCNT ; HWA_STATUS.C_STATUS.CLSWCNT = HWA_CONFIG.C_CONFIG.CLRSTCNT ;
for ( i = 0 ; i < HWA_CONFIG.C_CONFIG.DELAY ; i++ ) {
nop
wait_for_issue_cycle() ; // block until next C_FSM instruction
}
while( 1 ) {
wait_for_issue_cycle() ; // block until next C_FSM instruction if ( HWAOP* instruction ) {
wc = HWA_STATUS.C_STATUS.CWBANK % C BANKS rc = HWA_STATUS.C_STATUS.CRBANK % C BANKS bb =
HWA_STATUS.C_STATUS.BBANK % B BANKS wr = HWA_STATUS.C_STATUS.CWROW % ROWS
rr = HWA_STATUS.C_STATUS.CRROW    % ROWS for( i = 0 ; i < ELEMENTS ; i++ ) {
op = (HWA_STATUS.C_STATUS.OP%2) ?
HWA_CONFIG.OPERATION1 : HWA_CONFIG.OPERATION0    ;
C[wc][wr][i] = op( A x B[bb] , C[rc][rr][i] ) ;
}
HWA_STATUS.C_STATUS.CWROW += 1 ; // update C write counter HWA_STATUS.C_STATUS.CRROW += 1 ; //
```

```
update C read counter
if ( HWAOP* instruction ) {
if ( HWA_STATUS.C_STATUS.OP == 0 ) {
if ( ! --HWA_STATUS.C_STATUS.OP0CNT && HWA_CONFIG.C_CONFIG.OP1PER ) {
// change operation selector HWA_STATUS.C_STATUS.OP = 1
HWA_STATUS.C_STATUS.OP1CNT = HWA_CONFIG.C_CONFIG.OP1PER
}
} else {
if ( ! --HWA_STATUS.C_STATUS.OP1CNT && HWA_CONFIG.C_CONFIG.OP0PER ) {
// change operation selector HWA_STATUS.C_STATUS.OP = 0
HWA_STATUS.C_STATUS.OP0CNT = HWA_CONFIG.C_CONFIG.OP0PER
}
}
}
if ( ! --HWA_STATUS.C_STATUS.BSWCNT ) {
// swap B bank HWA_STATUS.C_STATUS.BBANK += 1
HWA_STATUS.C_STATUS.BSWCNT = HWA_CONFIG.C_CONFIG.BSWPER
}
if ( ! --HWA_STATUS.C_STATUS.CRSWCNT ) {
// swap C read bank HWA_STATUS.C_STATUS.CRBANK += 1
HWA_STATUS.C_STATUS.CRSWCNT = HWA_CONFIG.C_CONFIG.CRSWPER
}
if ( ! --HWA_STATUS.C_STATUS.CWSWCNT ) {
// swap C write bank HWA_STATUS.C_STATUS.CWBANK += 1
HWA_STATUS.C_STATUS.CWSWCNT = HWA_CONFIG.C_CONFIG.CWSPER
}
if ( ! --HWA_STATUS.C_STATUS.CRRSTCNT ) {
// reset C read row counter

HWA_STATUS.C_STATUS.CRROW    = HWA_CONFIG.C_CONFIG.CROFFSET HWA_STATUS.C_STATUS.CRRSTCNT =
HWA_CONFIG.C_CONFIG.CRRSTPER
}
if ( ! --HWA_STATUS.C_STATUS.CWRSTCNT ) {
// reset C write row counter
HWA_STATUS.C_STATUS.CWROW    = HWA_CONFIG.C_CONFIG.CWOFFSET HWA_STATUS.C_STATUS.CWRSTCNT =
HWA_CONFIG.C_CONFIG.CWRSTPER
}
if ( HWALD* instruction ) {
wc = HWA_STATUS.C_STATUS.CLBANK % C BANKS wr = HWA_STATUS.C_STATUS.CLROW    % ROWS for( i = 0 ; i <
ELEMENTS ; i++ ) {
C[wc][wr][i] = HWALD*C* operand element i
}
HWA_STATUS.C_STATUS.CLROW += 1 ; // update C write counter if ( ! --HWA_STATUS.C_STATUS.CLSWCNT ) {
// reset C operand load bank
HWA_STATUS.C_STATUS.CLBANK    = HWA_CONFIG.C_CONFIG.CLSTART HWA_STATUS.C_STATUS.CLRSTCNT =
HWA_CONFIG.C_CONFIG.CLSWPER
}
if ( ! --HWA_STATUS.C_STATUS.CLRSTCNT ) {
// reset C operand write row counter HWA_STATUS.C_STATUS.CLROW    = HWA_CONFIG.C_CONFIG.CLOFFSET
HWA_STATUS.C_STATUS.CLRSTCNT = HWA_CONFIG.C_CONFIG.CLRSTPER
}
}
}
```

4.1.2.4.4 Computation Dataflow

The C FSM controls subfunctions that make up the main computation in the MMA unit. The order of subfunctions is fixed, but the subfunction operation is parametrized by fields in the HWA_CONFIG.C_CONFIG register.

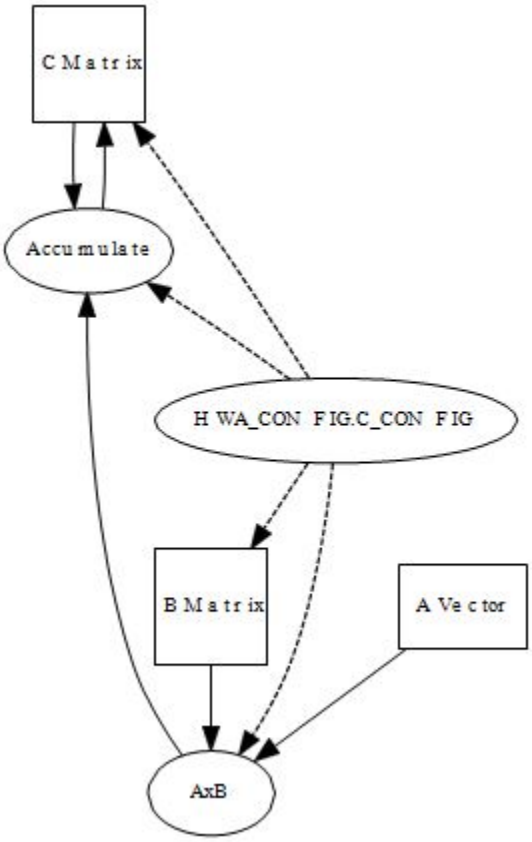


Figure 4-15. Computation Dataflow

4.1.2.4.5 Accumulation Operations

The C FSM configuration allows two accumulation operations to be specified in the C_CONFIG.OPERATION0 and C_CONFIG.OPERATION1 fields. A period can be defined in the C_CONFIG.OP0PER and C_CONFIG.OP1PER fields that specifies the number of pipeline advances caused by HWAOP* instructions between switching between the current operation. An expected usage of this facility is in long sequences of matrix operations that switch between = and += operations. It is useful for software pipelined loop code generation to have the MMA hardware internally switch the operation rather than changing the instruction stream.

If C_CONFIG.OP0PER is not equal to zero and C_CONFIG.OP1PER is equal to zero, then the C FSM never switches to operation 1. If C_CONFIG.OP1PER is not equal to zero and C_CONFIG.OP0PER is equal to zero, then the C FSM never switches to operation 0. If C_CONFIG.OP1PER is equal to zero and C_CONFIG.OP0PER is equal to zero, then C FSM behavior is undefined.

4.1.2.4.6 C Row Address Calculations for Calculations

The C FSM maintains separate counters and bank selections for read and write operations involving the C matrix storage. All accesses to the C matrix are row-major, thus the read and write counters are interpreted as row addresses. The HWA_CONFIG.C_CONFIG register provides individual initial values for the read and write counters, and separate initial C matrix storage banks.

4.1.2.4.7 C Element Addressing and Scaling for Load Instructions

The C FSM maintains a row counter and a bank selection for vector writes into the C matrix storage banks that are not shared with the row and bank controls for computations. Instructions can write into the C storage concurrently with computations if the limitations on the number of simultaneous reads and writes to the individual C storage banks are not violated.

Each C matrix vector row is 2048 bits, and the C matrix operand from a HWALD* instruction is constrained to 512 bits. To work with this mismatch in operand width and storage width, special modes for array addressing and data expansion are provided.

Data expansion is controlled by the HWA_CONFIG.C_CONFIG.HWLDTYPE and HWA_CONFIG.C_CONFIG.LDDST values. In X1 mode, the data is copied directly into the C matrix storage as a 512-bit value and no processing is performed on the data. Also in this mode, the HWA_CONFIG.CLOFFSET value is shifted left by two positions to align with the row offset in the CLROW field. In X4_* modes, sign extension or zero extension to the target element width is performed based on the type specified in HWLDTYPE. Next, a left shift is executed by a multiple of the operand size specified in HWA_CONFIG.C_CONFIG.HWLDTYPE.

The bit width written into the C matrix storage is implied by the HWA_CONFIG.C_CONFIG.HWLDTYPE and HWA_CONFIG.C_CONFIG.LDDST values.

Table 4-72. Title TBD

HWLDTYPE - Source Vector Element Type	HWLDDST - Load Destination Type	Implied Data Write Width	Source Vector Bits Used
(u)int8_t	X1	512	[511:0]
(u)int8_t	X4_*	2048	[511:0]
(u)int16_t	X1	512	[511:0]
(u)int16_t	X4_*	2048	[511:0]
(u)int32_t	X1	512	[511:0]
(u)int32_t	X4_*	2048	[511:0]

X1 mode allows all bits of the C storage to be written. This can be used in conjunction with a similar C output mode to implement state save/restore operations for task switching or diagnostic purposes. Because the write width is 512 bits in this case, it takes up to 256 clock cycles to load a single C matrix storage bank for 8-bit elements.

X4_* modes transform an input operand in one of the native operand sizes supported by the MMA unit to the native size of the C matrix element size corresponding to the operand size. A multiple {0,1,2,3} of the operand element size can be specified for signed operands. Only the 1x multiple of the operand size is available for general unsigned operands. In cases where this scaling does not provide the required C matrix values, processing in the C7x CPU or offline can be used with the X1 mode to preload the C matrix. In X4 mode, it takes up to 64 clock cycles to load a single C matrix storage bank for 8-bit elements.

When the operand is written as 512 bits in the C matrix storage, the low two bits ([1:0])of the HWA_STATUS.C_STATUS.CLROW value are interpreted as a 512-bit offset in the 2048-bit C matrix storage rows. In this case, the actual matrix row written is HWA_STATUS.C_STATUS.CLROW[7:2].

When the operand is expanded and written as 2048 bits in the C matrix storage, the value of HWA_STATUS.C_STATUS.CLROW[5:0] determines the row where the data is written.

4.1.2.4.8 Undefined Cases

Calculation results are undefined if:

- The type of the A vector loaded by the A FSM is inconsistent with the C FSM configuration.
- The type of the B matrix loaded by the B FSM is inconsistent with the C FSM configuration.
- The type of the C matrix loaded by the C FSM loaded by a prior configuration of the C FSM is inconsistent with the current C FSM configuration.

In general, do not load a MMA operand in one data type and then try to operate on it in a different data type. Chaining separate calculations with different operand data types is not supported.

See also:

HWA_CONFIG register definition

HWA_STATUS register definition

4.1.2.5 X Finite State Machine

4.1.2.5.1 X Finite State Machine Inputs

The X FSM advances to the next state when any MMA instruction is executed that loads transfer buffer data. These instructions are any instruction matching the pattern HWA.*XFER. In addition, the X finite state machine responds to the HWAOPEN instruction by initializing its state register and counters.

4.1.2.5.2 X Finite State Machine Outputs

The primary outputs of the X finite state machine are the read addresses and read enables on the C matrix storage, the write addresses and enables for the transfer buffer memory, and the controls for C data transformations.

4.1.2.5.3 X FSM Operation

The X FSM implements the state transitions, as shown in Figure 4-16.

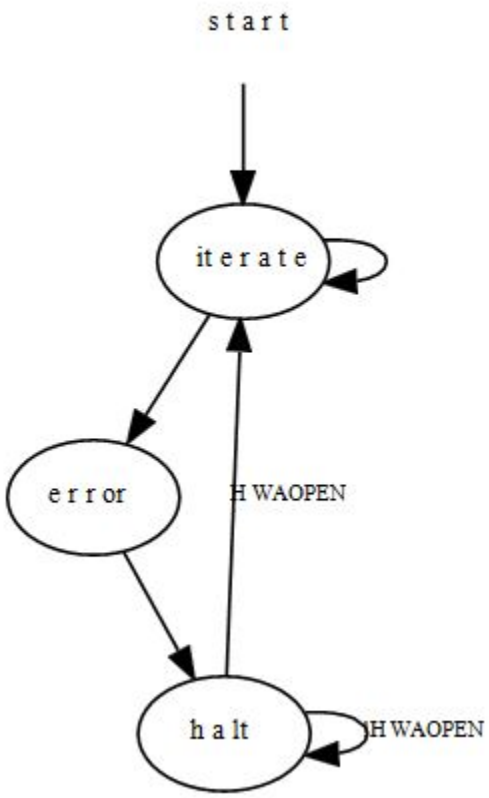


Figure 4-16. State Transitions

The operation of the X FSM is parametrized by the X_CONFIG field of the HWA_CONFIG register. Software may update the state of the configurations while the X FSM is iterating.

While the FSM is in the halted or error states, any input data is ignored.

The following pseudocode summarizes the X FSM operation:

```
// assume C storage is C[C BANKS][ROWS][COLS]
// assume X storage is X[XROWS] restart:
// initialize C bank, C read offsets, X write offsets, counters HWA_STATUS.X_STATUS.CBANK
= HWA_CONFIG.X_CONFIG.CSTART    ; HWA_STATUS.X_STATUS.CRROW    = HWA_CONFIG.X_CONFIG.COFFSET    ;
HWA_STATUS.X_STATUS.CSWCNT    = HWA_CONFIG.X_CONFIG.CSWPER    ;

HWA_STATUS.X_STATUS.CRRSTCNT = HWA_CONFIG.X_CONFIG.CRRSTPER ;
while( 1 ) {
wait_for_issue_cycle() ; // block until next X_FSM instruction
```

```
cb = HWA_STATUS.X_STATUS.CBANK    % C BANKS rr = HWA_STATUS.X_STATUS.CRROW    % ROWS
wr = HWA_STATUS.X_STATUS.XWROW    % XROWS
X[wr] = output_processing(C[cb][rr]) ;
HWA_STATUS.X_STATUS.CRROW += 1 ; // update C read counter HWA_STATUS.X_STATUS.XWROW += 1 ; // update
X write counter
if ( ! --HWA_STATUS.X_STATUS.CSWCNT ) {
// swap C bank HWA_STATUS.X_STATUS.CBANK += 1
HWA_STATUS.X_STATUS.BSWCNT = HWA_CONFIG.X_CONFIG.CSWPER
}
if ( ! --HWA_STATUS.X_STATUS.CRRSTCNT ) {
// reset C read row counter
HWA_STATUS.X_STATUS.CRROW    = HWA_CONFIG.X_CONFIG.COFFSET HWA_STATUS.X_STATUS.CRRSTCNT =
HWA_CONFIG.X_CONFIG.CRRSTPER
}
}
```

4.1.2.5.4 Output Processing Dataflow

The X FSM controls subfunctions that make up the output processing block in the MMA unit. The order of subfunctions is fixed, but the subfunction operation is parametrized by fields in the HWA_CONFIG.X_CONFIG register.

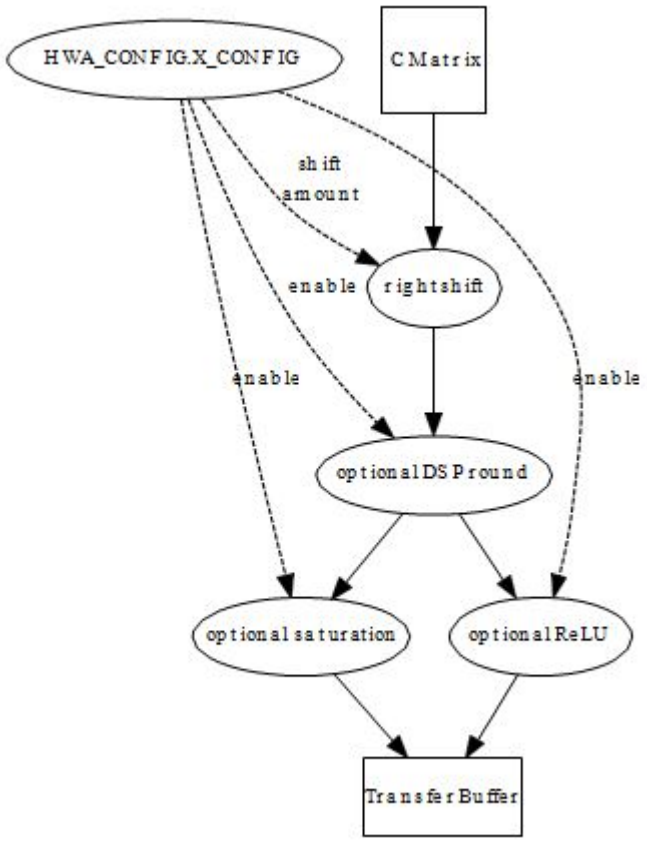


Figure 4-17. Output Processing Dataflow

The shift amount must be less than width of the element size of the C Matrix. The DSP round step is a 1/2-LSB-weight increment followed by truncation, as shown in Equation 2.

$$\text{rounded} = ((\text{C value} >> (\text{shift}-1))+1) >> 1 \quad (2)$$

where

- shift is the HWA_CONFIG.X_CONFIG.SHIFT value
- the shift is an arithmetic shift that sign-extends or zero- extends according to the C element type

- the rounding increment is performed with one additional bit of precision for consistency with C7x rounding operations.

Rounding is suppressed when the HWA_CONFIG.X_CONFIG.SHIFT value is zero.

ReLU is implemented as a special case of saturation that changes the lower limit from the smallest representable number to zero. In both ReLU and SAT modes, numbers too large to represent in the destination format are converted to the maximum representable number in the destination format. ReLU and SAT modes should not be simultaneously enabled.

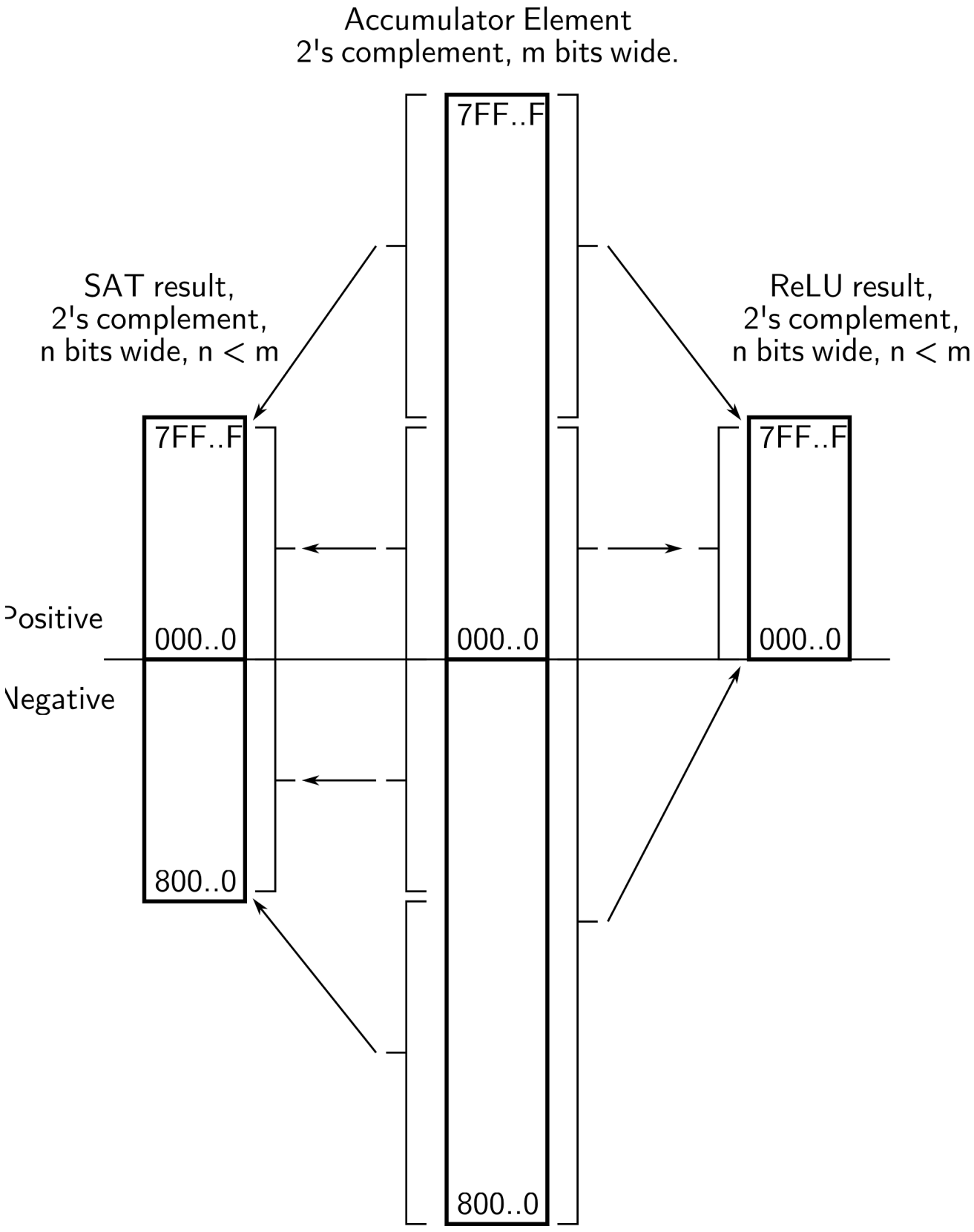


Figure 4-18. Signed Mapping to SAT or ReLU

4.1.2.5.5 C Matrix and Transfer Buffer Addressing

The X FSM maintains a counter and bank selection for C matrix read operations. All accesses to the C matrix are row-major, so the counter normally is interpreted as a row address. When the output format element bit width multiplied by the number of elements exceeds the width of the transfer buffer entries (512 bits), then low-order row address bits are interpreted as field select bits to specify power-of-two aligned fields in the 2048-bit C matrix row. This addressing is similar that described in the C FSM section for direct operand loading into the C matrix storage.

Table 4-73. Title TBD

CTYPE - C Matrix element Type	XTYPE - Transfer Buffer Element Type	Implied C Matrix Read Width	CRROW row bits	CRROW field select bits
(u)int32_t	(u)int8_t	2048	[5:0]	NA
(u)int32_t	(u)int16_t	1024	[6:1]	[0]
(u)int32_t	(u)int32_t	512	[7:2]	[1:0]
(u)int64_t	(u)int8_t	2048	[5:0]	NA
(u)int64_t	(u)int16_t	2048	[5:0]	NA
(u)int64_t	(u)int32_t	1024	[6:1]	[0]
(u)int64_t	(u)int64_t	512	[7:2]	[1:0]
(u)int128_t	(u)int8_t	2048	[5:0]	NA
(u)int128_t	(u)int16_t	2048	[5:0]	NA
(u)int128_t	(u)int32_t	2048	[5:0]	NA
(u)int128_t	(u)int64_t	1024	[6:1]	[0]
(u)int128_t	(u)int128_t	512	[7:2]	[1:0]

The HWA_CONFIG.X_CONFIG register provides an initial value for the read counter and initial C matrix storage bank.

The X FSM generates the sequence of push operations that load data into the transfer buffer FIFO. Pushes into the FIFO are decoupled from the read operations and generate overflow errors if there is insufficient space for the push operation.

See also:

HWA_CONFIG register definition

HWA_OFFSET register definition

4.1.2.6 MMA Instructions

HWALDA src1

Load 512 bits into A vector storage

Class	MMA
Valid execution modes	Unprotected
Latency	1
Unit	.L2
Description	The HWALDA instruction reads a 512-bit operand from a C7x vector register or streaming engine and deposits the value in the A vector memory in the MMA unit. The vector src1 operand may be one of VB0-VB15, VBL0-VBL7, SE0, SE1, SE0++, or SE1++. The type of the elements in the 512-bit value and any pre-processing steps are controlled by the A FSM. HWALDA can be used the same execute packet as other non-L2-unit HWA instructions. HWALDA is predictable.
Examples	`HWALDA    .L2 SE0++`

HWALDAB src1 , src2

Load 512 bits into A vector storage

Class	MMA
Valid execution modes	Unprotected
Latency	1
Unit	.L2
Description	The HWALDAB instruction reads two 512-bit operands from a C7x vector register or streaming engines. The left value (src1) is deposited into A vector memory. The vector src1 and src2 operands may be one of VB0-VB15, VBL0- VBL7, SE0, SE1, SE0++, or SE1++. The element types and any pre-processing on the A vector data is controlled by the A FSM. The right (src2) value is deposited into B matrix memory. The element types, pre-processing and storage locations in the B matrix are controlled by the B FSM. HWALDAB can be used in the same execute packet as any other non-L2-unit HWA instructions. In typical matrix operations, HWALDAB is in parallel with HWAOPXFER and HWARCV instructions. HWALDAB is predictable.
Examples	

```
HWALDAB    .L2 SE0++,SE1++    // A gets data from SE0, B gets data from SE1
||    HWAOPXFER .S1    // MAC and move results to transfer buffer
||    HWARCV    .S2 HWA0,VB0    // move result from transfer buffer to C7x VB0
```

HWALDB src1

Load 512 bits into B matrix storage

Class	MMA
Valid execution modes	Unprotected
Latency	1
Unit	.L2
Description	The HWALDB instruction reads a 512-bit operand from a C7x vector register or streaming engine and deposits the value in the B vector storage in the MMA unit. The vector src1 operand may be one of VB0-VB15, VBL0-VBL7, SE0, SE1, SE0++, or SE1++. The B FSM defines the type of the elements in the 512-bit value and the location that the elements are written in the B matrix memory. HWALDB can be in the same execute packet as other non-L2-unit HWA instructions. HWALDB is predictable.
Examples	`HWALDB .L2 SE0++`

HWALDBC src1, src2

Load 512 bits into B matrix storage and 512 bits into C matrix storage

Class	MMA
Valid execution modes	Unprotected
Latency	1
Unit	.L2
Description	The HWALDBC instruction reads two 512-bit operands from a C7x vector register or streaming engines. The vector src1 and src2 operands may be one of VB0-VB15, VBL0-VBL7, SE0, SE1, SE0++, or SE1++. The left value (src1) is deposited into B matrix memory. The element types, pre-processing and storage locations in the B matrix are controlled by the B FSM. The right value (src2) is deposited into C matrix memory. The element types, pre- processing, and storage locations in the C matrix are controlled by the C FSM. HWALDBC can be in the same execute packet as other non-L2-unit HWA instructions. HWALDBC is predictable.
Examples	`HWALDBC .L2 SE0++,SE1++ ; // B gets data from SE0, C gets data from SE1`

HWALDC src1

Load 512 bits into C matrix storage

Class	MMA
Valid execution modes	Unprotected
Latency	1
Unit	.L2
Description	The HWALDC instruction reads a 512-bit operand from a C7x vector register or streaming engine and deposits the value in the C matrix memory in the MMA unit. The vector src1 operand may be one of VB0-VB15, VBL0-VBL7, SE0, SE1, SE0++, or SE1++. The C FSM defines the type of the elements in the 512-bit value and the location that the elements are written in the C matrix memory. HWALDC can be in the same execute packet as other non-L2-unit HWA instructions. HWALDC is predictable.
Examples	`HWALDC .L2 SE0++`

HWAOP src1

Execute one vector-by-matrix operation step

Class	MMA
Valid execution modes	Unprotected
Latency	1
Unit	.S1
Description	The HWAOP instruction uses the C FSM to perform an operation on the A vector and B matrix. Typically the operation computes a row vector composed of the dot products of the A vector and all of the columns of the B matrix, and the result of the computation is written to the C matrix memory. The C FSM configuration determines the actual operation that is computed, the location of C matrix operands, and the location in the C matrix where results are stored. The src1 operand is a 64-bit register or immediate value and is reserved for future definition. Software should use the value 0 for compatibility with future devices. HWAOP can be in the same execute as any non-S1-unit HWA instructions. HWAOP is predictable.
Examples	`HWALOP .S1 0`

HWAOPXFER src1

Execute one vector-by-matrix operation step and move C matrix data

Class	MMA
Valid execution modes	Unprotected
Latency	1
Unit	.S1
Description	The HWAOPXFER instruction uses the C FSM to perform an operation on the A vector and B matrix. Typically the operation computes a row vector composed of the dot products of the A vector and all of the columns of the B matrix, and the result of the computation is written to the C matrix memory. In parallel with the computation, the HWAOPXFER instruction uses the X FSM to move a row of C matrix data to the transfer buffer that can be accessed by the C7x CPU. The src1 operand is a 64-bit register or immediate value and is reserved for future definition. Unlike the HWAXFER instruction, the immediate value does not select non C matrix data sources for the transfer buffer. Software should use the value 0 for compatibility with future devices. The C FSM configuration determines the actual operation that is computed, the location of C matrix operands, and the location in the C matrix where results are stored. The X FSM configuration determines source row of C matrix data, the destination location in the transfer buffer, and the output processing the occurs before the C matrix data is stored in the transfer buffer. HWAOPXFER can be in the same execute packet as other non-S1-unit HWA instructions. HWAOPXFER is predictable.
Examples	`HWALOPXFER    .S1 0`

HWAXFER src1

Move data to the transfer buffer

Class MMA

Valid execution modes Unprotected

Latency 1

Unit .L1

Description

In normal operation, the HWAXFER instruction uses the X FSM to move a row of C matrix data to the transfer buffer. The transfer buffer can be accessed by the C7x CPU. The HWAXFER instruction is also used to move contents of the status and control registers into the transfer buffer. Regardless of the data source, data always enters and leaves the transfer buffer FIFO in order. Status and control register reads are not visible on the transfer buffer output before the results of earlier HWAXFER or HWAOPXFER instructions.

The src1 operand is a 4-bit immediate value. The immediate value selects the data that is pushed into the transfer buffer FIFO.

In the common case of moving data from the C matrix storage, the X FSM configuration determines the source row of C matrix data, as well as the output processing that occurs before the C matrix data is stored in the transfer buffer. Other data sources do not advance the X FSM and the data from those sources does not pass through the output processing pipeline.

src1 Value	Data Source	Notes
4'b0000	C matrix storage	X FSM sequences C matrix addresses and result processing
4'b1000	HWA_CONFIG	No X FSM update
4'b1001	HWA_OFFSET	No X FSM update
4'b1010	HWA_STATUS	No X FSM update
4'b1011	HWA_BUSY	No X FSM update

HWAXFER can be in the same execute packet as other non-L1-unit HWA instructions.

HWAXFER is predictable.

Examples

```
HWALXFER    .L1 0
```

HWARCVS hwaimm,dst

Move a vector from the MMA transfer buffer to a C7x vector register

Class	MMA
Valid execution modes	Unprotected
Latency	4
Unit	.S2
Description	The HWARCVS and HWARCVC instructions are the primary instructions for reading information from the MMA unit. These instructions move oldest contents the transfer buffer to a vector register in the C7x CPU. The destination argument (dst) can be VB0-VB7, VBL0-VBL7, or VBM0-VBM7. No post-processing is performed on the data. The HWARCV(S|C) instructions are the only MMA instructions that returns an error indication to the C7x CPU. An error indication is returned when the HWA_STATUS.FirstErrorCode field is non-zero. HWARCVS does not advance the A FSM, B FSM, C FSM, or X FSM. The results of executing the HWARCVS instruction are undefined between the time HWACLOSE is issued and the time HWAOPEN executes. HWARCVS is predictable and has a 4 cycle latency. The hwaimm argument should be zero. The results of any other values are undefined.
Examples	`HWALRCVS .S2 0,VB0 ; // move MMA data output register HWA0 to VB0.`

HWAOPEN src1, src2, hwaimm

Initialize the MMA unit

Class	MMA
Valid execution modes	Unprotected
Latency	1
Unit	.L2
Description	The HWAOPEN instruction updates the control registers in the MMA unit. A FSM, B FSM, C FSM, and X FSM require their configuration to be initialized by HWAOPEN prior to the start of operations. The left operand (src1) provides data for HWA_CONFIG. The right operand (src2) provides data for HWA_OFFSET. Vector operands src1 and src2 may be one of VB0-VB15, VBL0-VBL7, SE0, SE1, SE0++, or SE1++. The 4-bit immediate value, hwaimm, changes the level of initialization that is performed. HWAOPEN does not update the data in the data storage for the A vector, B matrix, or C matrix. Different types of computations using the same data in these data storage arrays may straddle HWAOPEN instruction executions. HWAOPEN changes are pipelined. The state machines observe the changes to the and registers after the pipeline delay specific to that state machine. State machines can continue to operate while HWA_CONFIG and HWA_OFFSET changes are in flight. For example, assume some sequence of operations has programmed the X FSM to move data from the C matrix into the transfer buffer and this is immediately followed by a HWAOPEN instruction that modifies the configuration of the X FSM. The earlier X FSM operations continue to execute until the pipelined value of the new configuration reaches the X FSM. HWAOPEN does not affect the transfer buffer FIFO. When hwaimm is 4'b0000, HWA_CONFIG and HWA_OFFSET are updated. All the state machines and associated periodic counters are re-initialized. HWAOPEN clears the error code, halted and error flags in the HWA_STATUS register for each state machine. HWAOPEN does not clear the FirstErrorCode or LastErrorCode fields of HWA_STATUS. When hwaimm is 4'b0010, HWA_CONFIG fields that specify periods are updated. Current iteration counts are updated to the result of min(count,new HWA_CONFIG value). When hwaimm is 4'b0011, HWA_CONFIG fields that specify periods are updated. Current iteration counts are updated to the result of max(count,new HWA_CONFIG value). HWAOPEN is predictable.
Examples	`HWAOPEN .L2 VB0,VB1,0 ; // initialize HWA_CONFIG and HWA_OFFSET`

HWACLOSE src1

End HWA operations

Class	MMA
Valid execution modes	Unprotected
Latency	1
Unit	.S1
Description	The HWACLOSE instruction is used by software to inform the C7x CPU that no additional MMA instructions are executed until the next HWAOPEN instruction. The contents of any state elements (other than specific HWA_STATUS fields noted below) in the MMA unit are undefined until the next HWAOPEN instruction. Software should not assume internal MMA state persists from the time the HWACLOSE instruction is issued to the time the subsequent HWAOPEN instruction executes. Some MMA implementations may put the MMA unit in a powered- down sleep mode when the HWACLOSE instruction is executed. It is not necessary to execute an HWACLOSE instruction before executing an HWAOPEN instruction. The HWACLOSE instruction always clears the transfer buffer FIFO valid flags, effectively emptying the FIFO. HWACLOSE clears any non-zero values in the HWA_STATUS LastErrorCode and FirstErrorCode fields. The src1 operand is a 64-bit register or immediate value and is reserved for future definition. Software should use the value 0 for compatibility with future devices. HWACLOSE can be in the same execute packet as any non-S1-unit HWA instructions. HWACLOSE is predictable.
Examples	`HWACLOSE    .S1 0`

HWARES1 src1 , src2

	Reserved .L2 unit instruction for MMA
Class	MMA
Valid execution modes	Unprotected
Latency	Undefined
Unit	.L2
Description	The HESRES1 instruction is a reserved opcode. It can read two 512-bit operands from a C7x vector register or streaming engines. HWARES1 can be used in the same execute packet as any other non-L2-unit HWA instructions. HWARES1 is predictable.
Examples	`HWARES1 .L2 SE0++,SE1++ // do something with SE0/SE1 data`

4.1.3 Appendix 1 - Code Examples

The following information is for example purposes only.

Matrix Multiply Kernel

```
kernel:
HWALDAB    .L2 SE0++,SE1++    ; // push A and B operands into MMA

||    HWAOPXFER .S1    ; // matrix MAC and transfer result
||    HWARCV    .S2 HWA0,VB0    ; // move MMA result to VB0
||    VST64B    .D2 VB0,*D0(SA0++)    ; // store MMA result in VB0 in memory
||    TICK    ;
||    BNL    .B1 kernel    ;
```

Matrix Multiply Kernel with Output Processing

```
kernel:
HWALDAB    .L2 SE0++,SE1++    ;
||    HWAOPXFER .S1    ;
||    HWARCV    .S2 HWA0,VB0    ; // VB0 gets MMA result
||    LUTRD    .D1X    VB0,LTBR0,VB1 ; // pass MMA through LUT
||    VST64B    .D2 VB1,*D0(SA0++)    ; // store LUT result
||    TICK    ;
||    BNL    .B1 kernel    ;
```

Matrix Multiply Kernel with Input Processing

```
kernel:
VMV    .C2 SE0++,VB0    ; // get operand from SE0
||    LUTRD    .D1X VB0,LTBR0,VB1    ; // pass operand through LUT
||    HWALDAB    .L2 VB1,SE1++    ; // push operands into MMA
||    HWAOPXFER .S1    ;
||    HWARCV    .S2 HWA0,VB2    ;
||    VST64B    .D2 VB2,*D0(SA0++)    ;
||    TICK    ;
||    BNL    .B1 kernel    ;
```

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (https:www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265