NXP Semiconductors
Reference Manual

Document identifier: vspa2_ism_16au_la9310
Rev. 0, 06/2021

VSPA-16SP ISA-v2.0 Instruction Set Manual for LA9310

Contents

Chapter 1
Preface

1.1 Introduction

The primary purpose of this document is to describe the architecture and functionality of the Vector Signal Processing Acceleration (VSPA) platform. This manual includes instruction sets for two main functional units of VSPA - VCPU and IPPU as well as VSPA IP and Debug IP register definitions.

1.2 Variants of VSPA

The VSPA instruction set architecture (ISA) has evolved significantly since it was first developed, and will continue to undergo further development. This ISM elaborates VSPA ISA v2.0. The significant changes in comparison to VSPA ISA v1.0 are described in New features in VSPA ISA-v2.0

VSPA can be classified based on the number of arithmetic units (AU). This ISM supports VSPA-16SP for LA9310. LA9310 has 1 instance of VSPA 16AU single precision (SP). This means that, VSPA core for LA9310 supports:

- 16 complex MAC operations per clock cycle / single precision
- Single precision math

1.3 New features in VSPA ISA-v2.0

1.3.1 Overview

This instruction set architecture (ISA-v2.0) extends the former version by supporting features listed below:

- Improved compute efficiency
- Improved compiler efficiency
- Source code backward compatibility with VSPA1 with the help of ‘VSPA1onVSPA2’ compiler switch
  - Software written for VSPA1 will compile and run on VSPA2 without any changes
- Significant improvements to the vector data path
  - New SAU (square-root, atan, and so on) microinstruction can execute in parallel with existing AU operations (mac, mad, and so on)
  - New pre-add SAU instruction can reduce cycles for FIR filters by half
  - New look-up-table (LUT) SAU instruction allows non-linear function evaluation
- Significant improvements to the control plane
  - Increase in the VLIW program word from 56 to 64 bits
  - Addition of many new instructions with enhanced indirect addressing
  - Simplified register model; 3 sets (as*X*,a*X*,g*X*) to 2 sets (a*Y*,g*Y*)
  - Half-word (16 bits) address resolution

1.3.2 Enhancements in vector data path

Vector data path enhancements are as follows:

1. New OpVsau microinstruction to enable parallel execution of SAU and AU operations
2. New pre-add SAU instruction to reduce clock cycles in filtering algorithms with coefficient symmetry
3. New vector complex table look-up (LUT) SAU operation to approximate general function evaluation, f(|x|)

4. Option to feed SAU operation result directly to AU operation input
    - This is useful in a.*f(|x|) operations
5. Introduction of left rotation unit to complement existing right rotations
6. New instructions to broadcast scalar g*X* registers to VRA
7. Elimination of real/complex mode
    - Independent real and complex versions of AU instructions are available
    - ‘Hidden’ is still available for backward compatibility
8. Elimination of VRA page registers
    - Increased size of page registers
    - ‘Hidden’ is still available for backward compatibility
9. New instruction for inter-VRA register moves
10. Extrema search unit support for single precision and return of extrema in g*X*

1.3.3 Enhancements in program control and scalar data path

Program control and scalar data path enhancements are as follows:

1. Elimination of address storage register set (as*X*) to simplify programming model and improve coding efficiency
    - as*X* registers have been replaced with 16 additional address registers (a4-a19)
    - as*X* are still supported in tools for backward compatibility
2. Scalar g*X* registers now available for use as memory pointers
3. Expansion of program counter from 16 to 24 bits
4. Support for conditional relative branching
5. New illegal instruction error event
6. Increased size of loop counter from 10 to 16 bits
7. New loop-break instruction
    - Execution proceeds on first instruction outside of loop
8. New complete data type conversion instructions
9. OpA forms of VRA pointer control for improved coding efficiency
10. OpA and OpB moves between VRA pointers and scalar registers
11. Support for 32 bit immediate scalar ALU operations
12. Increased conditional scalar ALU operations
13. New instructions to improve IP register access efficiency
14. Scalar register bit set/clear instructions
15. Improved support for PMEM overlays

1.3.4 Enhancements in data memory access

Data memory access enhancements are listed below:

1. Support for half-word (16bit) addressing reduces data memory consumption
    - Existing 32-bit is still supported for backward compatibility
2. Support for partial vector stores

3. New OpB forms of loads and stores
4. Support for two times larger tightly coupled data memory
5. Improved load/store-multiple memory efficiency
6. Larger immediate value for indirect addressing
7. Support for indirect loads/stores using stack pointer
8. New two times larger indirect access pointer modifier

1.4 Using this manual

The information in this manual is broadly organized as follows:

1. VSPA Overview and Architecture
   - *Chapter 2*: Introduces VSPA architecture with the help of a block diagram and gives a brief overview of each of the functional units of VSPA.
2. VSPA functional unit - VCPU
   - *Chapter 3*: Primarily describes control plane and data plane in detail. Also, introduces some other features of VCPU like the VCPU's internal floating point representation, VCPU GO events and byte order.
   - *Chapter 4*: Lists instruction families and formats. Includes an instruction set summary table for all VCPU instructions with VSPA instruction set mnemonic conventions listed first. The VCPU instruction set summary consists of different groups of instructions each group consisting of a list of opcodes and operands and the family, number of cycles and a brief description for them. Quick references to different groups of instructions are given at the beginning of the VCPU instruction set summary table. The chapter also includes elaborate topics on each of these instruction groups.
3. VSPA functional unit - IPPU
   - *Chapter 5*: Introduces IPPU with a brief overview and lists IPPU features. Also describes IPPU core, operating states, memory access considerations and initialization.
   - *Chapter 6*: Lists size and hardware definitions for IPPU followed by an IPPU instruction set summary table. The summary table contains a list of IPPU instructions with number of cycles for each instruction and a quick reference link to detailed descriptions that follow.
4. VSPA functional unit - DMA controller
   - *Chapter 7*: Includes detailed description of the Direct memory access unit (DMA). Also, lists the features that the DMA does not support.
5. VSPA functional unit - Mailboxes
   - *Chapter 8*: Includes detailed description of the VSPA mailboxes.
6. VSPA functional unit - AXI Slave
   - *Chapter 9*: Includes detailed description of the AXI Slave.
7. VSPA Debug architecture
   - *Chapter 10*: Introduces VSPA Debug architecture and provides a general overview of the top-level blocks and their respective functionality.
   - *Chapter 15*: Contains a memory map and register definitions for VSPA debugger registers.
8. VSPA Interrupts
   - *Chapter 11*: Includes detailed description of the VSPA interrupts.
9. VSPA Initialization
   - *Chapter 12*: Describes VSPA Initialization
10. VSPA functional unit - FECU

- *Chapter 13*: Introduces Forward error correction unit (FECU) along with its sub-modules.
- *Chapter 16*:Includes memory map and register definitions for FECU. FECU registers are a part of VSPA memory map starting at offset address 0x300.

11. VSPA IP registers
    - *Chapter 14*: Contains a VSPA memory map listing the offset addresses, bit widths, access types, and reset values for all VSPA IP registers. Also, includes detailed register definitions with register diagrams and description for register bit fields.

1.5 Conventions

The following conventions are used throughout this document.

- All numbers are decimal values unless otherwise specified:
  - 0b*nnnn* denotes a number in binary format.
  - 0x*nnnn* denotes a number in hexadecimal format.
- Bits in registers, instructions, and instruction fields:
  - Bits are numbered beginning with 0 for the least significant bit on the right and ending with the most significant bit on the left.
  - A range of bits is specified by 2 numbers separated by a colon ":". For example, y:x denotes bits x through y.
- V[y:x] can have 2 interpretations, depending on the context of V:
  - If V is a scalar register or a scalar bus, then V[y:x] denotes the subfield of V, from bit position x through bit position y. For example, *aa* registers (a0 through a3) are all scalar registers. *a0*[4:0] denotes the lower five bits of register *a0*.
  - If V is a vectored register or a vectored bus, then V[y:x] denotes a subvector consisting of the x^{th} element of V through y^{th} element of V (both elements are inclusive).
- {x, y} denotes the concatenation of 2 values. For example, {010, 111} is the same as 010111.
- x^n means x is raised to the n^{th} power.
- For any duplication pattern (y, ..., x), y corresponds to the most significant bit and x corresponds to the least significant bit. For example, in the duplication pattern (real, imag, -imag, real) the leftmost real in parenthesis corresponds to the set of most significant bits.

Naming Conventions:

- Vector register names begin with an upper case letter, for example, S0, S1, S2, and so on. Scalar register names always begin with a lower case letter.
- For readability, names may be intermingled with the underscore character "_".

1.6 Acronyms and abbreviations

AU	Arithmetic unit
CREG	System control register
DI	Deinterleaving
DMA	Direct memory access
FECU	Forward error correction unit
FFT	Fast Fourier transform

Table continues on the next page...

Table continued from the previous page...

FSM	Finite state machine
GP	General purpose
GPIO	General purpose input output
HF	Half fixed
HP	Half precision
IDMEM	IPPU data memory
IP	Internal peripheral
IPPU	Intervector permutation processing unit
IPRAM	IPPU program memory
ISM	Instruction set manual
LUT	Look up table
MAG	Memory address generation
NCO	Numerically controlled oscillator
PMEM	Program memory
RF	Register file
RAG	Register file address generation
SAU	Special arithmetic unit
SOC	System on a chip
SP	Single precision
SUPV	Supervisor
VAU	Vector arithmetic unit (composed of arithmetic units (AU) and special arithmetic units (SAU))
VCPU	Vector central processing unit
VDMEM	VCPU data memory
VLIW	Very long instruction word
VPRAM	VCPU program memory
VRA	Vector register array
VSP	Vector signal processor
VSPA	Vector signal processing acceleration
VPRED	Vector predication

Chapter 2
VSPA Architecture Overview

2.1 VSPA architecture introduction

VSPA is a signal processing platform which leverages a Single Instruction Multiple Data (SIMD) data path and VLIW control plane to provide extremely high compute capability per milli-watt and/or silicon area. The instruction set and scalable data path are optimized for a broad range of applications. Some of these include low-complexity modems such as Bluetooth or AM/FM car radio, multi-channel audio, and the antenna signal processing used in highly complex 5G massive-MIMO cellular base stations. The architecture employs a very simple control plane so it is not optimum for control dominant or packet switching and layer 2 applications. VSPA is typically used in SOC's with a more traditional general purpose host controller such as an ARM which provide access to general peripherals. The architecture is also very suitable as a math co-processor for more general compute platforms.

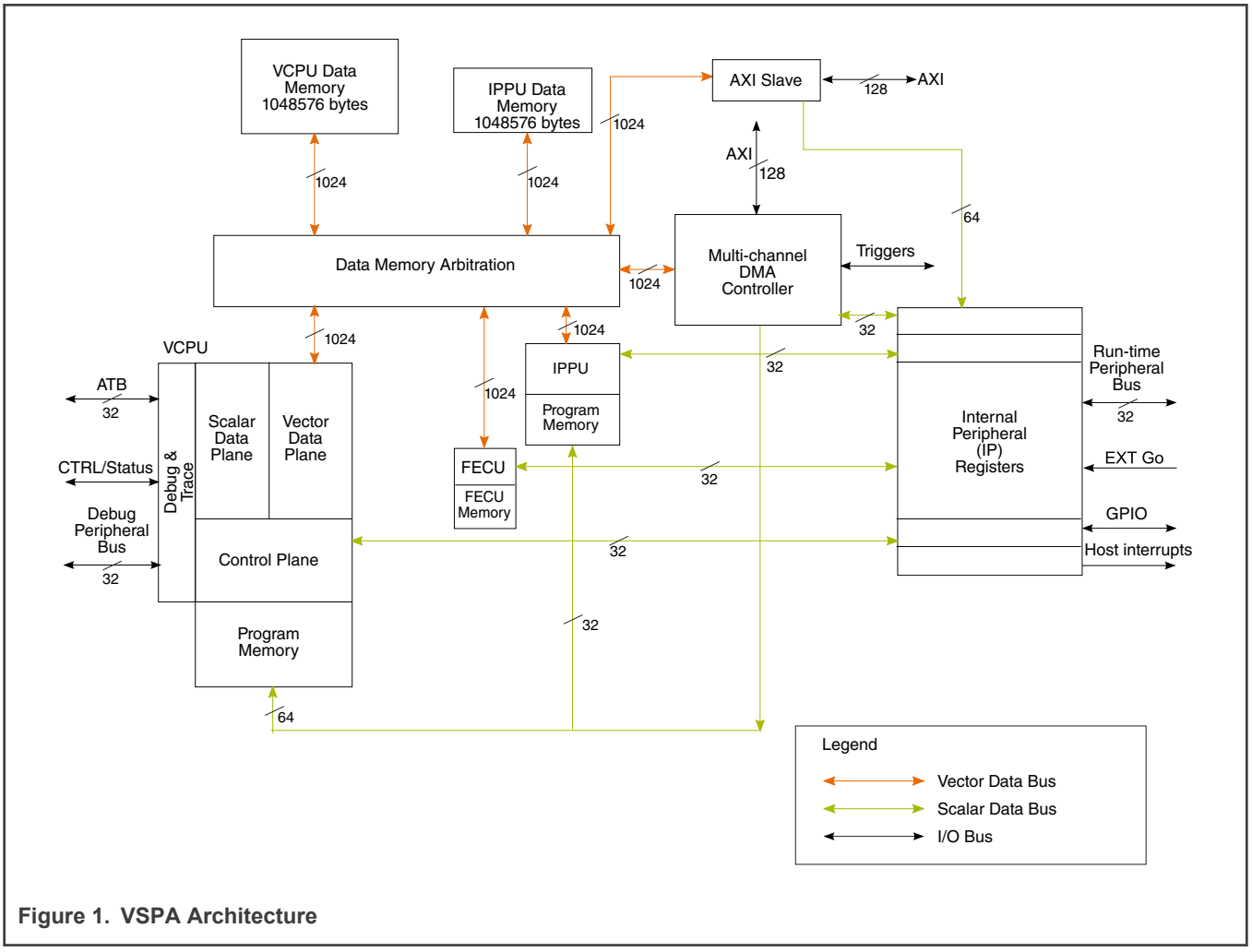


Figure 1. VSPA Architecture

The architecture consists of the following functional units:

2.1.1 VCPU introduction

VCPU stands for Vector Central Processing Unit.

The VSPA architecture consists of control and data planes.

- Control plane: The control plane is SIMD and can execute several operations on the data plane in each cycle. It uses a single issue pipeline and is optimized for efficient data operations. Software threads run to completion and begin with a 'go' or wake-up event initiated by an external event or DMA transfer completion. The 'done' instruction is the last instruction executed in a thread and puts the machine in a low power state.
- Data plane: All vector data math operations are implemented in software on the VCPU. The VCPU data path is organized into *vector* data plane and *scalar* data plane.

The Intervector Permutation Processing Unit and Vector Central Processing Unit are programmable machines supported by the tool chain.

Table 1. Vector/Scalar data plane features

Vector data plane	Scalar data plane
• 1024-bit vector data bus • 64 SP RMAC/RMAD operations per clock cycle • Vector sqrt(|x|), 1/x, 1/sqrt(|x|) operations • Vector NCO operations • Vector spreading/scrambling code generator for WCDMA • Independent vector compare engine • Multi-port, 8-line vector register array • Transparent intra-vector permutations on each VRA port • 2 vector rotate units • 256-bit vector sign register	• 32-bit data plane • Full-featured arithmetic and logic unit • 12 scalar registers • Stack pointer • 20 memory pointer registers with modulo buffer and re-ordering algorithm support

2.1.2 IPPU introduction

The Intervector Permutation Processing Unit (IPPU) is an independent programmable machine which is responsible for reordering buffers for efficient utilization of the vector data path on the VCPU. It has independent program and data memories so it can run in parallel with the VCPU. This allows buffers to be pipelined, allowing vector math of buffer N to be processed on the VCPU while buffer N+1 is reordered on the IPPU. The IPPU has a limited instruction set optimized for flexible construction of vectors from scalars of various data types. The IPPU and VCPU data memories are visible to the other core through an arbiter to allow efficient transfer of buffers.

2.1.3 DMA controller

The Direct Memory Access (DMA) Controller is an independent state machine which is responsible for movement of data buffers between VSPA core and other cores, on-chip peripherals, memories and any other memory mapped component. It is compliant with the AMBA AXI3 bus protocol with independent read and write busses and a configurable size depending on the target chips. The module supports up to 32 independent channels (active simultaneous transfers) serviced in a round-robin manner on a 16-beat burst interval. Data transfers can target both IPPU and VCPU data memories. Program images can also be loaded using the DMA. Each DMA channel can be configured to start execution of a software thread when the transfer completes. The thread can reside on the IPPU or VCPU. The DMA control resides in the IP register map and is visible to both the host (ARM) and VCPU. The boot-up sequence requires the host to configure one of the DMA channels to download the initial VCPU program image.

2.1.4 IP registers

Control of the IPPU, DMA, GPIO, external go events and other miscellaneous peripherals is configured with a set of memory mapped 32-bit internal peripheral (IP) bus registers. Both the host and VCPU have access to these registers, although the physical addresses are different as viewed from each core.

2.1.5 Vector data memory

Data is organized into 1024-bit lines, although the physical structure may consist of multiple parallel instantiations of smaller memory blocks. Independent tightly coupled data memories exist for both the IPPU and VCPU to allow simultaneous execution of programs and data memory accesses. Both memories are visible to the DMA, VCPU and IPPU machines through a data arbiter module.

There is a total of 2097152 bytes of memory, allocated as 1048576 bytes for the VCPU data memory (VDMEM) and 1048576 byes for the IPPU data memory (IDMEM). Accesses from the VCPU or DMA can be made as though the VDMEM and IDMEM are a contiguous memory. The behavior of accesses to address locations greater than 2097152 are dependent on the chip level memory integration. Such accesses should be avoided as they may cause the memory to be corrupted.

2.1.6 Data Memory Arbitration

The arbitration interface (DRI) arbitrates between VSPA units attempting to access the vector data memory. It has a fixed priority arbitration. If two or more units access the same memory in a cycle, the lower priority unit(s) will be stalled by the DRI until the high priorty access(es) have completed.

Table 2. DRI aribitration priority

priority	Unit
Highest	DMA
...	Axi Slave
...	IPPU
...	FECU
...	VCPU
...	VCPU parallel load/store
Lowest	Debug

Chapter 3
VCPU Architecture

3.1 Control plane

In a traditional general purpose microprocessor, an instruction is decoded at the decode stage(s). The resulting control signals that are generated from decoding this instruction are then passed down to subsequent pipe stages. The hardware in the subsequent pipe stages then uses these control signals to carry out the required function for that instruction. This approach incurs extra hardware complexities, but usually results in smaller instruction words.

Figure 2 shows how the control signals in a traditional general-purpose microprocessor are generated at the decode stages and are then passed down to the subsequent pipe stages (Execute 0, Execute 1, Execute 2 and Writeback).

Pipestages	Time				
	Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4
Fetch	i4	i5	i6		
Decode 0	i3	i4	i5	i6	
Decode 1	i2 (control for i2)	i3 (control for i3)	i4 (control for i4)	i5 (control for i5)	i6 (control for i6)
Execute 0	i1 (control for i1)	i2 (control for i2)	i3 (control for i3)	i4 (control for i4)	i5 (control for i5)
Execute 1		i1 (control for i1)	i2 (control for i2)	i3 (control for i3)	i4 (control for i4)
Execute 2			i1 (control for i1)	i2 (control for i2)	i3 (control for i3)
Writeback				i1 (control for i1)	i2 (control for i2)

Figure 2. Pipeline Flow for a General Purpose Microprocessor

In a very long instruction word (VLIW) engine, an instruction is decoded at the decode stages. However, the resulting control signals that are generated for this instruction are used to control all of the functional units in all pipe stages. In a VLIW engine, the generated control signals generally are not passed down to the subsequent pipe stages.

Figure 3 shows how the control signals are generated at Decode Stage 0 and Decode Stage 1 and are distributed to all pipe stages from Decode Stage 1.

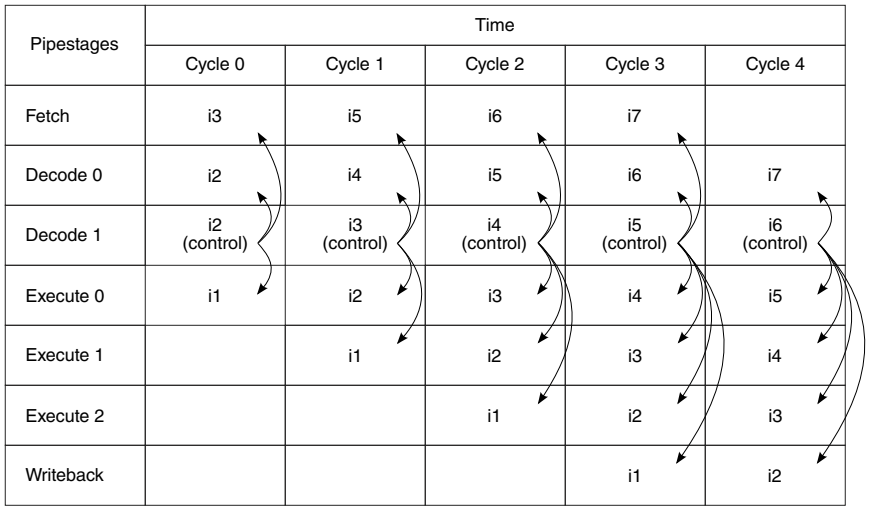


Figure 3. Pipeline Flow for a VLIW Engine

In general, operations encoded in a VLIW instruction are not limited to operations meant for one specific task. Instead, these operations specify multiple parallel operations performed at all pipe stages throughout the engine. Specifically, the parallel operations specified in a single VLIW instruction are used for operations in different pipe stages. For example, in Figure 3, when instruction *i5* reaches Decode-1 stage in cycle 3, it sends its control signals to Fetch, Decode-0, Execute-0, Execute-1, Execute-2 and Writeback in that same clock cycle.

NOTE

It is the programmer's responsibility to understand all of the concurrent activities that are taking place in the neighboring pipe stages when an instruction reaches the Decode-1 stage, so that the right control functions can be specified.

A traditional microprocessor is more "instruction centric" in the sense that an instruction specifies a precise architectural function that needs to be performed. The architectural state between two instructions (or at the instruction boundary) can be clearly identified. In such traditional machines, the pipeline operations can be clearly described by an instruction pipeline.

On the other hand, the VSPA Engine is more "data centric" in the sense that the instructions are coded around the data flow. That is, the instructions are coded to maximize the throughput of data flowing through the pipeline.

The architectural state of a VLIW engine is often ill-defined across instruction boundaries since each instruction in the engine specifies operations that affect data operations in other pipe stages. In such VLIW engines, the pipeline operations are best described by two closely connected pipelines: an instruction pipeline and a data pipeline. The two pipelines must work perfectly in tandem to produce the correct results and to achieve maximum throughput rate.

3.1.1 Program memory

The Program Memory (PMEM) is used to store program codes that are written in VSPA macroinstructions. The PMEM is composed of Program RAM (PRAM).

See Figure 4.

- The PRAM address spaces contain up to 32768 locations.
- Each location can store one 64-bit macroinstruction.
- PRAM address space starts at address 0

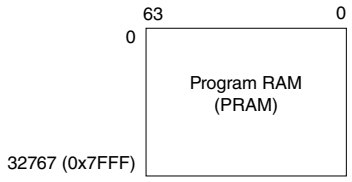


Figure 4. Program memory address space

The actual physical sizes of the PRAM address space are implementation-specific. However, it will be no larger than 32768 lines.

3.1.2 Program control

The Program Control Unit controls all VCPU instruction fetches, control-flow re-directions, and loop execution. The instruction set supports the following control-flow re-directions:

- Conditional and unconditional jumps
- Conditional and unconditional subroutine calls
- Return from subroutine

There are no sources of interrupts. Software threads begin with a wake-up or 'go' event, and complete with execution of the 'done' instruction, which puts the VCPU into a low power state.

NOTE

The done instruction will immediately halt the execution of the VCPU. Software must make sure that all instructions have been completed before 'done' is executed. Instructions with extra latency, such as load/store, should have an appropriate number of other instructions before a 'done'.

All jumps take 3 clock cycles to complete. They will execute 2 branch delay slots. That is, the 2 instructions immediately following the jump instruction will always be executed, even if the jump is taken.

2 adds executed after a jmp

```
jmp target;
add a2,a3;
add g0,g0,g1;
```

When the jmp instruction is executed, the following two 'add' instructions will also be executed, even though the jump is taken.

mv and ld executed after rts is taken

```
rts;
mv   a0,0;
ld   [a0]+a1;
```

In this example, the 'mv' and the 'ld' instructions will be executed, even though the rts is taken.

3.1.2.1 Return address stack

The 'jsr' instruction allows a program to conditionally or unconditionally jump to a subroutine. Sometime later, the subroutine will execute an 'rts' instruction, where program execution returns to the calling function.

The VCPU maintains a 16-deep Return Address Stack (RAS). This allows up to 16 sequential subroutine calls without an 'rts'. Note, when a jsr instruction is executed, the address 2 words ahead of the jsr instruction is pushed onto the RAS.

CAUTION

VCPU RAS pointer is four bit. There is no hardware recovery mechanism for RAS overflows. The maximum permissible depth for nested subroutine calls is 16. Overflow is not reported, however, the simulator will generate an error when the RAS overflows. It is the user responsibility to avoid error conditions such as underflow and overflow.

3.2 Data plane

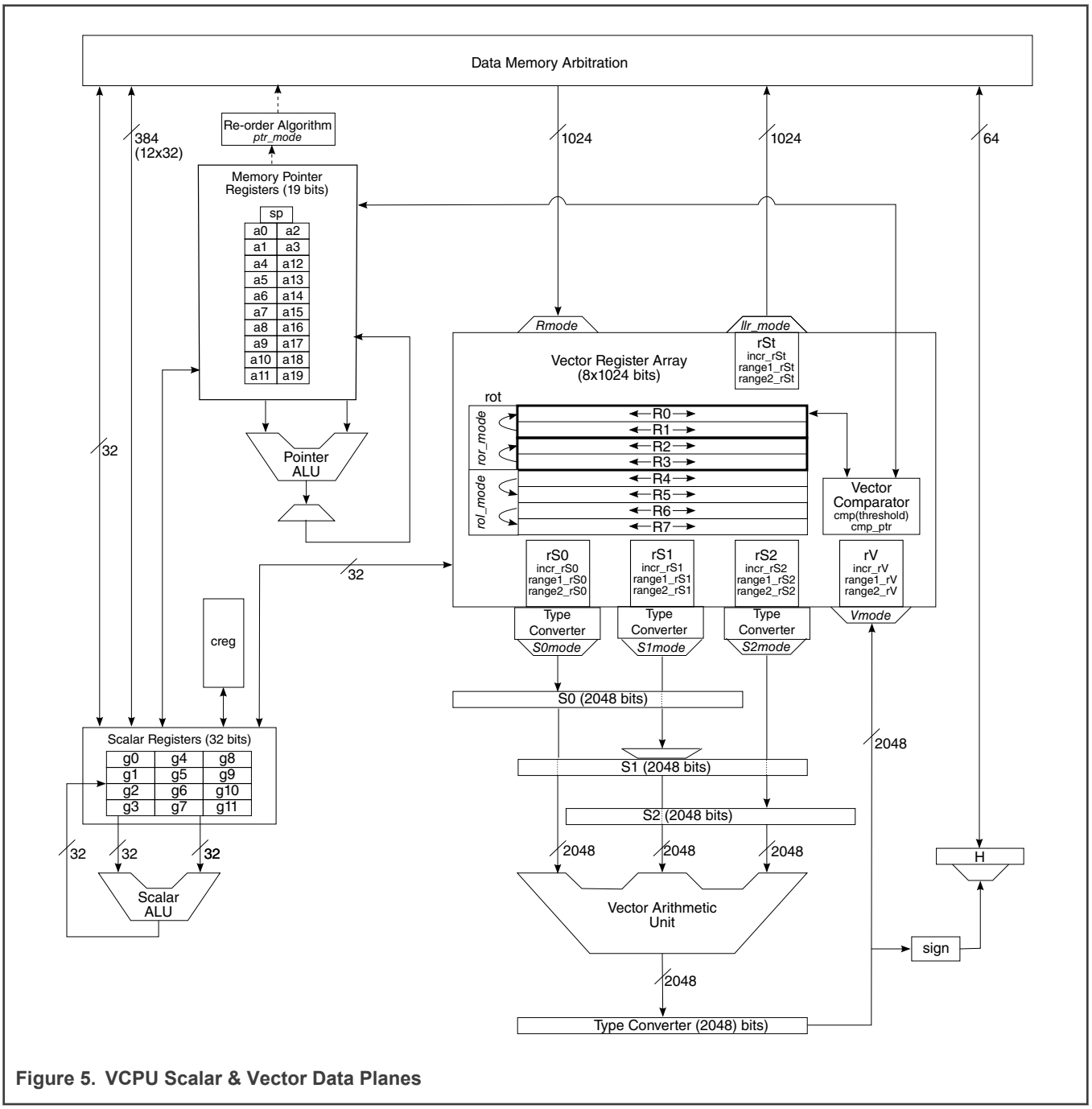


Figure 5. VCPU Scalar & Vector Data Planes

Table 3 shows all major functional units in the VCPU data plane.

Table 3. VSPA Functional Units

Unit		Descriptions
VRA	Vector Register Array	The array consists of eight 1024-bit rows or registers. • The array is addressed as a one-dimensional array in half-word increments. • Each register contains 64 half-words.
S0*mode* S1*mode* S2*mode*	VAU Operand Source Register Muxes	Used for intra-vector permutation of data read from the VRA. **NOTE** Data type conversion is also implemented here.
S0 S1 S2	Source Operand Registers	Store source operands for the VAU. Each source operand register is 2048-bits wide.
V*mode*	VAU Output Mux	Performs various intra-vector permutations on the output results of the VAU before they are written back to the VRA.
R*mode*	DMEM Load Mux	Performs various intra-vector permutations on the data read from the DMEM before it is written to the VRA.
llr_*mode*	DMEM Store Mux	Performs various data compression operations on data read from the VRA before it is written to DMEM.
VAU	Vector Arithmetic Unit	Vector containing 16 individual arithmetic units (AUs). • Each AU can perform a single complex operation or 4 real operations. • Each pair of AUs can also perform a decimation in time (DIT) or a decimation in frequency (DIF) butterfly operation.
rot	Vector Rotate Unit	Performs rotate functions on VRA rows: R0R1-combined, R0-only, R1-only, R2R3-combined, R2-only, or R3-only. R4R5-combined, R4-only, R5-only, R6R7-combined, R6-only, or R7-only.
NCO	Vector Numerically-Controlled Oscillator	Generates vectors of complex exponential samples for use as twiddle factors in FFT and mixing operations.
a*X*, pAU, *ptr_mode*	DMEM Pointer Registers, Pointer AU, & reorder algorithm (*ptr_mode)*	Generates word addresses that are used to access DMEM. Supports various arithmetic operations and useful reorder algorithms.
rS0, rS1, rS2, rV, rSt	VRA pointers	5 sets of pointers that are used for addressing the contents of the VRA. There is a dedicated pointer for each port in the VRA: • S0 read-port • S1 read-port • S2 read-port • DMEM store read-port

Table continues on the next page...

Table 3. VSPA Functional Units (continued)

Unit		Descriptions
		• VAU output write-port These pointers support auto-increment, auto-decrement, and modulo operations (for buffer management).
VCMPU	Vector Compare Unit	Independent engine that performs various vector compare operations. The results of these compare operations set condition bits used to determine branch directions.
g*X*, & sALU	General Purpose Scalar Registers and Arithmetic and Logic Unit	• Performs general purpose arithmetic, including fixed-point add, subtract, logical and, logical or, shift left, logical or arithmetic shift right, and so on. • Also performs data movements between scalar registers and other hardware resources (such as IP-registers and pointer registers).
H	Vector Sign Register	Captures the sign of each AU operation. • Provides storage for up to 4 VAU operations in a single register. • Some operations are provided that are useful in peak-clipping algorithms.
creg	Control Register	An array of data plane control parameters. The array is addressed in nibbles.
Type Converter		Converts between the various data types used in the VAU and VRA.
VPx	Vector predicate registers	Four 128-bit registers holding predicate bits (zero or non-zero flags).
VPRED	Vector predication unit	Provides support for vector predication of real-mode vector instructions.

3.2.1 Data memory

The VCPU Data Memory (DMEM) is used for temporary program variable storage. It is organized into multiple lines where each line is 1024-bits wide.

Each DMEM address uniquely identifies a single hword in the DMEM. The first DMEM line starts at address 0; the second DMEM line starts at address 64; the third DMEM line starts at address 128 and so on.

A DMEM line follows the little endian addressing convention, that is, the lowest address unit is the right-most (or the least significant) hword on the line. Figure 6 shows how data is packed in a DMEM line. Complex samples are composed of real (I) and imaginary (Q) components, where the real portion is at the lower address unit(s) and the imaginary portion is at the higher address(es). For half-precision representation, the real part occupies the lower 16 bits and the imaginary part occupies the higher 16 bits.

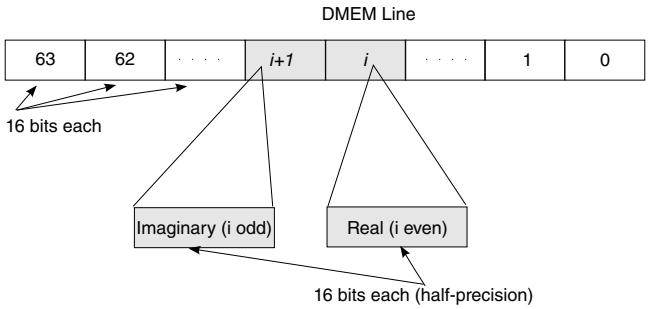


Figure 6. Data Packing in a DMEM Line

3.2.2 Data memory pointers

DMEM is addressed by pointer registers. Each pointer is 19-bits, but only 22 bits are used to address DMEM (the most significant portion indicates the line index and the least significant portion identifies a hword in the line).

The pointer AU is an independent unit, which is useful for modifying pointers in parallel with memory accesses. It performs operations on pointer registers, denoted as a*X* registers.

3.2.2.1 Pointer registers

There are 20 a*X* registers: a0 - a19. The following operations can be performed with the a*X* registers:

- Address DMEM using the content of an a*X* register. The a*X* register is then optionally post-incremented or post-decremented.
- Load a 19-bit unsigned immediate value from an instruction into an a*X* register.
- Perform an arithmetic operation on 2 a*X* registers, and write the result to an a*X* register.

3.2.2.2 Hardware buffer management

The a0, a1, a2 and a3 registers (of a*X* registers) also have a modulo mode that is useful for circular buffer management.

The boundaries of a circular buffer are defined by a range_a*X* register, where:

- The range_a0 register defines the boundaries of the circular buffer associated with a0.
- The range_a1 register defines the boundaries of the circular buffer associated with a1.
- The range_a2 register defines the boundaries of the circular buffer associated with a2.
- The range_a3 register defines the boundaries of the circular buffer associated with a3.

Table 4 shows all the a*X* registers.

Table 4. Address Registers (a*X*)

a*X* Registers	Modulo addressing support	Modulo buffer registers
a0	Yes	range_a0
a1	Yes	range_a1
a2	Yes	range_a2
a3	Yes	range_a3

Table continues on the next page...

Table 4. Address Registers (a*X*) (continued)

a*X* Registers	Modulo addressing support	Modulo buffer registers
a4 - a19	No	NA

Upon a hardware reset, the range_a*X* registers are initialized for linear addressing (non-modulo).

3.2.2.3 Pointer reordering algorithms

A pointer address can be modified according to a reorder algorithm specified by the instruction parameter, *ptr_mode*. Two algorithms are available:

- Normal Mode (no reorder)
- Bit-Reversal Mode

Figure 7 shows the DMEM address generation flow.

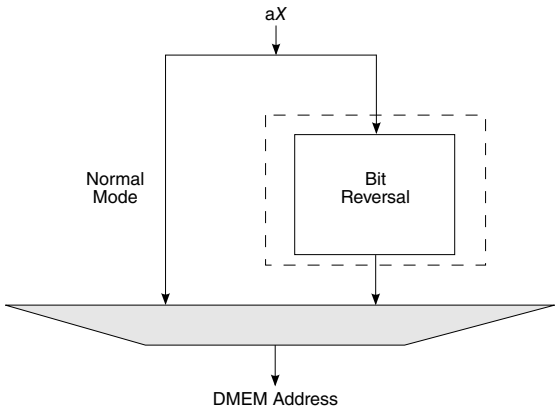


Figure 7. Pointer reordering algorithm block diagram

3.2.2.4 Normal mode

When a DMEM address is generated using Normal mode, the address is pulled directly from an a*X* register. No additional computation is performed.

3.2.2.5 Bit-reversal mode

Bit-Reversal mode is used to generate an index into an array of FFT samples.

Before accessing DMEM using Bit-Reversal mode, you must initialize the following:

- set.br, *fft_size* Initializes the br state machine for pointer a*X* with a specific fft size.

Table 5 defines the bit reversal algorithm where certain bit positions of the a*X* register are reversed according to the specific FFT size.

Table 5. Bit Reversal Mode

FFT Size	Bits Reversed in the lower portion of a*X* register	DMEM Address
32	a*X*[5:1]	{a*X*[18:6], a*X*[1:5]}

Table continues on the next page...

Table 5. Bit Reversal Mode (continued)

FFT Size	Bits Reversed in the lower portion of a*X* register	DMEM Address
64	a*X*[6:1]	{a*X*[18:7], a*X*[1:6]}
128	a*X*[7:1]	{a*X*[18:8], a*X*[1:7]}
256	a*X*[8:1]	{a*X*[18:9], a*X*[1:8]}
512	a*X*[9:1]	{a*X*[18:10], a*X*[1:9]}
1024	a*X*[10:1]	{a*X*[18:11], a*X*[1:10]}
2048	a*X*[11:1]	{a*X*[18:12], a*X*[1:11]}
4096	a*X*[12:1]	{a*X*[18:13], a*X*[1:12]}
8192	a*X*[13:1]	{a*X*[18:14], a*X*[1:13]}
16384	a*X*[14:1]	{a*X*[18:15], a*X*[1:14]}
32768	a*X*[15:1]	{a*X*[18:16], a*X*[1:15]}
65536	a*X*[16:1]	{a*X*[18:17], a*X*[1:16]}

3.2.3 Vector register array

The Vector Register Array(VRA) behaves like a cache between DMEM and the AUs. It consists of 8 registers (R0 - R7), where each register is 1024 bits wide. The width of these registers matches the width of the DMEM lines. Figure 8 shows how data is packed into the VRA.

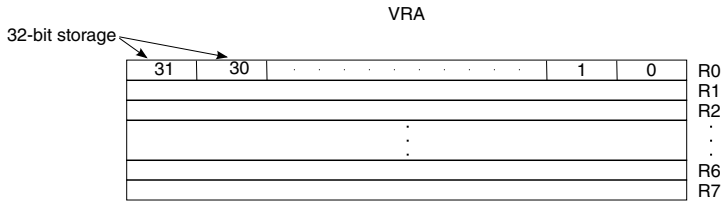


Figure 8. Data Packing in the VRA

Similar to a DMEM line, each register in the VRA can contain either 32 words or 64 half-words. Each register in the VRA follows the little endian addressing convention, that is, the lowest addressable unit is the right-most (or the least significant) half-word in the register line. Within a 32-bit word, in half-precision representation, the real part occupies the lower 16-bits and the imaginary part occupies the higher 16-bits. Figure 9 shows how data is packed in a VRA register line.

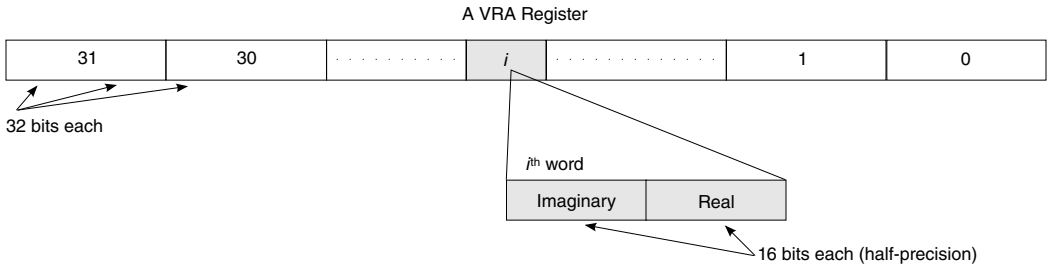


Figure 9. Data packing in a VRA register line

The VRA can store either 8 * 64 = 512 half-words (or 8 * 32 = 256 full-words). Ignoring the VRA register boundaries, the VRA can be addressed as a 1-D array in half-word increments. The array is accessed using pointers.

3.2.3.1 VRA read/write ports

The VRA ports enable up to 6 reads and 3 writes to occur during each clock cycle. However, not all read and write operations can access all registers in the VRA. Certain operations have restrictions on which registers they can access. Table 6 shows all possible read and write operations, and associated access restrictions.

Table 6. VRA Read and Write operations

Port connection	Read or Write port	Register access restrictions	How Source/Destination registers are specified	Pointer register width	Access granularity
S0	Read	None	rS0	9 bits	line or half-word
S1	Read	None	rS1	9 bits	line or half-word
S2	Read	None	rS2	9 bits	line or half-word
DMEM Store	Read	None	rSt	3 bits	line only
VAU output	Write	None	rV	9 bits	line or half-word
DMEM Load	Write	None	Load instruction 'ld Rx' parameter	NA	whole or part of line
rot input	Read	R0/R1 together or R2/R3 together (right rotate) R4/R5 together or R6/R7 together (left rotate)	set.rot,...' instruction	NA	line or two lines
rot output	Write	R0/R1 together or R2/R3 together (right rotate) R4/R5 together or R6/R7 together (left rotate)	'set.rot,...' instruction	NA	line or two lines
VCMPU	Read	None	rS2	NA	half-word

3.2.3.2 VRA operation source and destination

The VRA source or destination register can be specified implicitly or explicitly using pointers, as shown in Table 6. There are 5 ports that utilize a specific pointer for accesses to the VRA: rS0, rS1, rS2, rV, rSt.

3.2.3.3 VRA access granularity

The VRA's read and write ports have different access granularities defined in Table 6. Accordingly, the pointers that are used to address the VRA also have different widths.

3.2.3.4 VRA pointer control registers

Operation of each VRA pointer is specified with a set of 3 registers: r*X*.incr, r*X*.range1, r*X*.range2. Figure 10 shows all 5 VRA pointer sets.

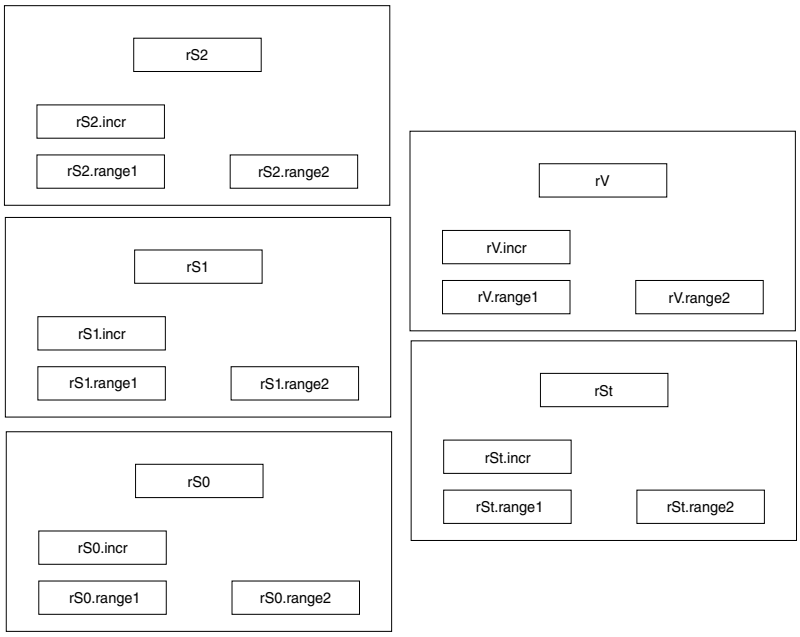


Figure 10. VRA pointer register sets

The functionality of the registers is as follows:

- r*X.incr* - Contains the 9 bit offset of the post increment/decrement for r*X*. This is a signed integer.
- r*X*.range1 - Contains the 9 bit beginning and wrap addresses of the 1st modulo buffer in the VRA. If the result of a post increment/decrement operation is equal to the wrap address, then the pointer is set to the beginning address of the buffer. Refer RAG pointer update algorithm. The 1st modulo buffer is disabled by setting the r*X*.range1 register to zero.
- r*X*.range2 - Contains the 9 bit beginning and wrap addresses of the 2nd modulo buffer in the VRA. If the result of a post increment/decrement operation is equal to the wrap address, then the pointer is set to the beginning address of the buffer. Refer RAG pointer update algorithm. The 2nd modulo buffer is disabled by setting the r*X*.range2 register to zero.The 2nd modulo buffer can only be enabled if the 1st modulo buffer is also enabled.

Collectively, these pointer register sets allow you to configure a total of 10 circular buffers inside the VRA. Upon reset, the r*X*.range1 and r*X*.range2 registers are set to zero, disabling both circular buffers for each VRA pointer.

The VRA pointer registers can be configured using the following instructions:

- setA.VRAptr r*X*,...- There is also an OpB version of this instruction. See VCPU instruction set summary
- set.VRAincr r*X*,...

- set.VRAptr,... - Configures the 9-bits of all pointers in 1 instruction(cycle).
- set.VRAincr,... - Configures the 9-bits of all increments in 1 instruction(cycle).
- set.VRArange1 r*X*,... - Configures the 9-bits of the range registers (start1 and wrap1 addresses) in 1 instruction(cycle).
- set.VRArange2 r*X*,... - Configures the 9-bits of the range registers (start2 and wrap2 addresses) in 1 instruction(cycle).

The "clr.VRA" instruction can be used to clear the entire set of all VRA pointer registers.

3.2.3.5 VRA data conflicts

There are 4 data input ports on the VRA elements

In order of write priority (highest to lowest):

- `Port 0 – Load port, used by OpV ld.<mode> instructions, ldB Rx & mv Rz, Ry.`
  - `ldB Rx may be used in parallel with ld.<mode> Ry provided they write different vector registers.`
  - `mv Rz, Ry must not be in any cycle between a vector memory load (ld [aX]) and a subsequent ld.<mode> instruction.`
- `Port 1 – Write back port, used by write back (wr.<mode>) from AU, mv [rV], gX; fill [rV], gX; lsb2rf [rV], gX; mv [rV] I;`
- `Port 2 – Rotate port, used exclusively by VRA rotate instructions`
- `Port 3 – Zone mask port, used exclusively by clr.VRA gX, gY;`

If a conflict occurs within a port, the results are undefined and an error should occur in the assembler. If there is no conflict between ports no error should occur.

The write back port (1) is the general purpose port for the VRA, any instructions not listed, which modify the VRA, will probably utilize this port.

3.2.4 DMEM address space vs. VRA address space

The address space for Data Memory (DMEM) and the address space for the Vector Register Array (VRA) represent 2 distinct data storage spaces. Address 0 for DMEM, for example, is a different storage location than address 0 for the VRA. The 2 address spaces are distinguished by these facts:

- The DMEM is an external memory which can be shared with other VSPA processing units, such as the IPPU.
- The access granularity for DMEM is a 16-bit half-word. The valid address space for DMEM ranges from 0 through 524287, where each address in this space points to a unique half-word.
- The access granularity for the VRA can be either a word or a half-word.
- Address generations for DMEM is handled by the MAG, which generates 19-bit addresses to access the DMEM.
- The VRA is an internal register array which can only be accessed by the VCPU.
- Address generations for the VRA is handled by the RAG unit. The RAG generates 9 bits addresses to access the VRA.
  - When the VSPA Engine operates on a vector using a "complex" instruction, only the upper 8 bits of the address are used; the least significant bit of the address is ignored.
  - When the VSPA Engine operates on a vector using a "real" instruction, all 9 bits of the address are used.

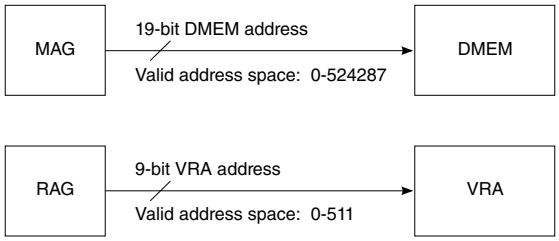


Figure 11. Address spaces for DMEM and VRA

3.2.5 Vector rotate unit

The Vector Rotate Unit performs rotate-right and/or rotate-left operations in half-word increments (which are specified by the *rot_mode* parameter of the 'set.rot ...' instruction). Rotations of 1, 2, 4, and 8 half-words are available. This is an independent machine and it operates in parallel with the other components in the vector data plane. The unit supports rotations on the following VRA registers:

- R1R0 where R0[0] is rotated into R1[63], and R1[0] is shifted right into R0[63]
- R0-only where R1[0] is shifted right into R0[63]
- R1-only where R0[0] is rotated into R1[63]
- R3R2 where R2[0] is rotated into R3[63], and R3[0] is shifted right into R2[63]
- R2-only where R3[0] is shifted right into R2[63]
- R3-only where R2[0] is rotated into R3[63]
- R5R4 where R4[63] is rotated into R5[0], and R5[63] is shifted left into R4[0]
- R4-only where R5[63] is shifted left into R4[0]
- R5-only where R4[63] is rotated into R5[0]
- R7R6 where R6[63] is rotated into R7[0], and R7[63] is shifted left into R6[0]
- R6-only where R7[63] is shifted left into R6[0]
- R7-only where R6[63] is rotated into R5[0]

3.2.6 VAU operand source registers

There are 3 source registers (S0, S1, S2) used as the input to all VAU operations. Each register is 2048-bits wide and contains 64 single precision variables.

3.2.7 VAU operand source register muxes

There are 3 VRA ports allocated to feed the VRA source registers with data each clock cycle. These instructions use the syntax, 'rd S0', 'rd S1', or 'rd S2'.

- A bit is allocated for each port in the instruction word, so that all 3 reads can occur in a given cycle.
- The operation completes in 2 cycles but is pipelined, which enables single-cycle throughput to the Vector AU.

3.2.7.1 Port permutation modes

Data read from the VRA into the VAU operand source registers can be re-arranged, according to a permutation algorithm or mode. Many permutation modes are available for each port, although each set is not identical. A given port's set of permutation modes have been determined by benchmarking a large number of signal processing algorithms on the core. More modes are available on the S1 port, which feeds one leg of the VAU multiplier. In many single input algorithms (such as an FIR filter), this

port carries the input sample stream. The other leg of the multiplier is fed by the S0 port from the VRA. It can read the filter coefficients in an FIR filter. The S2 port feeds the adder in the VAU and has the fewest possible permutation modes.

The intra-vector permutation mode for ports, S0, S1, and S2 is specified by the instruction parameter *S0mode*, *S1mode* and *S2mode,* respectively, using the following instruction:

- set.Smode ...

This is a sticky operation, which means that once the permutation mode is set, it remains in this state until changed by the next 'set.Smode ...' instruction. See Rotate register instructions for more details.

3.2.7.2 Data type conversion

These data paths from the source register VRA ports are also responsible for data type conversion. VAU operations can be in single precision. However, the contents of the VRA can be half-fixed (16 bits), half precision floating point (16 bits), single (32 bits). Proper operation in the VAU requires type conversion to occur prior to latching data in the operand source registers. The precision of the data read from VRA port, S0, S1, and S2 is specified by the instruction parameter, *S0prec*, *S1prec* and *S2prec*, respectively, using the following instruction:

- set.prec

Data type conversion will occur if there is a mismatch between the VAU precision (specified by *AUprec*) and the source register port precision.

This is a sticky operation, which means that once the precision parameter is set, it remains in this state until changed by the next 'set.prec' instruction.

3.2.8 Vector arithmetic unit

The VAU is composed of arithmetic units (AU) and special arithmetic units (SAU). Each AU has a throughput of 1 complex operation or 4 real operations per clock cycle. It has a pipeline latency of 4 clock cycles for SP operations. One Pair of AUs can implement a radix-2 butterfly operation. There are 16 AUs that support single precision operations. Each SAU has a throughput of 1 operation per clock cycle which may be performed in parallel with other AU operations. The composite throughput of the VAU can be summarized:

- 64 SP real linear operations per clock cycle
- 16 SP complex linear operations per clock cycle
- 32 SP non-linear operations per clock cycle
- 8 SP butterfly operations per clock cycle

The VAU precision is specified by the *AUprec* parameter using the 'set.prec ...' instruction. The mode is sticky, so the VAU remains in a set mode until it is changed by another set instruction.

3.2.8.1 Arithmetic unit

The AU implements the linear operations within the VAU which are:

- Multiply-add-real (rmad): V[i][n] = (S0[i][n-4]*S1[i][n-4]) + S2[i][n-4]
- Multiply-add-complex (cmad): V[i][n] = (S0[i][n-4]·S1[i][n-4]) + (S0[i+1][n-4]·S1[i+1][n-4]) + S2[i][n-4]
- Multiply-accumulate-real (rmac): V[i][n] = (S0[i][n-4]*S1[i][n-4]) + V[i][n-1]
- Multiply-accumulate-complex (cmac): V[i][n] = (S0[i][n-4]·S1[i][n-4]) + (S0[i+1][n-4]·S1[i+1][n-4]) + V[i][n-1]
- Multiply-add-feedback (maf): V[i][n] = (V[i][n-4]*S1[i][n-4]) + S2[i][n-4]
- Multiply-add-sign (mads): V[i][n] = (S0[i][n-4]*S1[i][n-4]) + sign((S0[i][n-4]*S1[i][n-4]))*S2[i][n-4]
- Multiply-feedback-accumulate (mafac): V[i][n] = (V[i][n-4]*S1[i][n-4]) + V[i][n-1]
- Pipelined with 4-cycle latency for SP operations

NOTE

'n' is the cycle count and 'i' is the vector element number.

The radix-2 butterfly operations by a pair of AU's include:

- Decimation-in-time Butterfly (cmad)
- Decimation-in-frequency Butterfly (dif.sau)
- SP support only
- Pipelined with 4-cycle latency

The VAU operation results are stored into the register (V), which is hidden in the programming model.

3.2.8.2 Special arithmetic unit

The SAU implements the non-linear operations within the VAU. The results of the SAU operation may then be used as a source operand for a subsequent AU operation, or they may be written directly back to the VRA.

The non-linear operations supported include:

- Reciprocal (rcp): SAUout = 1/S1
- Reciprocal Square-root (rrt): SAUout = 1/sqrt(S1)
- Square-root (srt): SAUout = sqrt(S1)
- Pre-Add (padd): SAUout = S1+S2
- Pipelined with two cycle latency

The result of the SAU operation may be used as an input to an AU operation by using the suffix '.sau' which is available on several AU instructions:

The AU operations which support using the SAU result are:

- Multiply-add (rmad.sau/cmad.sau): V = S0*SAUout + S2
- Multiply-accumulate (rmac.sau/cmac.sau): V = S0*SAUout + V
- Multiply-add-sign (mads.sau): V = S0*SAUout + sign(S0*SAUout)*|S2|
- Decimation-in-frequency Butterfly (dif.sau)

The result of the SAU operation may be written back to the VRA by using either of the following instructions:

- wr.fn: Write an entire vector of SAU results to the VRA
- wr.fn1: Write one element SAU result to the VRA

The SAU operation results are stored in an internal register (SAUout), which is hidden in the programming model.

3.2.9 VAU destination mux

The contents of the VAU output register are written back into the VRA through a dedicated port using the pointer, rV.

Data type conversion is controlled using the *Vprec* parameter of the 'set.prec ...' instruction. This specifies the precision of the data being written into the VRA. The contents of V are always stored in single precision, depending on the state of *AUprec*.

Permutation modes are also available in this path and are specified with the *Vmode* parameter of the write-result instruction, 'wr.Vmode R[rV] ...'. This is a single cycle operation, thus data is written from V into the VRA in one clock cycle. Note, Vmode is *not* sticky, and it must be specified in each write-result instruction.

3.2.10 Vector sign capture register (H)

The vector sign register (H) captures the sign of each VAU operation, providing storage for up to four VAU operations in a single register and enabling continuity of peak detection across multiple blocks. When a peak search is performed, the H register is

scanned looking for a 0->1 transition. A “skip offset” value, which must be less than the width of the H register (256 bits), is also provided to indicate that the next search will resume after skipping ahead "n" samples from the point it finds the 0->1 transition. In cases where a 0->1 transition is detected at the end of the current block, the skip value may take the pointer to the next peak search start location to a position that is beyond the current block. In this case a “skip mask” value is captured so that the "skip ahead" region is marked. When the next block of 256 bits is loaded, the search should start from just a few samples into the block. This allows for an AND or OR of the “skip mask” to the new register to effectively remove all transitions from the skip region (OR is used for 0->1, while AND is used for 1->0 transitions).

The vector sign capture register can be reset by the following code sequence:

```
set.creg 12,0;
set.creg 12,6; // Enable sign capture with auto-increment, initial state = 0
```

See System control registers, H register control for a full description of the vector sign capture control.

3.2.11 Vector NCO

The Vector NCO is used to generate complex exponential sequences used in mixing algorithms and twiddle factors for DFT and FFT algorithms. The data is fed directly into the VAU through source operand S1. AUprec must be constrained to be single or F24. If AUprec=F24, the number of NCO operations per cycle is 32.

There are three modes available (which are specified with the 'set.nco ...' instruction):

- radix2: generates 8 twiddle factors per cycle, which are loaded into S1 for use in VAU butterfly operations.
- singles: generates a single twiddle factor per cycle and replicates it across all elements of S1.
- normal: generates 16 samples of a complex tone each cycle. The vector is generated into S1 in preparation for complex multiply operations. This mode is typically used for generation of the injection signal in a digital mixing algorithm.

The base frequency of the NCO is specified using the nco_freq register. The phase is accessible via the nco_phase register.

3.2.12 Scalar arithmetic & logic unit

The Scalar ALU performs general purpose scalar arithmetic and logic operations on a set of twelve 32-bit registers; g0, g1, ..., g11.

A rich set of operations is available and summarized in Table 7. Many of the instructions support conditional execution, optional sign extension, full-word and half-word immediate forms, among other useful features.

Table 7. Summary of scalar ALU operations

General function	Operations
Arithmetic Operations	add, subtract, multiply, divide, modulo, absolute-value, base-2 logarithm,find-first-one, find-first-0-to-1, find-normalize-shift
Logical Operations	bit-test, bit-clear, and, or, not, xor, shift-right, shift-left, linear-feedback-shift-register
Data Type Conversions	Fixed-point to floating-point & vice-versa
Data Movement	• Between 2 scalar registers • Between a scalar register and an IP register • Between a scalar register and a pointer storage register • Between a scalar register and a VRA element. • Between scalar registers and operand generator registers (such as the NCO)

The scalar ALU also serves as a control hub that connects various major functional units within VSPA. It is connected to the IP registers, VRA, pointer storage registers and some other hardware resources (such as NCO registers, and so on). The scalar ALU also has direct access to DMEM. Figure 12 shows the connectivity of the Scalar ALU to other components in VSPA.

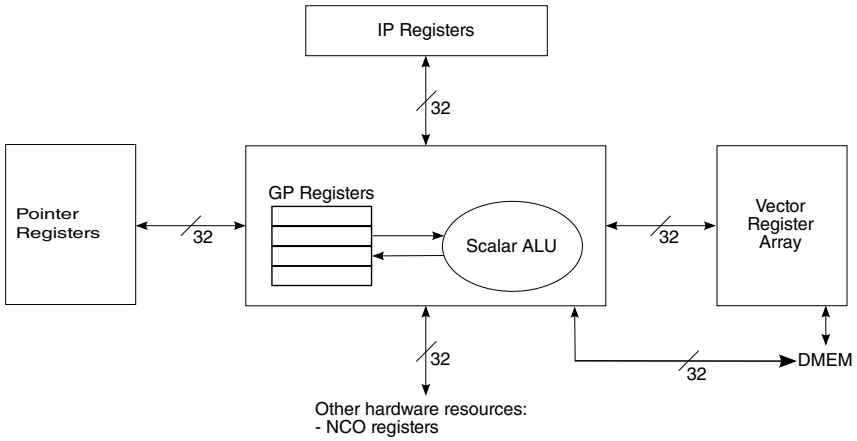


Figure 12. Scalar ALU connectivity

3.3 Data precision

This processor supports the following different data precisions:

Single (SP)	32-bit single-precision floating point
Half (HP)	16-bit half-precision floating point
Half-fixed (HF)	16-bit half-precision fixed point

Floating point representation is compliant with IEEE754(truncation) with some notable exceptions: (1) sub-normals are not supported and (2) NAN and Infinity are not supported (treated as large magnitude numbers).

Throughout this document the terms 'half', "half_float" and 'HP' will always refer to a 16-bit half-precision floating point value.

VSPA uses a signed-magnitude representation for HPfixed numbers as illustrated in Figure 13 below.

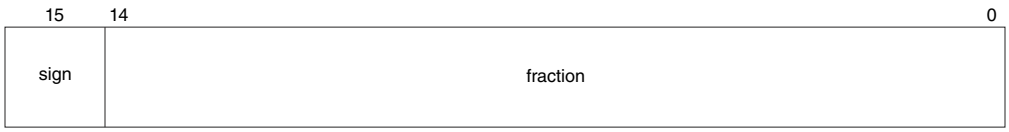


Figure 13. VSPA Half-fixed data precision

This format has two fields: a 1-bit *sign* and an unsigned 15-bit *fraction*. The *sign* field encodes the sign of the floating-point number (0 for positive and 1 for negative). The *fraction* field is a 15 bit integer in the range [0, 32767] with the most significant bit next to the sign bit.

The fractional number is given by:

$$(-1)^{sign} \times \frac{fraction}{2^{15}}$$

For example, 0x4000 represents 0.5, and 0xC000 represents -0.5. Note, this format can only represent numbers with a magnitude less than one. Therefore, 0x7FFF represents the maximum positive number 0.999969482421875 and 0xFFFF represents the maximum negative number -0.999969482421875. Both positive and negative zero representations are possible.

One benefit of the signed magnitude representation is the quantization noise observed in many algorithms will not have a DC bias.

The table below provides the constansts for AU single and AU F24 data precisions.

Table 8. Supported machine dependent constants

		AUprec single	AUprec F24	Notes
Safe minimum, such that 1/sfmin does not overflow	sfmin	2^-126=1.18e-38	2^-14=6.10e-5	No sub-normal support
Base of the machine	base	2	2	-
eps*base	prec	2.38e-7	7.62e-6	eps*2
Rounding	rnd	0	0	-
Minimum exponent before (gradual) underflow	emin	-126	-14	-
Underflow threshold - base**(emin-1)	rmin	2^-126=1.18e-38	2^-14=6.10e-5	No sub-normal support
Largest exponent before overflow	emax	128	16	-
Overflow threshold - (base**emax)*(1-eps)	rmax	2^128*(1-eps)=3.40e38	2^16*(1-eps)=6.55e4	-

3.3.1 Two's complement conversion

In some cases, data may be loaded from external devices into VSPA as a two's complement value. Unlike VSPA's internal data precisions, described in section Data precision, this data is asymetric. That is, it can represent a larger negative number than a positive number. For example the maximum negative number which may be represented in a 16-bit two's complement format is -32768 (0x8000) but the maximum positive number is 32767 (0x7fff).

VSPA has hardware which can convert this two's complement data into the half-fixed format shown in VSPA Half-fixed data precision. When doing these conversions the maximum negative 16-bit two's complement number cannot be converted to an exact value in the half-fixed format and instead must be saturated to the maximum negative value of that format. So the value -32768 (0x8000) in 16-bit two's complement will be converted to 0xffff in half fixed. This saturation applies to data converted using the DMA's AXI to DMEM format conversion and to data converted using the VCPU instruction 'ld.2scomp' when it is configured for 16-bit conversions.

NOTE

Saturation is not necessary for the 'ld.2scomp' instruction when configured for 8, 10 or 12-bit twos complement conversions since the maximum negative values in these formats can be exactly represented in the half-fixed format.

Data converted from half-fixed to 16-bit twos complement using the DMA's DMEM to AXI format conversion does not need to be saturated since the maximum negative value of half-fixed 0xffff can be exactly represented in the 16-bit twos complement format as 0x8001 (-32767). However a '-0' (0x8000) in half-fixed will be converted to '0' (0x0000) in 16-bit twos complement.

The representations of different numbers in two formats is shown in the table below as well as the treatment of the special cases.

Table 9. Special cases for twos complement conversions

Number[1]	2's complement hex	1's complement[2] hex	Special case
-32768	0x8000	NA	16-bit twos complement value is converted to 0xffff (-32767 1's complement)
-32767	0x8001	0xffff	No special conversion
-1	0xffff	0x8001	No special conversion
-0	NA	0x8000	1's complement value is converted to 0x0000 (0 16-bit twos complement)
0	0x0000	0x0000	No special conversion
1	0x0001	0x0001	No special conversion
32767	0x7fff	0x7fff	No special conversion
32768	NA	NA	No special conversion

1. VSPA's half-fixed format is used to represent fractional numbers and not integers as shown in this table.
2. 1's complement is a sign magnitude format similar to VSPA's half-fixed and is used here as it can represent integers.

3.4 Data types

The processor supports multiple data types.

Table 10. Supported Data Types

Data Type	Unit Size	Description
nibble	4 bits	Typically used to represent bit-log-likelihood ratios.
byte	8 bits	Typically used to represent bit-log-likelihood ratios.
half-word	16 bits	Typically used to represent a half-precision fixed/float point I or Q sample. The data memory and vector register array addressed in half-word increments.
full-word or word	32 bits	Can contain a complex sample in half-precision fixed/float point representation or an I or Q sample in single-precision floating point format.
double-word or double	64 bits	Can contain a complex sample in single-precision floating point representation.
vector	1024 bits	A vector consists of a finite number of scalar elements. An element can be complex or real and can be represented in any supported precision.

3.5 VCPU GO events

The VCPU can be made to GO by several different events. Each event must be acknowledged (cleared) by the VCPU or it will cause another Go immediately after the VCPU executes the DONE instruction.

The GO events are as follows:

- host_go: Occurs whenever an IPbus host sets the host go bit in the CONTROL register. The host software has generated an event that needs servicing by the VCPU. This event is cleared by the VCPU writing a 1 to the host_go bit in the CONTROL register.

- ippu_go: Occurs upon the completion of a 'go requesting' IPPU sequence, that is, the IPPU executed a DONE instruction and was programmed to generate a GO. This event is cleared by writing a 1 to the ippu_go flag in the CONTROL register.
- dma_go_chan[x]: Occurs upon the completion of a 'go requesting' DMA command, that is, a DMA channel finishes that was programmed to generate a GO. These events are cleared by writing 1's to the associated flags in the DMA_GO_STAT register.
- ext_go[x]: Indicates the detection of a rising edge on an ext_VCPU_go input. These events can only occur if armed by the associated bits in the ext_go_ena register. There are two ways to clear ext_go event:
    1. By clearing the corresponding bits in the EXT_GO_STAT register - Writing 1 to the EXT_GO_STAT register bits clears the ext_go event. The EXT_GO_STAT bit will assert again after next rising edge detected on the External GO inputs. This is the recommended way to clear a pending go_event.
    2. By clearing EXT_GO_ENA register - When EXT_GO_STAT is asserted and the EXT_GO_ENA register is negated, the ext_go event is negated. Note the ext_go will assert when EXT_GO_ENA register is asserted (written to 1). Additional ext_go event will not be detected when the EXT_GO_ENA register is cleared. To detect additional ext_go event, the ext_go event should be cleared by clearing EXT_GO_STAT register. Note that clearing an EXT_GO_ENA bit does not clear the corresponding EXT_GO_STAT bit.

3.6 Byte order

The VCPU operates in the little endian data format.

3.7 IRQ for thread killing

There is a corresponding IP register enable bit, and a status bit for vcpu_irq input, that reads the direct state of the vcpu_irq input.

When VSPA is in GO state (BUSY=1), an enabled VCPU_IRQ will force an immediate JMP to PC=([VEC_BASE] + 4 full word instructions). It also forces entry into SUPV mode. No state will be saved.

If VSPA is in DONE state (BUSY=0) when VCPU_IRQ is recognized, VSPA will execute a virtual GO. Rather than starting at PC=0 it will start at PC=[VEC_BASE]+4. To avoid an immediate virtual GO, the VCPU_IRQ status bit must be cleared before executing DONE.

Chapter 4
VCPU Instruction Set

4.1 VCPU instruction set overview

VSPA is a VLIW (very long instruction word) processor. On each clock cycle, the core issues an instruction word (referred to here as a macro-instruction). The macro-instructions contain 1 or more microinstructions that execute in parallel. This section:

- Defines the microinstruction operation code (opcode) mnemonics and their functionality.
- Describes the allowed combinations of microinstructions used to form a macro-instruction.

NOTE

In this section, the term *instruction* refers to a microinstruction, unless denoted otherwise.

4.2 Instruction set organization

The VSPA instruction set provides 4 different macro-instruction formats. Each format defines an allowed combination of microinstruction families suitable for parallel issue to the execution stage in the program pipeline.

See Table 15 for summary of all microinstructions, including opcode mnemonics, arguments, family and functional descriptions.

4.2.1 Instruction families

The microinstructions are organized into families according to functionality, data operands, required number of encode bits, and by need of parallel execution. Many communications signal processing algorithms have been analyzed to determine the optimum grouping of microinstructions into families to minimize cycle counts.

Instructions within a family cannot execute in parallel because among other reasons, they typically use a common hardware component (such as the Vector Arithmetic Unit). The instruction families and their general functionality are summarized in Table 11.

Table 11. Microinstruction families

Family ID		General Functionality
OpV	OpVr	Vector Register Array row store operation
	OpVs0	Vector AU S0 load operation
	OpVs1	Vector AU S1 load operation
	OpVs2	Vector AU S2 load operation
	OpVau	Vector AU math operations
	OpVd	Vector AU result store operation
	OpVrot	Vector Register Array row rotate operation
OpS		Scalar Data Operations
OpA		Vector Data memory access control
OpB		Vector Register Array access control and Do loop

Table continues on the next page...

Table 11. Microinstruction families (continued)

Family ID	General Functionality
OpC	Jumps and general control useful in parallel Vector Data operations
OpD	Multiple immediate data operations
OpZ	Return from Subroutine and Do Loop Start operations
OpX	Used for software breakpoints
OpVp	Vector predicate operations

4.2.2 Instruction formats

There are 4 allowed sets of microinstruction families which can execute in parallel. Each set is called a format and given a numeric identifier (defined in Table 12).

Table 12. Allowed macro-instruction sets

<table>
<tr><th rowspan="2">Format</th><th colspan="10">Bit Position</th></tr>
<tr><th>63</th><th>62</th><th>61</th><th>60</th><th>59 - 43</th><th>42 - 31</th><th>30 - 24</th><th>23 - 2</th><th>1</th><th>0</th></tr>
<tr><td>Format 4</td><td>0</td><td>0</td><td>0</td><td rowspan="4">OpX</td><td colspan="4">OpD</td><td colspan="2" rowspan="5">OpZ</td></tr>
<tr><td>Format 3</td><td>0</td><td>0</td><td>1</td><td colspan="2">OpS</td><td></td><td rowspan="3">OpV</td></tr>
<tr><td>Format 2</td><td>0</td><td>1</td><td>0</td><td colspan="3">OpC</td></tr>
<tr><td>Format 1</td><td>0</td><td>1</td><td>1</td><td>OpB</td><td colspan="2">OpA</td></tr>
<tr><td>Halfword</td><td>1</td><td></td><td colspan="2">OpX</td><td colspan="2">OpS-upper</td><td colspan="2">OpS-lower</td></tr>
</table>

Note that OpX and OpZ are present in all formats:

- OpX is used for software breakpoints.
- OpZ enables a zero cycle overhead for the return-from-subroutine (RTS) and Do-loop start operations.

The assembler supports the use of generic forms of certain instructions. The assembler should use the instruction based on the following rules:

- OpS should have priority over OpC and OpD.
- OpA should have priority over OpC.
- OpB should have priority over OpS, unless OpS can be packed with another OpS.
- OpA should have priority over OpS, unless OpS can be packed with another OpS.

NOTE

Bitfields highlighted in light gray are unused, ignored and should be 0 filled.

An overview of formats:

- Format 1 enables a Vector Data operation (OpV) to occur in parallel with memory accesses (OpA) and Vector Register accesses, among other useful operations (OpB). This is the dominant format used in heavy computational algorithms, because it best uses the parallel hardware resources in the machine. All 0 in this format acts as nop.
- Format 2 is the next most useful macro-instruction set, and is applicable with vector operations not requiring simultaneous memory accesses. Many algorithms do not require vector data memory accesses on each clock cycle and can operate on data in the Vector Register Array for several cycles between memory accesses. All 0 in this format acts as nop.
- Format 3 enables Scalar Data operations (OpS), which are typically in a general purpose RISC machine. Format 3 instruction set allows parallel Vector Data (OpV) operations, but does not allow parallel vector memory accesses (OpA). The OpS family does include scalar memory and register access operations. The bulk of the algorithm control software layer is implemented using instruction format 3. The control layer typically occupies a large portion of the program image. All 0 in this format acts as nop.
- Format 4 includes the OpD family of microinstructions, which require the most encoded bits. All 0 in this format acts as nop, but sets illegal opcode flag.
- Halfword format enables higher code density by encoding 2 OpS instructions into a single program memory word. In this format the 2 OpS instructions will execute sequentially, with the instruction in the ‘lower’ half word executing first and the instruction in the ‘upper’ half word executing next. All 0 in the ‘upper’ half word will be skipped as an execution cycle, effectively acting as a nop which executes in parallel with the ‘lower’ half word instruction. All 0 in the ‘lower’ half word is illegal.

Formats 1 and 2 enable efficient implementation of algorithms using Vector Data operations, such as FFT's, DFT's, equalization, and many other signal processing tasks.

The Vector Data operation (OpV) described in formats 1, 2 and 3 is composed of up to 7 parallel sub-operations (defined in Table 13). These sub-operations specify source, destination, and type of vector arithmetic operation. Note that OpVrot is used for vector rotations in the Vector Register Array.

Table 13. OpV sub-operations

OpVr	OpVs0	OpVs1	OpVs2	OpVau	OpVd	OpVrot

4.3 Internal operand generators

The ISA supports some internal operand generation that is useful in communications digital signal processing. Refer Vector NCO for details.

4.4 VCPU instruction set summary

The general form of a VSPA microinstruction is:

```
OpCode.x.y  destination_list,source_list.
```

The opcode extensions, *x* and *y*, specify optional operations for some of the instructions. The complete set of VSPA microinstructions are summarized in Table 15. Instructions are organized into functional groups. A brief description of each instruction is shown in the table.

The assembler mnemonic conventions used in the instruction set are summarized in Table 14.

Table 14. VSPA instruction set mnemonic conventions

Convention	Meanings
(*)	Implicit (optional) argument in a microinstruction. Also used to identify instructions hidden from the programmer and auto-generated by the build tools.

Table continues on the next page...

Table 14. VSPA instruction set mnemonic conventions (continued)

Convention	Meanings
[*]	Operation using memory or register array pointers. The result of the operation inside the brackets is an address used for the memory or register array access in the given instruction. **NOTE** The order of terms used in the expression is arbitrary.
{ * }	Select one of the operands enumerated within { }.
mv	Move - Indicates an immediate load or transfer between internal registers.
ld	Load - Indicates a read operation from internal data memory.
st	Store - Indicates a write operation to internal data memory.
set	Used to configure a mode or state in a hardware machine, typically references a write-only register.
rd	Read - Indicates a read operation from the Vector Register Array.
wr	Write - indicates a write operation to the Vector Register Array.
*.u	Update the data memory pointer used for a memory access, which can occur before or after the access, depending on the instruction syntax.
*.z	Indicates the 0 extension of a short immediate operand.
*.s	Indicates the sign extension of a short immediate operand.
*.cc	Indicates that the instruction execution depends on a logical test specified by "cc". Refer VCPU condition codes.
*.ucc	Indicates that the instruction will always update the VCPU Condition Codes based on the result of the operation.
*.h	Indicates an operation on a half-word element in a vector.
*.w	Indicates an operation on a full-word element in a vector.
*.e	Indicates an operation on a single element in a vector. The size of the element depends on the precision mode.
*.laddr	Indicates that the immediate offset is specified as number of lines. When extension is not present the immediate offset is interpreted as number of words.
creg	System control register address, where creg $\in$ {0, 1, 2, ..., 255} See Table 21 for a description of the creg registers.
sp	Stack pointer

Table continues on the next page...

Table 14. VSPA instruction set mnemonic conventions (continued)

Convention	Meanings
a*U*, a*V*, a*W*, a*X*, a*Y*, a*Z*	Data memory pointers, where a*U*∈{a0, a1, ..., a19}, a*V*∈{a0, a1, ..., a19}, a*W*∈{a0, a1, ..., a19}, a*X*∈{a0, a1, ..., a19}, a*Y*∈{a0, a1, ..., a19}, a*Z*∈{a0, a1, ..., a19}
g*X*, g*Y*, g*Z*	General purpose scalar register, where g*X*∈{g0, g1, ..., g11}, g*Y*∈{g0, g1, ..., g11}, g*Z*∈{g0, g1, ..., g11}
g*X*-*Y*	Indicates operation on a subset of scalar registers, beginning with g*X*, g*X*+1, ..., up to and including g*Y*.
ag*X*, ag*Y*, ag*Z*	Notation used to represent data memory pointers and general purpose registers together.
H	Vector element's signs register
R*x*	Vector Register Array (VRA) row vector, where R*x*∈{R0, R1, ..., R7}
r*X*	VRA pointer, where r*X*∈{rS0, rS1, rS2, rV, rSt}
r	Indicates operation on an entire set of VRA pointers.
R[r*X*]	Indicates a VRA element referenced by a pointer.
Rx[r*Y*]	Indicates an element referenced by a pointer within a single row of the array.
R[r*X*][*Y*]	Indicates element *Y* in a row referenced by a pointer.
S*x*	Vector Arithmetic Unit (AU) Source register, where S*x*∈{S0, S1, S2}
V	Vector AU Output register, which is not directly accessible via the instruction set.
IsX	Mnemonic used to represent an *X*-bit signed two's-complement immediate number in an instruction op-code.
IuX	Mnemonic used to represent an *X*-bit unsigned immediate number in an instruction op-code.
I	Mnemonic used to represent signed or unsigned immediate number in an instruction op-code.
llr_mode	Bit Log Likelihood Ratio (LLR) compression mode, where llr_mode∈{llr4, llr4half, llr8, llr8half}
fft_size	fft_size∈{32,64,128,256,512,1024, 2048, 4096, 8192, 16384, 32768, 65536}
S0mode	*S0mode*∈{S0hlinecplx, S0straight, S0cplx1, S0real1, S0zeros, S0abs, S0hword, S0word, S0i1r1i1r1, S0i1i1r1r1, S0group2nr, S0group2nc} with creg(15)=0 *S0mode*∈{S0hlinecplx, S0fft1, S0fft2, S0fft3, S0fft4, S0fft5, S0fftn} with creg(15)=1
S1mode	*S1mode*∈{S1hlinecplx, S1straight, S1cplx1, S1real1, S1real_conj, S1cplx_conj, S1nco, S1qline, S1i2i1r2r1, S1udfr, S1r2c, S1r2c_conj, S1r2c_im0, S1r2c_re0, S1interp2nr, S1interp2nc}
S2mode	*S2mode*∈{S2hlinecplx, S2straight, S2cplx1, S2real1, S2zeros, S2i1r1i1r1, S2i1i2r1r2} with creg(15)=0 *S2mode*∈{S2hlinecplx, S2fft1, S2fft2, S2fft3, S2fft4, S2fft5, S2fftn} with creg(15)=1
Rmode	*Rmode*∈{normal, zeros, l2l, h2h, h2l_l2h, l2h_h2l, h2l, l2h, replace_h, replace_l, qam}

Table continues on the next page...

Table 14. VSPA instruction set mnemonic conventions (continued)

Convention	Meanings
Vmode	*Vmode*∈{hlinecplx, even, fftn, fft7, fn, fn1, straight} with creg(15)=0 *Vmode*∈{hlinecplx, fft1, fft2, fft3, fft4, fft5, fft6} with creg(15)=1
S0prec	*S0prec*∈{half_fixed, half, single}
S1prec	*S1prec*∈{half_fixed, half, single}
S2prec	*S2prec*∈{half_fixed, half, single}
AUprec	*AUprec*∈{single, padd}
Vprec	*Vprec*∈{half_fixed, half, single}

rot_mode	*ror_mode*∈{
	R0R1r1, R0R1r2, R0R1r4, R0R1r8, R0R1rND1, R0R1rND2, R0R1rND4,
	R0r1, R0r2, R0r4, R0r8, R0rND1, R0rND2, R0rND4,
	R1r1, R1r2, R1r4, R1r8, R1rND1, R1rND2, R1rND4,
	R2R3r1, R2R3r2, R2R3r4, R2R3r8, R2R3rND1, R2R3rND2, R2R3rND4,
	R2r1, R2r2, R2r4, R2r8, R2rND1, R2rND2, R2rND4,
	R3r1, R3r2, R3r4, R3r8, R3rND1, R3rND2, R3rND4}
	rol_mode∈{
	R4R5l1, R4R5l2, R4R5l4, R4R5l8
	R4l1, R4l2, R4l4, R4l8
	R5l1, R5l2, R5l4, R5l8
	R6R7l1, R6R7l2, R6R7l4, R6R7l8
	R6l1, R6l2, R6l4, R6l8
	R7l1, R7l2, R7l4, R7l8}

nco_reg	*nco_reg*∈{nco_freq, nco_k, nco_phase}
quot	Quotient
rem	Remainder

Table 15. VCPU instructions summary

OpCode	Dest	Source	Family	Cycles	Description
VSPA instructions quick reference					
Data Memory Pointer instructions Load (from memory) instructions					Scalar Arithmetic instructions Logical Operation instructions

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
Store (to memory) instructions Vector Register Array (VRA) instructions Extrema Search Unit instructions Vector Arithmetic Unit Source Register instructions Vector Floating Point Arithmetic Unit instructions Vector Arithmetic Unit Write-Results instructions Vector Sign Register (H) instructions					Scalar Register-Set (gX) instructions Loop instructions Scalar Compare instructions Control Flow and State instructions Stack instructions Internal Peripheral Control instructions Operand Generator instructions Miscellaneous instructions
Data Memory Pointer instructions					VSPA instructions quick reference
mv	a*U*	Iu19	OpC	1	Move an immediate address to a*U*.
mvB	ag*X*	ag*Y*	OpB	1	Move the contents of ag*Y* to ag*X*.
mvS	ag*X*	ag*Y*	OpS	1	
mvS	sp	a*V*	OpS	1	Move the contents of a*V* to the stack pointer.
	a*U*	sp			Move the stack pointer into a*U*.
mvB	sp	ag*X*	OpB	1	Move the contents of ag*X* to the stack pointer.
	ag*X*	sp			Move the stack pointer into ag*X*.
addA	a*U*	a*V*,a*W*	OpA	1	Add a*V* and a*W* registers and write the result to a*U*.
subA	a*U*	a*V*,a*W*	OpA	1	Subtract a*V* from a*W* and write the result to a*U*.
add	a*V*	a*U*,Is19	OpC	1	Add a 19-bit signed immediate value to a*U* and write the result to a*V*.
	a*U*	sp,Is19			Add a 19-bit signed immediate value to stack pointer and write the result to a*U*.
add(.laddr)	a*U*, Is9		OpA	1	Add a 9-bit signed immediate value to aU and write the result to aU. If .laddr is included, update is by lines.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
set.range	a*Y*	ag*X*,lu19	OpC	1	Defines the beginning and ending address for a circular buffer in DMEM accessed by a*Y*. • Beginning address is stored in ag*X*. • Ending address is calculated, ag*X* + l - 1. **NOTE** The result is undefined when incrementing from an address within the circular buffer such that the beginning or ending address is overshot by a value greater than l.
	a*Y*	ag*X*,g*Y*			Alternative form of the previous instruction where the ending address is calculated = ag*X* + g*Y* - 1. **NOTE** The result is undefined when incrementing from an address within the circular buffer such that the beginning or ending address is overshot by a value greater than l.
set.br	ag*X*,fft_size		OpA	1	Enable the bit-reversed addressing mode for ag*X*, and set the size of the FFT in the address generator.
Load (from memory) instructions					VSPA instructions quick reference
ld	[ag*X*]+/-ag*Y*		OpA	3 [1]	Load a vector from DMEM using a pointer, then post-increment or post-decrement by the contents of ag*Y*.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
ld(.br)	[ag*X*]+/-ag*Y*				Load a word from DMEM using a pointer, then post-increment or post-decrement by the contents of ag*Y*. The address used for the memory access will be modified according to a re-order algorithm (bit reversal).
ldA(.laddr)	[ag*X*]+ls9				Load a vector from DMEM using a pointer, then post-increment or post-decrement by a signed immediate offset. If .laddr is included, update is by lines.
ld	g*Z*	[ag*X*]+/-ag*Y*	OpS	4 [1]	Load a 32-bit scalar from DMEM using a pointer and store the contents into g*Z*. post-increment or post-decrement by the contents of ag*Y*.
ld	g*Z*	[ag*X*]+/-ag*Y*x2			Load a scalar from DMEM using a pointer and store the contents into g*Z*. post-increment or post-decrement by 2 times the contents of ag*Y*.
ldS(.laddr)	g*X*	[ag*Y*]+ls9			Load a 32-bit scalar from DMEM using a pointer and store the contents into g*X*. post-increment by a signed, immediate offset.
ld	g*Z*	[ag*X*+/-ag*Y*]			Generate an address via a pre-increment or pre-decrement of ag*X*, then load a 32-bit scalar from this DMEM address and store the contents into g*Z*. The contents of a*gX* are not modified by this instruction.
ld.u	g*Z*	[ag*X*+/-ag*Y*]			Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ld	g*Z*	[ag*X*+/-ag*Y*x2]			Generate an address via a pre-increment or pre-decrement of ag*X* by 2 times ag*Y*, then load a 32-bit scalar from this DMEM address and store the contents into g*Z*. The contents of a*gX* are not modified by this instruction.
ld.u	g*Z*	[ag*X*+/-ag*Y*x2]			Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ldS(.laddr)	g*X*	[ag*Y*+ls9]			Generate an address via a pre-increment of ag*Y* by a signed, immediate number,

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					then load a 32-bit scalar from this DMEM address and store the contents into g*X*. The contents of ag*Y* are not modified by this instruction.
ldS(.laddr).u	g*X*	[ag*Y*+ls9]			Alternative form of the previous instruction where the address used in the memory access is stored back into the pointer ag*Y*.
ldh(.s)	g*Z*	[ag*X*]+/-ag*Y*			Load a 16-bit scalar from DMEM using a pointer and store the contents into g*Z*. If .s is included, the 16 bit value is sign extended into gZ, otherwise the value is zero extended. post-increment or post-decrement by the contents of ag*Y*.
ldh(.s)	g*Z*	[ag*X*+/-ag*Y*]			Generate an address via a pre-increment or pre-decrement of ag*X*, then load a 16-bit scalar from this DMEM address and store the contents into g*Z*. If .s is included, the 16 bit value is sign extended into gZ, otherwise the value is zero extended. The contents of a*gX* are not modified by this instruction.
ldh.u(.s)	g*Z*	[ag*X*+/-ag*Y*]			Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ldhS(.laddr)(.s)	g*Z*	[ag*X*]+ls9			Load a 16-bit scalar from DMEM using a pointer and store the contents into g*Z*. post-increment by a signed, immediate offset. If .s is included, the 16 bit value is sign extended into gZ, otherwise the value is zero extended.
ldhS(.laddr)(.s)	g*Z*	[ag*X*+ls9]			Generate an address via a pre-increment of ag*X* by a signed, immediate number, then load a 16-bit scalar from this DMEM address and store the contents into g*Z*. If .s is included, the 16 bit value is sign extended into gZ, otherwise the value is zero extended. The contents of ag*X* are not modified by this instruction.
ldhS(.laddr).u(.s)	g*Z*	[ag*X*+ls9]			Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
ld	g*Z*	I19			Load a 32-bit scalar from DMEM using an immediate address and store the contents into g*Z*.
ldh(.s)	g*Z*	I19			Load a 16-bit scalar from DMEM using an immediate address and store the contents into g*Z*. If .s is included, the 16 bit value is sign extended into g*Z*, otherwise the value is zero extended.
ldC	[ag*X*]+Is18		OpC	3	Load a vector from DMEM using a pointer, then post-increment or post-decrement by a signed immediate offset.
ldC	g*Y*	[ag*X*]+Is18	OpC	4	Load a 32-bit scalar from DMEM using a pointer and store the contents into g*Y*. post-increment by a signed, immediate offset.
ldC	g*Y*	[ag*X*+Is18]			Generate an address via a pre-increment of ag*X* by a signed, immediate number, then load a 32-bit scalar from this DMEM address and store the contents into g*Y*. The contents of ag*X* are not modified by this instruction.
ldC.u	g*Y*	[ag*X*+Is18]			Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ldC	ag*Y*	[sp+Is18]			Generate an address via a pre-increment of sp by a signed, immediate number, then load a 32-bit scalar from this DMEM address and store the contents into ag*Y*. The contents of sp are not modified by this instruction.
ldC.u	ag*Y*	[sp+Is18]			Alternative form of the previous instruction, where the address used in the memory access is stored into the stack pointer.
ldhC	g*Y*	[ag*X*]+Is18			Load a 16-bit scalar from DMEM using a pointer and store the contents as a zero extended 32-bit value into g*Y*. post-increment by a signed, immediate offset.
ldhC	g*Y*	[ag*X*+Is18]			Generate an address via a pre-increment of ag*X* by a signed, immediate number, then load a 16-bit scalar from this DMEM address and store the contents as a zero

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					extended 32-bit value into g*Y*. The contents of ag*X* are not modified by this instruction.
ldhC.u	g*Y*	[ag*X*+ls18]			Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ldhC.s	g*Y*	[ag*X*]+ls18			Load a 16-bit scalar from DMEM using a pointer and store the contents as a sign extended 32-bit value into g*Y*. post-increment by a signed, immediate offset.
ldhC.s	g*Y*	[ag*X*+ls18]			Generate an address via a pre-increment of ag*X* by a signed, immediate number, then load a 16-bit scalar from this DMEM address and store the contents as a sign extended 32-bit value into g*Y*. The contents of ag*X* are not modified by this instruction.
ldhC.u.s	g*Y*	[ag*X*+ls18]			Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ldhC	g*Y*	[sp]+ls18			Load a 16-bit scalar from DMEM using the stack pointer and store the contents as a zero extended 32-bit value into g*Y*. post-increment by a signed, immediate offset.
ldhC	g*Y*	[sp+ls18]			Generate an address via a pre-increment of sp by a signed, immediate number, then load a 16-bit scalar from this DMEM address and store the contents as a zero extended 32-bit value into g*Y*. The contents of sp are not modified by this instruction.
ldhC.u	g*Y*	[sp+ls18]			Alternative form of the previous instruction, where the address used in the memory access is stored into the stack pointer.
ldhC.s	g*Y*	[sp]+ls18			Load a 16-bit scalar from DMEM using the stack pointer and store the contents as a sign extended 32-bit value into g*Y*. post-increment by a signed, immediate offset.
ldhC.s	g*Y*	[sp+ls18]			Generate an address via a pre-increment of sp by a signed, immediate number, then

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					load a 16-bit scalar from this DMEM address and store the contents as a sign extended 32-bit value into g*Y*. The contents of sp are not modified by this instruction.
ldhC.u.s	g*Y*	[sp+ls18]			Alternative form of the previous instruction, where the address used in the memory access is stored into the stack pointer.
Store (to memory) instructions					VSPA instructions quick reference
st	[ag*X*]+/-ag*Y*		OpA	1 [2]	Store a vector from R[rSt] to DMEM using a pointer, then post-increment or post-decrement by the contents of ag*Y*.
st.llr_mode	[ag*X*]+/-ag*Y*				Store a vector from R[rSt] to DMEM using a pointer, then post-increment or post-decrement by the contents of ag*Y*. llr mode is used for bit log-likelihood-ratio (LLR) type conversion, read a vector from R[rSt], compress the data by 1/2 or 1/4 by converting the elements to fixed point, then quantize each to 4 or 8 bits.
st.br	[ag*X*]+/-ag*Y*				Store a vector from R[rSt] to DMEM using a pointer, then post-increment or post-decrement by the contents of ag*Y*. In br mode, the address used will be modified according to a re-order algorithm (bit reversal) specified by .br.
st.uline	[ag*X*]				Store from R[rSt] to DMEM using a pointer. The number of elements written from VRA to DMEM must be pre-configured using the ST_UL_VEC_LEN IP register. The pointer will be post incremented by the number of elements stored.
stA(.laddr)	[ag*X*]+ls9				Store a vector from R[rSt] to DMEM using a pointer, then post-increment by a signed, 9-bit immediate offset.
st.w(.br)	[ag*X*]+/-ag*Y*				Store full-word element 0 from R[rSt] to DMEM using a pointer, then post-increment or post-decrement by the contents of ag*Y*. The address used can be optionally modified according to a re-order algorithm (bit reversal) specified by .br.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
stA(.laddr).w	[ag*X*]+Is9				Store full-word element 0 from R[rSt] to DMEM using a pointer, then post-increment by a signed, 9-bit immediate offset.
st	[ag*X*]+/-ag*Y*	g*Z*	OpS	1 [1,2]	Store a 32-bit scalar from g*Z* into DMEM using a pointer. Post-increment or post-decrement by the contents of ag*Y*.
st	[ag*X*]+/-ag*Y*x2	g*Z*			Store a 32-bit scalar from g*Z* into DMEM using a pointer. Post-increment or post-decrement by 2 times the contents of ag*Y*.
st	[ag*X*+/-ag*Y*]	g*Z*			Generate an address via a pre-increment or pre-decrement of ag*X* by ag*Y*, then store a 32-bit scalar from g*Z* to this address in DMEM. The contents of a*gX* are not modified by this instruction.
st.u	[ag*X*+/-ag*Y*]	g*Z*			Alternative form of the previous instruction, where the address used in the memory access is stored back into the pointer a*gX*.
st	[ag*X*+/-ag*Y*x2]	g*Z*			Generate an address via a pre-increment or pre-decrement of ag*X* by 2 times ag*Y*, then store a 32-bit scalar from g*Z* to this address in DMEM. The contents of a*gX* are not modified by this instruction.
st.u	[ag*X*+/-ag*Y*x2]	g*Z*			Alternative form of the previous instruction, where the address used in the memory access is stored back into the pointer a*gX*.
stS(.laddr)	[ag*Y*]+Is9	g*X*			Store a 32-bit scalar from g*X* into DMEM using an pointer. Post-increment by a signed, immediate offset.
stS(.laddr)	[ag*Y*+Is9]	g*X*			Generate an address via a pre-increment or pre-decrement of ag*Y* with a signed, immediate number, then store a scalar from g*X* to this address in DMEM. The contents of ag*Y* are not modified by this instruction.
stS(.laddr).u	[ag*Y*+Is9]	g*X*			Alternative form of the previous instruction where the address used in the memory access is stored back into the pointer ag*Y*.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
sth	[ag*X*]+/-ag*Y*	g*Z*			Store a 16-bit scalar from g*Z* into DMEM using a pointer. Post-increment or post-decrement by the contents of ag*Y*.
sth	[ag*X*+/-ag*Y*]	g*Z*			Generate an address via a pre-increment or pre-decrement of ag*X* by ag*Y*, then store a 16-bit scalar from g*Z* to this address in DMEM. The contents of a*gX* are not modified by this instruction.
sth.u	[ag*X*+/-ag*Y*]	g*Z*			Alternative form of the previous instruction, where the address used in the memory access is stored back into the pointer a*gX*.
sthS(.laddr)	[ag*Y*]+Is9	g*Z*			Store a 16-bit scalar from g*X* into DMEM using a pointer. Post-increment by a signed, immediate offset.
sthS(.laddr)	[ag*Y*+Is9]	g*Z*			Generate an address via a pre-increment or pre-decrement of ag*Y* with a signed, immediate number, then store a scalar from g*Z* to this address in DMEM. The contents of ag*Y* are not modified by this instruction.
sthS(.laddr).u	[ag*Y*+Is9]	g*Z*			Alternative form of the previous instruction where the address used in the memory access is stored back into the pointer ag*Y*.
st	[ag*X*]+Is15	Is16,Is16	OpD	1 [1,2]	Store an immediate complex number into DMEM using ag*X*. Post increment ag*X* by Is15. **NOTE** Format is real, imaginary.
st	I19	g*Z*	OpS	1 [2]	Store a 32-bit scalar from g*Z* to an immediate address in DMEM.
sth	I19	g*Z*			Store a 16-bit scalar from g*Z* to an immediate address in DMEM.
st	Iu19	I32	OpD	1 [2]	Store an immediate 32-bit number to an immediate address in DMEM.
sth	Iu19	I16			Store an immediate 16-bit number to an immediate address in DMEM.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
stC	[ag*X*]+ls18		OpC	1	Store a vector from R[rSt] to DMEM using a pointer, then post-increment by a signed, 18-bit immediate offset.
stC.w	[ag*X*]+ls18				Store full-word element 0 from R[rSt] to DMEM using a pointer, then post-increment by a signed, 18-bit immediate offset.
stC	[ag*X*]+ls18	g*Y*			Store a 32-bit scalar from g*Y* into DMEM using a pointer. Post-increment by a signed, immediate offset.
stC	[ag*X*+ls18]	g*Y*			Generate an address via a pre-increment or pre-decrement of ag*X* with a signed, immediate number, then store a scalar from g*Y* to this address in DMEM. The contents of ag*X* are not modified by this instruction.
stC.u	[ag*X*+ls18]	g*Y*			Alternative form of the previous instruction where the address used in the memory access is stored back into the pointer ag*X*.
stC	[sp+ls18]	ag*Y*			Generate an address via a pre-increment or pre-decrement of the stack pointer with a signed, immediate number, then store a scalar 32-bit from g*Y* to this address in DMEM. The contents of the stack pointer are not modified by this instruction.
stC.u	[sp+ls18]	ag*Y*			Alternative form of the previous instruction where the address used in the memory access is stored back into the stack pointer.
stC	[sp]+ls18	ag*Y*			Store a 32-bit scalar from ag*Y* into DMEM using the stack pointer. Post-increment by a signed, immediate offset.
sthC	[ag*X*]+ls18	g*Y*			Store a 16-bit scalar from g*Y* into DMEM using a pointer. Post-increment by a signed, immediate offset.
sthC	[ag*X*+ls18]	g*Y*			Generate an address via a pre-increment or pre-decrement of ag*X* with a signed, immediate number, then store a 16-bit scalar from g*Y* to this address in DMEM.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					The contents of ag*X* are not modified by this instruction.
sthC.u	[ag*X*+ls18]	g*Y*			Alternative form of the previous instruction where the address used in the memory access is stored back into the pointer ag*X*.
sthC	[sp]+ls18	g*Y*			Store a 16-bit scalar from g*Y* into DMEM using the stack pointer. Post-increment by a signed, immediate offset.
sthC	[sp+ls18]	g*Y*			Generate an address via a pre-increment or pre-decrement of the stack pointer with a signed, immediate number, then store a scalar 16-bit from g*Y* to this address in DMEM. The contents of the stack pointer are not modified by this instruction.
sthC.u	[sp+ls18]	g*Y*			Alternative form of the previous instruction where the address used in the memory access is stored back into the stack pointer.
sth	[ag*X*]+ls9	I16			Store a 16-bit immediate value into DMEM using a pointer. Post-increment by a signed, immediate offset.
st.low	[ag*X*]+ag*Y*	Iu8			Store right justified partial vector from R[rSt] to DMEM using a pointer, then post-increment by the contents of ag*Y*. The index of the leftmost written element is given by Iu8.
st.low	[ag*X*]-ag*Y*	Iu8			Store right justified partial vector from R[rSt] to DMEM using a pointer, then post-decrement by the contents of ag*Y*. The index of the leftmost written element is given by Iu8.
st.low	[ag*X*]+ls16	Iu8			Store right justified partial vector from R[rSt] to DMEM using a pointer. Post-increment by a signed, immediate 16 bit offset. The index of the leftmost written element is given by Iu8.
st.low	[ag*X*]+ag*Y*	g*Z*			Store right justified partial vector from R[rSt] to DMEM using a pointer, then post-increment by the contents of ag*Y*. The

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					index of the left most written element is given in g*Z*.
st.low	[ag*X*]-ag*Y*	g*Z*			Store right justified partial vector from R[rSt] to DMEM using a pointer, then post-decrement by the contents of ag*Y*. The index of the left most written element is given in g*Z*.
st.low	[ag*X*]+ls16	g*Z*			Store right justified partial vector from R[rSt] to DMEM using a pointer. Post-increment by a signed, immediate 16 bit offset.The index of the left most written element is given in g*Z*.
st.high	[ag*X*]+ag*Y*	lu8			Store left justified partial vector from R[rSt] to DMEM using a pointer, then post-increment by the contents of ag*Y*. The index of the leftmost written element is given by lu8.
st.high	[ag*X*]-ag*Y*	lu8			Store left justified partial vector from R[rSt] to DMEM using a pointer, then post-decrement by the contents of ag*Y*.The index of the leftmost written element is given by lu8.
st.high	[ag*X*]+ls16	lu8			Store left justified partial vector from R[rSt] to DMEM using a pointer. Post-increment by a signed, immediate 16 bit offset.The index of the leftmost written element is given by lu8.
st.high	[ag*X*]+ag*Y*	g*Z*			Store left justified partial vector from R[rSt] to DMEM using a pointer, then post-increment by the contents of ag*Y*.The index of the left most written element is given in g*Z*.
st.high	[ag*X*]-ag*Y*	g*Z*			Store left justified partial vector from R[rSt] to DMEM using a pointer, then post-decrement by the contents of ag*Y*.The index of the left most written element is given in g*Z*.
st.high	[ag*X*]+ls16	g*Z*			Store left justified partial vector from R[rSt] to DMEM using a pointer. Post-increment by a signed, immediate 16 bit offset.The

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					index of the left most written element is given in gZ.
Vector Register Array (VRA) instructions					VSPA instructions quick reference
ld.normal	R*x*		OpVr	1	Load a vector from memRead bus into Rx.
ld.h2h	R*x*				Load high part of memRead bus into high part of Rx.
ld.h2l	R*x*				Load high part of memRead bus into low part of Rx.
ld.h2l_l2h	R*x*				Load high part of memRead bus into low part of Rx and load low part of memRead bus into high part of Rx+1. Rx can be R0, R2, R4 or R6.
ld.l2l	R*x*				Load low part of memRead bus into low part of Rx.
ld.l2h	R*x*				Load low part of memRead bus into high part of Rx.
ld.l2h_h2l	R*x*				Load low part of memRead bus into high part of Rx and load high part of memRead bus into low part of Rx+1. Rx can be R0, R2, R4 or R6.
ld.replace_h	R*x*				Replace the most significant word in Rx with a word on the memRead bus.
ld.replace_l	R*x*				Replace the least significant word in Rx with a word on the memRead bus.
ld.qam	R*x*				For modulation order M, take a M/32 line of memRead bus (32M bits), transform it into a full line, and write it into Rx. The transformation implements QAM modulation.
ld.2scomp	R*x*				Load partial line from memRead bus into a full line with type conversion of 2's complement value to half fixed value, write full line data into Rx.
mv.h	[rV]	Is16	OpD	1	Move a short immediate scalar into R[rV]. Post-increment the rV pointer by a signed integer value contained in incr_rV.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
mv.w	[rV]	Is16,Is16			Move a half-precision complex number into R[rV]. Post-increment the rV pointer by a signed integer value contained in incr_rV. **NOTE** Format is real, imaginary.
mvA.VRAptr	r*X*	ag*Y*	OpA	1	Move into a VRA pointer register from a scalar register
mvB.VRAptr	r*X*	ag*Y*	OpB		Move into a VRA pointer register from a scalar register
mvA.VRAptr	ag*Y*	r*X*	OpA		Move into a scalar register from a VRA pointer register
mvB.VRAptr	ag*Y*	r*X*	OpB		Move into a scalar register from a VRA pointer register
mvA.VRAincr	r*X*	ag*Y*	OpA		Move into a VRA increment register from a scalar register
mvB.VRAincr	r*X*	ag*Y*	OpB		Move into a VRA increment register from a scalar register
mvA.VRAincr	ag*Y*	r*X*	OpA		Move into a scalar register from a VRA increment register
mvB.VRAincr	ag*Y*	r*X*	OpB		Move into a scalar register from a VRA increment register
mvA.VRArange1	r*X*	ag*Y*	OpA		Move into a VRA range register from a scalar register
mvA.VRArange2	r*X*	ag*Y*	OpA		Move into a VRA range register from a scalar register
mvB.VRArange1	r*X*	ag*Y*	OpB		Move into a VRA range register from a scalar register
mvB.VRArange2	r*X*	ag*Y*	OpB		Move into a VRA range register from a scalar register
mvA.VRArange1	ag*Y*	r*X*	OpA		Move into a scalar register from a VRA range register

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
mvA.VRArange2	ag*Y*	r*X*	OpA		Move into a scalar register from a VRA range register
mvB.VRArange1	ag*Y*	r*X*	OpB		Move into a scalar register from a VRA range register
mvB.VRArange2	ag*Y*	r*X*	OpB		Move into a scalar register from a VRA range register
setA.VRAptr	r*X*	Iu9	OpA	1	Initialize a Vector Register Array pointer.
setB.VRAptr	r*X*	Iu9	OpB	1	Initialize a Vector Register Array pointer.
setA.VRAincr	r*X*	Is9	OpA	1	Initialize the post-increment register for r*X*.
setB.VRAincr	r*X*	Is9	OpB	1	Initialize the post-increment register for r*X*.
set.VRAptr	Iu9,Iu9,Iu9,Iu9,Iu3		OpD	1	Initialize all 5 VRA pointers in a single operation. Order of arguments: rS0, rS1, rS2, rV, rSt.
set.VRAincr	Is9,Is9,Is9,Is9,Is3				Initialize the post-increments for all 5 VRA pointers in a single operation. Order of arguments: rS0, rS1, rS2, rV, rSt.
set.VRArange1	r*X*	Iu9, Iu9	OpC	1	Initialize the range registers (start and wrap addresses) of the 1st buffer for r*X*. **NOTE** Format is start, wrap.
set.VRArange2	r*X*	Iu9, Iu9			Initialize the range registers (start and wrap addresses) of the 2nd buffer for r*X*. **NOTE** Format is start, wrap.
clr.VRA			OpB	1	Clear all the VRA pointers, increments, and ranges in a single operation.
clr	R*x*		OpVr	1	Clear all the elements of a VRA row vector in a single operation.
clr.VRA		g*X*,g*Y*	OpS	2	Clear a range of elements in the VRA bounded by g*X*-g*Y* to g*X*+g*Y*, inclusive of the endpoints.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
fill.w	[rV]	g*X*	OpB	1	Fill an entire vector with the full-word in g*X*.
mv.w	[rV]	g*X*			Move the contents of g*X* to a full-word element in R[rV]. Post-increment the rV pointer by a signed integer value ranging from contained in incr_rV.
fill.h	[rV]	g*X*			Fill an entire vector with the least significant short scalar in g*X*.
mv.h	[rV]	g*X*			Move the least significant short scalar in g*X* to a half-word element in R[rV]. Post-increment the rV pointer by a signed integer value contained in incr_rV.
fill.q	[rV]	g*X*:gX+1:gX+2:gX+3			Fill the entire vector with the contents of gX through gX+3.
mv.q	[rV]	g*X*:gX+1:gX+2:gX+3			Move the contents of gX through gX+3 into 4 full words in Rx.
fill.d	[rV]	g*X*:gX+1			Fill the entire vector with the contents of gX through gX+1.
mv.d	[rV]	g*X*:gX+1			Move the contents of gX through gX+1 into 2 full words in Rx
mv.w	g*X*	[rS0]		2	Move a full-word element in the register array to g*X*.
mv.h	g*X*	[rS0]			Move a half-word element in the register array to g*X*.
set.rot		(lt_mode), (rt_mode)		0	Configure the vector rotation unit. **NOTE** lt_mode and rt_mode are independent.
mv	R*x*	R*y*		2	Move the full vector Ry into Rx.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					NOTE The first cycle of this instruction moves Ry onto the memRead bus. Thus, it should be scheduled so as not to conflict with any memory read instructions. Memory read instructions populate the memRead bus in the 2nd cycle of their execution. This mv instruction also shares the same multiplexer that selects DMEM data to be written from Rz to DMEM. So, this instruction should not be used in the same cycle as an st instruction. This instruction overwrites data previously read from RAM using a ld instruction, and cannot occur in parallel with a st instruction.
ldB	R*x*			1	Load Rx from the memory buffer.
Extrema Search Unit instructions					VSPA instructions quick reference
set.xtrm	{signed, unsigned}, {max, min}, {even, all}, {value, index}, N		OpB	1	Configure the extrema search engine. This includes the number of half-word elements to search, N, which can be any power of 2 up to one half-line (32) or a multiple of half lines upto 64x32. The search engine uses rS2[msb:msb-5] to read vector data from the VRA. In 'all' mode, N must be greater

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					than or equal to 2 and in 'even' mode, N must be greater than or equal to 4.
xtrm	a*U*,g*X*			2+ceil(M/B)+log2[min(M,B)]	Start an extrema search and place the index of the extrema into either or both a memory pointer and scalar register. This is a multi-cycle instruction that depends on the number of elements in the search buffer and 'all' or 'even' mode. In 'all' mode, M=N and in 'even' mode, M=N>>1. B = 32 for elements in half precision and B = 16 for single precision.
	g*X*				
	a*U*				
Vector Arithmetic Unit Source Register instructions					VSPA instructions quick reference
rd	S0		OpVs*x*	2	Read data from the VRA and load an AU source register.
	S1				
	S2				
set.Smode	*S0mode*		OpB	0	Set S0 mode.
	S1mode				Set S1 mode.
	S2mode				Set S2 mode.
	S0mode,*S1mode*				Set S0 and S1 modes.
	S0mode, *S2mode*				Set S0 and S2 modes.
	S1mode, *S2mode*				Set S1 and S2 modes.
	S0mode, *S1mode*, *S2mode*				Set S0, S1 and S2 modes.
	S0chs, *S0mode*				Negate the output of the S0Mux prior to loading S0 register while setting S0 mode.
	S0chs, *S0mode*, *S1mode*				Negate the output of the S0Mux prior to loading S0 register while setting S0 and S1 modes.
	S0chs, *S0mode*, *S2mode*				Negate the output of the S0Mux prior to loading S0 register while setting S0 and S2 modes.
	S0chs, *S0mode*, *S1mode*, *S2mode*				Negate the output of the S0Mux prior to loading S0 register while setting S0, S1 and S2 modes.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
	S0conj, *S0mode*				Conjugate the output of the S0Mux prior to loading S0 register while setting S0 mode.
	S0conj, *S0mode*, *S1mode*				Conjugate the output of the S0Mux prior to loading S0 register while setting S0 and S1 modes.
	S0conj, *S0mode*, *S2mode*				Conjugate the output of the S0Mux prior to loading S0 register while setting S0 and S2 modes.
	S0conj, *S0mode*, *S1mode*, *S2mode*				Conjugate the output of the S0Mux prior to loading S0 register while setting S0, S1 and S2 modes.
	S0conj, S0chs, *S0mode*				Conjugate and negate the output of the S0Mux prior to loading S0 register while setting S0 mode.
	S0conj, S0chs, *S0mode*, *S1mode*				Conjugate and negate the output of the S0Mux prior to loading S0 register while setting S0 and S1 modes.
	S0conj, S0chs, *S0mode*, *S2mode*				Conjugate and negate the output of the S0Mux prior to loading S0 register while setting S0 and S2 modes.
	S0conj, S0chs, *S0mode*, *S1mode*, *S2mode*				Conjugate and negate the output of the S0Mux prior to loading S0 register while setting S0, S1 and S2 modes.
set.prec	*S0prec* *S1prec* *S2prec* *AUprec* *Vprec*		OpB	1 1 1 3 7	Set the precision state for the vector data path. This includes S0, S1, and S2 register writes, AU operations, and AU write-results operations.
Vector Floating Point Arithmetic Unit instructions					VSPA instructions quick reference
rmad			OpVau	4(sp) 5(DP)	Multiply and add.
rmad.sau					Multiply and add, using SAU result as S1 operand.
rmac					Multiply and accumulate.
rmac.sau					Multiply and accumulate, using SAU result as S1 operand.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
cmad					Complex multiply and add.
cmad.sau					Complex multiply and add, using SAU result as S1 operand.
cmac					Complex multiply and accumulate.
cmac.sau					Complex multiply and accumulate, using SAU result as S1 operand.
mads					Multiply and add with sign conversion.
mads.sau					Multiply and add with sign conversion.
mafac					Multiply feedback and accumulate.
maf					Multiply and add then feedback result to multiplier.
dif.sau					Decimation-in-frequency radix-2 butterfly,using SAU result as S1 operand
clr.au					Clear vector accumulators.
rcp			OpVsau	2	Vector reciprocal function.
rrt					Vector reciprocal square root function.
srt					Vector square root function.
nco				2	Iterate Numerically controlled oscillator.
padd				1	Pre-add S1 & S2.
rol			OpVrot	1	Perform a left shift or rotate vector operation.
ror					Perform a right shift or rotate vector operation.
Vector Arithmetic Unit Write-Results instruction					VSPA instructions quick reference
wr.even			OpVd	1	Write AU output into the VRA. Every other output shifting is performed at WbMux.
wr.fftn					Write AU output into the VRA. Perform shift function at WbMux for FFT nth stage.
wr.fft1					Write AU output into the VRA. Perform shift function at WbMux for FFT 1st stage.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
wr.fft2					Write AU output into the VRA. Perform shift function at WbMux for FFT 2nd stage.
wr.fft3					Write AU output into the VRA. Perform shift function at WbMux for FFT 3rd stage.
wr.fft4					Write AU output into the VRA. Perform shift function at WbMux for FFT 4th stage.
wr.fft5					Write AU output into the VRA. Perform shift function at WbMux for FFT 5th stage.
wr.fft6					Write AU output into the VRA. Perform shift function at WbMux for FFT 6th stage.
wr.fft7					Write AU output into the VRA. Perform shift function at WbMux for FFT 7th stage.
wr.fn					Write SAU output into the VRA. No special shifting function is performed at WbMux.
wr.fn1					Write single SAU output into the VRA. Output shifting is performed at WbMux.
wr.straight					Write AU output into the VRA. No special shifting function is performed at WbMux. (Real mode)
wr.hlinecplx					Write AU output into the VRA. No special shifting function is performed at WbMux. (Complex mode)
Vector Sign Register (H) instructions					VSPA instructions quick reference
ff0to1	g*Z*	g*X*	OpS	2	find-first-zero-to-one: Move into g*Z* the bit position of the first 0-to-1 transition (<trans_loc>) in the H register, then skip ahead g*X* bit positions (<skip>) and copy bit from H[<trans_loc>+<skip>-1] into all bits in H up to this point.
ff1to0	g*Z*	g*X*			find-first-one-to-zero: Move into g*Z* the bit position of the first 1-to-0 transition (<trans_loc>) in the H register, then skip ahead g*X* bit positions (<skip>) and copy bit from H[<trans_loc>+<skip>-1] into all bits in H up to this point.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
fa0to1		Iu2	OpB	1	find-all-zero-to-ones: Returns single bit for each adjacent pair of 0-1 bits in a quadrant of H register. Quadrant is specified with immediate constant. Result stored in dH register which can be accessed with bin2num instruction.
add(.cc)	g*Z*	g*X*,H	OpS	1	If the optional condition test is true, then sum the 64 least significant bits in the H register then add to the contents of g*X* and store the result in g*Z*. The H-register creg must be configured for 'no auto-increment'. Note, the sign capture operation clears the unused bits of the H register.
	g*Z*	H,g*Y*			Alternate form of the above instruction.
	g*Z*	H,H			Sum the 64 least significant bits in the H register then add to itself and store the result in g*Z*. The H-register creg must be configured for 'no auto-increment'. Note, the sign capture operation clears the unused bits of the H register.
bin2num	R*x*		OpVr	1	Converts each of 256 bits in dH to a numeric value of 0 or 0.5 (HP-fixed) and writes result into Rx.
and	H		OpS	1	Logical AND of H register with mask generated by the hardware and hidden from the programmer.
or	H				Logical OR of H register with mask generated by the hardware and hidden from the programmer.
ld	H	[ag*X*]+/-ag*Y*	OpS	4	Load a sign vector from DMEM using a pointer and store the contents into H. Post-increment or decrement by the contents of ag*Y*.
	H	[ag*Y*]+ls9			Load a sign vector from DMEM using a pointer and store the contents into H. Post-increment by a signed, short immediate offset.
ld	H	[ag*X*+/-ag*Y*]			Generate an address via a pre-increment or pre-decrement of ag*X* then load a sign vector from this DMEM address and store

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					the contents into H. The contents of ag*X* are not modified by this instruction.
ld.u	H	[ag*X*+/-ag*Y*]			Alternative form of the previous instruction where the address used in the memory access is stored into the pointer ag*X*.
ld	H	[ag*Y*+ls9]			Generate an address via a pre-increment of ag*Y* by a signed, immediate number then load a sign vector from this DMEM address, and store the contents into H. The contents of ag*Y* are not modified by this instruction.
ld.u	H	[ag*Y*+ls9]			Alternative form of the previous instruction where the address used in the memory access is stored back into the pointer ag*Y*.
ld	H	I19		4	Load a sign vector from DMEM using an immediate address and store the contents into H. **NOTE** lsb of I19 is ignored if legacy_mem_addr=0.
st	[ag*X*]+/-ag*Y*	H	OpS	1	Store a sign vector from H into DMEM using a pointer. Post-increment or post-decrement by the contents of ag*Y*.
	[ag*Y*]+ls9	H			Store a sign vector from H into DMEM using a pointer. Post-increment by a signed, immediate offset.
st	[ag*X*+/-ag*Y*]	H		1	Generate an address via a pre-increment or decrement of ag*X* by ag*Y* then store a sign vector from H to this address in DMEM. The contents of ag*X* are not modified by this instruction.
st.u	[ag*X*+/-ag*Y*]	H			Alternative form of the previous instruction where the address used in the memory access is stored back into the pointer ag*X*.
st	[ag*Y*+ls9]	H			Generate an address via a pre-increment or decrement of ag*Y* with a signed,

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					immediate number then store a sign vector from H to this address in DMEM. The contents of ag*Y* are not modified by this instruction.
st.u	[ag*Y*+Is9]	H			Alternative form of the previous instruction where the address used in the memory access is stored back into the pointer ag*Y*.
st	I19	H		1	Store a sign vector from H to an immediate address in DMEM. **NOTE** lsb of I19 is ignored if legacy_mem_addr=0.
Scalar Arithmetic instructions					VSPA instructions quick reference
addS(.ucc)(.cc)	g*Z*	g*X*,g*Y*	OpS	1	If the optional condition test is 'true', then add g*X* to g*Y* and store the result in g*Z*. **NOTE** Use of sp is allowed. sp can replace any one or more of the g*X*,g*Y*,or g*Z* operands. If the optional field (.ucc) is used, then the instruction will update the condition codes based on the result of the operation. See Table 72.
addS(.ucc).z	(g*Z*)	g*Z*,Iu16			Add an unsigned short immediate scalar to g*Z* and store the result in g*Z*. If the optional field (.ucc) is used, then the instruction will update the condition codes based on the result of the operation. See Table 72. **NOTE** Use of sp is allowed. sp can replace g*Z*.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
addS(.ucc).s	(g*Z*)	g*Z*,ls16			Sign extend, then add a short immediate scalar to g*Z* and store the result in g*Z*. If the optional field (.ucc) is used, then the instruction will update the condition codes based on the result of the operation. See Table 72. **NOTE** Use of sp is allowed. sp can replace g*Z*.
addD(.ucc)(.cc)	g*X*	g*Y*,l32	OpD	1	If the optional condition test is true, then add an immediate scalar to g*Y* and store the result in g*X*. If the optional field (.ucc) is used, then the instruction will update the condition codes based on the result of the operation. See Table 72. **NOTE** Use of sp is allowed.
subS(.ucc)(.cc)	g*Z*	g*X*,g*Y*	OpS	1	If the optional condition test is 'true', then subtract g*Y* from g*X* and store the result in g*Z*. If the optional field (.ucc) is used, then the instruction will update the condition codes based on the result of the operation. See Table 72. **NOTE** Use of sp is allowed. sp can replace any one or more of the g*X*,g*Y*,or g*Z* operands.
subS(.ucc).z	(g*Z*)	g*Z*,lu16			Subtract an unsigned short immediate scalar from g*Z* and store the result in g*Z*. If the optional field (.ucc) is used, then the instruction will update the condition codes based on the result of the operation. See Table 72.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					NOTE Use of sp is allowed. sp can replace g*Z*.
subS(.ucc).s	(g*Z*)	g*Z*,ls16			Sign extend, then subtract a short immediate scalar from g*Z* and store the result in g*Z*. If the optional field (.ucc) is used, then the instruction will update the condition codes based on the result of the operation. See Table 72. **NOTE** Use of sp is allowed. sp can replace g*Z*.
subD(.ucc)(.cc)	g*X*	g*Y*,l32	OpD	1	If the optional condition test is true, then subtract an immediate scalar from g*Y* and store the result in g*X*. If the optional field (.ucc) is used, then the instruction will update the condition codes based on the result of the operation. See Table 72. **NOTE** Use of sp is allowed.
rsub.z	(g*Z*)	g*Z*,lu16	OpS	1	Subtract g*Z* from an unsigned short immediate number and store the result in g*Z*. **NOTE** Use of sp is allowed.
rsub.s	(g*Z*)	g*Z*,ls16			Sign extend a short immediate scalar, then subtract g*Z* from it and store the result in g*Z*.
mpy(.cc)(.s)	g*Z*	g*X*,g*Y*	OpS	4	If the optional condition test is 'true', then multiply g*X* and g*Y* and store the signed result in g*Z*.
mpyS.z	(g*Z*)	g*Z*,lu16			Multiply an unsigned short immediate scalar with g*Z* and store the result in g*Z*.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
mpyS.s	(g*Z*)	g*Z*,ls16			Sign extend a short immediate scalar, then multiply it with g*Z* and store the result in g*Z*.
mpyD(.cc)	g*X*	g*Y*,l32	OpD	4	If the optional condition test is true, then multiply an immediate scalar with g*Y* and store the result in g*X*.
div(.cc)(.s)	g*Z*	g*X*,g*Y*	OpS	7	If the optional condition test is 'true', then g*Z* = floor (g*X*/g*Y*). The result g*Z* is signed.
div.z	(g*Z*)	g*Z*,lu16			Divide g*Z* by an unsigned short immediate scalar and store the unsigned quotient in g*Z*.
div.s	(g*Z*)	g*Z*,ls16			Sign extend an immediate scalar, then divide g*Z* by it and store the signed quotient in g*Z*.
rdiv.z	(g*Z*)	g*Z*,lu16			Divide an unsigned short immediate scalar by g*Z* and store the unsigned quotient in g*Z*.
rdiv.s	(g*Z*)	g*Z*,ls16			Sign extend a short immediate scalar, then divide it by g*Z* and store the signed quotient in g*Z*.
mod(.cc)(.s)	g*Z*	g*X*,g*Y*	OpS	7	If the optional condition test is 'true', then g*Z* = rem(g*X*/g*Y*). The result g*Z* is signed.
mod.z	(g*Z*)	g*Z*,lu16			Divide g*Z* by an unsigned short immediate scalar and store the unsigned remainder in g*Z*.
mod.s	(g*Z*)	g*Z*,ls16			Sign extend a short immediate scalar, then divide g*Z* by it and store the signed remainder in g*Z*.
rmod.z	(g*Z*)	g*Z*,lu16			Divide an unsigned short immediate scalar by g*Z,* and store the unsigned result in g*Z*.
rmod.s	(g*Z*)	g*Z*,ls16			Sign extend a short immediate scalar, then divide it by g*Z* and store the signed result in g*Z*.
abs	g*Z*	g*X*	OpS	1	Take the absolute value of g*X* and store the result in g*Z*.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
log	g*Z*	g*X*	OpS	2	Take the base-2 logarithm of a single precision floating point number in g*X* and store the result in g*Z*.
ff1	g*Z*	g*X*	OpS	1	find-first-one: Move into g*Z* the bit number of the most significant bit in g*X* that is set.
fns	g*Z*	g*X*		1	Move into g*Z* the number of shifts required to normalize the unsigned contents of g*X*.
fix2float	g*X*	g*Y*	OpB	1	Convert a 16-bit 2's complement integer in g*Y* to a 32-bit floating-point scalar in g*X*.
float2fix	g*X*	g*Y*			Convert a 32-bit single precision scalar in g*Y* to a 16-bit 2's complement integer in g*X*.
hfixtofloatsp	g*X*	g*Y*			Convert a short scalar located in the least significant portion of g*Y* from 16 bit sign magnitude fixed-point to single precision floating-point.
floatsptohfix	g*X*	g*Y*			Convert a scalar located in g*Y* from single precision floating-point to 16 bit sign magnitude fixed-point.
floathptofloatsp	g*X*	g*Y*			Convert a short scalar located in the least significant portion of g*Y* from 16 bit floating-point to single precision floating-point.
floatsptofloathp	g*X*	g*Y*			Convert a scalar located in g*Y* from single precision floating-point to 16 bit floating-point.
floatx2n	(g*X*)	g*X*,g*Y*	OpB	1	Scale a floating point number in gX by 2^n, where the integer exponent n is stored in gY.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					NOTE The value in gY is treated as an 8-bit signed integer. For example, 0xFFFF is interpreted as -1 and 0xFFFE as -2 (NOT positive values 65535, 65534); likewise, 0xFFFFFF00 and 0xFFFFFF01 will be interpreted as 0 and 1, respectively (NOT -256 and -255).
floatx2n	g*X*	g*Y*,I32	OpD	1	Scale a floating point number in gY by 2^n, using I32 as the exponent and store the result in g*X*.
sr.s	g*X*	g*Y*,I5	OpS	1	Shift-right the signed contents of g*Y* by an immediate number of bit positions and store the result in g*X*.
Logical Operation instructions					VSPA instructions quick reference
btst	g*Y*,I5		OpS	1	Test a bit in g*Y* and set the condition code.
bset(.cc)	g*Z*	g*X*,Iu5	OpS	1	If the optional condition test is true, set bit number Iu5, or with g*X* put result in g*Z*. g*X* is not changed.
bclr(.cc)	g*Z*	g*X*,g*Y*	OpS	1	If the optional condition test is true, then clear the bits in g*X* using the mask in g*Y* and store the result in g*Z*.
bclr(.cc)	g*Z*	g*X*,Iu5	OpS	1	If the optional condition test is true, clear bit number Iu5 from gX and write result to gZ. gX is not changed.
and(.cc)	g*Z*	g*X*,g*Y*	OpS	1	If the optional condition test is true, then perform the logical AND of g*X* with g*Y* and store the result in g*Z*. **NOTE** Use of sp is allowed.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
andS(.z)	(g*X*)	g*X*,I16			Logical AND of short immediate with g*X* and store the result in g*X*. The most significant 16 bits of g*X* are not modified. **NOTE** Use of sp is allowed.
andD(.cc)	g*X*	g*Y*,I32	OpD	1	If the optional condition test is true, then perform logical AND of long immediate with g*Y*and store the result in g*X*. **NOTE** Use of sp is allowed.
or(.cc)	g*Z*	g*X*,g*Y*	OpS	1	If the optional condition test is true, then perform the logical OR of g*X* with g*Y* and store the result in g*Z*.
orS	(g*X*)	g*X*,I16			Logical OR of signed immediate with g*X* and store the result in g*X*. The most significant 16 bits of g*X* are not modified.
orD(.cc)	g*X*	g*Y*,I32	OpD	1	If the optional condition test is true, then perform logical OR of long immediate with g*Y* and store the result in g*X*.
not(.cc)	g*Z*	g*X*	OpS	1	If the optional condition test is true, then perform the logical NOT of g*X* and store the result in g*Z*.
xor(.cc)	g*Z*	g*X*,g*Y*	OpS	1	If the optional condition test is true, then perform the logical XOR of g*X* with g*Y* and store the result in g*Z*.
xorS	g*X*	g*X*,I16			Logical XOR of signed immediate with g*X* and store the result in g*X*. The most significant 16 bits of g*X* are not modified.
xorD(.cc)	g*X*	g*Y*,I32	OpD	1	If the optional condition test is true, then perform logical XOR of long immediate with g*Y* and store the result in g*X*.
sr(.cc)	g*Z*	g*X*,g*Y*	OpS	1	If the optional condition test is true, then shift-right the unsigned contents of g*X* by the number of bit positions indicated in g*Y* and store the result in g*Z*.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
sr(.cc).s	g*Z*	g*X*,g*Y*			If the optional condition test is true, then shift-right the signed contents of g*X* by the number of bit positions indicated in g*Y* and store the result in g*Z*.
sr	g*X*	g*Y*,I5			Shift-right the unsigned contents of g*Y* by an immediate number of bit positions and store the result in g*X*.
sl(.cc)	g*Z*	g*X*,g*Y*	OpS	1	If the optional condition test is true, then shift-left the contents of g*X* by the number of bit positions indicated in g*Y* and store the result in g*Z*.
sl	g*X*	g*Y*,I5			Shift-left the contents of g*Y* by an immediate number of bit positions and store the result in g*X*.
lfsr	(g*X*)	g*X*,g*Y*	OpB	1	Linear feedback shift register operation. Shift-left the contents of gX by one bit position, then replace its least significant bit with the bit-wise XOR of gY.
lfsr	g*X*	g*Y*,I32	OpD	1	Linear feedback shift register operation. Shift-left the contents of g*Y* by one bit position, then replace its least significant bit with the bit-wise XOR of I32, store the result in g*X*.
lsb2rf	[rV]	g*X*	OpB	1	Convert the least significant bit of g*X* to half-fixed representation (0.0 or 0.5) and store into R[rV] . Post-increment the rV pointer by a signed integer value contained in incr_rV.
lsb2rf.sr	[rV]	g*X*			Convert the least significant bit of g*X* to half-fixed representation (0.0 or 0.5) and store into R[rV], then perform a logical shift-right operation on g*X*.
Scalar Register-Set (gX) instructions					VSPA instructions quick reference
mv(.cc)	g*Z*	g*X*	OpS	1	If the optional condition test is true, then move g*X* to g*Z*.
mvS.z	g*Z*	Iu16	OpS	1	Move an unsigned immediate short scalar into g*Z*.
mvS.s	g*Z*	Is16			Sign extend an immediate short scalar then move it into g*Z*.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
mvD(.cc)	g*X*	I32	OpD	1	If the optional condition test is true, then move an immediate scalar into g*X*.
mvh.s(.cc)	g*Z*	g*X*	OpS	1	If the optional condition test is true, then sign extend the least significant 16 bits of g*X* and move to g*Z*.
mv.w	[rV]	g*X*	OpB	1	Move the contents of g*X* to a full-word element in R[rV]. Post-increment the rV pointer by a signed integer value contained in incr_rV.
mv.h	[rV]	g*X*			Move the least significant short scalar in g*X* to a half-word element in R[rV]. Post-increment the rV pointer by a signed integer value contained in incr_rV.
mv.w	g*X*	[rS0]	OpB	2	Move a full-word element in the register array to g*X*.
mv.h	g*X*	[rS0]			Move a half-word element in the register array to g*X*.
clr.g		I12	OpS	1	Clear a set of g*X* registers indicated by an immediate mask.
mv(.cc)	g*Z*	pc	OpS	1	If the optional condition test is true, then move current pc to g*Z*.
mv(.cc)	g*Z*	quot			If the optional condition test is true, then move the quotient result of the previous modulo operation to g*Z*.
mv(.cc)	g*Z*	rem			If the optional condition test is true, then move the remainder result of the previous divide operation to g*Z*.
Loop instructions					VSPA instructions quick reference
set.loop	Iu10, Iu16		OpC	2	Set the number of loop iterations and the size (number of instructions) in a loop with immediate numbers. • 1st argument is loop iteration count. • 2nd argument is size.
setC.loop	Iu16				Set the number of iterations using an immediate number.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
set.loop	ag*X*, Iu10				Set the size of a loop with an immediate scalar, and set the iteration count with the contents of ag*X*.
set.loop	Iu10		OpB	2	Set the number of iterations using an immediate number.
set.loop	ag*X*				Set the number of iterations using the contents of ag*X*. If ag*X* = 0, the number of iterations will be 65536.
set.loop	ag*X*,Is5			1	Allows an OpB set loop instruction with a 5 bit signed instruction count
loop_begin			OpZ	1	Begin loop execution.
loop_stop			OpB	1	Early termination of a loop.
loop_end			OpZ	1	End loop execution. **NOTE** Assembler directive to indicate end of the loop.
Scalar Compare instructions					VSPA instructions quick reference
cmp	a*U*,Iu19		OpC	2	Compare a data memory pointer with an immediate number. **NOTE** The second cycle in 'cmp' is used to set up a conditional instruction.
cmpD(.cc)	g*Y*,I32		OpD	2	If the optional condition test is true, then compare g*Y* with an immediate scalar.
cmpS.z	g*Z*,Iu16		OpS	2	Compare g*Z* with a short unsigned immediate scalar.
cmpS.s	g*X*,Is16				Sign extend an immediate short scalar, then compare it with g*X*.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
cmp(.cc)	g*X*,g*Y*				If the optional condition test is true, then compare 2 scalar registers. Similar to subtract but without a destination register. **NOTE** Use of sp is allowed.
Control Flow and State instructions					VSPA instructions quick reference
jmp(.cc)	g*X*		OpC	3	If the optional condition test is true, then jump to the address in g*X*.
jsr(.cc)	g*X*				If the optional condition test is true, then jsr to the address in g*X*.
jmp(.cc)	Iu25				If the optional condition test is true, then jump to an immediate address.
jsr(.cc)	Iu25				If the optional condition test is true, then jsr to an immediate address.
jmp(.cc)	[pc+Is25]				If the optional condition test is true, then jump to a PC relative (PC+Is25) address.
jsr(.cc)	[pc+Is25]				If the optional condition test is true, then jsr to a PC relative (PC+Is25) address.
jmp(.cc)	[pc+g*X*]				If the optional condition test is true, then jump to a PC relative (PC+g*X*) address.
jsr(.cc)	[pc+g*X*]				If the optional condition test is true, then jsr to a PC relative (PC+g*X*) address.
loop_break(.cc)	Iu25		OpC	3	If the optional condition test is true, then stop loop execution and jump to an immediate address. **NOTE** This must be in parallel with OpZ break, which is added by the assembler.
swi(.cc)	Iu16		OpC	3	If the optional condition test is true, then jsr to software interrupt handler and enter supervisor mode.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
					NOTE This must be in parallel with OpZ break, which is added by the assembler.
rts			OpZ	3	Return from subroutine.
mv	cc	Iu4	OpS	1	Move an immediate number into the condition codes.
set.creg	creg	Iu4	OpS	1	Initialize the Control register (creg) with an immediate scalar. Refer Control register latency to understand exceptions to one cycle latency.
setB.creg	creg	Iu4	OpB	1	Alternate (OpB) form of the above instruction.
swbreak			OpX	0	Software breakpoint.
done			OpD	1	Enter the low power done state.
Stack instructions					VSPA instructions quick reference
mv	sp	I19	OpS	1 [3]	Initialize the stack pointer.
mvB	sp	ag*X*	OpB	1	Move contents of ag*X* to the stack pointer.
	ag*X*	sp			Move the stack pointer to ag*X*.
mvS	sp	a*V*	OpS	1 [3]	Move contents of a*V* to the stack pointer.
	a*U*	sp			Move the stack pointer to a*U*.
mv(.cc)	sp	g*X*	OpS	1 [3]	If the optional condition test is true, then move contents of g*X* to the stack pointer.
	g*Z*	sp			If the optional condition test is true, then move the stack pointer to g*Z*.
stS	[sp+Is10]	g*X*	OpS	1 [2,3]	Push g*X* onto the stack at an immediate offset from the current stack pointer. **NOTE** The stack pointer does not change.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
stS	[sp]+ls10	g*X*			Push g*X* onto the stack, then post-increment the stack pointer by an immediate number.
st	[sp+ls10]	H		2[2,3]	Push H onto the stack at an immediate offset from the current stack pointer. **NOTE** The stack pointer does not change.
st	[sp]+ls10	H			Push H onto the stack, then post-increment the stack pointer by an immediate number.
ldS	g*X*	[sp+ls10]	OpS	4 [3]	Pop g*X* using an immediate offset from the current stack pointer. **NOTE** The stack pointer is not updated with the new address.
ldS.u	g*X*	[sp+ls10]			Pop g*X* using an immediate offset from the current stack pointer. The stack pointer is updated with the new address.
ld	H	[sp+ls10]			Pop H using an immediate offset from the current stack pointer. **NOTE** The stack pointer is not updated with the new address.
ld.u	H	[sp+ls10]			Pop H using an immediate offset from the current stack pointer. The stack pointer is updated with the new address.
stm	[sp+ls16]	ag*X*,ag*Y*,...,ag*Z*	OpD	1	Store an ordered list of registers on the stack at an immediate offset from the current stack pointer. **NOTE** The stack pointer does not change.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
ldm	ag*X*,ag*Y*,...,ag*Z*	[sp+ls16]		4 [3]	Load an ordered list of registers from the stack at an immediate offset from the current stack pointer. **NOTE** The stack pointer does not change.
pushm	ag*X*,ag*Y*,...,ag*Z*		OpS	1[2,3]	Increment the stack pointer to the start of the next DMEM line (if needed), then push an ordered list of scalar registers or pointer registers onto the stack. The stack pointer is post-modified such that it points to the start of the next available (empty) full DMEM line.
popm	ag*X*,ag*Y*,...,ag*Z*		OpS	4 [3]	Decrement the stack pointer to the start of the previous DMEM line, then pop an ordered list of scalar registers or pointer registers from the stack.
pushm	a*X*,a*Y*,...,a*Z*		OpS	1[2,3]	Increment the stack pointer to the start of the next DMEM line (if needed), then push an ordered list of pointer registers onto the stack. The stack pointer is post-modified such that it points to the start of the next available (empty) full DMEM line.
popm	a*X*,a*Y*,...,a*Z*		OpS	4 [3]	Decrement the stack pointer to the start of the previous DMEM line, then pop an ordered list of pointer registers from the stack.
push	l32		OpD	1	Store an immediate scalar to the stack.
Internal Peripheral Control instructions					VSPA instructions quick reference
mvip	lu9	l32	OpD	1	Move an immediate scalar to an IP register. • The immediate address of the IP register specifies the destination. Note that the immediate address needs to be in word index format, while the IP register will be in byte offset format.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
mvip	Iu9	g*X*,I32			Move a subset of a scalar register to an IP register. • An immediate integer bit mask is used to specify the source data in g*X*. • The immediate address of the IP register specifies the destination. Note that the immediate address needs to be in word index format, while the IP register will be in byte offset format.
mvip	g*X*	Iu9,I32		1 or 2[4]	Move an IP register to a scalar register. • An immediate integer bit mask is used to specify the source data in the IP register. • An immediate address identifies the IP register source. Note that the immediate address needs to be in word index format, while the IP register will be in byte offset format.
mvip	Iu9	g*X*,g*Y*	OpS	1	Move a subset of a scalar register to an IP register. • An integer bit mask in g*Y* is used to specify the source data in g*X* • The immediate address of the IP register specifies the destination. Note that the immediate address needs to be in word index format, while the IP register will be in byte offset format.
mvip	g*X*	Iu9,g*Y*		1 or 2[4]	Move an IP register to a scalar register. • An integer bit mask in g*Y* is used to specify the source data in the IP register. • An immediate address identifies the IP register source. Note that the immediate address needs to be in word index format, while the IP register will be in byte offset format.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
mvip	[g*Z*]	g*X*(,g*Y*)		1	Move into IP register given by index in g*Z*, g*X* is data, optional gY is mask (if not provided use implicit mask of 0xffffffff).
mvip	g*X*	[g*Z*](,g*Y*)		1 or 2[4]	Move into g*X* from IP register given by index in g*Z*, optional gY is mask (if not provided use implicit mask of 0xffffffff).
bclrip	Iu9	Iu5		1	Clear bit at position Iu5 in IP register
bclrip	Iu9	g*X*		1	Clear bit at position gX[0:5] in IP register
bsetip	Iu9	Iu5		1	Set bit at position Iu5 in IP register
bsetip	Iu9	g*X*		1	Set bit at position gX[0:5] in IP register
clrip	Iu9	I32	OpD	1	Clear all bits in IP register given by mask
clrip	Iu9	g*X*	OpS	1	Clear all bits in IP register given by gX mask
setip	Iu9	I32	OpD	1	Set all bits in IP register given by mask
setip	Iu9	g*X*	OpS	1	Set all bits in IP register given by gX mask
btstip	Iu9	Iu5	OpS	1	Test bit at position Iu5 in IP register **NOTE** This instruction cannot be used on a slow IP register.
btstip	Iu9	g*X*		1	Test bit at position gX[0:5] in IP register **NOTE** This instruction cannot be used on a slow IP register.
Operand Generator instructions					VSPA instructions quick reference
set.nco	{radix2, singles, normal}, Iu10, Is31		OpD	3	Initialization of the vector NCO generator mode and frequency parameters. Order of arguments is k, f.
mv	nco_k	Iu11	OpB	3	Set the vector NCO generator (k-parameter) to an immediate scalar. Note, only affects least significant 11 bits of the half word register.

Table continues on the next page...

Table 15. VCPU instructions summary (continued)

OpCode	Dest	Source	Family	Cycles	Description
add	nco_k	Is11			Offset the vector NCO generator (k-parameter) by a signed immediate scalar.
mv	nco_reg	g*X*	OpB	3(nco_k), 2(nco_phase), 3(nco_freq)	Move a value from a scalar register g*X* into an NCO configuration register.
mv	g*X*	nco_reg		1(nco_k), 1(nco_phase), 1(nco_freq)	Move a value from an NCO configuration register into a scalar register g*X*.
Miscellaneous instructions					VSPA instructions quick reference
nop			OpA, OpB, OpC, OpS, OpV, OpZ	1	No operation.

1. The address modifier update latency is one cycle.
2. The data stored to memory can be the source of a load instruction in the next cycle. However, there is another cycle required before the data is visible to the debugger in memory.
3. The stack pointer modification latency is one cycle.
4. Reads of normal IP registers require 1 cycle; reads of slow IP registers require 2 cycles.

Table 16. Macro-instruction format 1 combinations

OpX	OpA			OpB	OpV	OpZ
swbreak	add(.laddr)	a*U*	Is9	See Table 18	See Table 20	loop_begin
	addA	a*U*	a*V*, a*W*			loop_end
	subA	a*U*	a*V*, a*W*			rts
	set.br	ag*X*	fft_size			
	mvA.VRAptr	r*X*	ag*Y*			
	mvA.VRAptr	ag*Y*	r*X*			
	mvA.VRAincr	r*X*	ag*Y*			
	mvA.VRAincr	ag*Y*	r*X*			
	mvA.VRArange1	r*X*	ag*Y*			

Table continues on the next page...

Table 16. Macro-instruction format 1 combinations (continued)

OpX	OpA			OpB	OpV	OpZ
	mvA.VRArange2	r*X*	ag*Y*			
	mvA.VRArange1	ag*Y*	r*X*			
	mvA.VRArange2	ag*Y*	r*X*			
	setA.VRAptr	r*X*	Iu9			
	setA.VRAincr	r*X*	Is9			
	ldA(.laddr)	[ag*X*]+Is9				
	stA(.laddr)	[ag*X*]+Is9				
	stA(.laddr).w	[ag*X*]+Is9				
	ld(.br)	[ag*X*]+/-ag*Y*				
	st(.llr_mode)	[ag*X*]+/-ag*Y*				
	st.w(.br)	[ag*X*]+/-ag*Y*				
	st.uline	ag*X*				

Table 17. Macro-instruction format 2 combinations

OpX	OpC	OpV	OpZ
swbreak	See Table 19	See Table 20	loop_begin
			rts
			break (MUST be in parallel with OpC loop_break or swi) **NOTE** This OpZ instruction is added by the assembler and is not specified by the user.

Table 18. OpB instructions

OpB			OpB		
set.prec	S0prec, S1prec, S2prec, Auprec, Vprec		fill.h	[rV]	g*X*
mv	nco_reg	g*X*	mv.h	g*X*	[rS0]

Table continues on the next page...

Table 18. OpB instructions (continued)

OpB			OpB		
mv	g*X*	nco_reg	floatx2n		g*X*, g*Y*
mv	nco_k	lu11	lfsr		g*X*, g*Y*
add	nco_k	ls11	fix2float	g*X*	g*Y*
fa0to1		lu2	float2fix	g*X*	g*Y*
set.xtrm	{signed,unsigned},{min,max}, {even,all},{value,index} N		hfixtofloatsp	g*X*	g*Y*
xtrm	a*U*, g*X*		floatsptohfix	g*X*	g*Y*
xtrm	g*X*		floathptofloatsp	g*X*	g*Y*
xtrm	a*U*		floatsptofloathp	g*X*	g*Y*
set.loop	lu10		mvB	sp	ag*X*
set.loop	ag*X*		mvB	ag*X*	sp
loop_stop			mvB	ag*X*	ag*Y*
set.rot		(lt_mode), (rt_mode)			
clr.VRA					
ldB	R*x*				
mv	R*x*	R*y*			
lsb2rf	[rV]	g*X*	setB.creg	creg	lu4
lsb2rf.sr	[rV]	g*X*	set.Smode	set.Smode ((S0conj,) (S0chs,) (S0mode,) (S1mode,) (S2mode))	
fill.w	[rV]	g*X*	setB.VRAincr	r*X*	ls9
mv.w	[rV]	g*X*	setB.VRAptr	r*X*	lu9
setB.loop	ag*X*	ls5	nop		
mvB.VRAptr	r*X*	ag*Y*	mvB.VRArange1	r*X*	ag*Y*
mvB.VRAptr	ag*Y*	r*X*	mvB.VRArange2	r*X*	ag*Y*
mvB.VRAincr	r*X*	ag*Y*	mvB.VRArange1	ag*Y*	r*X*

Table continues on the next page...

Table 18. OpB instructions (continued)

OpB			OpB		
mvB.VRAincr	ag*Y*	r*X*	mvB.VRArange2	ag*Y*	r*X*
mv.h	[rV]	g*X*			
fill.q	[rV]	g*X*			
mv.q	[rV]	g*X*			
fill.d	[rV]	g*X*			
mv.d	[rV]	g*X*			
mv.w	g*X*	[rS0]			

Table 19. OpC instructions

OpC			OpC		
set.loop	lu10, lu16		ldhC.s	g*Y*	[sp]+ls18
setC.loop	lu16		ldhC	g*Y*	[sp+ls18]
set.loop	ag*X*, lu10		ldhC.s	g*Y*	[sp+ls18]
set.VRArange1	r*X*	lu9, lu9	ldhC.u	g*Y*	[sp+ls18]
set.VRArange2	r*X*	lu9, lu9	ldhC.u.s	g*Y*	[sp+ls18]
jsr(.cc)	lu25		stC	[ag*X*]+ls18	
jmp(.cc)	lu25		stC.w	[ag*X*]+ls18	
loop_break(.cc)	lu25		stC	[ag*X*]+ls18	g*Y*
jsr(.cc)	g*X*		stC	[ag*X*+ls18]	g*Y*
jmp(.cc)	g*X*		stC	[sp+ls18]	ag*Y*
jsr(.cc)	[pc+ls25]		stC	[sp]+ls18	ag*Y*
jmp(.cc)	[pc+ls25]		stC.u	[ag*X*+ls18]	g*Y*
jsr(.cc)	[pc+g*X*]		stC.u	[sp+ls18]	ag*Y*
jmp(.cc)	[pc+g*X*]		sthC	[ag*X*]+ls18	g*Y*
set.range	a*Y*	ag*X*, lu19	sthC	[ag*X*+ls18]	g*Y*
set.range	a*Z*	ag*X*, g*Y*	sthC.u	[ag*X*+ls18]	g*Y*

Table continues on the next page...

Table 19. OpC instructions (continued)

OpC			OpC		
mv	a*U*	lu19	sthC	[sp]+ls18	g*Y*
add	a*V*	a*U*, ls19	sthC	[sp+ls18]	g*Y*
add	a*U*	sp, ls19	sthC.u	[sp+ls18]	g*Y*
cmp	a*U*, lu19		sth	[ag*X*]+ls9	l16
ldC	[ag*X*]+ls18		st.low	[ag*X*]+ag*Y*	lu8
ldC	g*Y*	[ag*X*]+ls18	st.low	[ag*X*]-ag*Y*	lu8
ldC	g*Y*	[ag*X*+ls18]	st.low	[ag*X*]+ls16	lu8
ldC	ag*Y*	[sp+ls18]	st.low	[ag*X*]+ag*Y*	g*Z*
ldC.u	g*Y*	[ag*X*+ls18]	st.low	[ag*X*]-ag*Y*	g*Z*
ldC.u	ag*Y*	[sp+ls18]	st.low	[ag*X*]+ls16	g*Z*
ldhC	g*Y*	[ag*X*]+ls18	st.high	[ag*X*]+ag*Y*	lu8
ldhC.s	g*Y*	[ag*X*]+ls18	st.high	[ag*X*]-ag*Y*	lu8
ldhC	g*Y*	[ag*X*+ls18]	st.high	[ag*X*]+ls16	lu8
ldhC.s	g*Y*	[ag*X*+ls18]	st.high	[ag*X*]+ag*Y*	g*Z*
ldhC.u	g*Y*	[ag*X*+ls18]	st.high	[ag*X*]-ag*Y*	g*Z*
ldhC.u.s	g*Y*	[ag*X*+ls18]	st.high	[ag*X*]+ls16	g*Z*
ldhC	g*Y*	[sp]+ls18	swi(.cc)	lu16	

Table 20. OpV instructions

OpVr		OpVsx		OpVau	OpVd	OpVrot
ld.normal	R*x*	rd	S0	rmad	wr.hlinecplx	rol
ld.h2h	R*x*		S1	rmad.sau	wr.even	ror
ld.h2l	R*x*		S2	cmad	wr.straight	
ld.h2l_l2h	R*x*			cmad.sau	wr.fft1	
ld.l2l	R*x*			rmac	wr.fft2	
ld.l2h	R*x*			rmac.sau	wr.fft3	

Table continues on the next page...

Table 20. OpV instructions (continued)

OpVr		OpVsx		OpVau	OpVd	OpVrot
ld.l2h_h2l	R*x*			cmac	wr.fft4	
ld.replace_h	R*x*			cmac.sau	wr.fft5	
ld.replace_l	R*x*			mads	wr.fft6	
clr	R*x*			mads.sau	wr.fft7	
bin2num	R*x*			maf	wr.fftn	
ld.qam	R*x*			mafac	wr.fn	
ld.2scomp	R*x*			clr.au	wr.fn1	
				dif.sau		
				rcp		
				rrt		
				srt		
				atan		
				nco		
				padd		

4.5 System control registers

The System Control Register (CReg) unit contains multiple addressable fields for setting modes used by various VSPA instructions.These fields are detailed in Table 21.

Table 21. CREG Fields

index	Field Description	Width	Reset Value
0	Real/Complex mode (0=real; 1=complex) **NOTE** This bit will be forced to zero when creg 22 (legacy dmem addressing mode) bit is cleared; furthermore, after clearing creg 22, if the creg 0 bit (real/complex) is set, the behavior is undefined (ie, it is not recommended to set the real/complex bit after creg 22 is cleared).	1 bit	0
1	Reserved	1 bit	0
2	Reserved	--	--

Table continues on the next page...

Table 21. CREG Fields (continued)

index	Field Description	Width	Reset Value
3	VAU output mode control (0=normal; 1=sign; 2=binary) Mode 0 - normal: when written back to VRA, it returns normal values in specified precision Mode 1 - sign: when written back to VRA, negative values return -1 (if half_fixed, 0xffff), non-negative values return 1 (if half_fixed, 0x7fff) Mode 2 - binary: when you write back to VRA, negative values return 0, non-negative values return 1 (if half_fixed, 0x7fff)	2 bits	00
4	Condition Code update switch (0=no update; 1=update) See Condition code flags for additional details.	1 bit	0
5-9	Reserved	--	--
10	CReg read address - most significant nibble	4 bits	0000
11	CReg read address - least significant nibble	4 bits	0000
12	H register control bit 0: initial state bit (Z bit) bit 1: sign capture enable bit (0=no capture; 1=capture) bit 2: auto-increment enable (0=no auto-increment; 1=auto-increment) See Vector sign capture register (H) for details.	3 bits	000
13	HPfixed scale-by-2 switch bit 0: Vprec control bit 1: S1prec control (0=no scaling; 1=scale-by-2) S1 (input) is scaled up by 2, V (output) is scaled down by 2	2 bits	00
14	H register sub-address. Read only.	--	--
15	FFT permutations mode switch bit 0: controls Sx-mux mode bit 1: write enable mask for bit 0 (1=enable) bit 2: controls V-mux mode bit 3: write enable mask for bit 2 (1=enable) To change the state of bit 0 or bit 2 the respective write enable mask bit must be set, this allows either one or both bits to be changed with one register write.	4 bits	0000
16	Fractional interpolator numerator constant (0 -15). Restriction on numerator: D<N<2D, where N is the numerator constant and D is the denominator constant.	4 bits	0000

Table continues on the next page...

Table 21. CREG Fields (continued)

index	Field Description	Width	Reset Value
17	Fractional interpolator denominator constant (0 -15). Restriction on denominator: N/2<D<N, where N is the numerator constant and D is the denominator constant.	4 bits	0000
18	Fractional interpolator phase (0 -15). Restriction on phase: if NAU%N == 0, phase=0, else phase <N, where 'N' is the numerator constant and '%' is the modulus operator	4 bits	0000
19	SP VAU output width switch (0=Single line; 1=Two lines)	1 bit	0
20	SP VAU output lane switch This controls which portion of the AU output gets written to VRA line on write back under circumstances that the output width is larger than a single register line. Note: only applicable if creg(19)=0 Vprec=SP: 0,2=0: 1023; 1,3=1024: 2047	2 bits	00
21	Reserved	2 bits	00
22	legacy_dmem_addr 0=Turn off the mode; 1=Turn on the mode **NOTE** Clearing this bit will also have the effect of clearing the real/complex bit (creg 0).	1 bit	1
23	order_g This is the exponent of base 2 in determining number of elements (n) in group, see description of S0group2nr and S0group2nc. Value of order_g is restricted (1<2^order_g<16).	3 bits	000
24	order_i This is the exponent of base 2 in determining number of elements (n) in group, see description of S1interp2nr and S1interp2nc. Value of order_i is restricted (1<2^order_i<64).	3 bits	000
25-254	Reserved		
255	Used for setting multiple fields in a single instruction based on a 4-bit immediate index value The following table shows the modification of fields {0, 1, 3, 13, 15, 19} for each index value of creg index 255. Note that VDPW and AUOM are assigned 0 and 00, respectively, in all cases.	N/A	N/A

Table continues on the next page...

Table 21. CREG Fields (continued)

index	Field Description	Width	Reset Value

index	0	1	3	13	13	15	15	19
bit#	0	0	1:0	1	0	2	0	0
CREG State	RC	VDPW	AUOM	HPFS	HPFV	FFTV	FFTS	ALLAU
0	0	0	00	0	0	0	0	0
1	0	0	00	0	0	0	0	1
2	0	0	00	0	1	0	0	0
3	0	0	00	1	0	0	0	0
4	0	0	00	1	1	0	0	0
5	1	0	00	0	0	0	1	0
6	1	0	00	0	0	0	0	1
7	1	0	00	0	0	1	0	0
8	1	0	00	0	0	0	0	0
9	1	0	00	0	0	1	1	0
10	1	0	00	0	1	0	0	0
11	1	0	00	0	1	0	1	0
12	1	0	00	0	1	1	1	0
13	1	0	00	1	0	0	0	0
14	1	0	00	1	0	0	1	0
15	1	0	00	1	0	1	1	0

RC - Real or complex mode

NOTE

RC will not be set by special update if legacyMemAddr is 0. Setting real/complex when the creg register 22 is set is allowed but not recommended.

VDPW - Vector data path width (wide/narrow)

AUOM - AU output mode control (normal, sign, bin)

HPFS - Half fixed scale by 2 for S1

HPFV - Half fixed scale by 2 for V

FFTS - FFT mode control bits for S

FFTV - FFT mode control bits for V

ALLAU - All AU write back enable (for single mode)

These fields are set (written) using the 'set.creg' instruction and can be read using the opS 'mv' instruction (mv gZ,cc). In order to read a specific Creg field value, the "read address" fields (10,11) must first be written to the index value of the field to be read, setting field 10 with the most significant bits (nibble) of the index value and setting field 11 with the least significant bits of the index value. The data returned and stored in the GP destination register will be in the following format:

{0000000000000000bbbbiiiiiiiiNZVC}

where the upper 16 bits are 0s; bits [15:12] are the value of the referenced field; bits[11:4] are the index value of the field; bits [3:0] are the Condition Code (cc) bits (N,Z,V,C).

NOTE

Note that it is not possible to read back multiple fields in a single read operation (that is, setting fields 10,11 to 255 will result in 0 value for bits [15:12] above. If div/mod and mpy instructions perform write-back on the same cycle, CC bits will be based on mpy result.

4.5.1 Control register latency

NOTE

This section only applies to legacy_dmem_addressing mode (creg 22 register is set to 1), but is not relevant when that register is cleared (creg 22 register is cleared), since the instructions now carry the real/complex information. Furthermore, when legacy_dmem_addressing mode control register (creg 22) is cleared by the programmer, the real/complex mode control register (creg 0) is automatically cleared by the hardware. It is recommended that, the programmer flush the AU pipeline before performing this operation in complex mode.

In general, the creg bits have a one-cycle latency before taking affect.

The bits in the following control registers take two cycles before taking effect: 0, 3, 15, 19, 20.

The bits in the following control registers take three cycles before taking effect: 16, 17, 18.

The Au Real/Complex mode control, creg bit 0, is pipelined to affect the Source Register, Arithmetic Unit and the AU writeback data. This allows the programmer to intermix complex and real operation.

The Au Real/Complex mode affects the Source Register after two cycles.

The Au Real/Complex mode affects the Arithmetic Unit after four cycles.

The Au Real/Complex mode affects the AU writeback data after eight cycles when AUprec is Single Precision.

The Au Real/Complex mode affects the AU writeback data after nine cycles when AUprec is Double Precision due to the extra cycle needed in the AU for double precision operations.

Control register latency code example

```
//
// Assume that the Au mode bit is set to real.
// Assume that S0prec,S1prec,S2prec,Vprec = half_fixed and AUprec=single
//
set.creg 0,1;                       // Set Au mode to Complex mode
set.creg 0,0;                       // Set Au mode to Real mode
rd S0; rd S1; rd S2;                // Complex Load
rd S0; rd S1; rd S2;                // Real Load
cmad;                                // Complex Operation
rmad;                                // Real Operation
nop;
nop;
wr.hlinecplx;                 // Complex Writeback
wr.straight;                  // Real Writeback
```

4.5.2 Vector precision latency

The set.prec instruction is Pipelined to affect the Source Register, Arithmetic Unit and the AU writeback data. This allows the programmer to intermix different precision operations.

The Vprec affects the Source Register after one cycle.

The Vprec affects the Arithmetic Unit after three cycles.

The Vprec affects the AU writeback data after seven cycles when AUprec is Single Precision.

Vector precision latency code example

```
//
// Assume that S0prec,S1prec,S2prec,Vprec = half_fixed and AUprec=single
//
 set.prec single, single, single, single, single;
// Sprec = Single Precision affects Src Load
 set.prec half_fixed, half_fixed, half_fixed, single, half_fixed; rd S0; rd S1;
rd S2;
// Sprec = Half Fixed Precision affects Src Load
 rd S0; rd S1; rd S2;
 cmad; // AUprec = Single Precision affects AU unit
 cmad; // AUprec = Single Precision affects AU unit
 nop;
 nop;
 wr.hlinecplx; // Vprec=Single Precision effects AU writeback.
 wr.hlinecplx; // Vprec=Half Fixed Precision effects AU writeback
```

4.6 VCPU condition codes

The VCPU conditions codes consist of 4 bits (N,Z,V,C) which may be updated by various instructions as shown in Table 22. These bits represent the following conditions:

N - The result of an operation is negative (that is, the most significant bit (31) is a 1)

Z - The result of an operation is zero

V - A two's complement overflow has occurred (that is, the sign of the result is not as expected)

C - A carry (or borrow) has occurred from an add (or sub)

Table 22. Instructions affecting the CC bits

Instruction	N	Z	V	C	Update description
addD(.ucc)(.cc) g*X*, g*Y*, I32 (sp allowed)	+	+	+	+	Only updates condition codes if system control register cc_update OR .ucc bit of instruction is set
subD(.ucc)(.cc) g*X*, g*Y*, I32 (sp allowed)	+	+	+	+	Only updates condition codes if system control register cc_update OR .ucc bit of instruction is set
andD(.cc) g*X*, g*Y*, I32 (sp allowed)	+	+	-	-	Only updates condition codes if system control register cc_update is set
orD(.cc) g*X*, g*Y*, I32	+	+	-	-	Only updates condition codes if system control register cc_update is set
cmpD(.cc) g*Y*, I32	+	+	+	+	Always updates condition codes

Table continues on the next page...

Table 22. Instructions affecting the CC bits (continued)

Instruction	N	Z	V	C	Update description
xorD(.cc) g*X*, g*Y*, I32	+	+	-	-	Only updates condition codes if system control register cc_update is set
mpyD(.cc) g*X*, g*Y*, I32	+	+	+	+	Only updates condition codes if system control register cc_update is set
cmp a*U*, Iu19	+	+	+	+	Always updates condition codes
sr g*X*, g*Y*, I (logical)	0	+	-	-	Only updates condition codes if system control register cc_update is set
sr.s g*X*, g*Y*, I (arith)	-	+	-	-	Only updates condition codes if system control register cc_update is set
sl g*X*, g*Y*, I	+	+	-	-	Only updates condition codes if system control register cc_update is set
btst g*Y*, I	+	+	-	-	Always updates condition codes
andS(.z) g*X*, I16 (sp allowed)	+	+	-	-	Only updates condition codes if system control register cc_update is set
orS g*X*, Is	+	+	-	-	Only updates condition codes if system control register cc_update is set
xorS g*X*, Is	+	+	-	-	Only updates condition codes if system control register cc_update is set
mpy(.cc)(.s) g*Z*, g*X*, g*Y*	+	+	+	+	Only updates condition codes if system control register cc_update is set
div(.cc)(.s) g*Z*, g*X*, g*Y*	+	+	+	+	Only updates condition codes if system control register cc_update is set
mod(.cc)(.s) g*Z*, g*X*, g*Y*	+	+	+	+	Only updates condition codes if system control register cc_update is set
sr(.cc) g*Z*, g*X*, g*Y*	0	+	-	-	Only updates condition codes if system control register cc_update is set
sr(.cc).s g*Z*, g*X*, g*Y*	-	+	-	-	Only updates condition codes if system control register cc_update is set
addS(.ucc)(.cc) g*Z*, g*X*, g*Y* (sp allowed)	+	+	+	+	Only updates condition codes if system control register cc_update OR .ucc bit of instruction is set
subS(.ucc)(.cc) g*Z*, g*X*, g*Y* (sp allowed)	+	+	+	+	Only updates condition codes if system control register cc_update OR .ucc bit of instruction is set
cmp(.cc) g*X*, g*Y* (sp allowed)	+	+	+	+	Always updates condition codes
sl(.cc) g*Z*, g*X*, g*Y*	+	+	-	-	Only updates condition codes if system control register cc_update is set
and(.cc) g*Z*, g*X*, g*Y* (sp allowed)	+	+	-	-	Only updates condition codes if system control register cc_update is set
or(.cc) g*Z*, g*X*, g*Y*	+	+	-	-	Only updates condition codes if system control register cc_update is set
xor(.cc) g*Z*, g*X*, g*Y*	+	+	-	-	Only updates condition codes if system control register cc_update is set
bclr(.cc) g*Z*, g*X*, g*Y*	+	+	-	-	Only updates condition codes if system control register cc_update is set
not(.cc) g*Z*, g*X*	+	+	-	-	Only updates condition codes if system control register cc_update is set

Table continues on the next page...

Table 22. Instructions affecting the CC bits (continued)

Instruction	N	Z	V	C	Update description
abs g*Z*, g*X*	0	+	+	+	Only updates condition codes if system control register cc_update is set
mv cc, Iu4	+	+	+	+	Always updates condition codes
cmpS.s g*Z*, Is16	+	+	+	+	Always updates condition codes
div.s g*Z*, Is16	+	+	+	+	Only updates condition codes if system control register cc_update is set
mod.s g*Z*, Is16	+	+	+	+	Only updates condition codes if system control register cc_update is set
mpyS.s g*Z*, Is16	+	+	+	+	Only updates condition codes if system control register cc_update is set
rdiv.s g*Z*, Is16	+	+	+	+	Only updates condition codes if system control register cc_update is set
rmod.s g*Z*, Is16	+	+	+	+	Only updates condition codes if system control register cc_update is set
rsub.s g*Z*, Is16	+	+	+	+	Only updates condition codes if system control register cc_update is set
addS(.ucc).s g*Z*, Is16 (sp allowed)	+	+	+	+	Only updates condition codes if system control register cc_update OR .ucc bit of instruction is set
subS(.ucc).s g*Z*, Is16 (sp allowed)	+	+	+	+	Only updates condition codes if system control register cc_update OR .ucc bit of instruction is set

NOTE

'+' means that the bit will be updated by the instruction and is dependent on a result, '-' means that the bit is not affected by the instruction and '0' means that the bit will always be set to 0 by the instruction. Any instructions not listed in table Table 22 can never affect the CC bits.

4.6.1 Multiply condition codes

The condition code bits which result from the multiply instruction have slightly different meaning than other operations. The N bit is derived from bit 63 of the multiply result. The Z bit is only set if all 64 bits of the result are 0. The V bit is set if there is a carry out of the multiply operation. The C is set in the following cases: an unsigned multiply has any bit set in the upper 32 bits of the result or a signed multiply has any bit set in the upper 33 bits of the result but the upper 33 bits are not all set.

4.6.2 Divide condition codes

The condition code bits which result from the divide instruction have slightly different meaning than other operations. The N bit is derived from bit 31 of the quotient. The Z bit is set if the quotient is 0. The V bit is set if the divisor is 0. The C bit is set if remainder is not 0.

4.6.3 Modulus condition codes

The condition code bits which result from the modulus instruction have slightly different meaning than other operations. The N bit is derived from bit 31 of the remainder. The Z bit is set if the remainder is 0. The V bit is set if the divisor is 0. The C bit is set if quotient is not 0.

4.7 Data memory pointer instructions

The MAG instructions manipulate the following MAG-related registers:

- a*X* registers (a0-a19)
- g*X* registers (g0-g11)

In addition, these instructions also enable or initialize various buffer ranges (min and max registers) supported within the MAG.

Table 23. Set Range Registers

set.range a0 set.range a1 set.range a2 set.range a3	Sets the range (min and max) for a0 buffer, a1 buffer, a2 buffer, a3 buffer.

For the following instructions

- `add aX, aY,`
- `add aX, Is17`
- `sub aX, aY,`

if the destination a*X* is either a0, a1, a2 or a3 and the result of the *add* operation lies outside the buffer range of a*X*, then the resulting contents of a*X* will be wrapped.

For example, consider the instruction `add a2, a3`. If the buffer range a2 is [min, max], then the buffer size of a2 will be given by size = max - min + 1. Assume that sum = a2 + a3. Then a2 will be updated as follows:

If sum is greater than max, a2 will take the value sum - size. If sum is less than min, a2 will take the value sum + size. If both the conditions are not true, a2 equals sum.

NOTE

If a3 is greater than size, then sum - size may still be greater than max and sum + size may still be less than min which could result in a2 being outside the range.

The wrap operation will only occur if the destination of the *add* operation is either a2 or a3 and the result of the *add* operation lies outside the buffer range of destination register.

Initialize a MAG buffer range by setting its min and max pointers. All legal MAG buffer, <mag_buff> are listed in Table 24.

Table 24. MAG Buffers

<mag_buff>	Mag Buffer Descriptions
a0	a0 buffer
a1	a1 buffer
a2	a2 buffer
a3	a3 buffer

This instruction has two formats. The first format uses Iu19 to specify the buffer size. The second format uses the content of a g*X* register to specify the buffer size.

The MAG buffer's min pointer is set to the content of the a*X* register.

The MAG buffer's max pointer is set to either

- a*X* + Iu19 - 1 (first instruction format), or
- a*X* + g*Y* - 1 (second instruction format).

Both formats of the instructions may be used to disable the range, with min pointer set to 0 and buffer size set to 0. For example,

```
mv a4, 0;
set.range a2, a4, 0;
```

disables the range on a2.

4.8 Data memory load & store instructions

The load/store instructions perform load/store operations to or from DMEM.

4.8.1 DMEM address generation modes (ptr_mode)

The address used for DMEM access can be generated using the following mode(s):

- Normal Mode

These addresses are generated using the content of an a*X* register. See Pointer reordering algorithms for more details.

4.8.2 Post-modifications of a*X* registers

Upon generating a DMEM address, the content of the a*X* register is post-modified (post-incremented or post-decremented), using one of the following modifiers.

- an immediate value specified in the ld or st instruction.
- another a*X* register, also referred to as a modifying a*X* register.

Note that, while there will be a latency to the DMEM read buffer (as indicated in Table 15), there will also be a latency of one cycle to post-increment the a*X* register.

If an immediate value is used as a post-modifier, a user can specify the increment amount in terms of number of words, or in terms of number of DMEM lines. For example,

```
ld [a2]+32;      // post-increment a2 by 32
ld [a2]-16;      // post-decrement a2 by 16
ld.laddr [a2]+2; // post-increment a2 by 2lines
ld.laddr [a2]-3; // post-decrement a2 by 3lines
```

4.8.3 Special notes

Special care are needed when accessing DMEM using certain modes, as described below.

4.8.3.1 Loads

The load instruction (*ld*) reads a vector from DMEM and holds it in an internal buffer for a subsequent explicit destination ld.Rmode instruction to write it into a VRA register or if it has an implicit destination a portion of it is autonomously written into a scalar register.

DMEM data from a *ld* instruction is loaded into a VRA register no earlier than three cycles after the *ld* operation by an *ld.Rmode*. When the destination is a scalar register, the DMEM data is autonomouly loaded after three cycles, no extra instructions are needed. All instructions which load from DMEM are shown in Table 25. These instructions all use the same memory pipeline, including an internal data register, and loads from the memory must consume the data before a subsequent load reaches the final cycle of the memory pipeline. A 'done' instruction should not be executed until the load instruction is complete, ie within the 3 instructions following the load.

NOTE

Table 25. Summary of load instructions

Load Instructions	Dest	Source	Description
Instructions requiring explicit destination load			These instructions only load data from memory through the pipeline into the internal data register, but not into the VRA. They must be followed, after the appropriate delay, with an OpVr *ld* instruction to complete the load of the data into the VRA.
ld(.br)	[ag*X*]+/-ag*Y*		Load a vector from DMEM using a pointer, then post-increment or post-decrement by the contents of a*gY*. The address used for the memory access can be optionally modified according to a re-order algorithm (bit reversal) specified by .br.
ldA(.laddr)	[ag*X*]+ls9		Load a vector from DMEM using a pointer, then post-increment or post-decrement by a signed immediate offset. If .laddr is included, update is by lines.
Instructions with implicit destination load			These instructions will load data from memory through the pipeline into the internal data register and then into the specified destination register. No following instruction is required to complete the load.
ld	g*Z*	[ag*X*]+/-ag*Y*	Load a 32-bit scalar from DMEM using a pointer and store the contents into g*Z*. post-increment or post-decrement by the contents of ag*Y*.
ld	g*Z*	[ag*X*]+/-ag*Y*x2	Load a scalar from DMEM using a pointer and store the contents into g*Z*. post-increment or post-decrement by 2 times the contents of ag*Y*.
ldS(.laddr)	g*X*	[ag*Y*]+*ls*	Load a 32-bit scalar from DMEM using a pointer and store the contents into g*X*. post-increment by a signed, immediate offset.
ld	g*Z*	[ag*X*+/-ag*Y*]	Generate an address via a pre-increment or pre-decrement of ag*X* by ag*Y*, then load a 32-bit scalar from this DMEM address and store the contents into g*Z*. The contents of a*gX* are not modified by this instruction.
ld.u	g*Z*	[ag*X*+/-ag*Y*]	Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ld	g*Z*	[ag*X*+/-ag*Y*x2]	Generate an address via a pre-increment or pre-decrement of ag*X* by 2 times ag*Y*, then load a 32-bit scalar from this DMEM address and store the contents into g*Z*. The contents of a*gX* are not modified by this instruction.
ld.u	g*Z*	[ag*X*+/-ag*Y*x2]	Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ldS(.laddr)	g*X*	[ag*Y*+*ls*]	Generate an address via a pre-increment of ag*Y* by a signed, immediate number, then load a 32-bit scalar from this DMEM address and store the contents into g*X*. The contents of ag*Y* are not modified by this instruction.
ldS(.laddr).u	g*X*	[ag*Y*+*ls*]	Alternative form of the previous instruction where the address used in the memory access is stored back into the pointer ag*Y*.

Table continues on the next page...

Table 25. Summary of load instructions (continued)

Load Instructions	Dest	Source	Description
ldh(.s)	g*Z*	[ag*X*]+/-ag*Y*	Load a 16-bit scalar from DMEM using a pointer and store the contents into g*Z*. If .s is included, the 16 bit value is sign extended into g*Z*, otherwise the value is zero extended. post-increment or post-decrement by the contents of ag*Y*.
ldh(.s)	g*Z*	[ag*X*+/-ag*Y*]	Generate an address via a pre-increment or pre-decrement of ag*X*, then load a 16-bit scalar from this DMEM address and store the contents into g*Z*. If .s is included, the 16 bit value is sign extended into gZ, otherwise the value is zero extended. The contents of a*gX* are not modified by this instruction.
ldh.u(.s)	g*Z*	[ag*X*+/-ag*Y*]	Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ldhS(.laddr)(.s)	g*Z*	[ag*X*]+*Is*	Load a 16-bit scalar from DMEM using a pointer and store the contents into g*Z*. post-increment by a signed, immediate offset. If .s is included, the 16 bit value is sign extended into gZ, otherwise the value is zero extended.
ldhS(.laddr)(.s)	g*Z*	[ag*X*+*Is*]	Generate an address via a pre-increment of ag*X* by a signed, immediate number, then load a 16-bit scalar from this DMEM address and store the contents into g*X*. If .s is included, the 16 bit value is sign extended into gZ, otherwise the value is zero extended. The contents of ag*Y* are not modified by this instruction.
ldhS(.laddr).u(.s)	g*Z*	[ag*X*+*Is*]	Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ld	g*Z*	*I*	Load a 32-bit scalar from DMEM using an immediate address and store the contents into g*Z*.
ldh(.s)	g*Z*	*I*	Load a 16-bit scalar from DMEM using an immediate address and store the contents into gZ. If .s is included, the 16 bit value is sign extended into gZ, otherwise the value is zero extended.
ldC	[ag*X*]+Is18		Load a vector from DMEM using a pointer, then post-increment or post-decrement by a signed immediate offset.
ldC	g*Y*	[ag*X*]+Is18	Load a 32-bit scalar from DMEM using a pointer and store the contents into g*Z*. post-increment by a signed, immediate offset.
ldC	g*Y*	[ag*X*+Is18]	Generate an address via a pre-increment of ag*X* by a signed, immediate number, then load a 32-bit scalar from this DMEM address and store the contents into g*Y*. The contents of ag*X* are not modified by this instruction.
ldC.u	g*Y*	[ag*X*+Is18]	Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.

Table continues on the next page...

Table 25. Summary of load instructions (continued)

Load Instructions	Dest	Source	Description
ldC	ag*Y*	[sp+ls18]	Generate an address via a pre-increment of sp by a signed, immediate number, then load a 32-bit scalar from this DMEM address and store the contents into ag*Y*. The contents of sp are not modified by this instruction.
ldC.u	ag*Y*	[sp+ls18]	Alternative form of the previous instruction, where the address used in the memory access is stored into the stack pointer.
ldhC	g*Y*	[ag*X*]+ls18	Load a 16-bit scalar from DMEM using a pointer and store the contents as a zero extended 32-bit value into g*Y*. post-increment by a signed, immediate offset.
ldhC	g*Y*	[ag*X*+ls18]	Generate an address via a pre-increment of ag*X* by a signed, immediate number, then load a 16-bit scalar from this DMEM address and store the contents as a zero extended 32-bit value into g*Y*. The contents of ag*X* are not modified by this instruction.
ldhC.u	g*Y*	[ag*X*+ls18]	Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ldhC.s	g*Y*	[ag*X*]+ls18	Load a 16-bit scalar from DMEM using a pointer and store the contents as a sign extended 32-bit value into g*Y*. post-increment by a signed, immediate offset.
ldhC.s	g*Y*	[ag*X*+ls18]	Generate an address via a pre-increment of ag*X* by a signed, immediate number, then load a 16-bit scalar from this DMEM address and store the contents as a sign extended 32-bit value into g*Y*. The contents of ag*X* are not modified by this instruction.
ldhC.u.s	g*Y*	[ag*X*+ls18]	Alternative form of the previous instruction, where the address used in the memory access is stored into the pointer ag*X*.
ldhC	g*Y*	[sp]+ls18	Load a 16-bit scalar from DMEM using the stack pointer and store the contents as a zero extended 32-bit value into g*Y*. post-increment by a signed, immediate offset.
ldhC	g*Y*	[sp+ls18]	Generate an address via a pre-increment of sp by a signed, immediate number, then load a 16-bit scalar from this DMEM address and store the contents as a zero extended 32-bit value into g*Y*. The contents of sp are not modified by this instruction.
ldhC.u	g*Y*	[sp+ls18]	Alternative form of the previous instruction, where the address used in the memory access is stored into the stack pointer.
ldhC.s	g*Y*	[sp]+ls18	Load a 16-bit scalar from DMEM using the stack pointer and store the contents as a sign extended 32-bit value into g*Y*. post-increment by a signed, immediate offset.
ldhC.s	g*Y*	[sp+ls18]	Generate an address via a pre-increment of sp by a signed, immediate number, then load a 16-bit scalar from this DMEM address and store the

Table continues on the next page...

Table 25. Summary of load instructions (continued)

Load Instructions	Dest	Source	Description
			contents as a sign extended 32-bit value into g*Y*. The contents of sp are not modified by this instruction.
ldhC.u.s	g*Y*	[sp+ls18]	Alternative form of the previous instruction, where the address used in the memory access is stored into the stack pointer.
ldm	a*X*, a*Y*, ..., a*Z*, g*X*, g*Y*, ..., g*Z*	[sp+ls16]	Decrement the effective address of (sp+ls16) to the start of the 32-element boundary address in the DMEM line, then load the ordered list of scalar registers from this effective address.

4.8.3.2 Stores

The store instruction (*st*) reads a selected VRA or scalar register and stores it to DMEM.

Select a VRA register using rSt.ptr, then post-increment or decrement rSt.ptr by an amount specified by rSt.incr. Select a scalar register by using a st [], ag*X* instruction. Data from the VRA or scalar register will be written into DMEM two cycles after the execution of the *st* instruction. A 'done' instruction should not be executed until the store instruction is complete, that is, in the first instruction following the store.

The width of the data store to memory for the different *st* instructions is shown in Table 26.

NOTE

Table 26. Data write width for st instructions

Store Instruction	Address Source	Data Source	Width of store to VRA
st(.br)(.llr_mode)	[ag*X*]+/-ag*Y*	VRA	line
stA(.laddr)	[ag*X*]+ls9	VRA	line
st.w(.br)	[ag*X*]+/-ag*Y*	VRA	32 bits
stA(.laddr).w	[ag*X*]+ls9	VRA	32 bits
st	[ag*X*]+/-ag*Y*	g*Z*	32 bits
st	[ag*X*]+/-ag*Y*x2	g*Z*	32 bits
st	[ag*X*+/-ag*Y*]	g*Z*	32 bits
st.u	[ag*X*+/-ag*Y*]	g*Z*	32 bits
st	[ag*X*+/-ag*Y*x2]	g*Z*	32 bits
st.u	[ag*X*+/-ag*Y*x2]	g*Z*	32 bits
stS(.laddr)	[ag*Y*]+*ls*	g*X*	32 bits
stS(.laddr)	[ag*Y*+*ls*]	g*X*	32 bits

Table continues on the next page...

Table 26. Data write width for st instructions (continued)

Store Instruction	Address Source	Data Source	Width of store to VRA
stS(.laddr).u	[ag*Y*+*ls*]	g*X*	32 bits
sth	[ag*X*]+/-ag*Y*	g*Z*	16 bits
sth	[ag*X*+/-ag*Y*]	g*Z*	16 bits
sth.u	[ag*X*+/-ag*Y*]	g*Z*	16 bits
sthS(.laddr)	[ag*Y*]+*ls*	g*Z*	16 bits
sthS(.laddr)	[ag*Y*+*ls*]	g*Z*	16 bits
sthS(.laddr).u	[ag*Y*+*ls*]	g*Z*	16 bits
st	[ag*X*]+ls15	ls16,ls16	32 bits
st	*I*	g*Z*	32 bits
sth	*I*	g*Z*	16 bits
st	lu19	l32	32 bits
sth	lu19	l16	16 bits
stC	[ag*X*]+ls18	VRA	line
stC.w	[ag*X*]+ls18	VRA	32 bits
stC	[ag*X*]+ls18	g*Y*	32 bits
stC	[ag*X*+ls18]	g*Y*	32 bits
stC.u	[ag*X*+ls18]	g*Y*	32 bits
stC	[sp+ls18]	g*Y*	32 bits
stC.u	[sp+ls18]	g*Y*	32 bits
sthC	[ag*X*]+ls18	g*Y*	16 bits
sthC	[ag*X*+ls18]	g*Y*	16 bits
sthC.u	[ag*X*+ls18]	g*Y*	16 bits
sthC	[sp]+ls18	g*Y*	16 bits
sthC	[sp+ls18]	g*Y*	16 bits
sthC.u	[sp+ls18]	g*Y*	16 bits

Table continues on the next page...

Table 26. Data write width for st instructions (continued)

Store Instruction	Address Source	Data Source	Width of store to VRA
sth	[ag*X*]+ls9	I16	16 bits
st.low	[ag*X*]+ag*Y*	Iu8	partial line
st.low	[ag*X*]-ag*Y*	Iu8	partial line
st.low	[ag*X*]+ls16	Iu8	partial line
st.low	[ag*X*]+ag*Y*	g*Z*	partial line
st.low	[ag*X*]-ag*Y*	g*Z*	partial line
st.low	[ag*X*]+ls16	g*Z*	partial line
st.high	[ag*X*]+ag*Y*	Iu8	partial line
st.high	[ag*X*]-ag*Y*	Iu8	partial line
st.high	[ag*X*]+ls16	Iu8	partial line
st.high	[ag*X*]+ag*Y*	g*Z*	partial line
st.high	[ag*X*]-ag*Y*	g*Z*	partial line
st.high	[ag*X*]+ls16	g*Z*	partial line
st.uline	ag*X*	VRA	variable

For a description of the *st.llr_mode* instructions, see st.llr_mode instruction.

4.8.3.2.1 st.llr_mode instruction

These *llr_mode* store instructions can also optionally specify the 16-bit precision type (half_fixed or half_float) from which the llr's are to be compressed. If the type is not specified it will default to 16-bit floating point.

st.llr4 instruction

The "st.llr4" instruction converts VRA data in 16-bit half-fixed format to 4 bit llr.

The "st.llr4half" instruction converts VRA data in 16-bit half-float format to 4 bit llr.

Write to DMEM is aligned to 1/4 line for llr4, but the address pointer does not need to be aligned to the fraction. This means that in st.llr4half [aX]+aY; aX does not need to be 1/4-line aligned. However, to prevent overwriting on subsequent stores, the increment value (aY) must be a multiple of the line fraction being written.

The following provides a detailed description of the operations of the st.llr4 [a*X*]+a*Y* instruction. See also Table 27.

First, define a function called *flt2llr4*(), as follows:

```
Function: flt2llr4(x[15:0][,type]) -> y[3:0]
Input:    a half-word number and optional type
Output:   a signed 4-bit llr symbol

if (type != half_fixed)mant_msb=9
else     mant_msb=14
if (x[mant_msb:mant_msb-2] != 111b)
```

```
     temp[2:0] = x[mant_msb:mant_msb-2] + {0, 0, x[mant_msb-3]}; // rounding
else
     temp[2:0] = 111b; // saturation

y[3:0] = {x[15], temp[2:0]};
```

The flt2llr4() function takes a half-word number (x[15:0]) as input and outputs a two's complement 4-bit llr value (y[3:0]). The sign bit of the input will be used as the sign bit of the output.

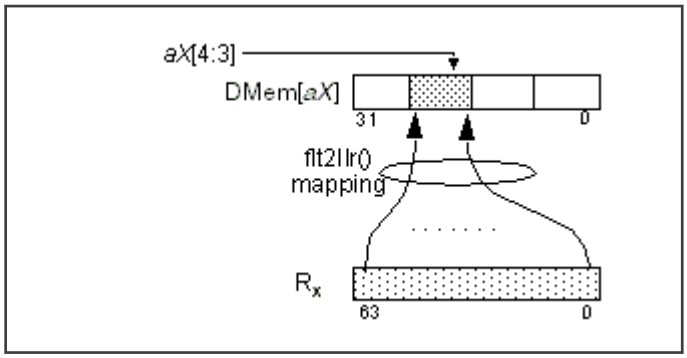
The mantissa bits to be used for the compression are determined by the optional type input. If the type is specified as half_fixed, the mantissa msb (mant_msb) is selected as 14, otherwise it is selected as 9.

If the three most significant bits of the mantissa of the input floating number (x[mant_msb:mant_msb-2]) are all ones, then set temp[2:0] to all ones (saturation operation). Otherwise, set temp[2:0] to be the sum of x[mant_msb:mant_msb-2] and x[mant_msb-3] (rounding operation). Lastly, set the ouput y[3:0] = {x[15], temp[2:0]}.

The operations of "st.llr4 [a*X*]+a*Y*" instruction are described in Table 27.

Table 27. The operations of "st [aXx]+aY, 4" instruction

st llr Instruction	Operations	Descriptions

st.llr4 [a*X*]+a*Y*	Rx <- VRA[rSt.*ptr*[2:0]];
	temp[255:0] = 0;
	i = 0;
	foreach real_elem in Rx {
	temp[255:0] = temp[255:0] | (flt2llr(real_elem) << (4*i));
	i=i+1;
	}
	memLine[1023:0] = DMem[*aX*];
	j = *aX*[4:3]; // quarter-line select
	memLine[256*(j+1)-1:256*j] = temp[255:0];
	DMem[*aX*] <- memLine[1023:0];
	aX <- *aX* + *aY*;
	rSt.*ptr* <- rSt.*ptr* + rSt.*incr*;
	 
		Read a VRA register, selected using rSt.ptr[2:0], and assign it to Rx.
		For each half-word in Rx, convert it into a 4-bit number, using the flt2llr() function. This conversion process produces a 256-bit data, or a quarter of a DMEM line. Assign this quarter line to temp[255:0].
		Overwrite a quarter of DMEM line located at *aX* by temp[255:0], using *aX*[4:3] as a quarter-line select within the DMEM line.
		Post-increment or decrement *aX* by adding or subtracting *aY*.
		Post-increment or decrement rSt.*ptr* by adding rSt.*incr*.
		Note: only one-quarter of DMEM line is written. The rest of DMEM line remains unchanged.

st.llr8 instruction

The "st.llr8" instruction converts VRA data in 16-bit half-fixed format to 8 bit llr.

The "st.llr8half" instruction converts VRA data in 16-bit half-float format to 8 bit llr.

Write to DMEM is aligned to 1/2 line for llr8, but the address pointer does not need to be aligned to the fraction. This means that in st.llr8half [aX]+aY; aX does not need to be 1/2-line aligned. However, to prevent overwriting on subsequent stores, the increment value (aY) must be a multiple of the line fraction being written.

The following provides a detailed description of the operations of the st.llr8 [a*X*]+a*Y* instruction. See also Table 28.

First, define a function called *flt2llr8*(), as follows.

```
      Function: flt2llr8(x[15:0][,type]) -> y[7:0]
      Input:    a half-word number and optional type
      Output:   a signed 8-bit llr value

      if (type != half_fixed)mant_msb=9
      else     mant_msb=14
      if (x[mant_msb:mant_msb-6] != 1111111b)
           temp[6:0] = x[mant_msb:mant_msb-6] + {0, 0, 0, 0, 0, 0, x[mant_msb-7]}; //
rounding
      else
           temp[6:0] = 1111111b; // saturation

      y[7:0] = {x[15], temp[6:0]};
```

The flt2llr8() function takes a half-word number (x[15:0]) as input and outputs a two's complement 8-bit llr value (y[7:0]). The sign bit of the input will be used as the sign bit of the output.

The mantissa bits to be used for the compression are determined by the optional type input. If the type is specified as fixed, the mantissa msb (mant_msb) is selected as 14, otherwise it is selected as 9.

If the seven most significant bits of the mantissa of the input floating number (x[mant_msb:mant_msb-6]) are all ones, then set temp[6:0] to all ones (saturation operation). Otherwise, set temp[7:0] to be the sum of x[mant_msb:mant_msb-6] and x[mant_msb-7] (rounding operation). Lastly, set the ouput y[7:0] = {x[15], temp[6:0]}.

The operations of "st.llr4 [a*X*]+a*Y*" instruction is described in Table 28.

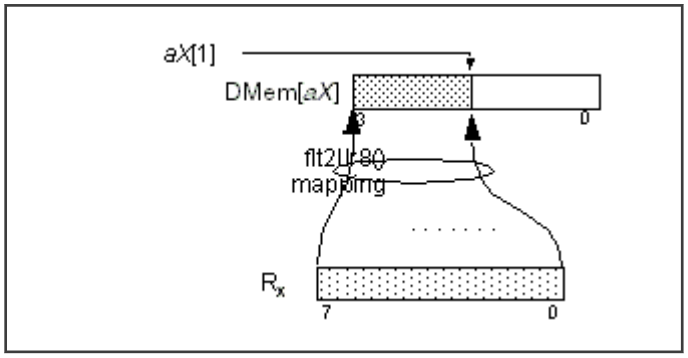
Table 28. The Operations of "st [aX]+aY, 8" Instruction

st.mem llr Instruction	Operations	Descriptions

st.llr4 [a*X*]+a*Y*	Rx <- VRA[rSt.*ptr*[2:0]];
	temp[511:0] = 0;
	i = 0;
	foreach real_elem in Rx {
	temp[511:0] = temp[511:0] | (flt2llr8(real_elem) << (8*i));
	i=i+1;
	}
	memLine[1023:0] = DMem[*aX*];
	j = *aX*[1]; // half-line select
	memLine[512*(j+1)-1:512*j] = temp[511:0];
	DMem[*aX*] <- memLine[1023:0];
	aX <- *aX* + *aY*;
	rSt.*ptr* <- rSt.*ptr* + rSt.*incr*;
		Read a VRA register, selected using rSt.ptr[2:0], and assign it to Rx.
		For each half-word element in Rx, convert it into a 8-bit number, using the flt2llr8() function. This conversion process produces a 512-bit data, or a half of a DMEM line. Assign this half line to temp[511:0].
		Overwrite a half of DMEM line located at *aX* by temp[511:0], using *aX*[1] as a half-line select within the DMEM line.
		Post-increment or decrement *aX* by adding or subtracting *aY*.
		Post-increment or decrement rSt.*ptr* by adding rSt.*incr*.

Table continues on the next page...

Table 28. The Operations of "st [aX]+aY, 8" Instruction

st.mem llr Instruction	Operations	Descriptions
		Note: only one-half of DMEM line is written. The rest of DMEM line remains unchanged.

4.8.3.2.2 st.uline instruction

This instruction will store a variable number of VRA elements to DMEM based on a value written to the ST_UL_VEC_LEN IP register. The VRA elements will be stored to DMEM given by the value in destination pointer agX. The MAG pointer may be truncated based on a configuration parameter and the data rotated into the truncated alignment. See VSPA register descriptions for the unalign bitfield description of the PARAM0 register and a description of the alignment. The source register Rx is indicated by the rSt VRA pointer.

See VSPA register descriptions for bit field descriptions of ST_UL_VEC_LEN register.

Store arbitrary length with arbitrary alignment

```
// on entry assume:
// a0 holds destination pointer
// g0 holds element count (# of 16-bit elements to be stored)
// rSt points to a RAG source register
sr g1, g0, SR_ELEM_TO_LINES;      // SR_ELEM_TO_LINES is a constant and a power
of 2
add g1, g1, 2;                    // +1 in case of fractional lines, and +1 in
case of misalignment
set loop g1, 1;                   // use a 1-instruction loop
mvip ST_VEC_LEN, g0;              // configure size of vector to be stored
using staa instruction
loop_begin; st.uline [a0]; loop_end; // store until finished
```

4.8.3.2.3 st.low

This instruction will store a partial aligned vector of elements from the lower portion of the vector register into the lower portion of the DMEM line beginning at 0. The index of the leftmost written element is given by lu8 or gZ depending on the form of the instruction used. The address pointer will be post-incremented.

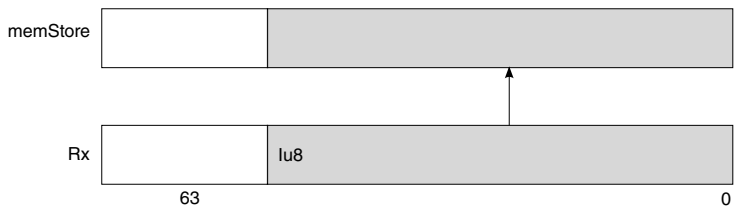


Figure 14. st.low

Table 29. Summary of st.low instructions

st.low Instructions	Dest	Source	Description
st.low	[ag*X*]+/-ag*Y*	Iu8	Store right justified partial vector from R[rSt] to DMEM using a pointer, then post-increment/decrement by the contents of agY. The index of the leftmost written element is given by Iu8.
st.low	[ag*X*]+Is16	Iu8	Store right justified partial vector from R[rSt] to DMEM using a pointer, then post-increment by 16 bit offset. The index of the leftmost written element is given by Iu8.
st.low	[ag*X*]+/-ag*Y*	g*Z*	Store right justified partial vector from R[rSt] to DMEM using a pointer, then post-increment/decrement by the contents of agY. The index of the leftmost written element is given in gZ.
st.low	[ag*X*]+Is16	g*Z*	Store right justified partial vector from R[rSt] to DMEM using a pointer, then post-increment by 16 bit offset. The index of the leftmost written element is given in gZ.

4.8.3.2.4 st.high

This instruction will store a partial aligned vector of elements from the upper portion of the vector register into the upper portion of the DMEM line beginning at 63. The index of the rightmost written element is given by Iu8 or gZ depending on the form of the instruction used. The address pointer will be post-incremented.

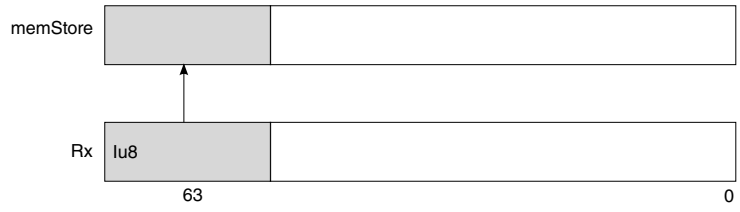


Figure 15. st.high

Table 30. Summary of st.high instructions

st.high Instructions	Dest	Source	Description
st.high	[ag*X*]+/-ag*Y*	Iu8	Store left justified partial vector from R[rSt] to DMEM using a pointer, then post-increment/decrement by the contents of agY. The index of the rightmost written element is given by Iu8.
st.high	[ag*X*]+Is16	Iu8	Store left justified partial vector from R[rSt] to DMEM using a pointer, then post-increment by signed 16 bit offset. The index of the rightmost written element is given by Iu8.
st.high	[ag*X*]+/-ag*Y*	g*Z*	Store left justified partial vector from R[rSt] to DMEM using a pointer, then post-increment/decrement by the contents of agY. The index of the rightmost written element is given in gZ.

Table continues on the next page...

Table 30. Summary of st.high instructions (continued)

st.high Instructions	Dest	Source	Description
st.high	[ag*X*]+ls16	g*Z*	Store left justified partial vector from R[rSt] to DMEM using a pointer, then post-increment by signed 16 bit offset. The index of the rightmost written element is given in gZ.

4.8.3.3 MAG update conflicts

Certain legal instruction combinations may try to write to the same hardware resources (or registers) in the same cycle. For example, a mv instruction and an add instruction may both try to write to a0 at the same time. In this case, a set of rules is used to govern which instruction has precedence and write back its result, and which will not. Only the instruction with a higher precedence will write back its result.

Table 31 lists all instruction combinations that exhibits potential hardware resource conflicts. The Table also shows the instruction precedence in each of these cases.

Table 31. Instruction Precedent for Parallel Instructions

First Instruction	Second Instruction	Shared Registers	Instruction with Higher Precedent
mv[1]	add[1]	a*X*	mv
	sub[1]	a*X*	
	ld[1]	*aX*	
	st[1]	*aX*	

1. Assuming the two parallel instructions are writing to the same *aX* register.

when an `mv` instruction has a resource conflict with another instruction, the former always wins.

Example 1: Upon executing the following code segment, a3 will be set to 0, instead of 1032.

```
mv a3, 1024;
mv a5, 0;
mv a3, a5; ld [a3]+8;          // mv takes precedence: a3 <- 0
```

4.9 Vector register array instructions

The `ld.Rmode Rx` instruction loads the contents of the memRead bus into the VRA. A muxing logic block called LdMux performs various optional shift operations on the memRead bus data before the latter is written into the VRA. These optional shift operations are also referred to as Rmode.

Table 32. Load VRA Modes

Instructions	Brief Descriptions
ld.normal R*x*	Load memRead bus into R*x*[1]
ld.h2l_l2h R*x*	Load high part of memRead bus into low part of R*x* and load low part of memRead bus into high part of R*x*+1.[2]

Table continues on the next page...

Table 32. Load VRA Modes (continued)

Instructions	Brief Descriptions
	R*x* can be R0, R2, R4 or R6.
ld.l2h_h2l R*x*	Load low part of memRead bus into high part of R*x* and load high part of memRead bus into low part of R*x*+1. R*x* can be R0, R2, R4 or R6.
ld.h2l R*x*	Load high part of memRead bus into low part of R*x*.
ld.l2h R*x*	Load low part of memRead bus into high part of R*x*.
ld.replace_h R*x*	Replace the most significant word in R*x* with a word on the memRead bus.
ld.replace_l R*x*	Replace the least significant word in R*x* with a word on the memRead bus.
ld.h2h R*x*	Load high part of memRead bus into high part of R*x*.
ld.l2l R*x*	Load low part of memRead bus into low part of R*x*.
ld.qam R*x*	For modulation order M, take a M/32 line of memReadbus (32M bits), transform it into a full line, and write it into R*x*. The transformation implements QAM modulation.
ld.2scomp R*x*	Load partial line from memRead bus into a full line with type conversion of 2's complement value to half fixed value, write full line data into Rx

1. R*x* refers to a VRA register.
2. R*x*+1 refers to the next sequential VRA register following R*x*. For example, if R*x* is R2, then R*x*+1 is R3.

Descriptions

- In the following table, a*X* refers to the content of the a*X* register that was used to initiate the corresponding *ld* operation two or more cycles earlier.
- a*X*[5:0] refers to the significant lower 6 bits of a*X*.
- R0[0] refers to the first (or least significant) element of R0.
- memRead[*i:j*] refers to the subvector of memRead bus that contains j^{th} through i^{th} words (inclusive) of memRead bus, where *i* >= *j* and *i, j*=0,...63
- Most *ld* instructions use the lower 6 bits of a*X* to determine how the shift operation is to be performed by LdMux.

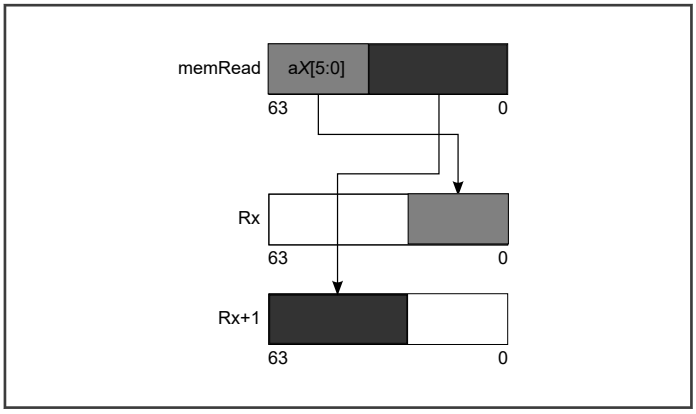
Table 33. VRA ld instructions

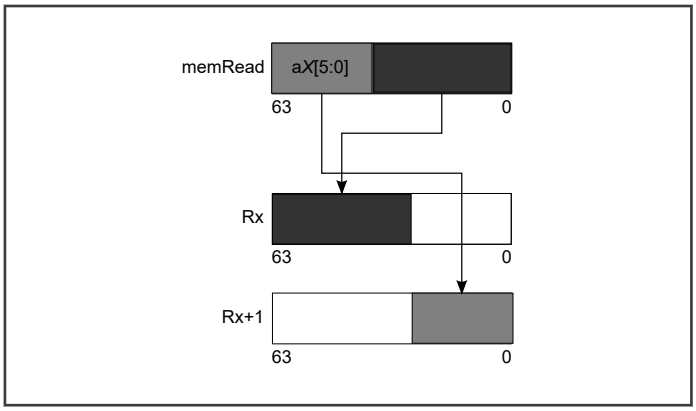
ld Instruction	Operation	Description
ld.normal R*x*	R*x*[63:0] <- memRead[63:0]; where R*x* = R0,...,R7	Load memRead bus into R*x*.

ld.h2l_l2h R*x*	R*x*[63- a*X*[5:0] :0] = memRead[63:a*X*[5:0]], where R*x*=R0,R2,R4 or R6.
	If (a*X*[5:0] != 0) {
	R*x*+1[63 : 64-a*X*[5:0]] = memRead[a*X*[5:0]-1 : 0]
	}
		Load high part of memRead bus into low part of R*x*;
		If a*X*[5:0] is not zero, then load low part of memRead bus into

Table continues on the next page...

Table 33. VRA ld instructions (continued)

ld Instruction	Operation	Description
	where R*x*+1 denotes the next sequential VRA register after R*x*. 	high part of R*x*+1; otherwise, R*x* +1 will not be modified.

ld.l2h_h2l R*x*	If (a*X*[5:0] != 0) {
	R*x*[63:64-a*X*[5:0]] = memRead[a*X*[5:0]-1:0];
	}
	where R*x*=R0,R2,R4 or R6.
	R*x*+1[63- a*X*[5:0] : 0] = memRead[63 : a*X*[5:0]]; where R*x*+1 denotes the next sequential VRA register after R*x*.
	 
		If a*X*[5:0] is not zero, then load low part of memRead bus into high part of R*x*; otherwise, R*x* will not be modified.
		Load high part of memRead bus into low part of R*x*+1.

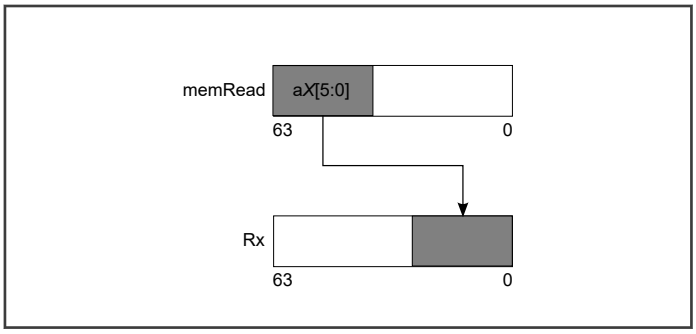
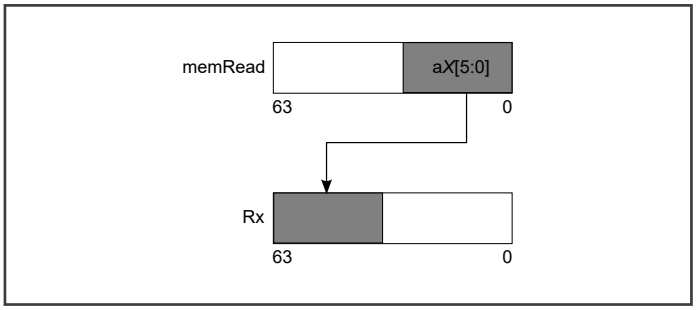
ld.h2l R*x*	R*x*[63- a*X*[5:0] : 0] = memRead[63 : a*X*[5:0]]; where R*x*=R0,...,R7. 	Load high part of memRead bus into low part of R*x*.

Table continues on the next page...

Table 33. VRA ld instructions (continued)

ld Instruction	Operation	Description

ld.l2h R*x*	If (a*X*[5:0] != 0) {
	R*x*[63 : 64-a*X*[5:0]] = memRead[a*X*[5:0]-1 : 0];
	}
	where R*x*=R0,...,R7.
	 
		If a*X*[5:0] is not zero, load low part of memRead bus into high part of R*x*; otherwise R*x* will not be modified.

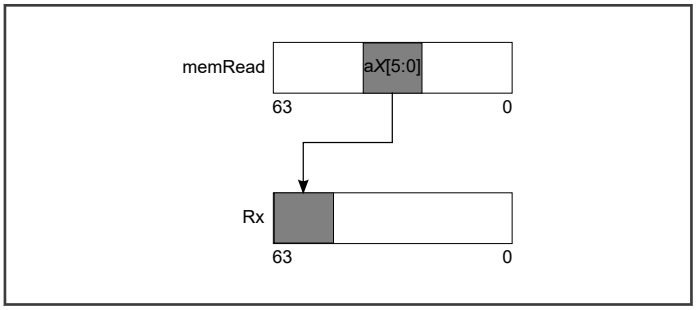
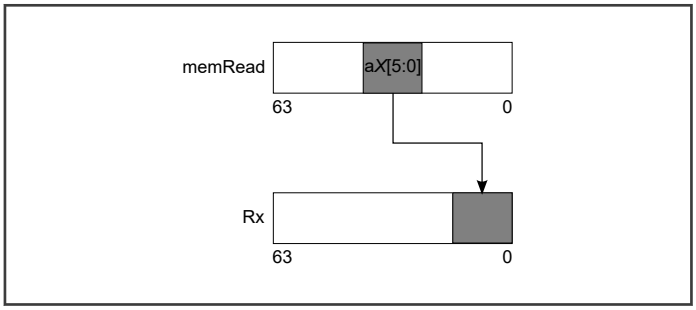
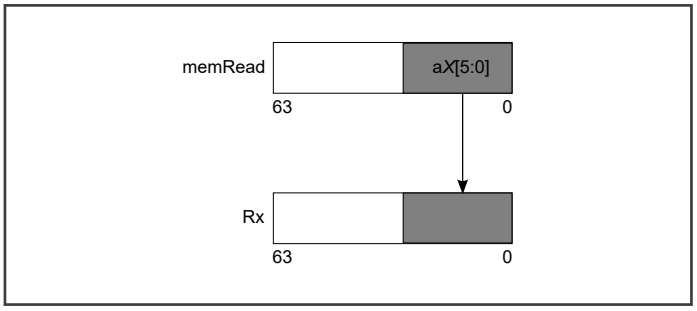
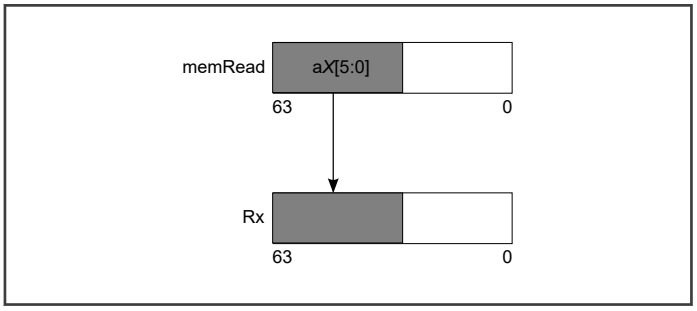
ld.replace_h R*x*	R*x* = {memRead[a*X*[5:0]], R*x*[62:0]}; where R*x*=R0,...,R7. 	Replace word R*x*[63] with a word on the memRead bus pointed to by a*X*[5:0]
ld.replace_l R*x*	R*x* = {R*x*[63:1], memRead[a*X*[5:0]]}; where R*x*=R0,...,R7. 	Replace word R*x*[0] with a word on the memRead bus pointed to by a*X*[5:0]
ld.l2l R*x*	R*x*[a*X*[5:0]-1 : 0] = memRead[a*X*[5:0]-1 : 0]; where R*x*=R0,...,R7.	Replace low part of R*x* by the low part of memRead bus.

Table continues on the next page...

Table 33. VRA ld instructions (continued)

ld Instruction	Operation	Description
		
ld.h2h R*x*	R*x*[63 : a*X*[5:0]] = memRead[63 : a*X*[5:0]]; where R*x*=R0,...,R7. 	Replace high part of R*x* by the high part of memRead bus.

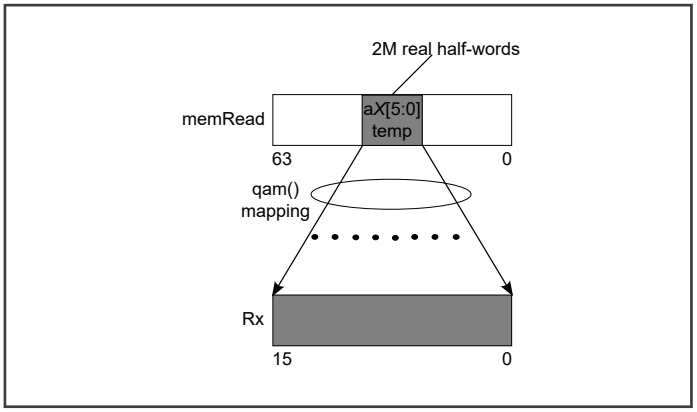
ld.qam R*x*	Modulation order: M (can be 1, 2, 4, 6, 8 or 10)
	temp = memRead[a*X* + (M-1): a*X*];
	for (i=0; i<64; i++) {
	j = temp & (2M-1);
	temp = temp >> M;
	R*x*[2*i + 0] = real (qam(j));
	R*x*[2*i + 1] = imag (qam(j));
	}
	 
		Extract 2M half-words (or M full-words) from memRead bus, starting from the word pointed to by a*X*. Assign this 32M-bit data to temp.
		For each M bits in temp, starting from the least significant bits, convert it into a 32-bit complex element (half fixed) using the corresponding modulation mapping functions, and write the resulting word into R*x* starting from element 0.
		The modulation order is configured by programming the LD_RF_CONTROL register. The modulation mapping functions are constructed by programming the IP registers LD_RF_TB_REAL_0 and LD_RF_TB_IMG_0 if modulation mode <= 0100. If modulation mode == 0110 or 1000, LD_RF_TB_REAL_0 to LD_RF_TB_REAL_7, and LD_RF_TB_IMAG_0 to

Table continues on the next page...

Table 33. VRA ld instructions (continued)

ld Instruction	Operation	Description
		LD_RF_TB_IMAG_7 are all used. If modulation mode == 1010, LD_RF_TB_REAL_0, LD_RF_TB_REAL_1, LD_RF_TB_IMAG_0 and LD_RF_TB_IMAG_1 are used. The modulation values are mapped into the IP registers according to the modulation index (i), with index i=0 occupying bits 1:0, i=1 in bits 3:2, and so on. According to the following formula: value for index (i) is stored in bits (2i+1):(2i).

ld.2scomp R*x*	Size S (can be 8, 10, 12, 16)
	temp = memRead[a*X* + ((S*2)-1): a*X*];
	for (i=0; i<64; i++) {
	temp = temp >> S*2;
	R*x*[2*i + 0] = {temp[S*2*i],|temp[(S*2*i)-1:0]|,0};
	R*x*[2*i + 1] = {temp[S*((i*2)+1)],|temp[(S*((i*2)+1))-1:0]|,0};
	}
		Extract 2S half-words (or S full words) from memRead bus, starting from the word pointed to by a*X*. Assign this 16S bit data to temp. If required, bit reverse each S bits in temp and assign to temp2 else assign temp to temp2.
		For each S bits in temp2, starting from the least significant bits, convert it into a 16-bit element (half fixed) using the corresponding two's complement type conversion functions, and write the resulting element into Rx starting from element 0.
		The 2's complement value will be left justified into the half fixed value such that the negative value with the largest absolute value will produce a negative half-scale (0xc000) result. For S=16, 2's complement 0x8000 will saturate to 0xffff."

ld R*x* example code

```
ld.normal R2;
ld.h2l_l2h R4;
ld.l2h_h2l R4;
ld.h2l R3;
```

```
ld.l2h R7;
ld.replace_h R6;
ld.replace_l R3;
ld.l2l R5;
ld.h2h R3;
clr R4;
ld.qam R2;
```

Special Notes

- Since a memory load takes three cycles, an *ld.Rmode* instruction must be executed at least three cycles after the correcponding *ld* instruction. For example,

```
mv a3, 0;
ld.laddr [a3]+1;                        // load memory line from address 0
ld.laddr [a3]+1;                        // load memory line from address 32
ld.laddr [a3]+1;                         // load memory line from address 64
ld.laddr [a3]+1; ld.normal R6;     // load memory line from address 96; load data
                                         // from address 0 into R6
                 ld.normal R5;     // load data from address 32 into R5
                 ld.normal R4;     // load data from address 64 into R4
                 ld.normal R7;     // load data from address 96 into R7
```

- The memRead bus will retain its data indefinitely until the next load from memory operation overwrites its contents. Consequently,
  - A *ld.Rmode* instruction can be executed more than two cycles after a corresponding load from memory instruction, if there is no intervening load from memory instruction.
  - In addition, multiple *ld.Rmode* can be executed with a single load from memory instruction. In this case, the same data will be loaded into the VRA multiple times.

4.9.1 RAG instructions

The RAG instructions manipulate the RAG registers, which include the following five RAG register sets:

- rS0
- rS1
- rS2
- rV
- rSt

Each of the above RAG register sets has six registers: ptr, incr, start1, wrap1, start2 and wrap2.

A register within a RAG register set is denoted by <RAGset>.<register>. For example, the ptr register of rS0 is denoted by rS0.ptr.

Table 34. RAG Instructions

Instruction	Description
setA.VRAptr r*X*, Iu9	Set the 9 bits in ptr register of a RAG register set.
setB.VRAptr r*X*, Iu9	Set the 9 bits in ptr register of a RAG register set.
setA.VRAincr r*X*, Iu9	Set the 9 bits in incr register of a RAG register set.
setB.VRAincr r*X*, Iu9	Set the 9 bits in incr register of a RAG register set.
set.VRAptr Iu9, Iu9, Iu9, Iu9, Iu3	Set the 9 bits in ptr register of rS0, rS1, rS2 and rV RAG register sets and set the 3 bits in ptr register of rSt RAG register set.
set.VRAincr Is9, Is9, Is9, Is9, Is3	Set the 9 bits in incr register of rS0, rS1, rS2 and rV RAG register sets and set the 3 bits in incr register of rSt RAG register set.
set.VRArange1 r*X*, Iu9, Iu9	Specifies the range of the first buffer (start1 and wrap1 registers) for a RAG register set.
set.VRArange2 r*X*, Iu9, Iu9	Specifies the range of the second buffer (start2 and wrap2 registers) for a RAG register set.
mvA.VRAptr r*X*, ag*Y*	Move into a VRA pointer register from a scalar register
mvB.VRAptr r*X*, ag*Y*	Move into a VRA pointer register from a scalar register
mvA.VRAptr ag*Y*, r*X*	Move into a scalar register from a VRA pointer register
mvB.VRAptr ag*Y*, r*X*	Move into a scalar register from a VRA pointer register
mvA.VRAincr r*X*, ag*Y*	Move into a VRA increment register from a scalar register
mvB.VRAincr r*X*, ag*Y*	Move into a VRA increment register from a scalar register
mvA.VRAincr ag*Y*, r*X*	Move into a scalar register from a VRA increment register
mvB.VRAincr ag*Y*, r*X*	Move into a scalar register from a VRA increment register
mvA.VRArange1 r*X*, ag*Y*	Move into a VRA range register from a scalar register
mvA.VRArange2 r*X*, ag*Y*	Move into a VRA range register from a scalar register
mvB.VRArange1 r*X*, ag*Y*	Move into a VRA range register from a scalar register
mvB.VRArange2 r*X*, ag*Y*	Move into a VRA range register from a scalar register
mvA.VRArange1 ag*Y*, r*X*	Move into a scalar register from a VRA range register
mvA.VRArange2 ag*Y*, r*X*	Move into a scalar register from a VRA range register
mvB.VRArange1 ag*Y*, r*X*	Move into a scalar register from a VRA range register

Table continues on the next page...

Table 34. RAG Instructions (continued)

Instruction	Description
mvB.VRArange2 ag*Y*, r*X*	Move into a scalar register from a VRA range register
clr.VRA	Clear all RAG registers (ptr, incr, start1, wrap1, start2, wrap2) for all RAG register sets.

NOTE

- When reading or writing a single element to a VRA register (such as storing SAU output to R7), the lower bits of RAG.ptr register are used to specify which element is being accessed.
  - In real mode, a half-word element is selected by ptr[4:0].
  - In complex mode, a full-word element is selected by ptr[4:1], with ptr[0] being ignored.

4.9.1.1 RAG circular buffers configurations

RAG.ptr is updated each time the corresponding VRA port is accessed. For example, rS0.ptr is updated each time Source Operand Register S0 is loaded.

RAG.ptr is updated based on the following register contents (see VRA pointer control registers for more details):

- the magnitude and sign of RAG.incr register,
- the setting of the first circular buffer, RAG.start1 and RAG.wrap1, and
- the setting of the second circular buffer, RAG.start2 and RAG.wrap2.

The first circular buffer is disabled if and only if both RAG.start1 and RAG.wrap1 are set to zero. Likewise, the second circular buffer is disabled if and only if both RAG.start2 and RAG.wrap2 are set to zero.

For each RAG register set, the following buffer configurations are supported:

- both buffers are enabled,
- both buffers are disabled, and
- the first buffer is enabled, but the second buffer is disabled.

NOTE

If the second circular buffer is enabled, but the first circular buffer is disabled, the hardware will behave as if both buffers are disabled. Programmers should avoid using this buffer configuration.

4.9.1.2 RAG pointer update algorithm

The following shows the RAG.ptr update algorithm.

```
update_rag_ptr:
   nxt_ptr = ptr + incr;

   if (both buffers are enabled) {
         if (ptr == wrap1)
            nxt_ptr = start2;
         else if (ptr == wrap2)
            nxt_ptr = start1;
      }
   } else if (first buffer is enabled) {
         if (ptr == wrap1)
            nxt_ptr = start1;
      }
   }
```

```
ptr = nxt_ptr;
```

In the above RAG.ptr update algorithm, if only the first buffer is enabled, ptr will wrap to wrap1 if incr is negative and ptr hits start1. Conversely, ptr will wrap to start1 if incr is positive and ptr hits wrap1.

If both buffers are enabled and incr is negative, ptr will wrap to wrap2 if it hits start1 or to wrap1 if it hits start2. Conversely, if both buffers are enabled and incr is positive, ptr will wrap to start2 if it hits wrap1 or to start1 if it hits wrap2. In these cases, the ptr will go back and forth between the two buffers.

CAUTION

The RAG register modulus operation is not a true modulus operation. For it to wrap around correctly, the result of <RAGset>.ptr + <RAGset>.incr must be exactly equal to <RAGset>.wrap1 or <RAGset>.wrap2. See the following example.

The following code examples exhibit incorrect behaviors.

```
set.VRArange1 rS0, 0, 3;    // rS0.start1 = 0, rS0.wrap1 = 3
set.VRAincr rS0, 2;         // rS0.incr = 2
```

In the example above, rS0.ptr increments as follows: 0, 2, 4, and so on. Since rS0.ptr was never equal to 3, it will never wrap back to 0.

```
set.VRArange1 rS0, 0, 4;    // rS0.start1 = 0, rS0.wrap1 = 4
set.VRAincr rS0, 2;         // rS0.incr = 2
```

In the example above, rS0.ptr increments as follows: 0, 2, 4, and so on. Whenever rS0.ptr equals 4, it will wrap back to 0. Thus rS0.ptr will be updated as follows: 0, 2, 4, 0, 2, 4, 0, 2, 4, and so on.

4.9.1.3 Syntax for RAG pointer value

RAG pointer values are used in the following RAG instructions:

- setA.VRAptr r*X*, Iu9
- setB.VRAptr r*X*, Iu9
- set.VRAptr Iu9, Iu9, Iu9, Iu9, Iu3
- set.VRArange1 r*X*, Iu9, Iu9
- set.VRArange2 r*X*, Iu9, Iu9
- mv.VRArange1 rX, agY - rSxwrap1=agY[24:16], rSxstart1=agY[8:0]
- mv.VRArange2 rX, agY - rSxwrap2=agY[24:16], rSxstart2=agY[8:0]
- mv.VRArange1 agY, rX - agY[24:16]=rSxwrap1, agY[8:0]=rSxstart1
- mv.VRArange2 agY, rX - agY[24:16]=rSxwrap2, agY[8:0]=rSxstart2
- mv.VRAincr rX, agY - rSxincr=agY[8:0]
- mv.VRAincr agY, rX - agY[8:0]=rSxincr
- mv.VRAptr rX, agY - rSxptr=agY[8:0]
- mv.VRAptr agY, rX - agY[8:0]=rSxptr

The VRA has eight registers, each with 64 half-word elements. It can be thought of as a linear array indexed from element 0 through element 511, where element 0 refers to R0 element 0, and 511 refers to R7 element 63.

A half-word element in the VRA can be specified using an integer value ranging from 0 through 63.

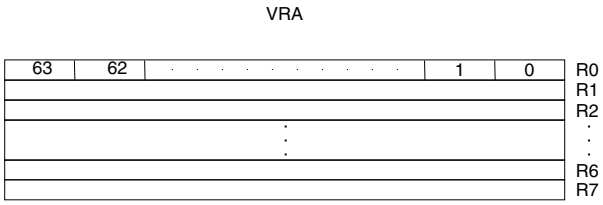


Figure 16. Data Packing in the VRA (half-word)

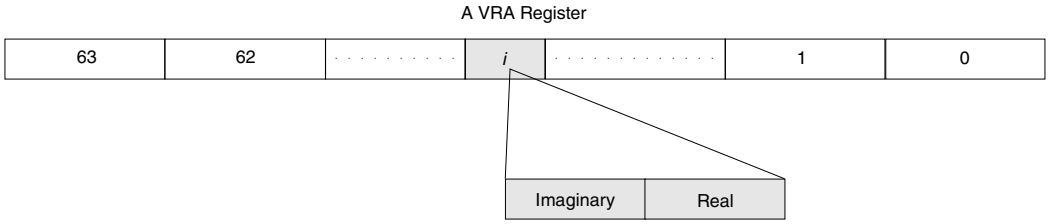


Figure 17. Data packing in a VRA register line (half-word)

4.9.1.4 Syntax for RAG increment value

RAG increment values are used in the following RAG instructions:

- set.VRAincr r*X*, Is9
- set.VRAincr Is9, Is9, Is9, Is9, Is3
- mvA.VRAincr r*X*, ag*Y*
- mvB.VRAincr r*X*, ag*Y*
- mvA.VRAincr ag*Y*, r*X*
- mvB.VRAincr ag*Y*, r*X*

A RAG increment value can be specified by using a signed integer value ranging from -256 to 255

4.9.1.5 RAG destination conflicts

Certain legal instruction combinations may try to write to the same hardware resources (or registers) in the same cycle. For example, a *setA.VRAptr* instruction and an *rd* instruction may both try to write to rS0 at the same time. In this case, a set of rules is used to govern which instruction has precedence. Only the instruction with a higher precedence will write back its result.

Table 35 lists all instruction combinations that exhibits potential hardware resource conflicts. The table also shows the instruction precedence in each of these cases.

Table 35. Instruction Precedent for Parallel Instructions

First Instruction	Second Instruction	Shared Registers	Higher Precedent
any of: • setA.VRAptr r*X*, Iu9 • setB.VRAptr r*X*, Iu9	rd S0	rS0	setA/setB
	rd S1	rS1	

Table continues on the next page...

Table 35. Instruction Precedent for Parallel Instructions (continued)

First Instruction	Second Instruction	Shared Registers	Higher Precedent
• mv.[h,w] gX, [rS0] • mv.[h,w,q] [rV], gX • fill.[h,w,q] [rV], gX	rd S2	rS2	
	wr	rV	
	st	rSt	

When a setA/setB instruction has a resource conflict with another instruction, the former always win.

4.10 Rotate register instructions

The rotate register (rot) instructions perform shift or rotate operations, either right, left or both, on the vector registers. The rotate right instructions operate on R0, R1, R2 or R3 or by chaining both R0 and R1 or both R2 and R3 based on the setting of the rotate right mode register (ror_mode_reg). The rotate left instructions operate on R4, R5, R6 or R7 or by chaining both R4 and R5 or both R6 and R7 based on the setting of the rotate left mode register (rol_mode_reg). These latter registers dictate how the shift or rotate operations are to be performed.

The ror_mode_reg/rol_mode_reg can be set prior to a shift or rotate operation or it can also be set during the same cycle where the shift/rotate operation is to be performed.

Once set, the rotate mode registers will retain its setting until the next set.rot instruction that explicitly sets or updates that register.

Table 36 shows all rot instructions. In this table, rot_mode denotes a supported rot mode. See Rotate-register modes for all supported rot_mode.

Table 36. Rrot Instructions

Rrot Instructions	Instruction Format	Descriptions
set.rot (ror_mode),(rol_mode)	OpB	Set ror_mode_reg with ror_mode. Set rol_mode_reg with rol_mode.

4.10.1 Rotate-register modes

Following sections show all supported <ror_mode> and all supported <rol_mode>.

Four VRA registers are involved in the right rotate/shift operations: R0, R1, R2 and R3. Specifically, R0 and R1 can be chained together to perform rotate/shift operations. Likewise, R2 and R3 can be chained together to perform rotate/shift operations. There are 3 special rotation modes for each register or register combination which are used to rotate data which has been up-sampled.

The ror_mode format is a combination of the VRA register or combination of registers, along with the direction and amount of shift or rotation. For example, ror_mode R0r2 is a combination of the following:

- "R0", which indicates the shift is to occur on the R0 VRA register and
- "r2", in which "r" indicates that the shift is a right shift and "2" indicates a shift of 2 real elements (half words).

The special rotation mode mentioned above has a slightly different format which is similar to the example R0r2 used above except that the "r2" portion of the ror_mode that represents the direction and amount of shift is substituted with "rND2", which means it shifts right by 2 real elements according to the fractional sample rate N/D.

- "r" indicates the shift is a right shift,
- "N" is the numerator of the fractional sample rate,
- "D" is the denominator of the fractional sample rate, and

- "2" indicates a shift of 2 real elements (half words).

The fractional sample rate is derived from values programmed into CREG fields 16, 17 and 18.

Four VRA registers are involved in the left rotate/shift operations: R4, R5, R6 and R7. Specifically, R4 and R5 can be chained together to perform rotate/shift operations. Likewise, R6 and R7 can be chained together to perform rotate/shift operations.

The rol_mode format is a combination of the VRA register or combination of registers, along with the direction and amount of shift or rotation. For example, rol_mode R4l2 is a combination of the following:

- "R4", which indicates the shift is to occur on the R4 VRA register and
- "l2", in which "l" indicates that the shift is a left shift and "2" indicates a shift of 2 real elements (half words).

4.10.1.1 Right rotate/shift on R0 and R1

In R0R1r1, R0R1r2, R0R1r4, R0R1r8, R0R1rND1, R0R1rND2 and R0R1rND4 modes, R0 and R1 are stitched together and form a circular chain. For each of R0R1r1, R0R1r2, R0R1r4 and R0R1r8 modes, the chain is then rotated right by 1, 2, 4 or 8 real elements, respectively. The R0R1rND1, R0R1rND2 and R0R1rND4 modes are special modes to rotate data for up-sampling by 2. For each of R0R1rND1, R0R1rND2 and R0R1rND4 modes, the chain is then rotated right by 1, 2 or 4 real elements, respectively, according to fractional sample rate N/D.

In R0r1, R0r2, R0r4 and R0r8 modes, R0 is shifted right by 1, 2, 4 or 8 real elements, respectively. The lower elements of R1 are shifted into the higher elements of R0. The R0rND1, R0rND2 and R0rND4 modes are special modes to rotate data for up-sampling by 2. In R0rND1, R0rND2 and R0rND4 modes, R0 is shifted right by 1, 2 or 4 real elements, respectively, according to fractional sample rate N/D. The lower elements of R1 are shifted into the higher elements of R0.

In R1r1, R1r2, R1r4 and R1r8 modes, R1 is shifted right by 1, 2, 4 or 8 real elements, respectively. The lower elements of R0 are shifted into the higher elements of R1. The R1rND1, R1rND2 and R1rND4 modes are special modes to rotate data for up-sampling by 2. In R1rND1, R1rND2 and R1rND4 modes, R1 is shifted right by 1, 2 or 4 real elements, respectively, according to fractional sample rate N/D. The lower elements of R0 are shifted into the higher elements of R1.

4.10.1.2 Right rotate/shift on R2 and R3

Rotate/shift operations performed on R2 and R3 are similar to those for R0 and R1. See descriptions in the preceding section, but with R0 replaced by R2, and R1 replaced by R3.

Descriptions

- Table 37 shows all possible <ror_mode> for R0 and R1, along with a description for each of these modes.
- Table 38 shows all possible <ror_mode> for R2 and R3, along with a description for each of these modes.
- Note that Table 37 and Table 38 are identical, except that the former describes rot modes that are performed on R0 and R1, while the latter describes rot modes that are performed on R2 and R3 - they are mirror of each other.
- R0, R1, R2 and R3 are assumed to be vectored buses, each containing 64 elements. Each element is 16 bits wide.
- In these Tables, "mse" denotes the index of the most significant element, which is mse=63.
- The amount of rotation for each element of the register in the fractional rotate or shift modes (R0R1rND1, R1rND2, and so on) is calculated using a fractional sample rate, N/D, derived from values programmed in the system control register (CREG) fields 16, 17 and 18, where N=value of field 16 and D=value of field 17.

Table 37. All Supported <ror_mode> - For R0 and R1

rot_mode	Descriptions
R0R1r1	Rotate right R0/R1 by 1 real elements. { R1[mse:0], R0[mse:0] } = { R0[0], R1[mse:0], R0[mse:1] }
R0R1r2	Rotate right R0/R1 by 2 real elements.

Table continues on the next page...

Table 37. All Supported <ror_mode> - For R0 and R1 (continued)

rot_mode	Descriptions
	{ R1[mse:0], R0[mse:0] } = { R0[1:0], R1[mse:0], R0[mse:2] }
R0R1r4	Rotate right R0/R1 by 4 real elements. { R1[mse:0], R0[mse:0] } = { R0[3:0], R1[mse:0], R0[mse:4] }
R0R1r8	Rotate right R0/R1 by 8 real elements. { R1[mse:0], R0[mse:0] } = { R0[7:0], R1[mse:0], R0[mse:8] }
R0R1rND1	Fractional rotate right R0/R1 by 1 real elements[1].
R0R1rND2	Fractional rotate right R0/R1 by 2 real elements[1].
R0R1rND4	Fractional rotate right R0/R1 by 4 real elements[1].
R0r1	Shift right R0 by 1 real elements. R0[mse:0] = { R1[0], R0[mse:1] }
R0r2	Shift right R0 by 2 real elements. R0[mse:0] = { R1[1:0], R0[mse:2] }
R0r4	Shift right R0 by 4 real elements. R0[mse:0] = { R1[3:0], R0[mse:4] }
R0r8	Shift right R0 by 8 real elements. R0[mse:0] = { R1[7:0], R0[mse:8] }
R0rND1	Fractional shift right R0 by 1 real elements[1].
R0rND2	Fractional shift right R0 by 2 real elements[1].
R0rND4	Fractional shift right R0 by 4 real elements[1].
R1r1	Shift right R1 by 1 real elements. R1[mse:0] = { R0[0], R1[mse:1] }
R1r2	Shift right R1 by 2 real elements. R1[mse:0] = { R0[1:0], R1[mse:2] }
R1r4	Shift right R1 by 4 real elements. R1[mse:0] = { R0[3:0], R1[mse:4] }
R1r8	Shift right R1 by 8 real elements. R1[mse:0] = { R0[7:0], R1[mse:8] }
R1rND1	Fractional shift right R1 by 1 real elements[1].

Table continues on the next page...

Table 37. All Supported <ror_mode> - For R0 and R1 (continued)

rot_mode	Descriptions
R1rND2	Fractional shift right R1 by 2 real elements[1].
R1rND4	Fractional shift right R1 by 4 real elements[1].

1. The amount of rotation for each element of the register is calculated using a fractional sample rate, N/D, derived from values programmed in system control registers 16, 17 and 18, where N=value of field 16 and D=value of field 17.

Table 38. All Supported <ror_mode> - For R2 and R3

rot_mode	Descriptions
R2R3r1	Rotate right R2/R3 by 1 real elements. { R3[mse:0], R2[mse:0] } = { R2[0], R3[mse:0], R2[mse:1] }
R2R3r2	Rotate right R2/R3 by 2 real elements. { R3[mse:0], R2[mse:0] } = { R2[1:0], R3[mse:0], R2[mse:2] }
R2R3r4	Rotate right R2/R3 by 4 real elements. { R3[mse:0], R2[mse:0] } = { R2[3:0], R3[mse:0], R2[mse:4] }
R2R3r8	Rotate right R2/R3 by 8 real elements. { R3[mse:0], R2[mse:0] } = { R2[7:0], R3[mse:0], R2[mse:8] }
R2R3rND1	Fractional rotate right R2/R3 by 1 real elements[1].
R2R3rND2	Fractional rotate right R2/R3 by 2 real elements[1].
R2R3rND4	Fractional rotate right R2/R3 by 4 real elements[1].
R2r1	Shift right R2 by 1 real elements. R2[mse:0] = { R3[0], R2[mse:1] }
R2r2	Shift right R2 by 2 real elements. R2[mse:0] = { R3[1:0], R2[mse:2] }
R2r4	Shift right R2 by 4 real elements. R2[mse:0] = { R3[3:0], R2[mse:4] }
R2r8	Shift right R2 by 8 real elements. R2[mse:0] = { R3[7:0], R2[mse:8] }
R2rND1	Fractional shift right R2 by 1 real elements[1].
R2rND2	Fractional shift right R2 by 2 real elements[1].
R2rND4	Fractional shift right R2 by 4 real elements[1].

Table continues on the next page...

Table 38. All Supported <ror_mode> - For R2 and R3 (continued)

rot_mode	Descriptions
R3r1	Shift right R3 by 1 real elements. R3[mse:0] = { R2[0], R3[mse:1] }
R3r2	Shift right R3 by 2 real elements. R3[mse:0] = { R2[1:0], R3[mse:2] }
R3r4	Shift right R3 by 4 real elements. R3[mse:0] = { R2[3:0], R3[mse:4] }
R3r8	Shift right R3 by 8 real elements. R3[mse:0] = { R2[7:0], R3[mse:8] }
R3rND1	Fractional shift right R3 by 1 real elements[1].
R3rND2	Fractional shift right R3 by 2 real elements[1].
R3rND4	Fractional shift right R3 by 4 real elements[1].

1. The amount of rotation for each element of the register is calculated using a fractional sample rate, N/D, derived from values programmed in system control registers 16, 17 and 18, where N=value of field 16 and D=value of field 17.

4.10.1.3 Left rotate/shift on R4 and R5

Table 39 shows all possible <rol_mode> for R4 and R5, along with a description for each of these modes.

Table 39. All Supported <rol_mode> - For R4 and R5

rot_mode	Descriptions
R4R5l1	Rotate left R4/R5 by 1 real elements. { R5[mse:0], R4[mse:0] } = { R5[mse-1:0], R4[mse:0], R5[mse] }
R4R5l2	Rotate left R4/R5 by 2 real elements. { R5[mse:0], R4[mse:0] } = { R5[mse-2:0], R4[mse:0], R5[mse:mse-1] }
R4R5l4	Rotate left R4/R5 by 4 real elements. { R5[mse:0], R4[mse:0] } = { R5[mse-4:0], R4[mse:0], R5[mse:mse-3] }
R4R5l8	Rotate left R4/R5 by 8 real elements. { R5[mse:0], R4[mse:0] } = { R5[mse-8:0], R4[mse:0], R5[mse:mse-7] }
R4l1	Rotate left R4 by 1 real elements. R4[mse:0] = { R4[mse-1:0], R5[mse] }
R4l2	Rotate left R4 by 2 real elements. R4[mse:0] = { R4[mse-2:0], R5[mse:mse-1] }

Table continues on the next page...

Table 39. All Supported <rol_mode> - For R4 and R5 (continued)

rot_mode	Descriptions
R4l4	Rotate left R4 by 4 real elements. R4[mse:0] = { R4[mse-4:0], R5[mse:mse-3] }
R4l8	Rotate left R4 by 8 real elements. R4[mse:0] = { R4[mse-8:0], R5[mse:mse-7] }
R5l1	Rotate left R5 by 1 real elements. R5[mse:0] = { R5[mse:1], R4[mse] }
R5l2	Rotate left R5 by 2 real elements. R5[mse:0] = { R5[mse:2], R4[mse:mse-1] }
R5l4	Rotate left R5 by 4 real elements. R5[mse:0] = { R5[mse:4], R4[mse:mse-3] }
R5l8	Rotate left R5 by 8 real elements. R5[mse:0] = { R5[mse:8], R4[mse:mse-7] }

4.10.1.4 Left rotate/shift on R6 and R7

Table 40 shows all possible <rol_mode> for R6 and R7, along with a description for each of these modes.

Table 40. All Supported <rol_mode> - For R6 and R7

rot_mode	Descriptions
R6R7l1	Rotate left R6/R7 by 1 real elements. { R7[mse:0], R6[mse:0] } = { R7[mse-1:0], R6[mse:0], R7[mse] }
R6R7l2	Rotate left R6/R7 by 2 real elements. { R7[mse:0], R6[mse:0] } = { R7[mse-2:0], R6[mse:0], R7[mse:mse-1] }
R6R7l4	Rotate left R6/R7 by 4 real elements. { R7[mse:0], R6[mse:0] } = { R7[mse-4:0], R6[mse:0], R7[mse:mse-3] }
R6R7l8	Rotate left R6/R7 by 8 real elements. { R7[mse:0], R6[mse:0] } = { R7[mse-8:0], R6[mse:0], R7[mse:mse-7] }
R6l1	Rotate left R6 by 1 real elements. R6[mse:0] = { R6[mse-1:0], R7[mse] }
R6l2	Rotate left R6 by 2 real elements. R6[mse:0] = { R6[mse-2:0], R7[mse:mse-1] }

Table continues on the next page...

Table 40. All Supported <rol_mode> - For R6 and R7 (continued)

rot_mode	Descriptions
R6l4	Rotate left R6 by 4 real elements. R6[mse:0] = { R6[mse-4:0], R7[mse:mse-3] }
R6l8	Rotate left R6 by 8 real elements. R6[mse:0] = { R6[mse-8:0], R7[mse:mse-7] }
R7l1	Rotate left R7 by 1 real elements. R7[mse:0] = { R7[mse:1], R6[mse] }
R7l2	Rotate left R7 by 2 real elements. R7[mse:0] = { R7[mse:2], R6[mse:mse-1] }
R7l4	Rotate left R7 by 4 real elements. R7[mse:0] = { R7[mse:4], R6[mse:mse-3] }
R7l8	Rotate left R7 by 8 real elements. R7[mse:0] = { R7[mse:8], R6[mse:mse-7] }

4.10.1.5 Rotate-register modes examples

Following figure shows examples of rotate right by real elements:

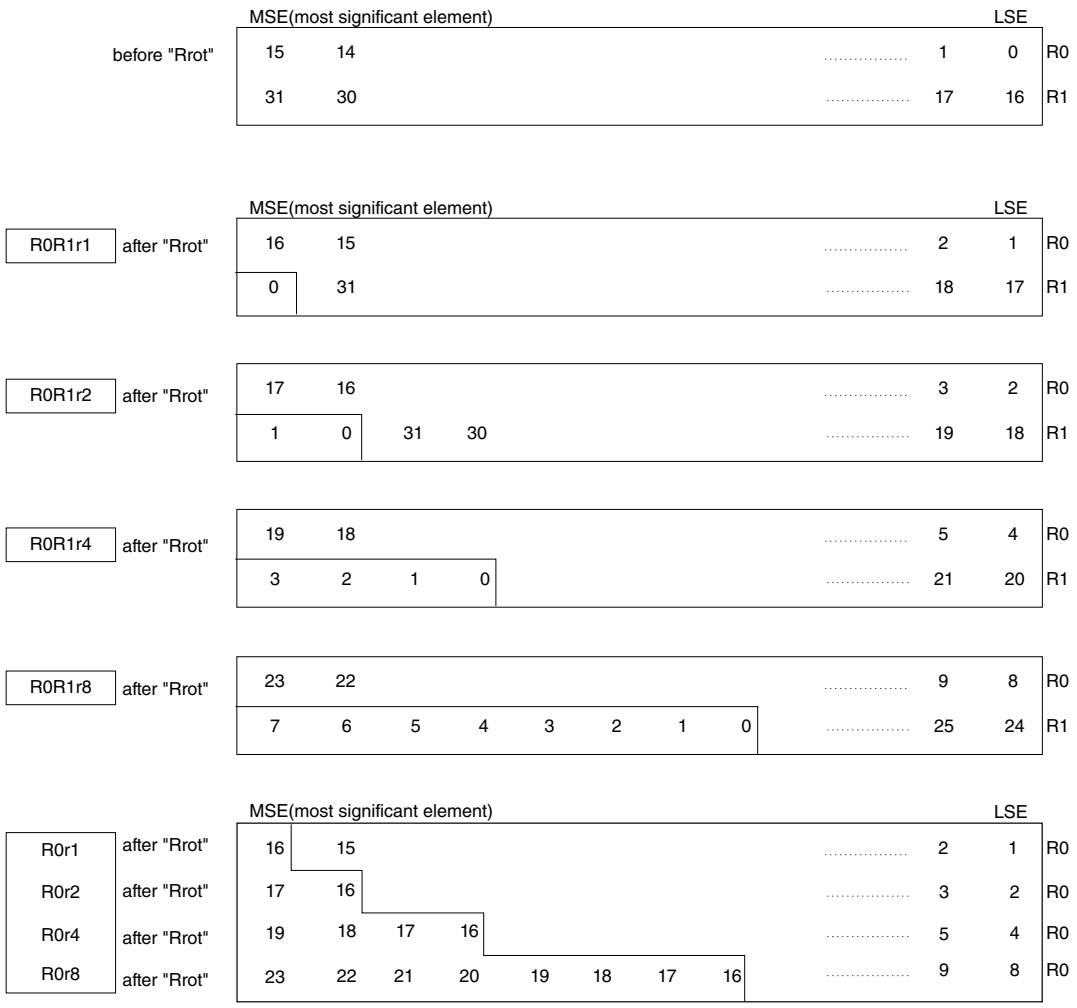


Figure 18. Rotate right by real elements examples

The following example is given as sample code for function that filters the complex, 16-bit fixed-point data, previously re-sampled by 3/2 (that is interpolated by 3 and then decimated by 2) with a filter described by real, 16-bit fixed-point coefficients, to create the complex, 16-bit fixed-point filtered data.

```
.global _firFilterUp3Dn2
.type _firFilterUp3Dn2, @function
.section .text
_firFilterUp3Dn2:
set.creg 16, 3;                                          // Interpolate by 3.
set.creg 17, 2;                                          // Decimate by 2.
    // a0:  pt to output
    // a2:  pt to filter taps
    // a3: pt to input data
    // g0: data block loop
    // g1: taps loop
    mv a4, a2;

ld.laddr [a3]+1;

clr.VRA ;
ld.laddr [a3]+0;
```

```
set.creg 18, 0;
ld.normal R0;                                                    // Initial phase 0.

set.loop g0, L01, L02;
ld.normal R1;

set.VRAincr rS1,NUM_AU*2;
ld.laddr [a2]+1;                                                 // s1: half dmem line of
taps

setA.VRApg rS1, 0, 1;
set.prec half_fixed, half_fixed, half_fixed, single, half_fixed;

ld.laddr [a2]+1;
set.Smode S0straight, S1r2c, S2zeros;

set.VRArange1 rS1, NUM_AU*4*2, NUM_AU*4*2+NUM_AU*2;
ld.normal R2;
L01:set.Rrot R0R1rND2;
ld.laddr [a2]+1;
rot;
rd S2;
rd S1;
rd S0;
loop_begin;

ld.laddr [a2]+1;
ld.normal R2;
rot;
rd S1;
rd S0;
// as4:  number of filter taps per phase /2 - 3

set.loop g1, 2;
rot;
rd S1;
rd S0;
rmad;

ld.normal R2;
rot;
rd S1;
rd S0;
rmac;

ld.laddr [a2]+1;
rot;
rd S1;
rd S0;
rmac;
loop_begin;

ld.normal R2;
rot;
rd S1;
rd S0;
rmac;
loop_end;

ld.laddr [a3]+1;
rot;
rd S1;
rd S0;
rmac;

ld.laddr [a3]+0;
ld.normal R2;
rd S1;
rd S0;
```

```
rmac;

set.creg 18,
2;
rmac;

ld.laddr [a2]+1;
ld.normal R0;
rmac;

set.loop g1, 2;
ld.normal R1;

ld.laddr [a2]+1;
rot;
rd S2;
rd S1;
rd S0;

setB.VRAptr rSt, 4;
ld.normal R2;
rot;
rd S1;
rd S0;

ld.laddr [a2]+1;
rot;
rd S1;
rd S0;
rmad;
wr.straight;

st.laddr [a0]+1;
ld.normal R2;
rot;
rd S1;
rd S0;
rmac;

ld.laddr [a2]+1;
rot;
rd S1;
rd S0;
rmac;
loop_begin;

ld.normal R2;
rot;
rd S1;
rd S0;
rmac;
loop_end;

ld.laddr [a3]+1;
rot;
rd S1;
rd S0;
rmac;

ld.laddr [a3]+0;
ld.normal R2;
rd S1;
rd S0;
rmac;

set.creg 18,
1;
rmac;
```

```
ld.laddr [a2]+1;
ld.normal R0;
rmac;

set.loop g1, 2;
ld.normal R1;

ld.laddr [a2]+1;
rot;
rd S2;
rd S1;
rd S0;


st.laddr [a0]+1;
ld.normal R2;
rot;
rd S1;
rd S0;
rmac;

ld.laddr [a2]+1;
rot;
rd S1;
rd S0;
rmac;
loop_begin;

ld.normal R2;
rot;
rd S1;
rd S0;
rmac;
loop_end;

ld.laddr [a3]+1;
rot;
rd S1;
rd S0;
rmac;

ld.laddr [a3]+0;
rd S1;
rd S0;
rmac;

mvB a2,
a4;
rmac;                    // reset a2 to the top of taps

ld.laddr [a2]+1;
ld.normal R0;
rmac;                    // ld taps from the beginning

set.creg 18, 0;
ld.normal R1;

nop;

ld.normal R2;

ld.laddr [a2]+1;
wr.straight;
L02: st.laddr
[a0]+1;
loop_end;
rts;
nop;
nop;
```

```
.size    _firFilterUp3Dn2, .-_firFilterUp3Dn2
//end
```

The following example is given as a function that filters the complex, 16-bit fixed-point data, previously resampled by 8/5 (interpolated by 8 then decimated by 5) with a filter described by real, 16-bit fixed-point coefficients, to create the complex, 16-bit fixed-point filtered data. The number of filter coefficients per phase >= 8 and has to be even taps.

```
 #include "frcUp8Dn5_constants.h"

.global _frcFilterUp8Dn5
.type _frcFilterUp8Dn5, @function
.section .text
_frcFilterUp8Dn5:
// ---------------------------------------------------------------
// Initialize VSP to operate in real arithmetic mode, using the // full data path and normal
AU output:
// ---------------------------------------------------------------
    set.creg 13, 0;
// Disable 2x scaling on fixed-point.
// ---------------------------------------------------------------
// Set the fractional interpolation rotation mode parameters:
// Interpolation = 8, Decimation = 5, Phase = 0.
// ---------------------------------------------------------------
    set.creg 16, 8;            // Interpolate by 8.
    set.creg 17, 5;            // Decimate by 5.
    set.creg 18, 0;            // Initial phase 0.
// ---------------------------------------------------------------
// Set the input and output precision mode to 16-bit fixed-point
// (only AU computations done in 32-bit floating-point):
// ---------------------------------------------------------------------------
    set.prec half_fixed, half_fixed, half_fixed, single, half_fixed;
// ---------------------------------------------------------------
// Reset VRA pointers, increments and modulos:
// ---------------------------------------------------------------
    clr.VRA ;
// ---------------------------------------------------------------------------
// Initialize VRA pages and pointers:
// rS0 = R0[0] = input     : page 0, offset 0.
// rS1 = R2[0] = coeffs    : page 1, offset 0.
// rS2 = don't care        : page 0, offset 0.
// rVd = R3[0] = output    : page 1, offset NUM_AU * 4 (256 for 64-AU).
// rSt = R3[0] = output    : 3.
// ---------------------------------------------------------------------------
    set.VRAptr 0, 64*2, 0, 64*3, 3;
// ---------------------------------------------------------------------------
// Initialize VRA pointer increments:
// rS1 += NUM_AU*2 (128 for 64-AU) to go from 1/2-vector to 1/2-vector.
// Limit rS1 to R2lo and R2hi.
// ---------------------------------------------------------------------------
    set.VRAincr 0, +16*2, 0, 0, 0;
    set.VRArange1 rS1, 16*4*2, 16*4*2+16*2;
// ---------------------------------------------------------------------------
// Set VRA source modes:
// Load S0 with a vector of 16-bit fixed-point complex numbers.
// Load S1 with 1/2-vector of 16-bit fixed-point real numbers and expand
// them to complex.
// Load S2 with zeroes.
// ---------------------------------------------------------------------------
    set.Smode S0straight, S1r2c, S2zeros;
// ---------------------------------------------------------------
// Set filter coefficients pointer range:
// ---------------------------------------------------------------
    mv a4, _frcUp8Dn5_filterCoeff;
    set.range a2, a4, FRCUP8DN5_COEFSIZE;
// ---------------------------------------------------------------
```

```
// Move output size to general purpose registers
// to later set loop iteration count:
// ------------------------------------------------------------
    mv a5, FRCUP8DN5_VCPU_LOOPITER;           // a5: Number of output vectors.
    mv a6, FRCUP8DN5_TAPITER;                 // a6: Iteration count for #Taps
// ------------------------------------------------------------
// Initialize memory address registers,
// start loading input data from memory,
// set rotation mode to fractional interpolation mode:
// ------------------------------------------------------------
    ld [a1]+FRCUP8DN5_HISTPTROFF;
mvB a3, as1;

 // dummy load to adjust pointer to history samples
ld [a3]+FRCUP8DN5_HISTPTROFF;
 // dummy load to input pointer, to perform l2h_h2l operation
ld [a1]-FRCUP8DN5_HISTPTROFF;
mvB a0, as0;
// Start fetching data for output vector phase 0.
    ld.laddr [a3]+1;
mvB a2, a4;
    ld.laddr [a3]-1;
set.Rrot R0R1rND2;
// Fractional interpolation rotation for 16-bit complex data.
//------------------------------------------------------------
// Load coefficients from memory and load register file:
// ------------------------------------------------------------
ld.laddr [a2]+1;
ld.h2l R0;
// Start loading register file with data phase 0.

set.loop a5, L01, L02;
ld.l2h_h2l R0;
ld.laddr [a2]+1;
ld.l2h R1;
ld.normal R2;
L01: ld.laddr [a2]+1;
rd S0;
rd S1;
rd S2;
rot;
loop_begin;

set.loop a6, 2;
ld.normal R2;
rd S0;
rd S1;
rot;
ld.laddr [a2]+1;
rd S0;
rd S1;
rmad;
rot;
ld.normal R2;
rd S0;
rd S1;
rmac;
rot;
loop_begin;
ld.laddr [a2]+1;
rd S0;
rd S1;
rmac;
rot;
loop_end;

ld.laddr [a3]+1;
ld.normal R2;
rd S0;
```

```
rd S1;
rmac;
rot;

ld.laddr [a3]+1;
rd S0;
rd S1;
rmac;
rot;

ld.laddr [a3]-1;
ld.normal R2;
rd S0;
rd S1;
rmac;

ld.h2l R0;
rmac;
ld.laddr [a2]+1;
ld.l2h_h2l R0;
rmac;

ld.l2h R1;

ld.laddr [a2]+1;
rd S0;
rd S1;
rd S2;
rot;

set.loop a6;
ld.normal R2;
rd S0;
rd S1;
rot;

ld.laddr [a2]+1;
rd S0;
rd S1;
rmad;
rot;
wr.straight;              //Write Phase 0 output

ld.normal R2;
rd S0;
rd S1;
rmac;
rot;
loop_begin;
ld.laddr [a2]+1;
rd S0;
rd S1;
rmac;
rot;
loop_end;
    ld.laddr [a3]+1;
ld.normal R2;
rd S0;
rd S1;
rmac;
rot;
ld.laddr [a3]+1;
rd S0;
rd S1;
rmac;
rot;
ld.laddr [a3]-1;
ld.normal R2;
rd S0;
```

```
rd S1;
rmac;
st.laddr [a0]+1;
ld.h2l R0;
rmac;
//Store Phase 0 output
 ld.laddr [a2]+1;
ld.l2h_h2l R0;
rmac;

ld.l2h R1;
ld.laddr [a2]+1;
rd S0;
rd S1;
rd S2;
rot;

set.loop a6;
ld.normal R2;
rd S0;
rd S1;
rot;
ld.laddr [a2]+1;
rd S0;
rd S1;
rmad;
rot;
wr.straight;              //Write Phase 1 output

ld.normal R2;
rd S0;
rd S1;
rmac;
rot;
loop_begin;
ld.laddr [a2]+1;
rd S0;
rd S1;
rmac;
rot;
loop_end;
ld.laddr [a3]+1;
ld.normal R2;
rd S0;
rd S1;
rmac;
rot;
ld.laddr [a3]+1;
rd S0;
rd S1;
rmac;
rot;
ld.laddr [a3]-1;
ld.normal R2;
rd S0;
rd S1;
rmac;
st.laddr [a0]+1;
ld.h2l R0;
rmac;
//Store Phase 1 output
ld.laddr [a2]+1;
ld.l2h_h2l R0;
rmac;

ld.l2h R1;
ld.laddr [a2]+1;
rd S0;
rd S1;
```

```
rd S2;
rot;

set.loop a6;
ld.normal R2;
rd S0;
rd S1;
rot;
ld.laddr [a2]+1;
rd S0;
rd S1;
rmad;
rot;
wr.straight;              //Write Phase 2 output

ld.normal R2;
rd S0;
rd S1;
rmac;
rot;
loop_begin;
ld.laddr [a2]+1;
rd S0;
rd S1;
rmac;
rot;
loop_end;

ld.laddr [a3]+1;
ld.normal R2;
rd S0;
rd S1;
rmac;
rot;
ld.laddr [a3]+1;
rd S0;
rd S1;
rmac;
rot;
ld.laddr [a3]-1;
ld.normal R2;
rd S0;
rd S1;
rmac;
st.laddr [a0]+1;
ld.h2l R0;
rmac;
//Store Phase 2 output
ld.laddr [a2]+1;
ld.l2h_h2l R0;
rmac;

ld.l2h R1;

ld.laddr [a2]+1;
rd S0;
rd S1;
rd S2;
rot;

set.loop a6;
ld.normal R2;
rd S0;
rd S1;
rot;

ld.laddr [a2]+1;
rd S0;
rd S1;
```

```
rmad;
rot;
wr.straight;              //Write Phase 3 output

ld.normal R2;
rd S0;
rd S1;
rmac;
rot;
loop_begin;

ld.laddr [a2]+1;
rd S0;
rd S1;
rmac;
rot;
loop_end;
ld.laddr [a3]+1;
ld.normal R2;
rd S0;
rd S1;
rmac;
rot;

ld.laddr [a3]+1;
rd S0;
rd S1;
rmac;
rot;

ld.laddr [a3]-1;
ld.normal R2;
rd S0;
rd S1;
rmac;

st.laddr [a0]+1;
ld.h2l R0;
rmac;
//Store Phase 3 output
ld.laddr [a2]+1;
ld.l2h_h2l R0;
rmac;

ld.l2h R1;

ld.laddr [a2]+1;
rd S0;
rd S1;
rd S2;
rot;

set.loop a6;
ld.normal R2;
rd S0;
rd S1;
rot;
ld.laddr [a2]+1;
rd S0;
rd S1;
rmad;
rot;
wr.straight;              //Write Phase 4 output

ld.normal R2;
rd S0;
rd S1;
rmac;
rot;
```

```
loop_begin;

ld.laddr [a2]+1;
rd S0;
rd S1;
rmac;
rot;
loop_end;

ld.laddr [a3]+1;
ld.normal R2;
rd S0;
rd S1;
rmac;
rot;

ld.laddr [a3]+1;
rd S0;
rd S1;
rmac;
rot;

ld.laddr [a3]-1;
ld.normal R2;
rd S0;
rd S1;
rmac;

st.laddr [a0]+1;
ld.h2l R0;
rmac;
//Store Phase 4 output
 ld.laddr [a2]+1;
ld.l2h_h2l R0;
rmac;

ld.l2h R1;

ld.laddr [a2]+1;
rd S0;
rd S1;
rd S2;
rot;

set.loop a6;
ld.normal R2;
rd S0;
rd S1;
rot;

ld.laddr [a2]+1;
rd S0;
rd S1;
rmad;
rot;
wr.straight;              //Write Phase 5 output

ld.normal R2;
rd S0;
rd S1;
rmac;
rot;
loop_begin;

ld.laddr [a2]+1;
rd S0;
rd S1;
rmac;
rot;
```

```
loop_end;
ld.laddr [a3]+1;
ld.normal R2;
rd S0;
rd S1;
rmac;
rot;

ld.laddr [a3]+1;
rd S0;
rd S1;
rmac;
rot;
ld.laddr [a3]-1;
ld.normal R2;
rd S0;
rd S1;
rmac;
st.laddr [a0]+1;
mvB a7, a3;
ld.h2l R0;
rmac;                                   //Store Phase 5 output
ld.laddr [a2]+1;
ld.l2h_h2l R0;
rmac;

ld.l2h R1;
ld.laddr [a2]+1;
rd S0;
rd S1;
rd S2;
rot;

set.loop a6;
ld.normal R2;
rd S0;
rd S1;
rot;
ld.laddr [a2]+1;
rd S0;
rd S1;
rmad;
rot;
wr.straight;            //Write Phase 6 output

ld.normal R2;
rd S0;
rd S1;
rmac;
rot;
loop_begin;
ld.laddr [a2]+1;
rd S0;
rd S1;
rmac;
rot;
loop_end;
ld.laddr [a3]+1;
ld.normal R2;
rd S0;
rd S1;
rmac;
rot;

ld.laddr [a3]+1;
rd S0;
rd S1;
rmac;
rot;
```

```
ld.laddr [a3]-1;
ld.normal R2;
rd S0;
rd S1;
rmac;

st.laddr [a0]+1;
ld.h2l R0;
rmac;                                        //Store Phase 6 output

ld.laddr [a2]+1;
ld.l2h_h2l R0;
rmac;

ld.l2h R1;

nop;

nop;
wr.straight;              //Write Phase 7 output
L02: st.laddr [a0]+1;
mvB a1, a7;
loop_end;               //Store Phase 7 output

ld.laddr [a1]+0;
mvB a1, a2;           //Grab history samples to update the history buffer

mv a6, 0;
rts;
set.range a2, a6, 0;

ld.normal R3;

st.laddr [a1]+0;           //Updating the history buffer
.size _frcFilterUp8Dn5, .-_frcFilterUp8Dn5
```

4.11 Extrema instructions

The extrema instructions configure and run the extrema search engine. The extrema search engine finds an extremum point, maximum or minimum, in *N* half-word elements. The extrema engine can return either the resulting element index or the element value depending on the result mode (index,value). If the result mode is index, element index is returned into a general purpose register, an address register or both. If the result mode is value, the value will be returned into a general purpose register and the index will be returned into an address register.

The data flow is illustrated in Figure 19.

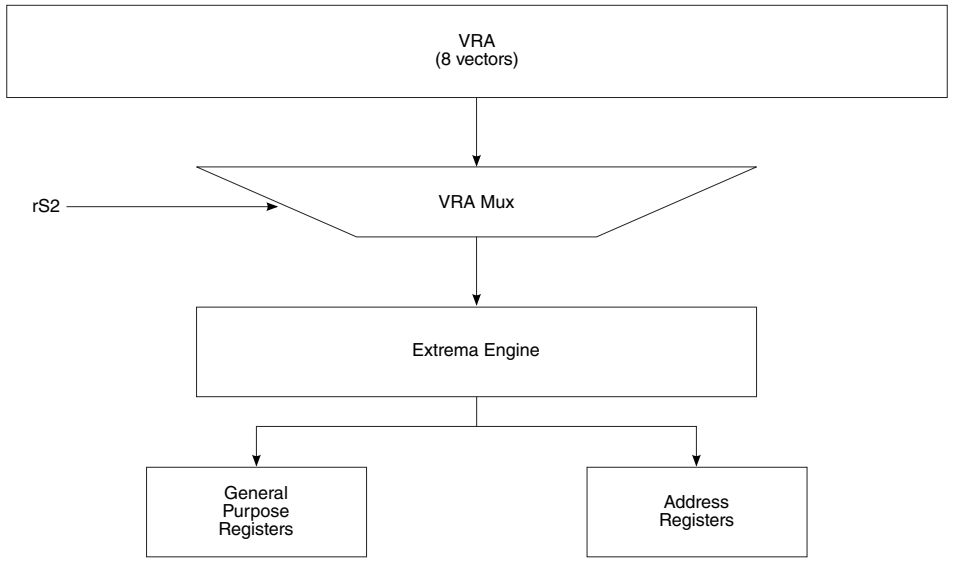


Figure 19. Extrema Engine Data Path

Table 41. Extrema Instructions

Functionality	Instruction Examples	Family
Configure Extrema Engine	set.xtrm {signed,unsigned}, {max,min}, {even,all}, {value,index}, N	OpB
Run Extrema Engine	xtrm a*X*, g*Y*	OpB
	xtrm a*X*	
	xtrm g*Y*	

4.11.1 Extrema configuration

The extrema engine can be configured in many different modes using the *set.xtrm* instruction. The engine will use the last configuration set by the user before running the *xtrm* instruction. If the *set.xtrm* instruction is called while the extrema engine is running, none of the configuration is modified. The engine uses the S2 precision settings.

4.11.1.1 Extrema

The extrema engine can be configured with four different extrema: signed max, signed min, unsigned max and unsigned min. The signed max will return the value closest to positive infinity. The signed min will return the value closest to negative infinity. The unsigned max will return the value farthest from zero. The unsigned min will return the value closest to zero.

4.11.1.2 Extrema modes

The extrema engine can be configured in two different modes: all and even. The all mode will find an extremum within all N elements. The even mode will find an extremum within the even elements of N elements. The resulting index in even mode represents the Xth even index in N elements. For example, if the resulting index in even mode is 8, it is actually the 16th element in N elements.

4.11.1.3 Extrema element count

The extrema engine element count, N, is the number of half-words that will be processed by the engine during a search. N can be any power of 2 up to one half line (32) or a multiple of half lines up to 2048. When in even mode, the minimum N is 4. The extrema engine compares up to B elements at one time. The value of B is 32 for elements in half precision and 16 for single precision. If N is greater than B, the extrema engine will update rS2 in order to compare the next B elements. To do so, the rS2 must be configured to increment to the next B elements to compare.

For example:

- To search for a signed max within 512 half precision elements, the following configuration can be used:

```
set.prec half, half, half, single, single; set.vraptr rS2, 0;
set.xtrm signed, max, all, value, 512; set.vraincr rS2, 32;
```

- To search for a signed max within 512 single precision elements, the following configuration can be used:

```
set.prec single, single, single, single, single; set.vraptr rS2, 0;
set.xtrm signed, max, all, value, 2*512; set.vraincr rS2, 2*16;
```

4.11.1.4 Extrema result mode

The extrema engine has two different result modes: index and value. The index mode will return the index of the extreme value into a general purpose register, an address pointer or both. The value mode will return the extreme value into a general purpose register, the index of the extreme value into an address pointer or both.

4.11.1.5 Extrema latency

The *xtrm* instruction is multi cycle and it depends on the number of elements N and the all/even set by the most recently executed set.xtrm instruction. The latency of xtrm in given by the formula:

2+ceil(M/B)+log2[min(M,B)]

Where M=N in ‘all’ mode and M=N>>1 in ‘even’ mode; B = 32 for elements in half precision and B = 16 for single precision.

NOTE

A 'done' instruction should not be executed until the xtrm instruction is complete.

4.11.2 Extrema Functionality

The extrema engine returns the index or the value of an extremum within a given range of N elements in the VRA. The engine uses the S2 precision settings and searches for extrema in groups of B elements (B = 32 in half precision, B=16 in single precision). The engine uses rS2 as the pointer to the elements to search within the VRA. The result can be written into a general purpose register, address register, or both.

4.11.2.1 RAG configuration

The extrema engine uses rS2 as the pointer into the VRA for the elements to compare. The rS2 pointer must be configured on a B element boundary where B = 32 for elements in half precision and B = 16 for single precision. When N is larger than B, the engine will update rS2 N/B -1 times to search all N elements. The rS2 increment register must be configured properly to increment to the next B elements in the VRA. If rS2 is not on a B-element boundary, the engine will use the elements starting at the closest B-element boundary. The rS2 pointer & increment registers and the S2 register cannot be used by any other instruction for ceil(N/B) cycles following the issue of the "xtrm" instruction.

4.11.2.2 Latency

The *set.xtrm* instruction has a 1 cycle latency. The *xtrm* instruction has variable latency depending on *N* number of elements to search. See Extrema latency for latency calculation.

After the *xtrm* latency, the index of the extrema value will reside in the desired general purpose or address register.

The next *set.xtrm* or *xtrm* instruction can occur during the last cycle of latency of the *xtrm* instruction. This is possible because the extrema engine is writing the result to the output register.

4.11.2.3 Multiple extrema

If multiple extrema exist in N elements, the extrema engine will return one of the extrema indexes. Depending on N and the location of the multiple extrema, the very first extremum is not guaranteed to be the returned extremum index.

4.11.3 Extrema instructions code example

```
set.prec single, single, single, single, single; set.vraptr rS2, 0;
set.xtrm signed, max, all, value, 512*2; set.vraincr rS2, NAU*2;

ld.laddr [g0]+1;
   ld.laddr [g0]+1;
   ld.laddr [g0]+1;
   ld.laddr [g0]+1; ld R0; xtrm a10, g10;  // xtrm operand:
   ld.laddr [g0]+1; ld R1;                 // R0
   ld.laddr [g0]+1; ld R2;                 // R0
   ld.laddr [g0]+1; ld R3;                 // R1
   ld.laddr [g0]+1; ld R4;                 // R1
                  ` ld R5;                 // R2
                    ld R6;                 // R2
                    ld R7;                 // R3
   nop;                                    // R3
   nop;                                    // R4
   nop;                                    // R4
   nop;                                    // R5
   nop;                                    // R5
   ld.laddr [g0]+1;                        // R6
   ld.laddr [g0]+1;                        // R6
   ld.laddr [g0]+1;                        // R7
   ld.laddr [g0]+1; ld R0;                 // R7
   ld.laddr [g0]+1; ld R1;                 // R0
   ld.laddr [g0]+1; ld R2;                 // R0
   ld.laddr [g0]+1; ld R3;                 // R1
   ld.laddr [g0]+1; ld R4;                 // R1
                    ld R5;                 // R2
                    ld R6;                 // R2
                    ld R7;                 // R3
   nop;                                    // R3
   nop;                                    // R4
   nop;                                    // R4
   nop;                                    // R5
   nop;                                    // R5
   nop;                                    // R6
   nop;                                    // R6
   nop;                                    // R7
   nop;                                    // R7
   nop;
   nop;
   nop;
   nop;
   nop;
```

4.12 Vector AU source register instructions

The Vector AU source register instructions perform the following operations:

- read data from the VRA; up to one vector each clock cycle per destination source register,
- data permutation and/or replication for maximum utilization of the VAU,
- optional data type conversion (that is: half-fixed to single) and
- update the S0, S1 and S2 VAU source registers.

The data permutation and/or replication are controlled by S0mode, S1mode and S2mode for each of the three VAU source register data paths, respectively. Data type conversion is controlled using S0prec, S1prec and S2prec. The data flow is illustrated in Figure 20.

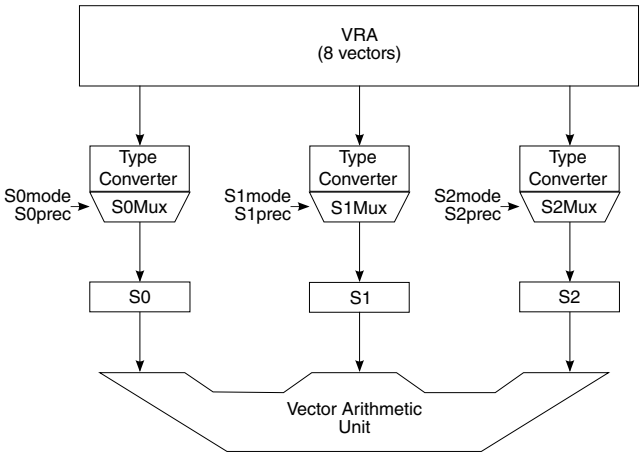


Figure 20. VRA to Source Register Data Path

One VRA pointer is allocated to each source register port to indicate which data to extract. The pointer is post-modified upon execution of the read operation. Instruction syntax and functionality is summarized in Table 42

Table 42. Source Operand Load Instructions

Functionality	Instruction Examples	Family	Update VRA Pointer
Data permutation and/or replication	set.Smode S0mode;	OpB	No
	set.Smode S1mode;		
	set.Smode S2mode;		
	set.Smode S0mode, S1mode;		
	set.Smode S0mode, S2mode;		
	set.Smode S1mode, S2mode;		
	set.Smode S0mode, S1mode, S2mode;		
	set.Smode S0chs, S0mode;		
	set.Smode S0chs, S0mode, S1mode;		
	set.Smode S0chs, S0mode, S2mode;		
	set.Smode S0chs, S0mode, S1mode, S2mode;		
	set.Smode S0conj, S0mode;		

Table continues on the next page...

Table 42. Source Operand Load Instructions (continued)

Functionality	Instruction Examples	Family	Update VRA Pointer
	set.Smode S0conj, S0mode, S1mode;		
	set.Smode S0conj, S0mode, S2mode;		
	set.Smode S0conj, S0mode, S1mode, S2mode;		
	set.Smode S0conj, S0chs, S0mode;		
	set.Smode S0conj, S0chs, S0mode, S1mode;		
	set.Smode S0conj, S0chs, S0mode, S2mode;		
	set.Smode S0conj, S0chs, S0mode, S1mode, S2mode;		
Data type conversion	set.prec S0prec,S1prec,S2prec,AUprec,Vprec	OpB	No
VRA data read	rd S0;	OpVs0	rS0
	rd S1;	OpVs1	rS1
	rd S2;	OpVs2	rS2
	rd S0; rd S1; rd S2;	OpVs0 + OpVs1 + OpVs2	rS0, rS1 & rS2
Combination of data permutation and/or replication and VRA data read[1]	set.Smode S0<S0mode>; rd S0;	OpB + OpVs	rS0
	set.Smode S1<S1mode>; rd S1;	OpB + OpVs	rS1
	set.Smode S2<S2mode>; rd S2;	OpB + OpVs	rS2

1. These instructions are of type Format-1 macro-instruction.

4.12.1 VRA data reads

Data is read from the VRA using the 'rd ...' instruction. Each source register port is assigned its own microinstruction family. Therefore, any single, pair, or all three paths can read data in a given cycle.

4.12.1.1 Pipeline delay

The 'rd' instructions have a two cycle pipeline delay so the destination VAU source register (S0, S1, or S2) cannot be used in the next cycle. However, the delay is pipelined so its possible to feed the source registers with new data every cycle. Consider the following example:

rd S0; rd S1; rd S2; // 1st read

rd S0; rd S1; rd S2; // 2nd read

rmac/cmac; // uses data from 1st read

rmac/cmac; // uses data from 2nd read

4.12.1.2 VRA pointers

Each VAU source register port in the VRA is assigned a pointer; rS0, rS1, and rS2 for S0, S1, and S2, respectively. The pointer contains the beginning address of a vector or element in the VRA which will be accessed with the next 'rd' instruction. The pointer is always post-modified according to the pointer mode after executing the read operation.

The RAG pointer Registers are 11 bits. The entire RAG address or only a portion of this is used depending on the number of AU's. For a 16AU design 9 bits are used. The most significant 3 bits of the VRA pointer identify the R register. The remaining bits indicate the column or element offset within the R register.

In the following, we will take rS0 as an example. Similar rules apply to rS1 and rS2.

- rS0[8:6], or the upper 3 bits of rS0, specify the VRA register to read from;
- rS0[5:0], or the lower bits of rS0, specify the element offset within the VRA register.

When loading the entire line from VRA, only the upper 3 bits of the *rag* register are used. The lower bits are ignored.

When loading a 16 bit data element from VRA, the lower bits are used, as follows.

- When accessing an element in real mode, all lower bits of rS0 (or rS0[5:0]) will be used.
- When accessing an element in complex mode, only rS0[5:1] will be used. The rS0[0] will be ignored.

When loading a 32 bit data element from VRA, the lower bits are used, as follows.

- When accessing an element in real mode, only rS0[5:1] will be used. The rS0[0] will be ignored.
- When accessing an element in complex mode, only rS0[5:2] will be used. The rS0[1:0] will be ignored.

When loading a 64bit data element from VRA, the lower bits are used, as follows.

- When accessing an element in real mode, only rS0[5:2] will be used. The rS0[1:0] will be ignored.
- When accessing an element in complex mode, only rS0[5:3] will be used. The rS0[2:0] will be ignored.

4.12.1.3 RAG pointer registers updates

The RAG pointer is updated each time the corresponding source mux is loaded. That is, RAG pointer is updated each time when a "Load Source Only" or "Load Source and Set Source Loading Mode" instruction is executed (see Table 42).

For example, rS2 is updated whenever one of the following two instructions is executed.

- rd S2;
- rd S2; set.Smode S2<s2mode>;

set.Smode S2<s2mode>, on the other hand, will not cause rS2 to be updated.

It is assumed that the user programmed the RAG pointer updates correctly for each Sxprec type.

If the Sxprec type is half fixed or half float, the RAG pointer should update by 1 in real mode and by 2 in complex mode.

If the Sxprec type is single precision, the RAG pointer should update by 2 in real mode and by 4 in complex mode.

See also RAG instructions for more details on RAG pointers update algorithm.

4.12.2 Data permutation and/or replication

Data permutation and/or replication is controlled using the *set.Smode* instruction. The S0 path allows for optional conjugation and/or sign inversion (change-sign). The ISA supports configuration of any individual source register path, any two, or all three in a single OpB microinstruction. These operations do not affect the VRA pointers or the source registers.

There are three groups of Source Operand Load Instructions detailed in Table 42:

- set source loading modes,
- load source only, and
- load source and set source loading modes

The first group of instructions are opB instructions. These instructions can also be combined in one instruction `set.Smode S0<s0mode>, S1<S1mode>, S2<s2mode>;`

The second group of instructions are opVs instructions.

The third group of instructions are a combination of opB and opVs instructions. This group of instructions, `set.Smode S0<s0mode>; rd S0;`, updates the source mode registers, the source registers and the corresponding RAG pointer.

The notations <s0mode>, <S1mode> and <s2mode> used in Table 42 denote the muxing modes for S0Mux, S1Mux and S2Mux, respectively. These source muxing modes are stored in s0_mode_reg, s1_mode_reg and s2_mode_reg, respectively.

Once s0_mode_reg is set using the `set.Smode S0<s0mode>` instruction, it will remain unchanged until the next `set.Smode S0<s0mode>` instruction is executed. The same can be said about the behaviors of <S1mode> and <s2mode>.

If an instruction updates only S0 but not s0_mode_reg, then the old muxing mode for S0Mux set by the previous update to s0_mode_reg will be used. That is, once s0_mode_reg is set by an instruction, it will retain its value until the next instruction that explicitly updates the register is executed. The same can be said about s1_mode_reg and s2_mode_reg.

For example, in the following code sequence, the first instruction sets s1_mode_reg to normal mode and then loads S1 using normal mode. The second instruction loads S1 using normal mode, as well.

```
rd S1; set.Smode S1hlinecplx;
rd S1;
```

4.12.3 VRA data type conversion

The data read from the VRA can be type converted before being loaded into the source register. The S0prec, S1prec and S2prec determine the data type stored in the VRA. The AUprec determines the final data type stored in the source register.

The diagrams that follow show bit width sizes. Each size represents the number of bits that are used before type conversion and the resulting number of bits after type conversion. For example, in a half fixed to single precision conversion 1024 bits are used before conversion and the conversion results in 2048 bits. The size after source muxing is not shown here because the source muxing operation can change the size of the data, and it's final size is dependent on Smode.

The valid Sxprec to AUprec conversions are shown in the following figures:

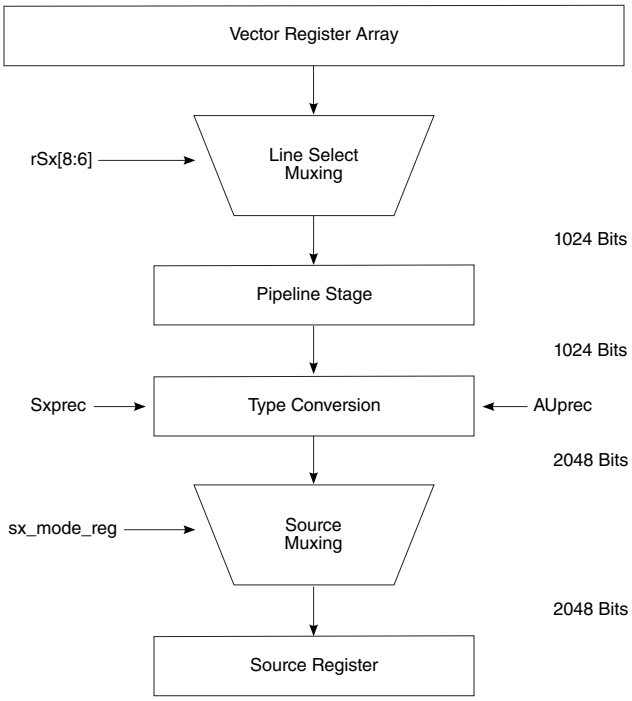


Figure 21. Half Fixed to Single Precision Source Load

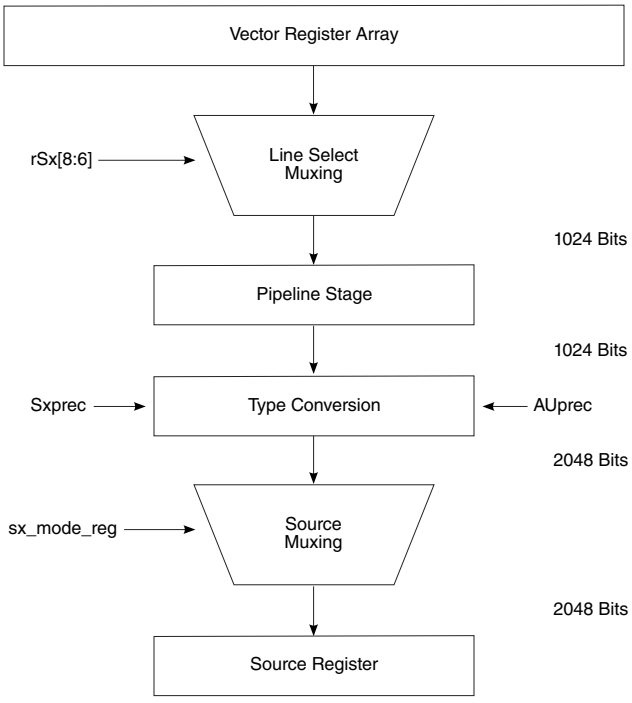


Figure 22. Half Precision to Single Precision Source Load

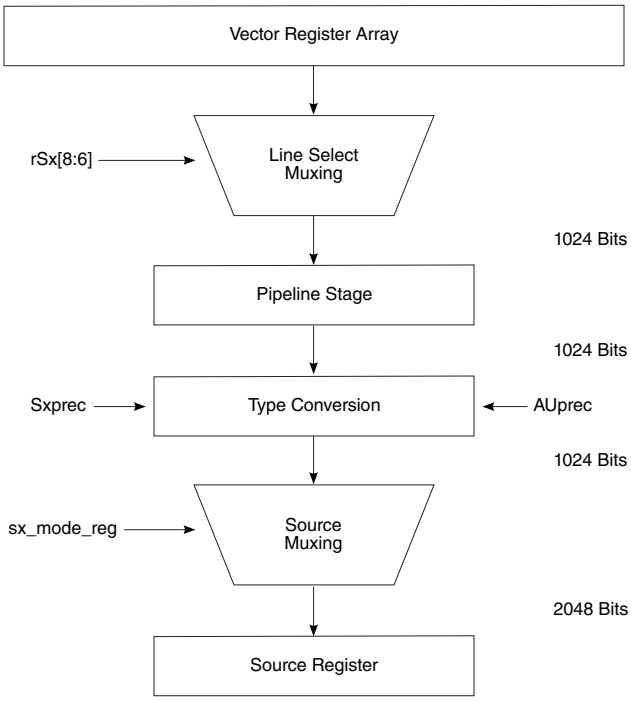


Figure 23. Single Precision to Single Precision Source Load

4.12.4 *S0mode* options and detailed description

Table 43 below shows all possible <s0mode> along with brief descriptions of these modes.

Rx denotes the VRA register where the data is read from, which is the register pointed to by rS0.

The output of the type converter contains 64 word elements. In some cases not all elements are valid out of the type converter. In those cases, the unused elements will have undefined values.

Sx (S0 for S0mode) is assumed to be a vectored bus containing 64 word elements.

In these instructions, conj (conjugate) and sign (change-sign) are optional operations. If specified, they conjugate and/or negate the output data of S0Mux, just before they are loaded into S0. See Table 43 for more details.

Table 43. All Supported *S0mode*

s0mode	Restrictions	Descriptions[1]
S0hlinecplx	Complex data creg(15[0])=x, creg(19)=x	Load S0 with output of the type converter, in preparation for complex multiplication (cmad or cmac). Each complex element is duplicated with the following pattern (real,imag,-imag,real). Update rS0.

Table continues on the next page...

Table 43. All Supported *S0mode* (continued)

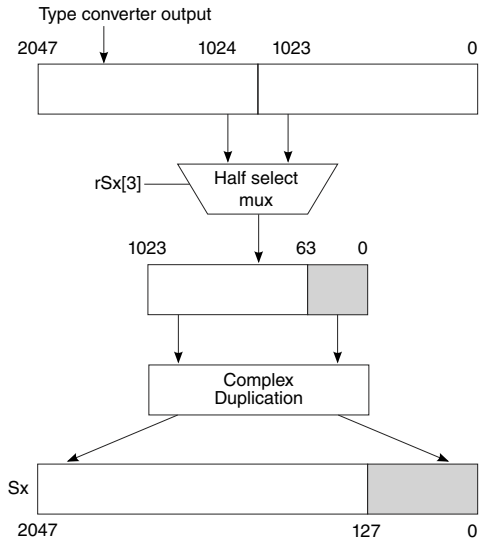
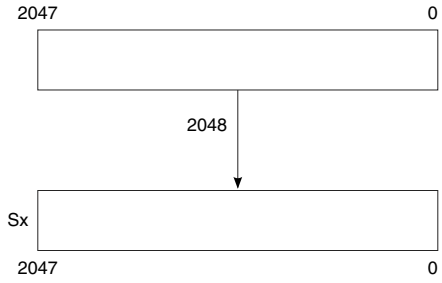
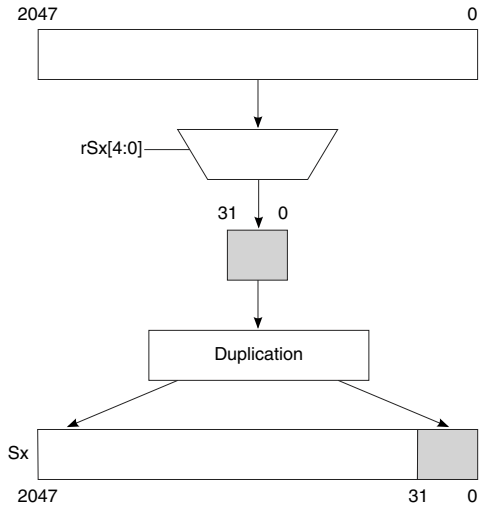
s0mode	Restrictions	Descriptions[1]
		 * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.
S0straight	Real data creg(15[0])=0, creg(19)=x	Load S0 with output of the type converter. Update rS0 value. 
S0hword	Real data creg(15[0])=0, creg(19)=x	Load a real element from output of the type converter, using rS0 as offset. Duplicate this element into all elements of S0. Update rS0.

Table continues on the next page...

Table 43. All Supported *S0mode* (continued)

s0mode	Restrictions	Descriptions[1]
		 * Note: If Sxprec is single, the rag bits used are rSx[5:2].

S0group2nr	Real data
	creg(15[0])=0, creg(23), creg(19)=x
		S0group2nr
		Load a group of real elements from output of the type converter, using rS0 as offset.
		Duplicate this group into all elements of S0:
		for (i = 0; i < 4*NAU/n; i++) {
		for (j = 0; j < n; j++) {
		S0[n*i+j] = r[j+rS0];
		}
		}
		Number of elements (n) in group is determined by 2^order_g
		Value of order_g is restricted such that:
		1<2^order_g<NAU*4
		Update rS0.

Table continues on the next page...

Table 43. All Supported *S0mode* (continued)

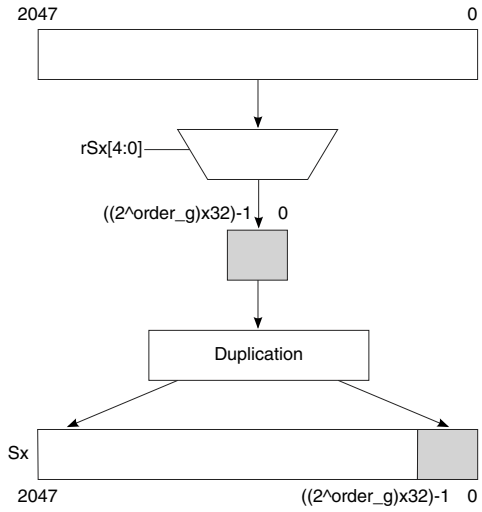
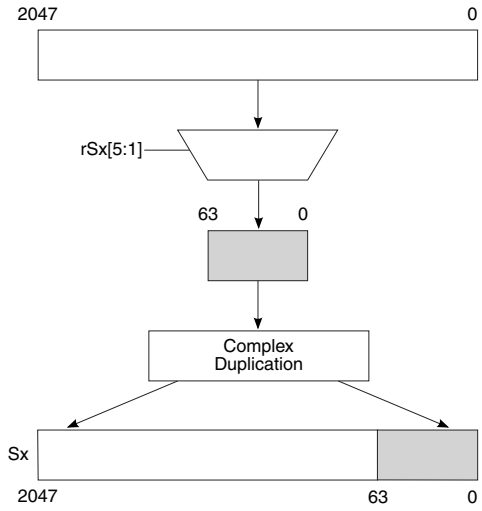
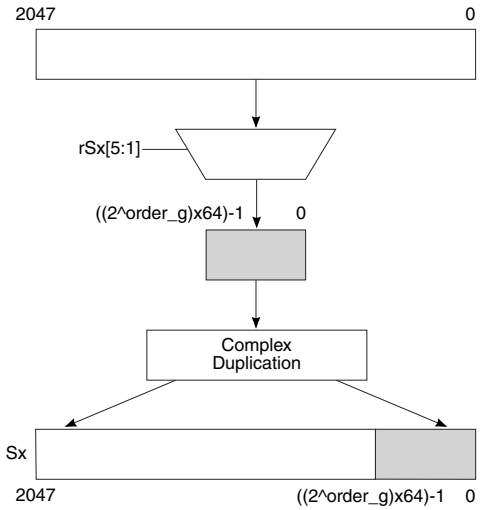
s0mode	Restrictions	Descriptions[1]
		
S0word	Complex data creg(15[0])=0, creg(19)=x	Load a complex element from output of the type converter, using rS0 as offset, in preparation for complex multiplication (cmad or cmac). Each complex element is duplicated with the following pattern (real,imag,-imag,real) into all elements of S0. Update rS0.  * Note: If Sxprec is single, the rag bits used are rSx[5:2].
S0group2nc	Complex data creg(15[0])=0, creg(23), creg(19)=x	Load a group of complex elements from output of the type converter, using rS0 as offset. Duplicate this group into all elements of S0: for (i = 0; i < NAU/n; i++){

Table continues on the next page...

Table 43. All Supported *S0mode* (continued)

s0mode	Restrictions	Descriptions[1]

		for (j = 0; j < n; j++) {
		S0[4*n*i+4*j] = r[2*(j+rS0)];
		S0[4*n*i+4*j+1] = -r[2*(j+rS0)+1];
		S0[4*n*i+4*j+2] = r[2*(j+rS0)+1];
		S0[4*n*i+4*j+3] = r[2*(j+rS0)];
		}
		}
		Number of elements (n) in group is determined by 2^order_g
		Value of order_g is restricted such that:
		1<2^order_g<NAU
		Update rS0.
		 

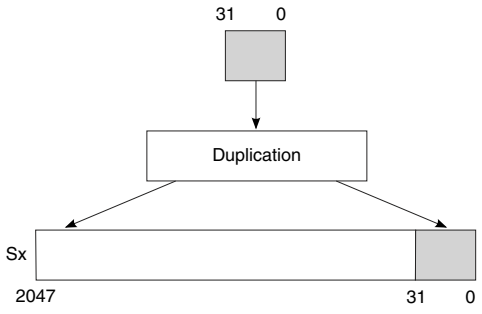
S0zeros	Real and Complex data creg(15[0])=0, creg(19)=x	Load all elements of S0 with floating-point constant "0". Update rS0. 
S0real1	Real data	Load all elements of S0 with floating-point constant "1".

Table continues on the next page...

Table 43. All Supported *S0mode* (continued)

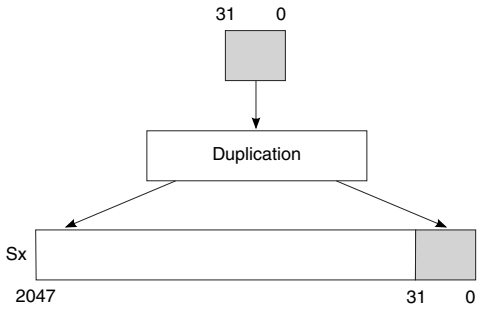
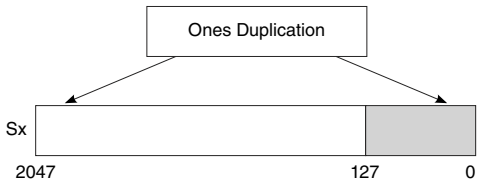
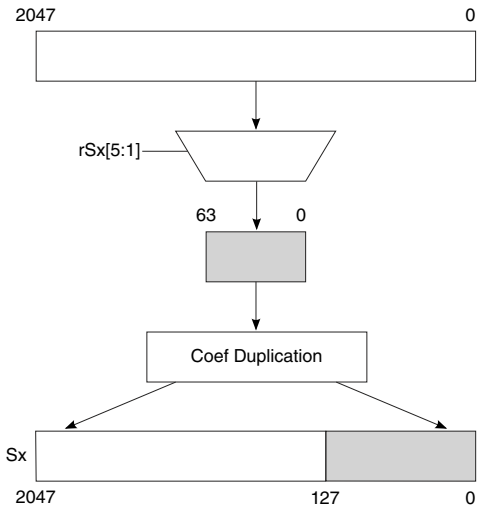
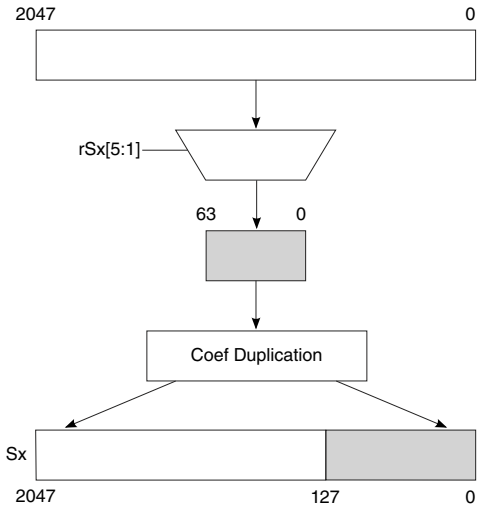
s0mode	Restrictions	Descriptions[1]
	creg(15[0])=0, creg(19)=x	Update rS0. 
S0cplx1	Complex data creg(15[0])=0, creg(19)=x	Load all complex elements of S0, in preparation for complex multiplication (cmad or cmac) with floating-point constant {0+1j, 1 + j0}. Update rS0. 
S0i1r1i1r1	Real and Complex data creg(15[0])=0, creg(19)=x	Load a complex element from output of the type converter, using rS0. The complex element is duplicated into 4 real elements in the following format (imag, real, imag, real) and replicated into all elements of S0. Update rS0.  * Note: If Sxprec is single, the rag bits used are rSx[5:2].

Table continues on the next page...

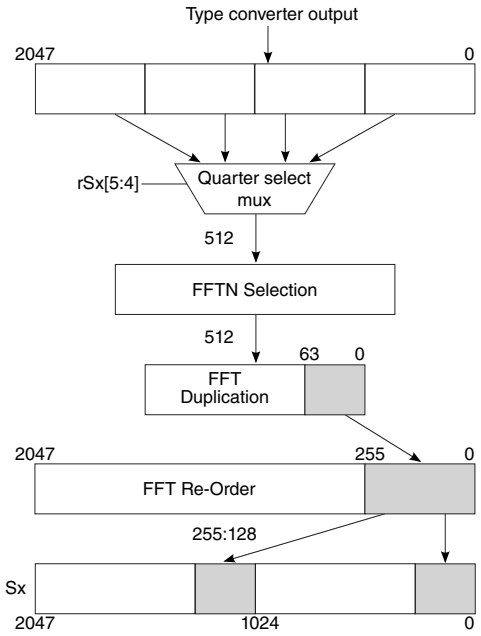
Table 43. All Supported *S0mode* (continued)

s0mode	Restrictions	Descriptions[1]
S0i1i1r1r1	Real and Complex data creg(15[0])=0, creg(19)=x	Load a complex element from output of the type converter, using rS0. The complex element is duplicated into 4 real elements in the following format (imag, imag, real, real) and replicated into all elements of S0. Update rS0.  * Note: If Sxprec is single, the rag bits used are rSx[5:2].

S0fftn;	Complex data only,
	S0prec=half_fixed /half/single,
	AUprec=single, F24.
		Using rS0, select 1 complex element from output of the type converter.
		Replicate these elements into S0, in preparation for the DIT or DIF butterfly operation.
		Each FFT complex element is duplicated with the following pattern (real,imag,-real,-imag,-imag,real,imag,-real).
		FFT re-orders the complex elements as follows:
		for(j=0; j<8; j++){
		// first half output
		mx_fft[127+j*128 : j*128] =
		mx_fft_orig[127+j*256 : j*256];
		// second half output
		mx_fft[1024+127+j*128 : 1024+j*128]=
		mx_fft_orig[127+j*256+128 : j*256+128];
		}
		Update rS0.

Table continues on the next page...

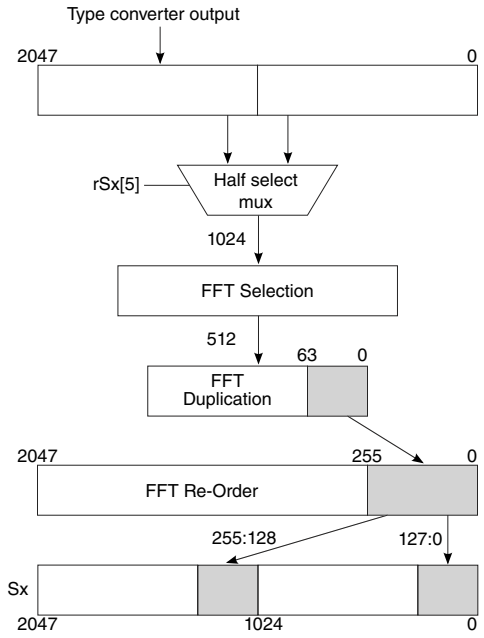
Table 43. All Supported *S0mode* (continued)

s0mode	Restrictions	Descriptions[1]
		 * Note: If Sxprec is single, the quarter select mux only selects from the lower two quarters for FFT duplication.

S0fft4;	Complex data only,
	S0prec=half_fixed /half/single,
	AUprec=F24.
		Using rS0, select 8 complex elements from the output of the type converter.
		Replicate these elements into S0, in preparation for the DIT or DIF butterfly operations
		Each FFT complex element is duplicated with the following pattern (real,imag,-real,-imag,-imag,real,imag,-real).
		FFT re-orders the complex elements as follows:
		for(j=0; j<8; j++){
		// first half output
		mx_fft[127+j*128 : j*128] =
		mx_fft_orig[127+j*256 : j*256];
		// second half output
		mx_fft[1024+127+j*128 :1024+j*128]=
		mx_fft_orig[127+j*256+128 : j*256+128];
		}
		Update rS0.

Table continues on the next page...

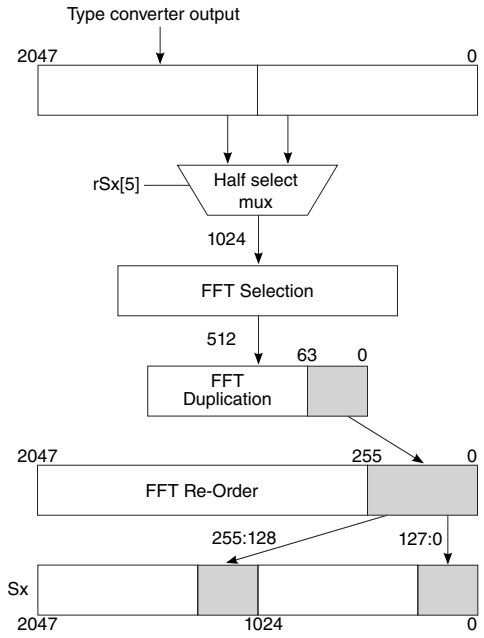
Table 43. All Supported *S0mode* (continued)

s0mode	Restrictions	Descriptions[1]
		 * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.

S0fft3;	Complex data only,
	S0prec=half_fixed /half/single,
	AUprec=single, F24.
		Using rS0, select 4 complex elements from the output of the type converter.
		Replicate these elements into S0, in preparation for the DIT or DIF butterfly operations
		Each FFT complex element is duplicated with the following pattern (real,imag,-real,-imag,-imag,real,imag,-real).
		FFT re-orders the complex elements as follows:
		for(j=0; j<8; j++){
		// first half output
		mx_fft[127+j*128 : j*128] =
		mx_fft_orig[127+j*256 : j*256];
		// second half output
		mx_fft[1024+127+j*128 :1024+j*128]=
		mx_fft_orig[127+j*256+128 : j*256+128];
		}
		Update rS0.

Table continues on the next page...

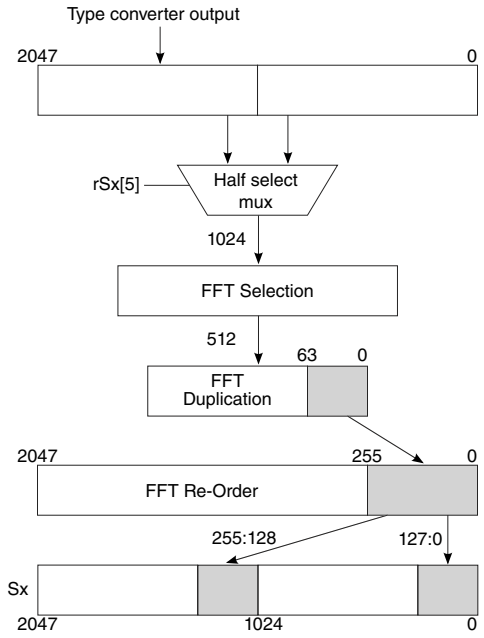
Table 43. All Supported *S0mode* (continued)

s0mode	Restrictions	Descriptions[1]
		 * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.

S0fft2;	Complex data only,
	S0prec=half_fixed /half/single,
	AUprec=single, F24.
		Using rS0, select 2 complex elements from the output of the type converter.
		Replicate these elements into S0, in preparation for the DIT or DIF butterfly operation.
		Each FFT complex element is duplicated with the following pattern (real,imag,-real,-imag,-imag,real,imag,-real).
		FFT re-orders the complex elements as follows:
		for(j=0; j<8; j++){
		// first half output
		mx_fft[127+j*128 : j*128] =
		mx_fft_orig[127+j*256 : j*256];
		// second half output
		mx_fft[1024+127+j*128 :1024+j*128]=
		mx_fft_orig[127+j*256+128 : j*256+128];
		}
		Update rS0.

Table continues on the next page...

Table 43. All Supported *S0mode* (continued)

s0mode	Restrictions	Descriptions[1]
		 * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.

S0fft1;	Complex data only,
	S0prec=half_fixed /half/single,
	AUprec=single, F24.
		Using rS0, select 1 complex elements from the output of the type converter.
		Replicate these elements into S0, in preparation for the DIT or DIF butterfly operation.
		Each FFT complex element is duplicated with the following pattern (real,imag,-real,-imag,-imag,real,imag,-real).
		FFT re-orders the complex elements as follows:
		for(j=0; j<8; j++){
		// first half output
		mx_fft[127+j*128 : j*128] =
		mx_fft_orig[127+j*256 : j*256];
		// second half output
		mx_fft[1024+127+j*128 :1024+j*128]=
		mx_fft_orig[127+j*256+128 : j*256+128];
		}
		Update rS0.

Table continues on the next page...

Table 43. All Supported *S0mode* (continued)

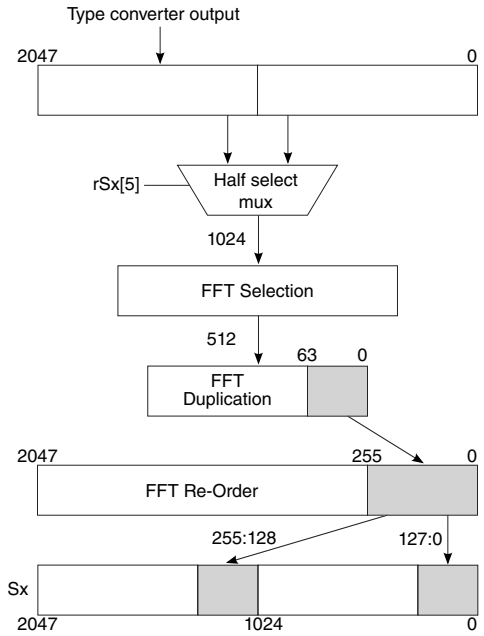
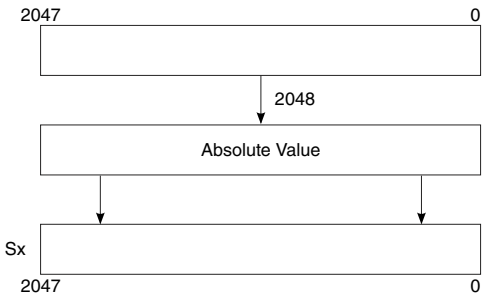
s0mode	Restrictions	Descriptions[1]
		 * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.
S0abs	Real and Complex data creg(15[0])=0, creg(19)=x	Load S0 with absolute value of each element. Update rS0. 
<s0mode>, sign;	Real and Complex data	Perform sign-bit inversion on all the data elements.

Table continues on the next page...

Table 43. All Supported *S0mode* (continued)

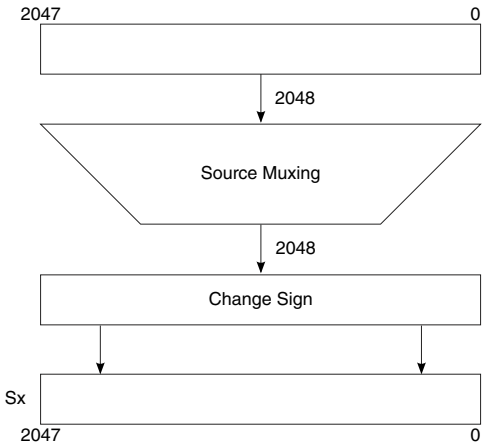
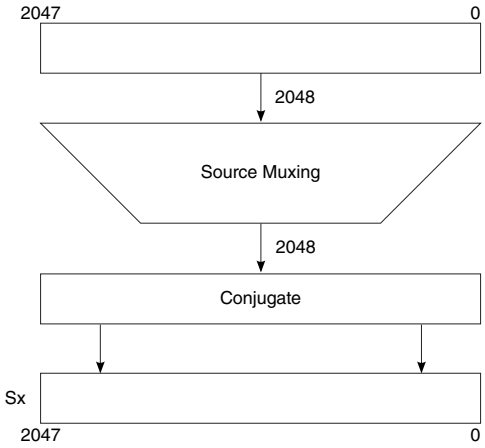
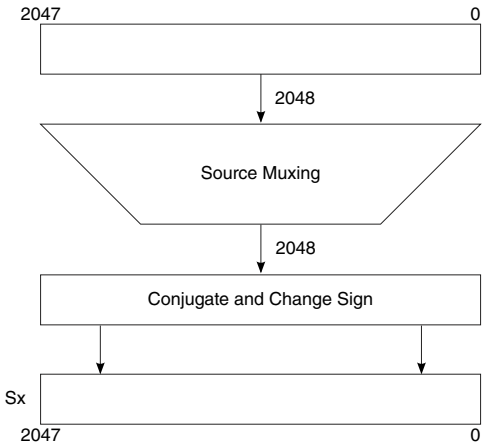
s0mode	Restrictions	Descriptions[1]
		
<s0mode>, S0conj;	Real and Complex data	Perform conjugate on all the complex data elements; that is, invert the sign bit of all imaginary elements. 
<s0mode>, S0conj, sign;	Real and Complex data	Perform both conjugate and sign inversion; that is, inverting the sign bit of all real elements.

Table continues on the next page...

Table 43. All Supported *S0mode* (continued)

s0mode	Restrictions	Descriptions[1]
		

1. abs, conj and sign operations are performed on the output of S0Mux. The results of these operations are then loaded into S0.

4.12.5 *S1mode* options and detailed description

Table 44 below shows all possible <S1mode> along with brief descriptions of these modes.

Rx denotes the VRA register where the data is read from, which is the register pointed to by rS1.

The output of the type converter contains 64 word elements. In some cases not all elements are valid out of the type converter. In those cases, the unused elements will have undefined values.

Sx (S1 for S1mode) is assumed to be a vectored bus containing 64 word elements.

Table 44. All Supported *S1mode*

S1mode	Restrictions	Descriptions
S1hlinecplx	Complex data	Load S1 with output of the type converter, in preparation for complex multiplication (cmad or cmac). Each complex element is duplicated with the following pattern (imag,real,imag,real). Update rS1.

Table continues on the next page...

Table 44. All Supported *S1mode* (continued)

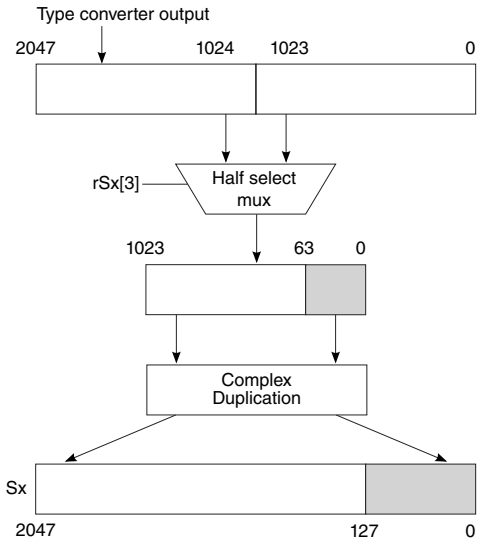
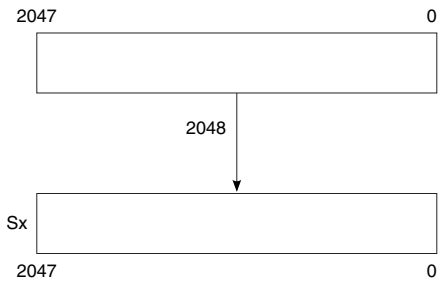
S1mode	Restrictions	Descriptions
		 * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.
S1straight	Real data	Load S1 with output of the type converter. Update rS1. 
S1real1	Real data	Load all elements of S1 with floating-point constant "1". Update rS1. **NOTE** This S1 mode will only produce floating point values and so may NOT be used in conjuction with AUprec padd or paddF24, which require values in half fixed format.

Table continues on the next page...

Table 44. All Supported *S1mode* (continued)

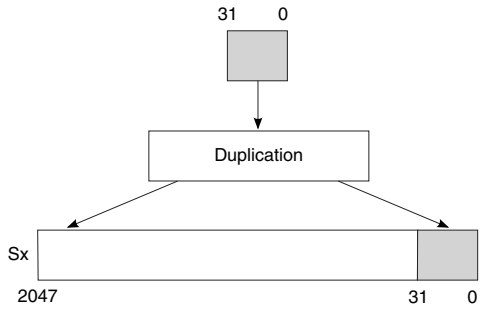
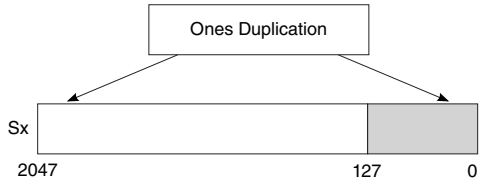
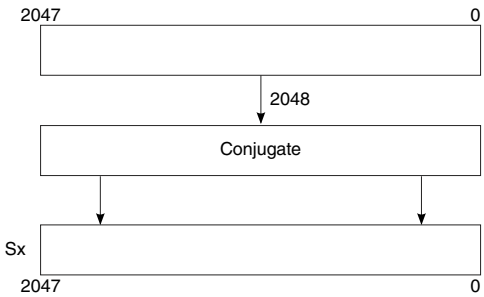
S1mode	Restrictions	Descriptions
		
S1cplx1	Complex data	Load all complex elements of S1, in preparation for complex multiplication (cmad or cmac) with floating-point constant {1 + j0, 1 + j0}. Update rS1. **NOTE** This S1 mode will only produce floating point values and so may NOT be used in conjuction with AUprec padd or paddF24, which require values in half fixed format. 
S1real_conj	Real data	Same as "straight" with post-conjugate operation; that is, invert the sign bit of the odd elements. 
S1cplx_conj	Complex data	Same as "hlinecplx" with post-conjugate operation; that is, invert the sign bit of the imaginary elements.

Table continues on the next page...

Table 44. All Supported *S1mode* (continued)

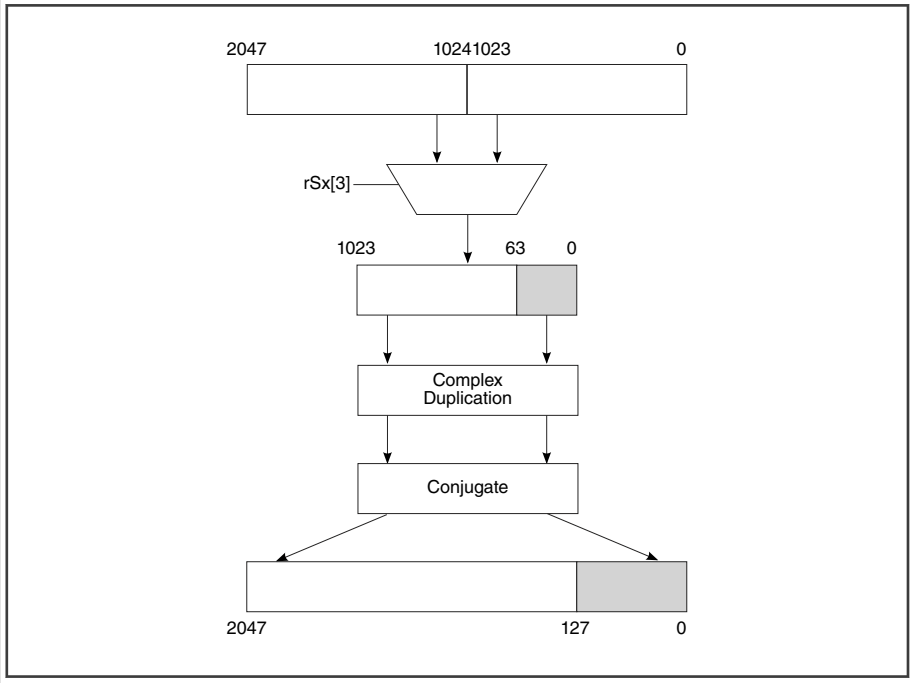
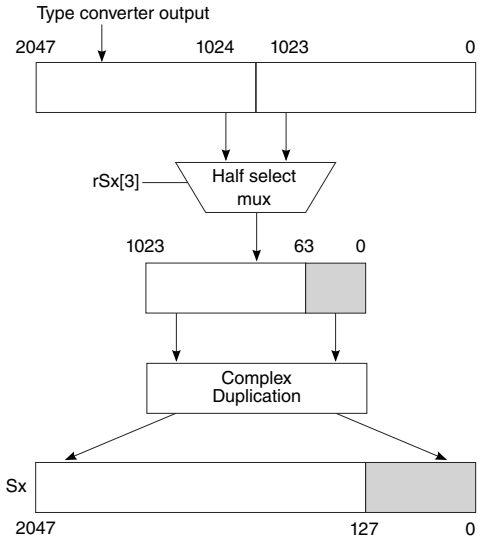
S1mode	Restrictions	Descriptions
		
S1r2c	Real and Complex data	Load S1 real-to-complex with output of the type converter. Each complex element is replicated as follows (imag,imag,real,real) into S1. Update rS1.  * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.
S1r2c_conj	Real and	Same as "r2c" with post-conjugate operation; that is, invert the sign bit of the imaginary elements.

Table continues on the next page...

Table 44. All Supported *S1mode* (continued)

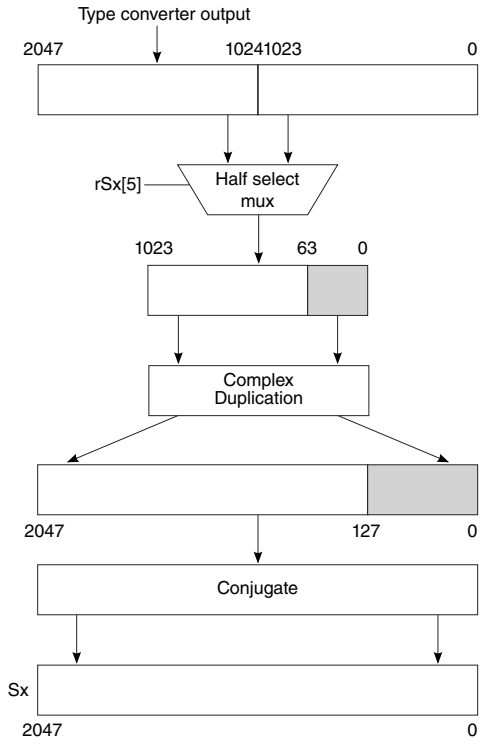
S1mode	Restrictions	Descriptions
	Complex data	 * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.
S1r2c_im0	Real and Complex data	Load S1 real-to-complex with output of the type converter. Each complex element is replicated as follows (0,imag,0,real} into S1. Update rS1.

Table continues on the next page...

Table 44. All Supported *S1mode* (continued)

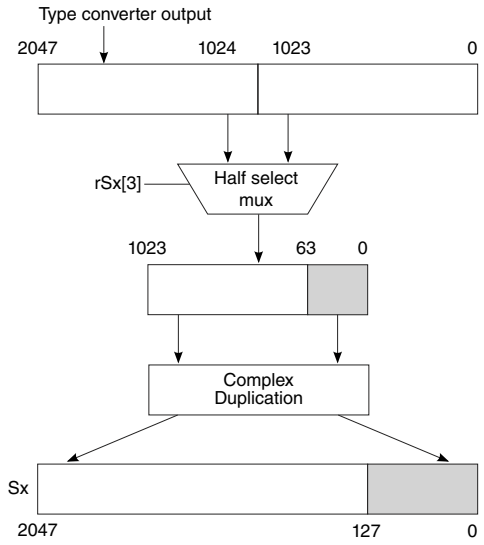
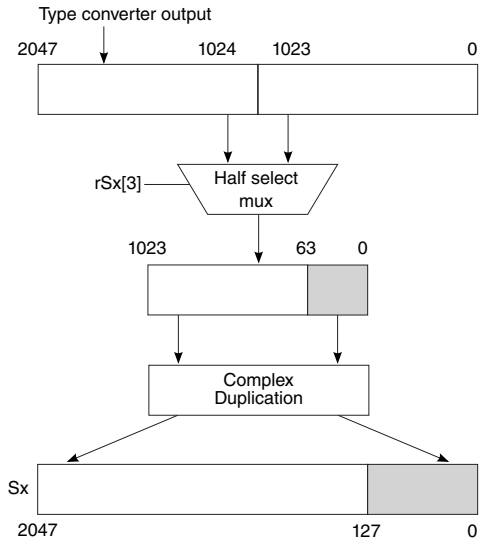
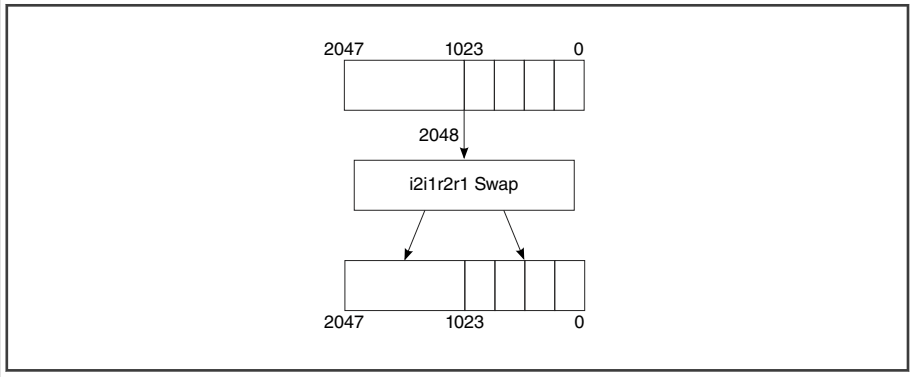
S1mode	Restrictions	Descriptions
		 * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.
S1r2c_re0	Real and Complex data	Load S1 real-to-complex with output of the type converter. Each complex element is replicated as follows (imag,0,real,0} into S1. Update rS1.  * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.
S1i2i1r2r1	Real data	Same as S1hlinecplx

Table continues on the next page...

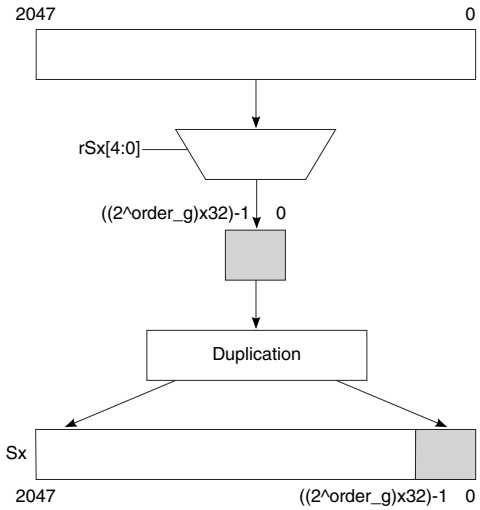
Table 44. All Supported *S1mode* (continued)

S1mode	Restrictions	Descriptions
	Complex data	The i2i1r2r1 mode is for up-down filter with complex inputs Load S1 with output of the type converter, swapping the 2nd and 3rd elements of each group of 4 elements. Update rS1. 
S1udfR	Real data	Same as "r2c" mode. **NOTE** This mode is only supported for backward compatibility, that is when set.creg is set to 1.
	Complex data	The udfR mode is for up-down filter with real inputs Same as "straight" mode.

S1interp2nr	Real data
	creg(24), creg(16), creg(17), creg(18), creg(19)=0
		Load real elements from the output of the type converter, using rS1 as offset.
		Duplicate elements into S1:
		for (i = 0; i < 4*NAU/n; i++) {
		for (j = 0; j < n; j++) {
		S1[n*i+j] = r[i+rS1];
		}
		}
		Number of elements (n) in group is determined by 2^order_i
		Value of order_i is restricted such that:
		1 < 2^order_i <= NAU*4
		Update rS1.

Table continues on the next page...

Table 44. All Supported *S1mode* (continued)

S1mode	Restrictions	Descriptions
		

S1interp2nc	Complex data
	creg(24), creg(16), creg(17), creg(18), creg(19)=0
		Load complex elements from the output of the type converter, using rS1 as offset.
		Duplicate elements into S1:
		for (i = 0; i < NAU/n; i++) {
		for (j = 0; j < n; j++) {
		S1[4*n*i+4*j] = r[2*(i+rS1)];
		S1[4*n*i+4*j+1] = r[2*(i+rS1)+1];
		S1[4*n*i+4*j+2] = r[2*(i+rS1)];
		S1[4*n*i+4*j+3] = r[2*(i+rS1)+1];
		}
		}
		Number of elements (n) in group is determined by 2^order_i
		Value of order_i is restricted such that:
		1 < 2^order_i <= NAU
		Update rS1.

Table continues on the next page...

Table 44. All Supported *S1mode* (continued)

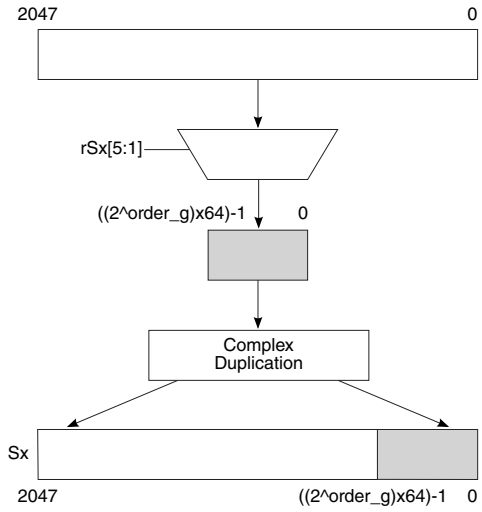
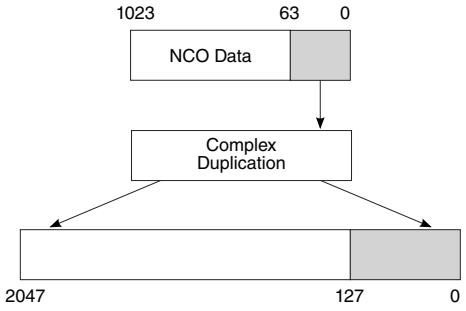
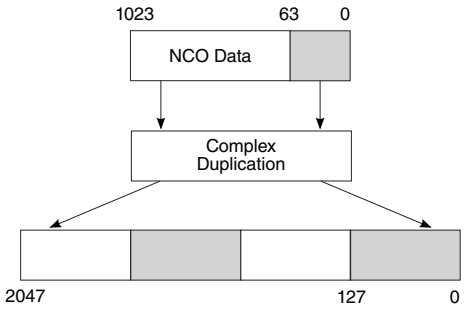
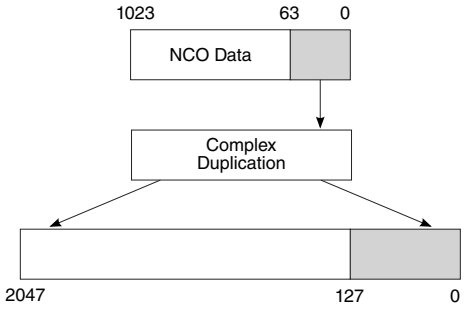
S1mode	Restrictions	Descriptions
		
S1nco	Complex data Only creg(25)=0, creg(19)=0	Load S1 with complex tone samples being generated by nco module. Update rS1. **NOTE** The Vector NCO is only capable of producing single-precision twiddle factors. If *S1nco*, then the programmer must constrain AUprec to be single. When nco_mode=singles, take first complex element and duplicate with the following pattern (imag,real,imag,real).  When nco_mode=radix2, take first half of complex elements and duplicate with the following pattern (imag,real,imag,real). Take the output and copy to the upper half of S1.

Table continues on the next page...

Table 44. All Supported *S1mode* (continued)

S1mode	Restrictions	Descriptions
		 When nco_mode=normal, duplicate each complex element with the following pattern (imag,real,imag,real). 

4.12.6 *S2mode* options and detailed description

Table 45 below shows all possible <s2mode> along with brief descriptions of these modes.

Rx denotes the VRA register where the data is read from, which is the register pointed to by rS2.

The output of the type converter contains 64 word elements. In some cases not all elements are valid out of the type converter. In those cases, the unused elements will have undefined values.

Sx (S2 for S2mode) is assumed to be a vectored bus containing 64 word elements.

Table 45. All Supported *S2mode*

S2mode	Restrictions	Descriptions
S2hlinecplx	Complex data creg(15[0])=x, creg(19)=x	Load S2 with output of the type converter, in preparation for complex multiplication (cmad or cmac). Each complex element is duplicated with the following pattern (0,imag,0,real). Update rS2.

Table continues on the next page...

Table 45. All Supported *S2mode* (continued)

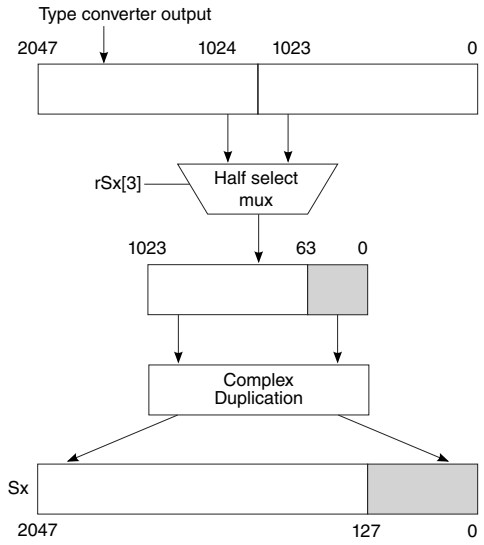
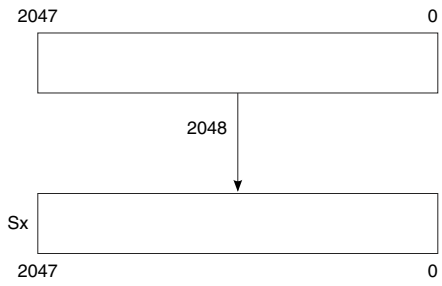
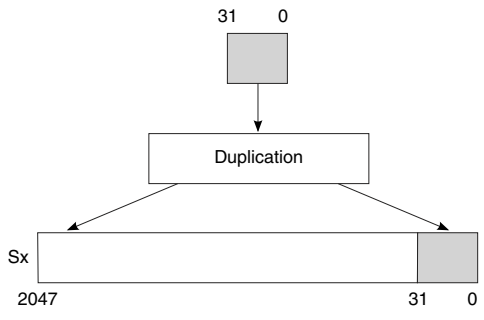
S2mode	Restrictions	Descriptions
		 * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.
S2straight	Real data	Load S2 with output of the type converter. Update rS2. 
S2zeros	Real and Complex data	Load all elements of S2 with floating-point constant "0". Update rS2. 

Table continues on the next page...

Table 45. All Supported *S2mode* (continued)

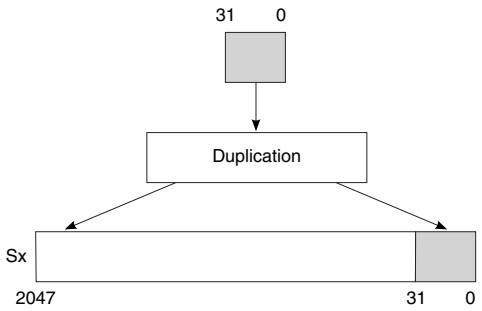
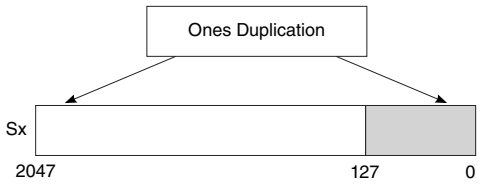
S2mode	Restrictions	Descriptions
S2real1	Real data	Load all elements of S2 with floating-point constant "1". Update rS2. **NOTE** This S2 mode will only produce floating point values and so may NOT be used in conjuction with AUprec padd or paddF24, which require values in half fixed format. 
S2cplx1	Complex data	Load all complex elements of S2, in preparation for complex multiplication (cmad or cmac) with floating-point constant {0+0j,1+0j}. Update rS2. **NOTE** This S2 mode will only produce floating point values and so may NOT be used in conjuction with AUprec padd or paddF24, which require values in half fixed format. 
S2i1r1i1r1	Real and Complex data	Load a complex element from the output of the type converter, using rS2. The complex element is expanded into 4 real elements in the following format (imag, real, imag, real) and replicated into all elements of S2. Update rS2.

Table continues on the next page...

Table 45. All Supported *S2mode* (continued)

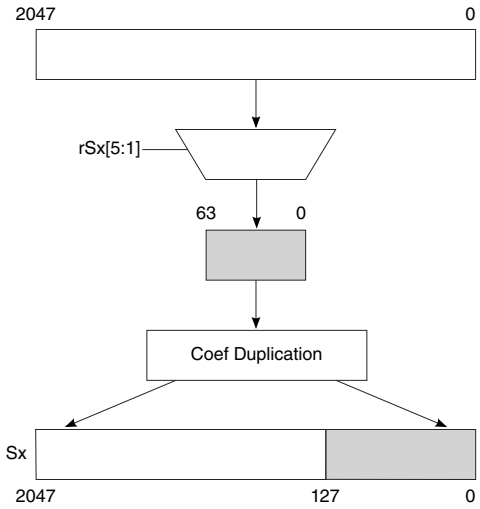
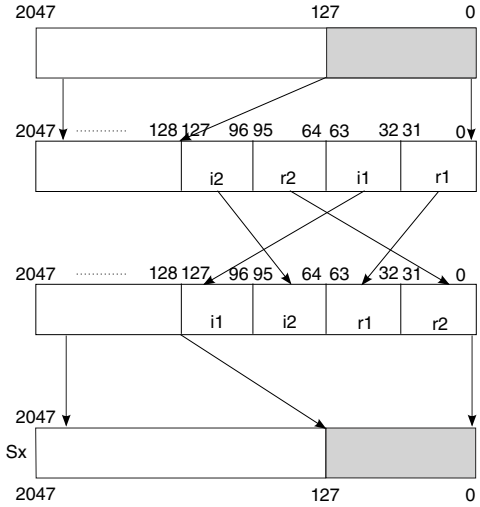
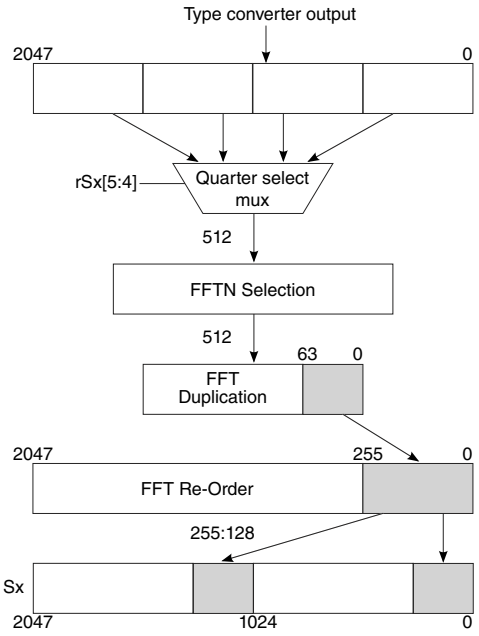
S2mode	Restrictions	Descriptions
		 * Note: If Sxprec is single, the rag bits used are rSx[5:2].
S2i1i2r1r2	Real and Complex data	Load complex elements from the output of the type converter, using rS2. The complex elements are re-ordered in the following format (imag1, imag2, real1, real2) and loaded into S2. 
S2fftn;	Complex data only, S2prec=half_fixed /half/single, AUprec=single, F24.	Using rS2, select 2 complex elements from the output of the type converter. Replicate these elements into S2, in preparation for the DIT or DIF butterfly operation. Each FFT complex element is duplicated with the following pattern (real,imag,0,imag,-imag,real,0,real). FFT re-orders the complex elements as follows: for(j=0; j<8; j++){

Table continues on the next page...

Table 45. All Supported *S2mode* (continued)

S2mode	Restrictions	Descriptions

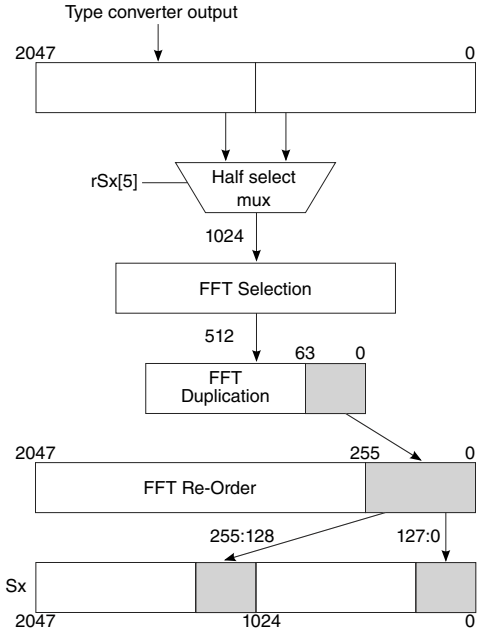
	creg(15[0])=1, creg(19)=0	// first half output
		mx_fft[127+j*128 : j*128] =
		mx_fft_orig[127+j*256 : j*256];
		// second half output
		mx_fft[1024+127+j*128 : 1024+j*128]=
		mx_fft_orig[127+j*256+128 : j*256+128];
		}
		Update rS2.
		 
		
		* Note: If Sxprec is single, the quarter select mux only selects from the lower two quarters for FFT duplication.

S2fft4;	Complex data only,
	S2prec=half_fixed /half/single,
	AUprec=F24.
	creg(15[0])=1, creg(19)=0
		Using rS2, select 8 complex elements from the output of the type converter.
		Replicate these elements into S2, in preparation for the DIT or DIF butterfly operations
		Each FFT complex element is duplicated with the following pattern (real,imag,0,imag,-imag,real,0,real).
		FFT re-orders the complex elements as follows:
		for(j=0; j<8; j++){
		// first half output
		mx_fft[127+j*128 : j*128] =
		mx_fft_orig[127+j*256 : j*256];

Table continues on the next page...

Table 45. All Supported *S2mode* (continued)

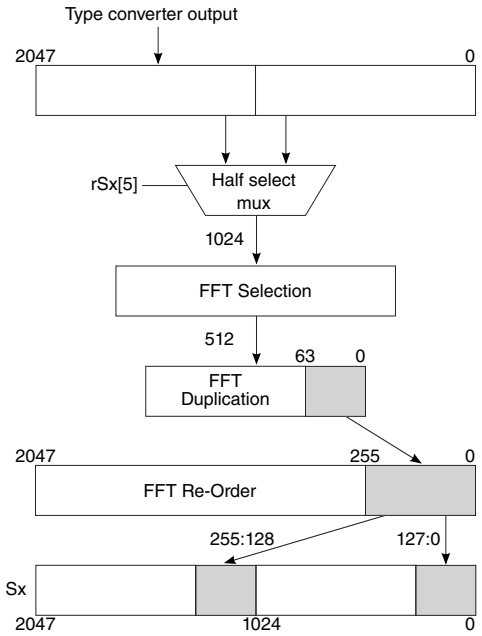
S2mode	Restrictions	Descriptions

		// second half output
		mx_fft[1024+127+j*128 : 1024+j*128]=
		mx_fft_orig[127+j*256+128 : j*256+128];
		}
		Update rS2.
		 
		
		* Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.

S2fft3;	Complex data only,
	S2prec=half_fixed /half/single,
	AUprec=single, F24.
	creg(15[0])=1, creg(19)=0
		Using rS2, select 4 complex elements from the output of the type converter.
		Replicate these elements into S2, in preparation for the DIT or DIF butterfly operations
		Each FFT complex element is duplicated with the following pattern (real,imag,0,imag,-imag,real,0,real).
		FFT re-orders the complex elements as follows:
		for(j=0; j<8; j++){
		// first half output
		mx_fft[127+j*128 : j*128] =
		mx_fft_orig[127+j*256 : j*256];
		// second half output
		mx_fft[1024+127+j*128 : 1024+j*128]=
		mx_fft_orig[127+j*256+128 : j*256+128];

Table continues on the next page...

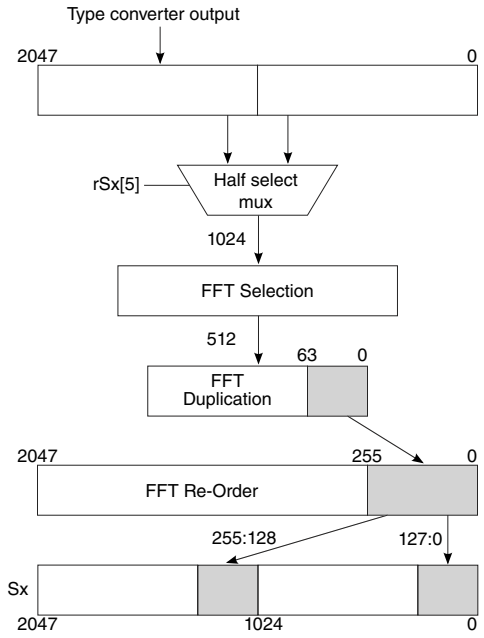
Table 45. All Supported *S2mode* (continued)

S2mode	Restrictions	Descriptions
		} Update rS2.  * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.

S2fft2;	Complex data only,
	S2prec=half_fixed /half/single,
	AUprec=single, F24.
	creg(15[0])=1, creg(19)=0
		Using rS2, select 2complex elements from the output of the type converter.
		Replicate these elements into S2, in preparation for the DIT or DIF butterfly operations
		Each FFT complex element is duplicated with the following pattern (real,imag,0,imag,-imag,real,0,real).
		FFT re-orders the complex elements as follows:
		for(j=0; j<8; j++){
		// first half output
		mx_fft[127+j*128 : j*128] =
		mx_fft_orig[127+j*256 : j*256];
		// second half output
		mx_fft[1024+127+j*128 : 1024+j*128]=
		mx_fft_orig[127+j*256+128 : j*256+128];
		}
		Update rS2.

Table continues on the next page...

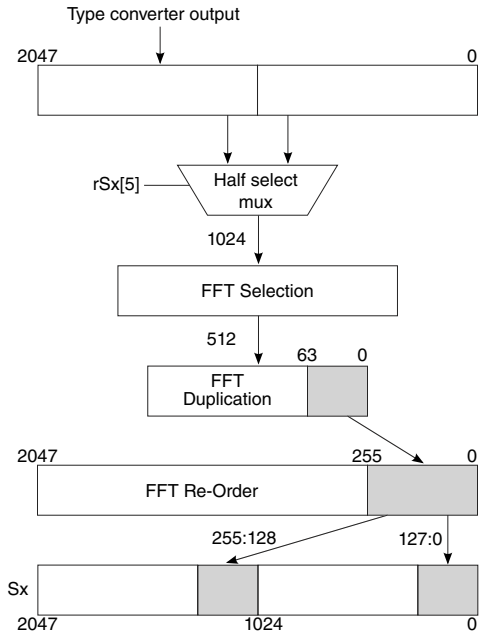
Table 45. All Supported *S2mode* (continued)

S2mode	Restrictions	Descriptions
		 * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.

S2fft1;	Complex data only,
	S2prec=half_fixed /half/single,
	AUprec=single, F24.
	creg(15[0])=1, creg(19)=0
		Using rS2, select 1 complex elements from the output of the type converter.
		Replicate these elements into S2, in preparation for the DIT or DIF butterfly operations
		Each FFT complex element is duplicated with the following pattern (real,imag,0,imag,-imag,real,0,real).
		FFT re-orders the complex elements as follows:
		for(j=0; j<8; j++){
		// first half output
		mx_fft[127+j*128 : j*128] =
		mx_fft_orig[127+j*256 : j*256];
		// second half output
		mx_fft[1024+127+j*128 : 1024+j*128]=
		mx_fft_orig[127+j*256+128 : j*256+128];
		}
		Update rS2.

Table continues on the next page...

Table 45. All Supported *S2mode* (continued)

S2mode	Restrictions	Descriptions
		 * Note: If Sxprec is single, the half select mux is not present. Only the lower half of the bus is used for duplication.

4.13 AU instructions

This group of instructions specify AU arithmetic operations.

- rmad - multiply add
- rmad.sau - multiply and add, using SAU result as S1 operand
- cmad - complex multiply add and decimation-in-time butterfly
- cmad.sau - complex multiply add and decimation-in-time butterfly that uses output of previous SAU operation instead of s1 register contents
- rmac - multiply accumulate
- rmac.sau - multiply and accumulate, using SAU result as S1 operand
- cmac - complex multiply accumulate
- cmac.sau - complex multiply accumulate that uses output of previous SAU operation instead of S1 register contents
- mads - special mad intended for AU-assisted data conversion between float and fixed point
- mads.sau - multiply and add with sign conversion.
- maf - multiply add with feedback to multiplier
- mafac - multiply-feedback-accumulate
- dif.sau - decimation-in-frequency radix-2 butterfly,using SAU result as S1 operand
- clr.au - clear vector accumulator

Table 46. AU Instructions[1],[2]

Mnemonic	Function Performed in Real Mode	Function Performed in Complex Mode
rmad	V[i][n] = (S0[i][n-4]*S1[i][n-4]) + S2[i][n-4][3] for i=0,1,..,63	Undefined
rmad.sau	V[i][n] = (S0[i][n-4]*SAUout[i][n-4]) + S2[i][n-4] for i=0,1,..,63	Undefined
cmad	Undefined	V[i][n] = (S0[i][n-4]*S1[i][n-4]) + (S0[i+1][n-4]*S1[i+1][n-4]) + S2[i][n-4] for i=0,2,4,6,...,62
cmad.sau	Undefined	V[i][n] = (S0[i][n-4]*SAUout[i][n-4]) + (S0[i+1][n-4]*SAUout[i+1][n-4]) + S2[i][n-4] for i=0,2,4,6,...,62
rmac	V[i][n] = (S0[i][n-4]*S1[i][n-4]) + V[i][n-1] for i=0,1,..,63	Undefined
rmac.sau	V[i][n] = (S0[i][n-4]*SAUout[i][n-4]) + V[i][n-1] for i=0,1,..,63	Undefined
cmac	Undefined	V[i][n] = (S0[i][n-4]*S1[i][n-4]) + (S0[i+1][n-4]*S1[i+1][n-4]) + V[i][n-1] for i=0,2,4,6,...,62
cmac.sau	Undefined	V[i][n] = (S0[i][n-4]*SAUout[i][n-4]) + (S0[i+1][n-4]*SAUout[i+1][n-4]) + V[i][n-1] for i=0,2,4,6,...,62
mads	V[i][n] = (S0[i][n-4]*S1[i][n-4]) + sign[4]((S0[i][n-4]*S1[i][n-4])) · S2[i][n-4] for i=0,1,..,63	Undefined
mads.sau	V[i][n] = (S0[i][n-4]*SAUout[i][n-4]) + sign((S0[i][n-4]*SAUout[i][n-4])) · S2[i][n-4] for i=0,1,..,63	Undefined
maf	V[i][n] = (V[i][n-4]*S1[i][n-4]) + S2[i][n-4] for i=0,1,..,63	Undefined
mafac	V[i][n] = (V[i][n-4]*S1[i][n-4]) + V[i][n-1] for i=0,1,..,63	Undefined

Table continues on the next page...

Table 46. AU Instructions[1],[2] (continued)

Mnemonic	Function Performed in Real Mode	Function Performed in Complex Mode
dif.sau	Undefined	V[i][n]=(S0[i][n-4]*SAUout[i][n-4]) + (S0[i+1][n-4]*SAUout[i+1][n-4]) + (SAUout[i+2][n-4]*S2[i+2][n-4]) + (SAUout[i+3][n-4]*S2[i+3][n-4]); V[i+2][n]=S0[i+2][n-4] + S2[i+2][n-4]; for i=0,4,..,60
clr.au	Clear Vector Accumulator V[i] = 0 for i=0,1,..,63	Clear Vector Accumulator V[i] = 0 for i=0,1,..,63

1. s0[n], s1[n], s2[n] and V are vectored-buses. Each of these buses contains 64 real elements in SP mode.
2. s0[*i*][n] denotes the i^{th} element of the s0[n] bus, V[*i*][n] denotes the i^{th} element of the V[n] bus, and so on. 'n' denotes the cycle count.
3. This table provides information for single precision.
4. sign() function returns 1 or -1 based on the sign bit of the input.

4.13.1 AU latency

When AUprec is set to single precision the latency is four.

For example, a rmad/cmad instruction with AUprec set to single precision will take four cycles in total. After four cycles, the auOut bus will contain the results of the mad instruction and can be written back to the VRA. So loading the source operand registers (S0, S1 and S2) through writing the AU results back to VRA will take seven cycles; two for the source operand load, four for the AU instruction execution and one for the VRA writeback.

4.13.2 AU instructions code example

```
set.prec half_fixed, half_fixed, half_fixed, single, half_fixed;
set.Smode S0hlinecplx, S1hlinecplx, S2hlinecplx; rd S0; rd S1; rd S2;
nop;
cmad;
nop;
nop;
nop;
wr.hlinecplx;
```

4.13.3 Multiply add functionality

Table 47. MAD Descriptions

Instructions	Real/ Complex Modes	AUprec	Operations	Descriptions

rmad	Real	Single	for (i=0; i<64; i++) {
			V[i][n] = (S0[i][n-4]*S1[i][n-4]) + S2[i][n-4]
			}
				Perform 64 parallel multiply-add operations on S0/S1/S2 elements.

Table continues on the next page...

Table 47. MAD Descriptions (continued)

Instructions	Real/ Complex Modes	AUprec	Operations	Descriptions

cmad	Complex	Single	for (i=0; i<64; i +=2) {
			V[i][n] = (S0[i][n-4]*S1[i][n-4]) + (S0[i+1][n-4]*S1[i+1][n-4]) + S2[i][n-4]
			}
				This can perform 16 complex multiply-add operations, where V[0] represent the real outputs and V[2] represent the imaginary outputs.

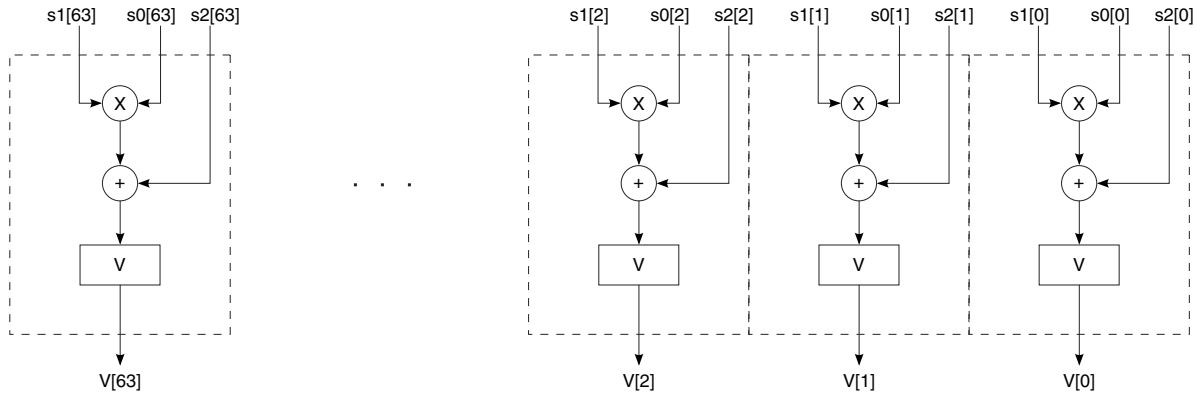


Figure 24. Real mad Operation (Single Precision)

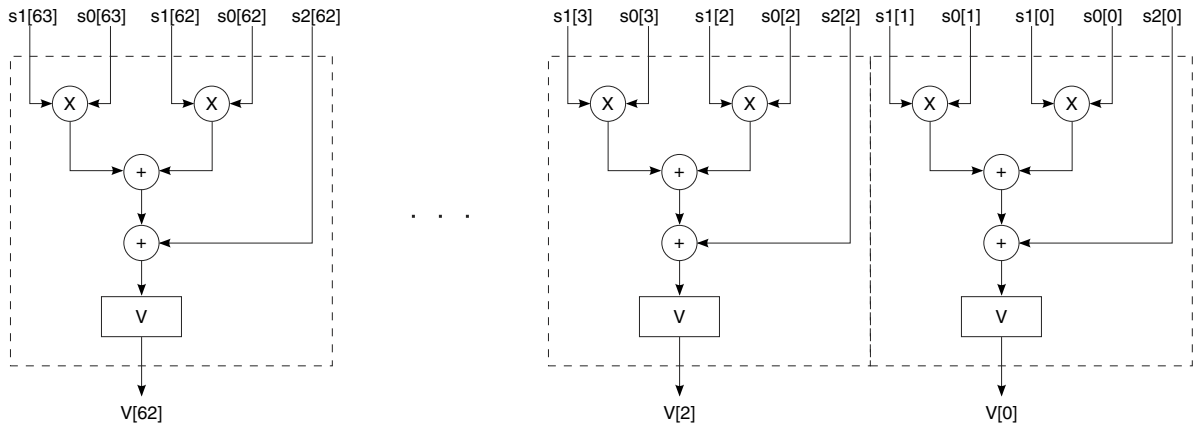


Figure 25. Complex mad Operation (Single Precision)

4.13.4 Multiply and add with sign conversion

Table 48. MADS Description

Instructions	Real/ Complex Modes	AUprec	Operations	Descriptions

mads	Real Only	Single	for (i=0; i<64; i++) {
			V[i][n] = (S0[i][n-4]*S1[i][n-4]) + sign((S0[i][n-4]*S1[i][n-4]))*S2[i][n-4]
			}
			where sign() function returns 1 or -1 based on the sign bit of the input.
				Perform 64 parallel multiply-add operations on S0/S1/S2 elements, where the sign of the adder operand (from S2) is inverted if the product of s0 and s1 is negative.
				This instruction is useful in using the AU to perform float-to-fixed or fixed-to-float conversion

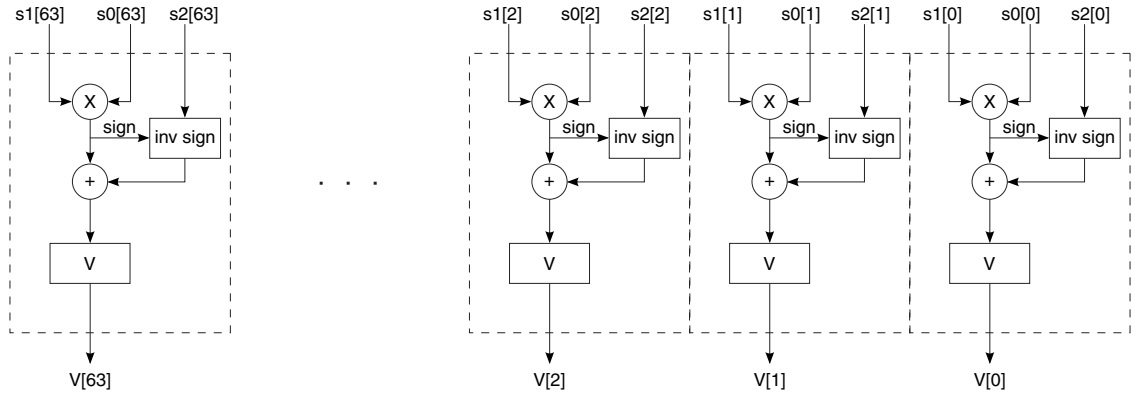


Figure 26. mads Operation in Real Mode (Single Precision)

4.13.5 Multiply accumulate functionality

Table 49. MAC Descriptions

Instructions	Real/ Complex Modes	AUprec	Operations	Descriptions

rmac	Real	Single	for (i=0; i<64; i++) {
			V[i][n] = (S0[i][n-4]*S1[i][n-4]) + V[i][n-1]
			}
				Perform 64 parallel multiply-accumulate operations on S0/S1 elements.

cmac	Complex	Single	for (i=0; i<64; i +=2) {
			V[i][n] = (S0[i][n-4]*S1[i][n-4]) + (S0[i+1][n-4]*S1[i+1][n-4]) + V[i][n-1]
			}
				This can perform 16 complex multiply-accumulate operations, where V[0] represent the real accumulators and V[2] represent the imaginary accumulators.

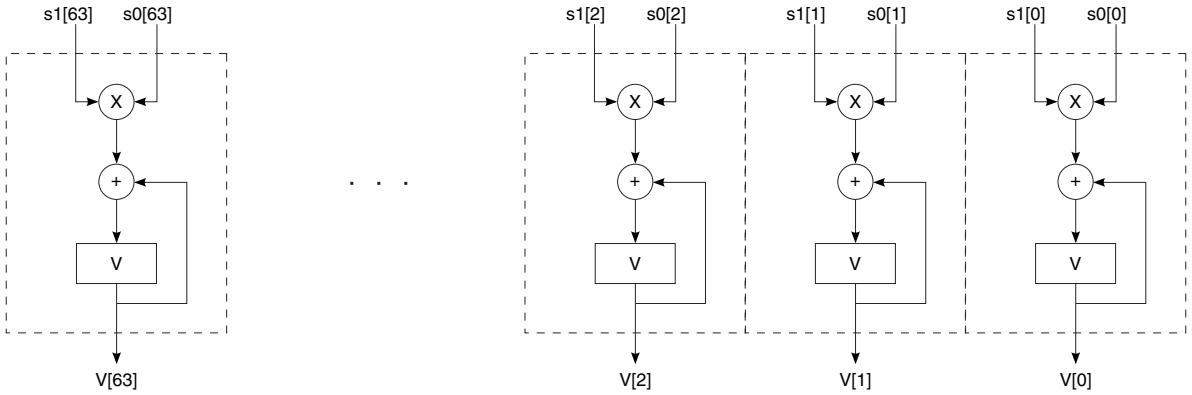


Figure 27. Real mac Operation (Single Precision)

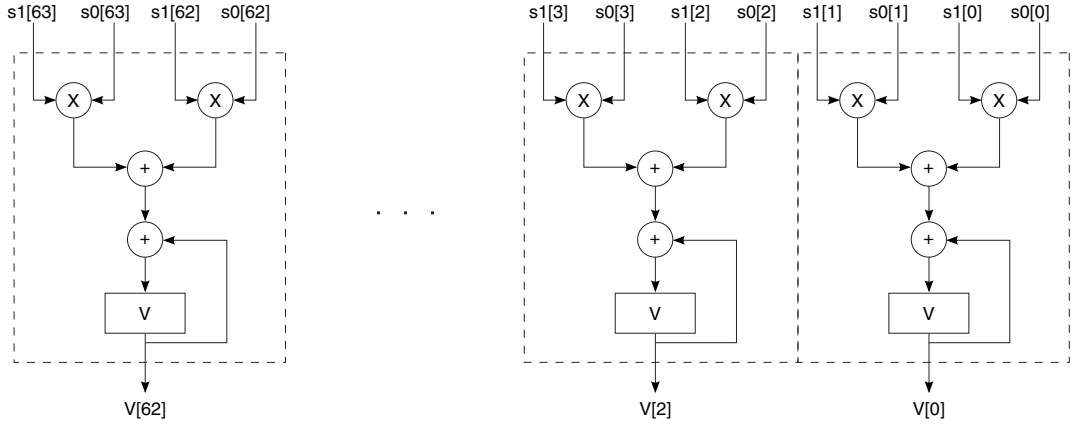


Figure 28. Complex mac Operation (Single Precision)

RMAC code example

```
// a0 = output pointer; a1 = input pointer for a; a2 = input pointer for b; a3
= input pointer for c
// performs a + b*c
set.creg 19, 1;        // set VAU output width to 2 lines
clr.VRA;            // reset VRA pointers
set.VRAptr 64*2, 64*4, 0, 64*6, 6;      // Set VRA pointers to rS0 = R2[0]; rS1
= R4[0]; rS2 = R0[0]; rV = R6[0]; rSt = R6
ld.laddr [a1]+1;                  // Load "a" first 32 elements

ld.laddr [a1]+1;                  // Load "a" second 32 elements
ld.laddr [a2]+1;               // Load "b" first 32 elements
ld.laddr [a2]+1;
ld.normal R0;                  // Load "b" second 32 elements
ld.laddr [a3]+1;
ld.normal R1;                  // Load "c" first 32 elements
ld.laddr [a3]+1;
ld.normal R2;                  // Load "c" second 32 elements

ld.normal R3;

ld.normal R4;
ld.normal R5;
set.Smode S0straight, S1straight, S2straight;
rd s0; rd S1; rd s2;
```

```
// set Smodes to load "direct"
nop;
rmad;
nop;
nop;
nop;
wr.straight;                        // writes output to R6-R7
setB.VRAptr rSt, 7;
st.laddr [a0]+1;              // store output first 32 elements
st.laddr [a0]+1;              // store output second 32 elements
```

4.13.6 Multiply add with feedback functionality

Table 50. MAF Descriptions

Instructions	Real/ Complex Modes	AUprec	Operations	Descriptions

maf	Real	Single	for (i=0; i<64; i++) {
			V[i][n] = (V[i][n-4]*S1[i][n-4]) + S2[i][n-4]
			}
				Perform 64 parallel multiply-accumlate with feedback operations with S1 and S2 elements.

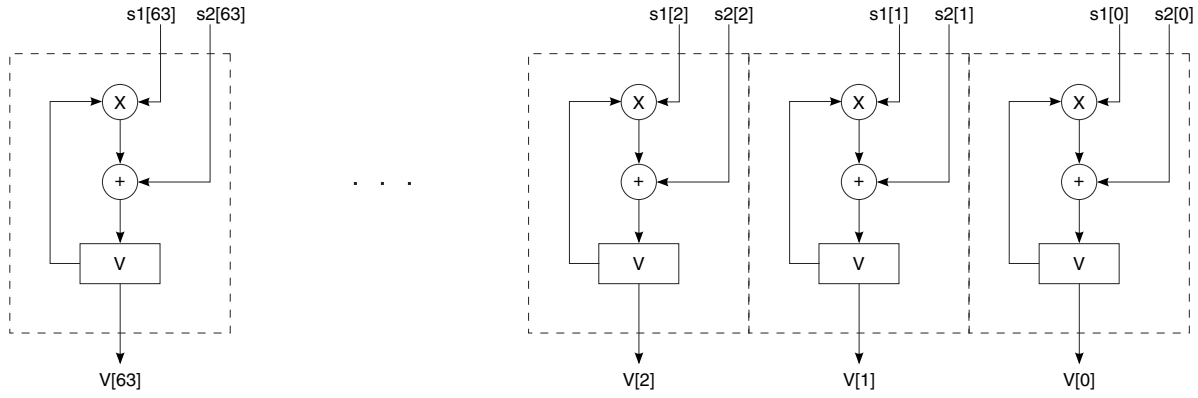


Figure 29. maf Operation (Single Precision)

4.13.7 Decimation in time and frequency butterfly functionality

Table 51. Decimation Descriptions

Instructions	Real/ Complex Modes	AUprec	Operations	Descriptions
cmad	Complex Only	Single Precision Only	for (i=0; i<64; i +=2) {	Perform 32 parallel multiply-add-add operations on S0/S1/S2 elements.

Table continues on the next page...

Table 51. Decimation Descriptions (continued)

Instructions	Real/ Complex Modes	AUprec	Operations	Descriptions
			V[i][n] = (S0[i][n-4]*S1[i][n-4]) + (S0[i+1][n-4]*S1[i+1][n-4]) + S2[i][n-4] }	With S0 and S2 in a FFT mode, S1 in NC0 mode and with NCO in nco_mode=radix2, this can perform 16 complex DIT butterfly operations, where V[0] represent the real outputs and V[2] represent the imaginary outputs.

dif.sau	Complex Only	Single Precision Only	for (i=0; i<64; i +=4) {
			V[i][n]=(S0[i][n-4]*SAUout[i][n-4]) + (S0[i+1][n-4]*SAUout[i+1][n-4]) + (SAUout[i+2][n-4]*S2[i+2][n-4]) + (SAUout[i+3][n-4]*S2[i+3][n-4]);
			V[i+2][n]=S0[i+2][n-4] + S2[i+2][n-4];
			}
				Perform 32 parallel FFT-specific operations on S0/S1/S2 elements.
				With S0 and S2 in a FFT mode, S1 in NC0 mode and with NCO in nco_mode=radix2, this can perform 16 complex DIF butterfly operations, where V[0] represent the real outputs and V[4] represent the imaginary outputs.

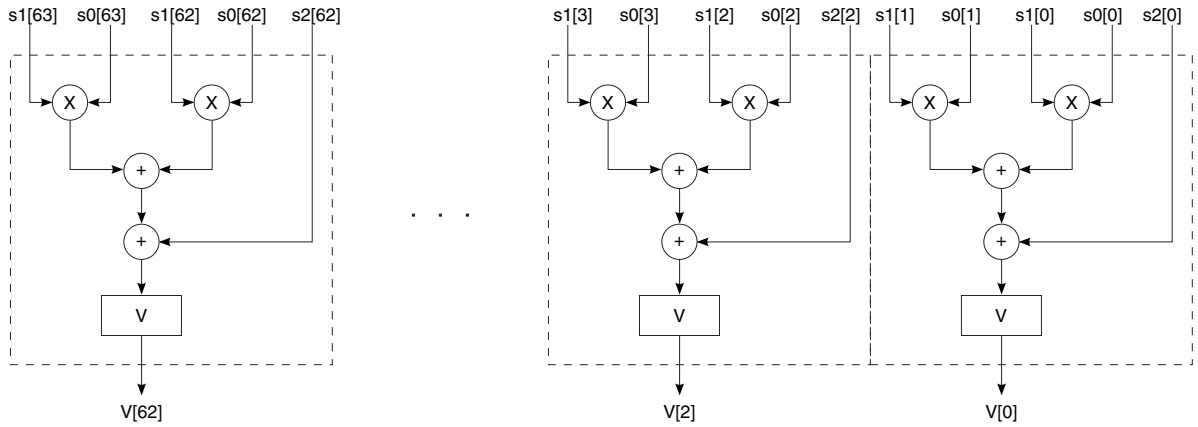


Figure 30. cmad Operation (Single Precision)

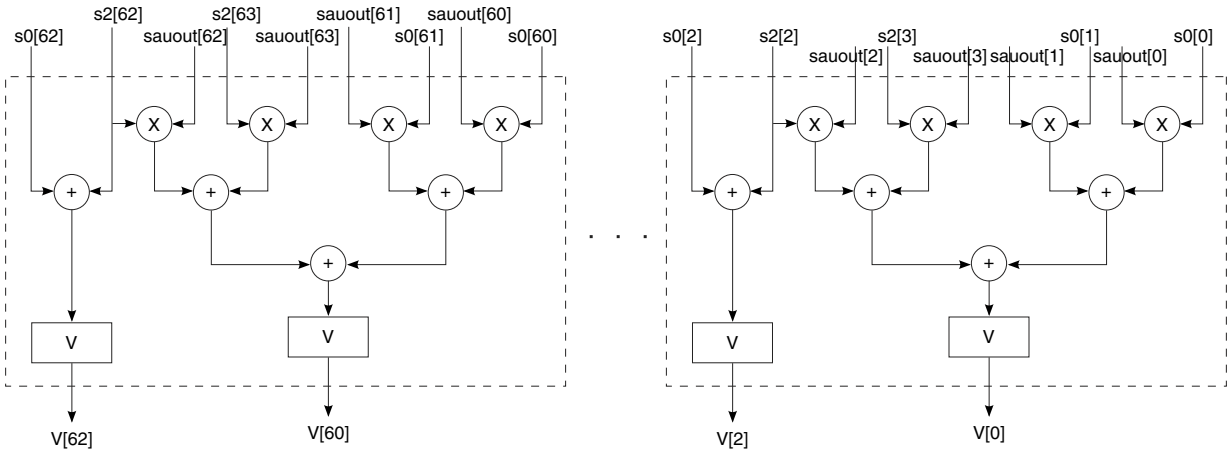


Figure 31. dif Operation (Single Precision)

4.14 Special AU instructions

This group of instructions specifies special AU arithmetic operations.

The SAU instructions calculate non-linear functions. The input is either 32-bit floating point data or 16-bit fixed point data. The output is either one data point (for real calculation) or two data points (for complex calculations). The SAU output is always in single precision format. The 16-bit half fixed input data is between -1 and 1 in sign-magnitude format. 32-bit floating point data is in single precision format. Certain functions ignore the input data sign. Built-in lookup tables (LUTs) are used to look up data points that are then used in the SAU interpolator logic to calculate one or two result data points. The result is provided as the SAU output data. The LUT instruction is a part of the SAU functionality.

- rcp - reciprocal on 32 elements
- rrt - reciprocal square root on 32 elements
- srt - square root on 32 elements
- nco - generates complex exponential sequences
- padd - adds 2 vectors of half-fixed numbers as loaded from the VRA into s2 and s1

Table 52. SAU Instructions

Mnemonic	Function Performed
rcp	V[i*2] = 1/s1[i*2] SAUout[i*2] = 1/s1[i*2] SAUout[i*2+1] = 1/s1[i*2] for i=0,1,..,31 input: single precision output: single precision
rrt	V[i*2] = 1/√|s1[i*2]| SAUout[i*2] = 1/√|s1[i*2]| SAUout[i*2+1] = 1/√|s1[i*2]|

Table continues on the next page...

Table 52. SAU Instructions (continued)

Mnemonic	Function Performed
	for i=0,1,..,31 input: single precision output: single precision
srt	V[i*2] = √|s1[i*2]| SAUout[i*2] = √|s1[i*2]| SAUout[i*2+1] = √|s1[i*2]| for i=0,1,..,31 input: single precision output: single precision
nco	See Numerically controlled oscillator (NCO) instructions
padd	V[i]=s2[i]+s1[i] SAUout[i] = s2[i]+s1[i] for i=0,1,...,63 input: half-fixed precision output: single or F24 precision

4.14.1 SAU input and output vector

The SAU uses the same input and output buses as the AU.

4.14.2 SAU latency

After the number of cycles defined by the latency of an SAU instruction, the SAU output will exist on the even indices of the AU output bus and can be written into the VRA. For example, for two cycle latency, from the source register load, through the SAU execution and into the VRA, will take 5 cycles; 2 for the source register load, 2 for the sau instruction, 1 for the SAU writeback.

There must be a minimum of one instruction between a "rd S1" operation and a SAU instruction. See Special AU instructions code example.

4.14.3 Special AU instructions code example

```
set.prec half_fixed, half_fixed, half_fixed, single, half_fixed;
set.Smode S1r2c; rd S1;
nop; // a minimum of one instruction needed between rd S1 and rcp/rrt/srt operations
rcp;
nop;
wr.fn;
```

4.14.4 Reciprocal functionality

Table 53. Reciprocal Descriptions

Instruction	Real/Complex Mode	AUprec	Operation	Description
rcp	Real or Complex	Single	for (i=0; i<64; i+=2) V[i] = 1/s1[i];	Calculate the reciprocal of all real elements of s1. **NOTE** The result of rcp is an approximation

4.14.5 Reciprocal square root functionality

Table 54. Reciprocal Square Root Descriptions

Instruction	Real/Complex Mode	AUprec	Operation	Description
rrt	Real or Complex	Single	for (i=0; i<64; i+=2) V[i] = 1/√|s1[i]|;	Calculate the reciprocal square root of all real elements of s1. **NOTE** The result of rrt is an approximation.

4.14.6 Square root functionality

Table 55. Square Root Descriptions

Instruction	Real/Complex Mode	AUprec	Operation	Description
srt	Real or Complex	Single	for (i=0; i<64; i+=2) V[i]= √|s1[i]|;	Calculate the square root of all real elements of s1. **NOTE** The result of srt is an approximation.

4.14.7 Pre adder functionality

Table 56. Pre adder Descriptions

Instruction	Real/Complex Mode	AUprec	Operation	Description

padd	Real or Complex	padd	for (i=0; i<64; i+=1)
			{
			V[i]=s2[i]+s1[i]
			}
				Adds 2 vectors of half-fixed numbers as loaded from the VRA into s2 and s1. Under “padd” mode of AU precision no type conversion happens on the source register load and the inputs are always treated as half-fixed numbers. The output vector from the pre-adder is available at the SAU output either for use by any of rmac.sau/ cmac.sau/rmad.sau/cmad.sau instruction or for writing to VRA. If written back to the VRA only half of the output vector (every alternate output sample) is written back.

padd	Real or Complex	Single/Double	Not applicable	-

4.15 Store AU/SAU output instructions

The *wr* instruction stores the AU or SAU output into a VRA register.

As part of the VRA writeback operation, a muxing logic block called WbMux performs a variety of shifting functions on the AU/SAU results before writing them back into the VRA.

Table 57 shows all *wr* instructions and their supported muxing modes for WbMux. These modes dictate which shifting functions are to be performed by the WbMux prior to writing back to VRA.

Table 57. Store AU and Store SAU Instructions

Instruction	Description
wr.even;	Write AU output into VRA. Every other output shifting is performed at WbMux.
wr.fftn;	Write AU output into VRA. Perform shift function at WbMux for FFT nth stage.
wr.fft5;	Write AU output into VRA. Perform shift function at WbMux for FFT 5th stage.
wr.fft4;	Write AU output into VRA. Perform shift function at WbMux for FFT 4th stage.
wr.fft3;	Write AU output into VRA. Perform shift function at WbMux for FFT 3rd stage.
wr.fft2;	Write AU output into VRA. Perform shift function at WbMux for FFT 2nd stage.
wr.fft1;	Write AU output into VRA. Perform shift function at WbMux for FFT 1st stage.
wr.fn;	Write SAU output into VRA. No special shifting function is performed at WbMux.

Table continues on the next page...

Table 57. Store AU and Store SAU Instructions (continued)

wr.fn1;	Write single SAU output into VRA. Output shifting is performed at WbMux.
wr.straight;	Write AU output into the VRA. No special shifting function is performed at WbMux.
wr.hlinecplx;	Write AU output into the VRA. No special shifting function is performed at WbMux.

4.15.1 VAU data type conversion

The data from the Vector Arithmetic Unit can be type converted before being stored into the VRA. The AUprec determines the data type of the Vector Arithmetic Unit. The Vprec determines the final data type stored in the VRA.

The diagrams that follow show bit width sizes. Each size represents the number of bits that are used before type conversion and the resulting number of bits after type conversion. For example, in a single precision to half fixed conversion 2048 bits are used before conversion and the conversion results in 1024 bits.

The valid AUprec to Vprec type conversions are shown in the following figures:

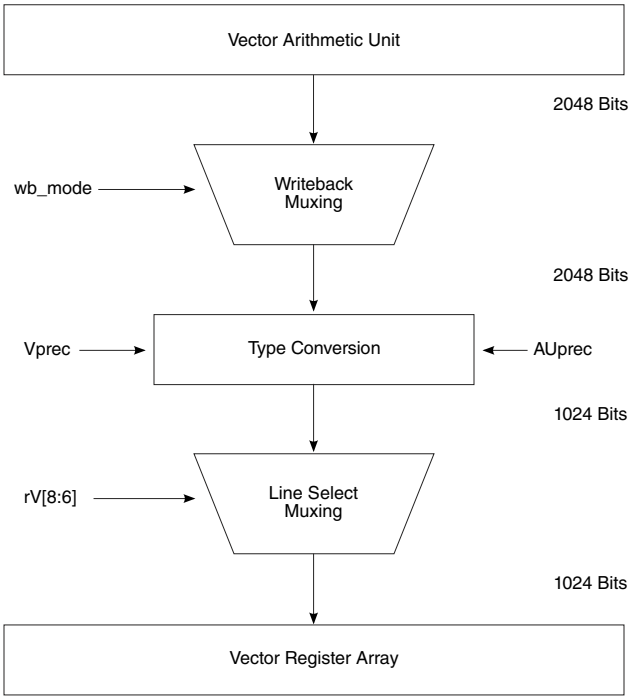


Figure 32. Single Precision to Half Fixed VRA store

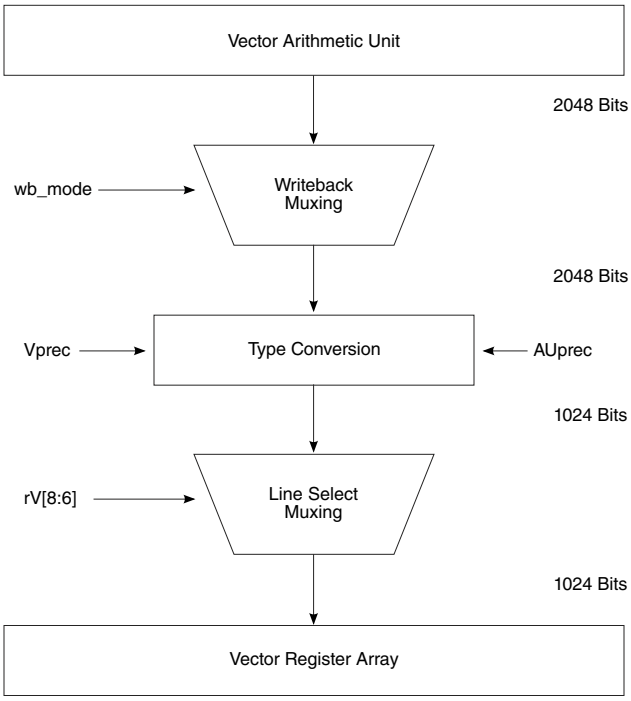


Figure 33. Single Precision to Half Precision VRA store

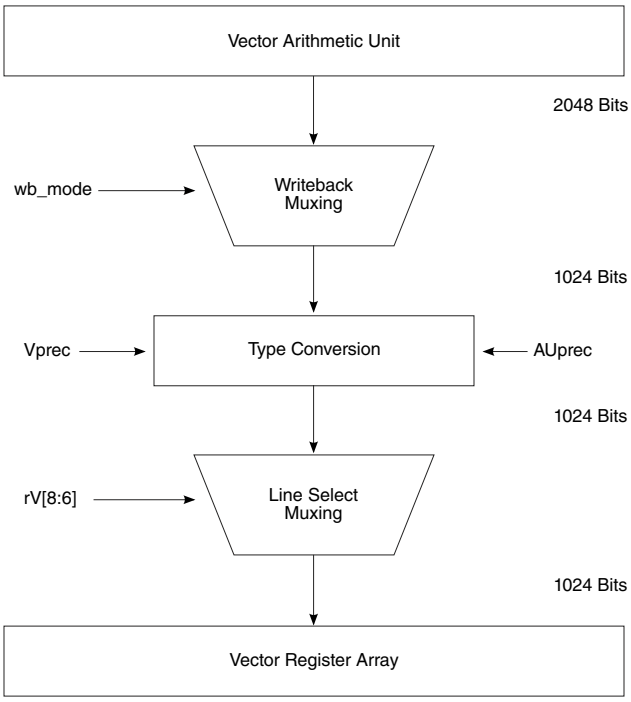


Figure 34. Single Precision to Single Precision VRA store

Descriptions

- In the following table, R*x* denotes the VRA register to which the data is written.

- To select a VRA register, the VRA array is indexed by the upper three bits of the rV register.
- V, R*x*, are assumed to be vectored buses. Each vectored bus contains 64 elements if AUprec is single precision.
- Q*x* represents a quarter of the elements in the register. For example, Q2 is the third quarter of elements in the register.
- All the R*x* values come from before the type conversion. Refer to VAU data type conversion for how the mux results end up in V after type conversion. The calculations shown are assuming type conversion to half-fixed/half. Also, refer creg 19/20 in Table 21

Table 58. All Supported Vmode, Write-back Modes

Instruction	Restrictions	Operation	Description

wr.fn;	Real &
	Complex data
	creg(15[2])=0, creg(25), creg(21), creg(19)
		R*x* = VRA [rV[8:6]];
		for (i=0,j=0;j < 64;j =+2,i++) {
		R*x* [i] = V [j];
		}
		update_rag(rV);
			Store the vector output of the SAU into the VRA. Only the even elements of V are used.
			Update rV.

wr.fn1;	Real & Complex data creg(15[2])=0, creg(25), creg(21), creg(19)	Rx [rV[5:0]] = V [0]; update_rag(rV);	Store the first SAU output into a line of register a VRA register pointed to by rV. Update rV.

wr.even;	Real data
	creg(15[2])=0, creg(19)
		R*x* = VRA [rV[8:6]];
		for (i=0,j=0;j < 64;j +=2,i++) {
		R*x* [i] = V [j];
		}
		update_rag(rV);
			Store the output of AU complex operation into the vector register array. Only the even elements of V are used.
			Update rV.
			Note that, wr.even only works for half_fixed or half_float precisions.

wr.fftn;	Complex data only,
	Vprec=half-fixed/half/ single
	creg(15[2])=0, creg(19)
		R*x* = VRA [rV[8:6] & 0x6];
		R*x*+1 = VRA [rV[8:6] & 0x6 + 1];
		switch (rV[5:4]) {
		// store to Q0 of R*x*, R*x*+1
		0: i_list = {0,...,15};
		// store to Q1 of R*x*, R*x*+1
		1: i_list = {16,...,31};
		// store to Q2 of R*x*, R*x*+1
		2: i_list = {32,...,47};
		// store to Q3 of R*x*, R*x*+1
		3: i_list = {48,...,63};
		j_list = {2,6,...,58,62};
		foreach (i, j) in (i_list , j_list) {
			Store the output of AU fft operation into vector register array.
			This mode is intended for DIF FFT stage 8 and for DIT FFT stage *N*-7.
			Only supports an even register as the starting register. That is, R*x* = R0, R2, R4, or R6
			R*x* is the output for the top leg of butterfly, while R*x*+1 is the output of the bottom one.
			Update rV.

Table continues on the next page...

Table 58. All Supported Vmode, Write-back Modes (continued)

Instruction	Restrictions	Operation	Description

		Rx [i] = V [j];
		Rx+1[i] = V [j - 2];
		}
		update_rag(rV);	

wr.fft5;	Complex data only,
	Vprec=half-fixed/half
	creg(15[2])=1, creg(19)
		Rx = VRA [rV[8:6]];
		if (rV[5]== 0)
		i_list = {0-31}
		else
		i_list = {32-63}
		j_list = {2,6,...,58,62};
		foreach (i, j) in (i_list , j_list) {
		Rx [i] = V [j];
		Rx [i + 32] = V [j - 2];
		}
		update_rag(rV);
			Store the output of AU fft operation into the vector register array.
			This mode is intended for DIF FFT stage 5 and for DIT FFT stage *N*-4.
			Update rV.

wr.fft4;	Complex data only,
	Vprec=half-fixed/half/ single
	creg(15[2])=1, creg(19)
		Rx = VRA [rV[8:6]];
		if (rV[5]== 0)
		i_list = {0-15,..., 16-31}
		else
		i_list = {32-47,..., 48-63}
		j_list = {2,6,...,58,62};
		foreach (i, j) in (i_list , j_list) {
		Rx [i] = V [j];
		Rx [i + 16] = V [j - 2];
		}
		update_rag(rV);
			Store the output of AU fft operation into the vector register array.
			This mode is intended for DIF FFT stage 4 and for DIT FFT stage *N*-3.
			Update rV.

wr.fft3;	Complex data only,
	Vprec=half-fixed/half/ single
	creg(15[2])=1, creg(19)
		Rx = VRA [rV[8:6]];
		if (rV[5]== 0)
		i_list = {0-7,..., 24-31}
		else
		i_list = {32-39,..., 56-63}
		j_list = {2,6,...,58,62};
		foreach (i, j) in (i_list , j_list) {
			Store the output of AU fft operation into the vector register array.
			This mode is intended for DIF FFT stage 3 and for DIT FFT stage *N*-2.
			Update rV.

Table continues on the next page...

Table 58. All Supported Vmode, Write-back Modes (continued)

Instruction	Restrictions	Operation	Description

		Rx [i] = V [j];
		Rx [i + 8] = V [j - 2];
		}
		update_rag(rV);	

wr.fft2;	Complex data only,
	Vprec=half-fixed/half/ single
	creg(15[2])=1, creg(19)
		Rx = VRA [rV[8:6]];
		if (rV[5]== 0)
		i_list = {0-3,..., 28-31}
		else
		i_list = {32-35,..., 60-63}
		j_list = {2,6,...,58,62};
		foreach (i, j) in (i_list , j_list) {
		Rx [i] = V [j];
		Rx [i + 4] = V [j - 2];
		}
		update_rag(rV);
			Store the output of AU fft operation into the vector register array.
			This mode is intended for DIF FFT stage 2 and for DIT FFT stage *N*-1.
			Update rV.

wr.fft1;	Complex data only,
	Vprec=half-fixed/half/ single
	creg(15[2])=1, creg(19)
		Rx = VRA [rV[8:6]];
		if (rV[5]== 0)
		i_list = {0-1,..., 30-31}
		else
		i_list = {32-33,..., 62-63}
		j_list = {2,6,...,58,62};
		foreach (i, j) in (i_list , j_list) {
		Rx [i] = V [j];
		Rx [i + 2] = V [j - 2];
		}
		update_rag(rV);
			Store the output of AU fft operation into the vector register array.
			This mode is intended for DIF FFT stage 1 and for DIT FFT stage *N*.
			Update rV.

wr.straight	Real data
	creg(15[2])=0, creg(19)
		Rx = VRA [rV[10:8]];
		for (i=0;i<64; i++) {
		Rx [i] = V [i];
		}
		update_rag(rV);
			Store the output of AU real operation into the vector register array.
			Update rV.

Table continues on the next page...

Table 58. All Supported Vmode, Write-back Modes (continued)

Instruction	Restrictions	Operation	Description

wr.hlinecplx	Complex data
	creg(15[2])=x, creg(19)
		Rx = VRA [rV[10:8]];
		for (i=0,j=0;j <64;j =+2,i++) {
		Rx [i] = V [j];
		}
		update_rag(rV);
			Store the output of AU complex operation into vector register array. Only the even elements of V are used.
			Update rV.

NOTE

The wr.fft<x> mirrors the behavior of the corresponding S2fft<x> and S0fft<x> modes. That is, for FFT butterfly inputs chosen from columns of VRA based on S2 and S0 pointers, the outputs are written to the same columns, with a possible quarter or half VRA register shift, (upper butterfly output to S2 columns, lower butterfly output to S0 columns) as the input S0/S2 modes.

4.16 GP instructions

The General Purpose Arithmetic Unit (GP Unit) performs some commonly used arithmetic, logical and data format conversion operations on twelve General Purpose Registers (GP Registers): g0-g11. These GP registers are each 32 bit wide.

4.16.1 GP move instructions

The gp move instructions allows data movement between GP registers and various hardware resources within VSPA:

- some VRA elements,
- hardware registers.
- VSPA scalar registers,
- IP registers,

4.16.1.1 Move vector register array instructions

Table 59 describes the syntax and operations of instructions which move data between the GP registers and the vector register array (VRA). These are all OpC family instructions.

Table 59. GP Move VRA Instructions

Move Data Between	Instruction Syntax	Operation	Description
A GP register and a VRA element	mv.h g*X*, [rS0];	g*X*[15:0] <- [rS0] update_rag(rS0);	Move a 16-bit real element from VRA, pointed to by rS0, to g*X*. rS0 is then updated.
	mv.w g*X*, [rS0];	g*X*[15:0] <- [rS0] g*X*[31:16] <- [rS0+1] update_rag(rS0);	Move a 32-bit complex element from VRA, pointed to by rS0, to g*X*. rS0 is then updated.
	mv.h [rV], g*X*;	[rV] <- g*X*[15:0]; update_rag(rV);	Move the lower 16 bits of g*X* to an element in Rx, pointed to by rV. rV is then updated.

Table continues on the next page...

Table 59. GP Move VRA Instructions (continued)

Move Data Between	Instruction Syntax	Operation	Description
	mv.w [rV], g*X*;	[rV] <- g*X*[15:0]; [rV+1] <- g*X*[31:16]; update_rag(rV);	Move the 32 bit contents of g*X* to an element in Rx, pointed to by rV. rV is then updated.

See VRA pointer control registers for information on RAG ptr updates.

4.16.1.2 Move hardware register instructions

Table 60 shows a list of supported hardware registers. Among these hardware registers, some are read-only, some are write-only and some are both readable and writeable. These are all OpC family instructions.

If a hardware register is readable, a mv from the hardware register to a GP register can be performed.

If a hardware register is writeable, a mv from a GP register to the hardware register can be performed.

Table 60. Hardware Registers Accessible by *mv* Instruction

Hardware Register <hw_reg>	Description	Related Blocks	No of Bits	Readable	Writeable	Related Instructions
nco_k	NCO accumulator	NCO	16	Yes	Yes	mv nco_k, g*X* mv g*X*, nco_k
nco_phase	NCO phase register		16	Yes	Yes	mv nco_phase, g*X* mv g*X*, nco_phase
nco_freq	NCO freq register		32	No	Yes	mv nco_freq, g*X*

4.16.1.3 Move scalar registers

Table 61 shows the instructions which move data between the various VSPA scalar registers. These registers include GP registers, address (and address storage registers), stack pointer, condition code bits and some special internal holding registers.

Table 61. Scalar registers accessible by mv instruction

Move Data Between	Family	Instruction Syntax	Operation	Description
Address/GP and address/GP	OpB	mvB ag*X* ag*Y*	ag*X*<-ag*Y*	Move ag*Y* into ag*X*
	OpS	mvS ag*X* ag*Y*	ag*X*<-ag*Y*	Move ag*Y* into ag*X*
Address/GP and stack pointer	OpS	mvS ag*X*, sp	ag*X*<-sp	Move stack pointer into ag*X*
		mvS sp, ag*X*	sp<-ag*X*	Move ag*X* into stack pointer
	OpB	mvB ag*X*, sp	ag*X*<-sp	Move stack pointer into ag*X*
		mvB sp, ag*X*	sp<-ag*X*	Move ag*X* into stack pointer

Table continues on the next page...

Table 61. Scalar registers accessible by mv instruction (continued)

Move Data Between	Family	Instruction Syntax	Operation	Description
GP register and GP register	OpS	mv(.cc) g*Y* g*X*	g*Y*<-g*X*	Move g*X* into g*Y*
GP register and stack pointer	OpS	mv(.cc) g*Y* sp	g*Y*[31:17] <- 0 g*Y*[16:0] <- sp	Move stack pointer into the lower 17 bits of g*Y* and zeros into the upper 15 bits of g*Y*
		mv(.cc) sp g*X*	sp <- g*X*[16:0]	Move the lower 17 bits of g*X* into stack pointer
GP register and condition code bits	OpS	mv g*Y* cc	g*Y*[31:16]<-0 g*Y*[15:4]<-creg g*Y*[3:0] <- CC	Move CC bits into the lower 4 bits of g*Y*. The upper 28 bits of g*Y* will be loaded with other system control register data. See System control registers for information on which bits will be loaded into the GP destination register.
		mv cc g*X*	cc <- g*X*[3:0]	Move the lower 4 bits of g*X* into CC bits
Internal register to GP register	OpS	mv.cc g*Y*, quot	g*Y*<-internal(quotient value)	Move signed/unsigned integer quotient result from prior "quot" or "rem" instruction.[1]
		mv.cc g*Y*, rem	g*Y*<-internal(remainder value)	Move signed/unsigned integer remainder result from prior "quot" or "rem" instruction.[1]
		mv.cc g*Y*, pc	g*Y*<-PC	Move current program counter into GP destination register.

1. Signed/unsigned quotient and remainder values are held in internal (hidden) registers until completion of next "quot" or "rem" instruction.

4.16.1.4 Move IP instruction

Table 62 shows a list of instructions which move data between the GP registers and the memory mapped IP registers.

Table 62. mvip instructions

Move Data Between	Family	Instruction Syntax	Operations	Descriptions
GP registers and memory mapped IP registers	opD	mvip Iu9 Iu32	ip[Iu9]<-Iu32	Move the 32 bit immediate data into IP register Iu9

Table continues on the next page...

Table 62. mvip instructions (continued)

Move Data Between	Family	Instruction Syntax	Operations	Descriptions
		mvip Iu9 gX, Iu32	ip[Iu9]<-(gX&Iu32)|(ip[Iu9]&~Iu32)	Move gX register data into IP register Iu9. Only gX bits with the corresponding bits in Iu32 set to one are written to the corresponding IP[Iu9] bits. IP[Iu9] bits with the corresponding bits in Iu32 cleared (zero) are not written and keep previous value.
		mvip gX Iu9, Iu32	gX<-(ip[Iu9]&Iu32)	Move IP register Iu9 data anded with Iu32 into gX register

4.16.2 Linear feedback shift register instructions

Table 63. lfsr instructions

Instruction Syntax	Operation	Description
lfsr g*X*, g*Y*	g*X* = (g*X*[30:0] << 1) | (g*Y*[0] ^ g*Y*[1] ^ g*Y* [2] ^... ^ g*Y*[31]);	Shift g*X* left by one bit, then replace bit 0 of g*X* with the result of the following operation: `gY[0] ^ gY[1] ^ gY[2] ^...^ gY[31]`
lfsr g*X*, Iu32	g*X* = (g*X*[30:0] << 1) | (Iu32[0] ^ Iu32[1] ^ Iu32 [2] ^... ^ Iu32[31]);	Using the 32-bit immediate data for the exclusive or (XOR) operation, shift g*X* left by one bit, then replace bit 0 of g*X* with the result of the following operation: `Iu32[0] ^ Iu32[1] ^ Iu32[2] ^ ...^ Iu32[31]`

4.16.3 Floating point generation instructions

Table 64. lsb2rf instructions

Instruction Syntax	Operation

lsb2rf [rV], g*X*	if (g*X*[0] == 0)
	[rV] = 0;
	else
	[rV] = 0.5;
	Update rV.

lsb2rf.sr [rV], g*X*	if (g*X*[0] == 0)
	[rV] = 0;
	else

Table continues on the next page...

Table 64. lsb2rf instructions (continued)

Instruction Syntax	Operation
	[rV] = 0.5; Update rV. g*X* = g*X* >> 1;

The *lsb2rf* instruction converts to half-fixed format, so it writes a half-fixed 0 or a half-fixed 0.5 (0x4000) into a real element in R*x* using rV as the element pointer, depending on the LSB of g*X* (or g*X*[0]).

If g*X*[0] is 0, a zero will be written into R*x*[rV]. Otherwise, a half-fixed 0.5 (0x4000) will be written into R*x*[rV]. The rV is then updated (see VRA pointer control registers for more details on rV updates).

The *lsb2rf.sr* instruction performs the same operations as the *lsb2rf* instruction, as well as a one-bit logical right-shift on g*X*.

4.16.4 Arithmetic instructions

Table 65. Arithmetic Instructions

Opera-tion	Family	CC[1]	Syntax	Operation	Description
add	opD	creg	addD(.ucc)(.cc) g*X*,g*Y*,I32	g*X*<-g*Y*+Iu32	If the optional condition test is 'true', then add an immediate scalar to g*Y* and store the result in g*X*. If .ucc bit of instruction is set, update condition codes accordingly.
add	opS	creg	addS(.ucc).z g*Z*,Iu16	g*Z*<-g*Z*+{16'h000,Iu16}	Add Zero extended 16 bit integer to g*Z* and store result back into g*Z*. If .ucc bit of instruction is set, update condition codes accordingly.
			addS(.ucc).s g*Z*,Is16	g*Z*<-g*Z*+{{16{Is[15]}},Is16}	Add Sign extended 16 bit integer to g*Z* and store result back into g*Z*. If .ucc bit of instruction is set, update condition codes accordingly.
			addS(.ucc)(.cc) g*Z* g*X*,g*Y*	g*Z*<-g*X*+g*Y*	If the optional condition test is 'true', then add contents of g*X* with the contents of g*Y* and store result in g*Z* if optional condition is 'true'. If .ucc bit of instruction is set, update condition codes accordingly.
cmp	opD	always	cmpD(.cc) g*X*,I32	g*X*-I32	If the optional condition test is 'true', then subtract 32 bit integer from g*X* and update condition codes accordingly.
cmp	opC	always	cmp a*X*,Iu19	aX-Iu19	Subtract 19 bit integer from a*X* memory pointer and update condition codes accordingly.

Table continues on the next page...

Table 65. Arithmetic Instructions (continued)

Operation	Family	CC[1]	Syntax	Operation	Description
cmp	opS	always	cmpS.z g*Z*,Iu16	g*Z*-{16'h0000,Iu16}	Subtract Zero extended 16 bit integer from g*Z* and update condition codes accordingly.
			cmpS.s g*Z*,Is16	g*Z*-{{16{Is[15]}},Is16}	Subtract 16 bit sign-extended integer from g*Z* and update condition codes accordingly.
			cmp(.cc) g*X*, g*Y*	g*X*-g*Y*	If the optional condition test is 'true', then subtract contents of g*Y* from contents of g*X* and update condition codes accordingly.
sub	opD	creg	subD(.ucc)(.cc) g*X*,g*Y*,I32	g*X*<-g*Y*-Iu32	If the optional condition test is 'true', then subtract 32 bit integer from g*Y* and store result back into g*X*. If .ucc bit of instruction is set, update condition codes accordingly.
sub	opS	creg	subS(.ucc).z g*Z*,Iu16	g*Z*<-g*Z*-{16'h0000,Iu16}	Subtract Zero extended 16 bit integer from g*Z* and store result back into g*Z*. If .ucc bit of instruction is set, update condition codes accordingly.
			subS(.ucc).s g*Z*,Is16	g*Z*<-g*Z*-{{16{Is[15]}},Is16}	Subtract Sign extended 16 bit integer from g*Z* and store result back into g*Z*. If .ucc bit of instruction is set, update condition codes accordingly.
			subS(.ucc)(.cc) g*Z* g*X*,g*Y*	g*Z*<-g*X*-g*Y*	If the optional condition test is 'true', then subtract contents of g*Y* from contents of g*X* and store result into g*Z* if optional condition is 'true'. If .ucc bit of instruction is set, update condition codes accordingly.
rsub	opS	creg	rsub.z g*Z*,Iu16	g*Z*<-{16'h0000,Iu16}-g*Z*	Subtract g*X* from Zero extended 16 bit integer and store result back into g*Z*.
			rsub.s g*Z*,Is16	g*Z*<-{{16{Is[15]}},Is16} -g*Z*	Subtract g*Z* from Sign extended 16 bit integer and store result back into g*Z*.
mpy[2]	opS	creg	mpy(.cc)(.s) g*Z* g*X*, g*Y*	g*Z*<-g*X**g*Y*	If the optional condition test is 'true', then multiply contents of g*X* with contents of g*Y* and store the signed result into g*Z*, if optional condition is 'true'.
			mpyS.z g*X*,Iu16	g*Z*<-g*Z**{16'h0000,Iu16}	Multiply contents of g*Z* with Zero extended 16 bit integer and store result back into g*Z*.
			mpyS.s g*X*,Is16	g*Z*<-g*Z**{{16{Is[15]}},Is16}	Multiply contents of g*Z* with Sign extended 16 bit integer and store result back into g*Z*.

Table continues on the next page...

Table 65. Arithmetic Instructions (continued)

Operation	Family	CC[1]	Syntax	Operation	Description
mpy	opD	creg	mpyD(.cc) g*X*,g*Y*,I32	g*X*<-g*Y**I32	If the optional condition test is 'true', then multiply an immediate scalar with g*Y* and store the result in g*X*.
div[3,2]	opS	creg	div(.cc)(.s) g*Z* g*X*, g*Y*	g*Z*<-g*X*/g*Y* internal<-g*X*%g*Y*	If the optional condition test is 'true', then divide contents of g*X* with contents of g*Y* and store the signed truncated integer quotient into g*Z*, if optional condition is 'true'. In addition, the signed remainder (modulus operation) is available in an internal register.
			div.z g*Z*, Iu16	g*Z*<-g*Z*/{16'h0000,Iu16} internal<-g*Z*%16'h0000,Iu16	Divide contents of g*Z* with Zero extended 16 bit integer and store unsigned truncated integer quotient back into g*Z*. In addition, the unsigned remainder (modulus operation) is available in an internal register.
			div.s g*Z*, Is16	g*Z*<-g*Z*/{{16{Is[15]}},Is16} internal<-g*Z*% {{16{Is[15]}},Is16}	Divide contents of g*Z* with Sign extended 16 bit integer and store truncated integer quotient back into g*Z*. In addition, the signed remainder (modulus operation) is available in an internal register.
rdiv[3,2]	opS	creg	rdiv.z g*Z*, Iu16	g*Z*<-{16'h0000,Iu16}/g*Z* internal<- {16'h0000,Iu16}%g*Z*	Divide Zero extended 16 bit integer with contents of g*Z* and store unsigned truncated integer quotient back into g*Z*.In addition, the unsigned remainder (modulus operation) is available in an internal register.
			rdiv.s g*Z*, Is16	g*Z*<-{{16{Is[15]}},Is16}/g*Z* internal<- {{16{Is[15]}},Is16}%g*Z*	Divide Sign extended 16 bit integer with contents of g*Z* and store signed truncated integer quotient back into g*Z*. In addition, the signed remainder (modulus operation) is available in an internal register.
mod[3,2]	opS	creg	mod(.cc)(.s) g*Z* g*X*, g*Y*	g*Z*<-g*X*%g*Y* internal<-g*X*/g*Y*	If the optional condition test is 'true', then divide contents of g*X* with contents of g*Y* and store the signed integer remainder into g*Z*, if optional condition is 'true'. In addition, the signed quotient (divide operation) is available in an internal register.
			mod.z g*Z*, Iu16	g*Z*<-g*Z*%{16'h0000,Iu16} internal<-g*Z*/{16'h0000,Iu16}	Divide contents of g*Z* with Zero extended 16 bit integer and store the unsigned integer remainder back into g*Z*. In addition, the unsigned quotient (divide operation) is available in an internal register.

Table continues on the next page...

Table 65. Arithmetic Instructions (continued)

Operation	Family	CC[1]	Syntax	Operation	Description
			mod.s g*Z*, ls16	g*Z*<-g*Z*%{{16{ls[15]}},ls16} internal<-g*Z*/{{16{ls[15]}},ls16}	Divide contents of g*Z* with Sign extended 16 bit integer and store signed integer remainder back into g*Z*. In addition, the signed quotient (divide operation) is available in an internal register.
rmod[3,2]	opS	creg	rmod.z g*Z*, lu16	g*Z*<-{16'h0000,lu16}%g*Z* internal<-{16'h0000,lu16}/g*Z*	Divide Zero extended 16 bit integer with contents of g*Z* and store unsigned integer remainder back into g*Z*. In addition, the unsigned quotient (divide operation) is available in an internal register.
			rmod.s g*Z*, ls16	g*Z*<-{{16{ls[15]}},ls16}%g*Z* internal<-{{16{ls[15]}},ls16}/g*Z*	Divide Sign extended 16 bit integer with contents of g*Z* and store signed integer remainder back into g*Z*. In addition, the signed quotient (divide operation) is available in an internal register.
abs	opS	creg	abs g*Z*, g*X*	g*Z*<-(g*X*[31]) ? ~g*X*[31:0] +'h1 : g*X*	Take the absolute value of g*X* and store result in g*Z*.
ff1	opS	never	ff1 g*Z*, g*X*	if (g*X*[31]) g*Z*<-31 else if (g*X*[30]) g*Z*<-30 else if (g*X*[29]) g*Z*<-29 . . . else if (g*X*[1]) g*Z*<-1 else if (g*X*[0]) g*Z*<-0 else g*Z*<- -1	Determine the most significant '1' in the value of g*X* and return the bit index in g*Z*. If no bits in g*X* are set -1 (32'hFFFFFFFF) is placed in g*Z*.
fns	opS	never	fns g*Z*, g*X*	if (g*X*[31]) g*Z*<-0 else if (g*X*[30]) g*Z*<-1 else if (g*X*[29]) g*Z*<-2 . . . else if (g*X*[1]) g*Z*<-30 else if (g*X*[0]) g*Z*<-31 else g*Z*<- -1	Determine the number of shifts required to normalize the value stored in g*X* and store that value into g*Z*. "Normalize" in this case refers to shifting the value in g*X* until the most significant '1' is in bit position 31. If no bits in g*X* are set, -1 (32'hFFFFFFFF) is placed in g*Z*.

1. Indicates under what condition the instruction updates the Condition Code bits. Always, never, or CREG (only when System Control Register 4 is set to a 1).
2. A 'done' instruction should not be executed until the div/mod/mpy/mac instruction is complete.
3. Signed/Unsigned quotient ("quot" instruction)/remainder ("rem" instruction) result held until completion of next quot/rem instruction. Refer to Scalar registers accessible by mv instruction

4.16.4.1 Log base 2 instruction

Assembler Syntax

```
log gY, gX;
```

Description

The input of the function is assumed to be in single precision floating point format. The function returns an output that is formatted as a 32 bit fixed point number with structure as shown below:

```
s  iiiiiiiiiiiiiiii.ffffffffffffffff
1  16 integer bits  15 fractional bits
```

4.16.4.2 Fixed to single precision float instruction

Assembler Syntax

`fix2float gX, gY;`

Description

Convert the 16-bit fractional value stored in the lower half of gY to a 32-bit single precision floating point value and store in gX. The short scalar is in 2's complement, assumed to be in the following format:

```
s.ffffffffffffff
1  15 fractional bits
```

4.16.4.3 Single precision float to fixed-point instruction

Assembler Syntax

`float2fix gX, gY;`

Description

Convert the 32-bit single precision floating-point value stored in gY to a 32-bit fixed-point value and store it in gX. The fixed-point value stored in g*X* will be in the following format:

```
s  iiiiiiiiiiiiiiii.ffffffffffffffff
1  16 integer bits  15 fractional bits
```

NOTE

's' indicates a sign bit. The fixed point is in 2's complement.

4.16.4.4 Half fixed to single precision float instruction

Assembler Syntax

`hfixtofloatsp gX, gY;`

Description

Convert the 16-bit half fixed value in VSPA format stored in the lower half of gY to a 32-bit single precision float value and store in gX. See Data precision for a description VSPA's half fixed format.

4.16.4.5 Single precision float to half fixed instruction

Assembler Syntax

`floatsptohfix gX, gY;`

Description

Convert the 32-bit single precision floating point value stored in gY to a 16-bit half fixed value in VSPA format and store in gX. See section 3.3 "Data precision" for a description VSPA's half fixed format.

4.16.4.6 Half precision float to single precision float instruction

Assembler Syntax

```
floathptofloatsp gX, gY;
```

Description

Convert the 16-bit half precision floating point value stored in the lower half of gY to a 32-bit single precision value and store in gX. The 16-bit floating point representation is compliant with IEEE754. See Data precision.

4.16.4.7 Single precision float to half precision float instruction

Assembler Syntax

```
floatsptofloathp gX, gY;
```

Description

Convert the 32-bit single precision floating point value stored in gY to a 16-bit half precision float value and store in gX. The 16-bit floating point representation is compliant with IEEE754. See Data precision.

4.16.4.8 Scale single precision float instruction

Assembler Syntax

```
floatx2n gX, gY;
floatx2n gX, Is8;
```

Description

The first form of the instruction scales the 32-bit single precision floating-point value stored in gX by 2^n, where the integer value n is stored in gY. The second form performs the same operation, except the integer value n is supplied as a short signed immediate value.

4.17 Hardware loop instructions

This group of instructions control all loop executions. The hardware supports up to 8 levels of nested loops. The following table lists all hardware loop instructions.

Table 66. Loop Instructions

Instruction	Description
set.loop	Set iteration count and loop size.
loop_begin	Marks the beginning of a loop. Start the loop executions using the loop size and iteration count setup earlier.
loop_end	Marks the end of a loop. This instruction has no effect on the macro-instruction word, but must be present to indicate the last instruction executed as part of the loop.
loop_stop	Set the current loop count to zero, effectively terminating the current loop execution.
loop_break	Stop loop execution and jump to an immediate address.

A *set.loop* instruction sets the loop size and iteration count of a loop. The loop size is the number of macro-instructions between the *loop_begin* and the *loop_end*, inclusive. See set.loop instruction. This instruction has a two cycle latency, there must be at least one instruction between *set.loop* and a subsequent *loop_begin*. Any *set.loop* which occurs before the *loop_begin*, will be taken as a new *set.loop* for the next *loop_begin* encountered, even if it does not have the required setup time.

A *loop_begin* instruction marks the beginning of a loop. The loop execution begins with the macro-instruction marked with *loop_begin*, using the loop count and iteration count information set up earlier by a *set.loop* instruction.

A *loop_end* instruction marks the end of a loop. The loop execution ends with the macro-instruction marked with *loop_end*, using the loop count and iteration count information set up earlier by a *set.loop* instruction.

The execution body of a loop includes all instruction between, and inclusive of, the *loop_begin* and *loop_end* statements of a loop, plus instructions contained in any subroutine called from the loop. The instruction containing the *loop_end* statement for the loop, plus the two preceding instructions, must be executed consecutively, with no change of flow, for the loop values to be removed from the hardware loop stack and the loop to be terminated correctly.

A *loop_stop* instruction sets the current loop count to zero terminating the current loop execution. A *loop_stop* instruction cannot be placed in the last three instructions of a loop and it must be within the execution body of the loop. A *loop_stop* instruction is typically used in conjunction with a conditional jmp instruction to execute the *loop_stop* instruction.

A *loop_begin* or *loop_end* instruction may be used in parallel with any other instruction, except done or set.loop instruction.

A *loop_break* instruction stops the current loop execution and jumps to an immediate address to continue instruction execution.

4.17.1 Hardware loop control mechanism

Programmers may take advantage of the assembler's capability to generate hardware loops. In the following example, Lstart and Lend are assembly labels:

```
   set.loop iter_count, Lstart, Lend;
```

The Lstart label must reference the first macro-instruction of the loop, which must also contain the *loop_begin* instruction. The Lend label must reference the last instruction of the loop, which must also contain the *loop_end* instruction.

Examples:

```
   Lstart: label for the beginning of the loop
   Lend: label for the end of the loop
```

4.17.1.1 General hardware loop format example

```
   set.loop iter_count, Lstart, Lend;  // The result of 'Lend - Lstart' must be greater
                                       // than or equal to zero, otherwise the assembler
                                       // flags an error
   nop;                 // Minimum 1 intervening instruction
Lstart:
   loop_begin; insn1;   // Parallel instructions are allowed with 'loop_begin'
   insn2;
   insn3;
   ...
   ...
   insn(n-3);
   insn(n-2);
   insn(n-1);
Lend:
   loop_end; insnn;     // Last instruction of the loop
   insn(n+1);           // Not part of the loop body
```

4.17.1.2 Two instruction hardware loop format example

```
   set.loop iter_count, Lstart, Lend; // Result of 'Lend-Lstart' must be greater
                                      // than or equal to zero, otherwise the assembler
                                      // flags an error
   nop;                     // Minimum 1 intervening instruction
Lstart:
   loop_begin; insn1;       // Parallel instructions are allowed with 'loop_begin'
Lend:
```

```
   loop_end; insn2;
   insn3;                   // Not part of the loop body
```

4.17.1.3 One instruction hardware loop format example

```
   set.loop iter_count, Lstart, Lend; // Result of 'Lend-Lstart' must be greater
                                      // than or equal to zero, otherwise the assembler
                                      // flags an error
   nop;                          // Minimum 1 intervening instruction
Lstart:
Lend:
   loop_begin; insn1; loop_end; // Parallel instructions are allowed with 'loop_begin'
   insn2;                        // Not part of the loop body
```

4.17.2 Overwriting a set.loop instruction

A *set.loop* instruction can be overwritten by another *set.loop* instruction.

If a *set.loop* instruction is followed by another *set.loop* instruction without an intervening *loop_begin* instruction, the second *set.loop* instruction will overwrite the first *set.loop* instruction. That is, the loop information provided by the first *set.loop* instruction will be overwritten and lost.

In the example below, a second *set.loop* instruction overwrites the first *set.loop* instruction.

```
 set.loop 4, 2;    // Set up a loop with 2 instructions and 4 iterations
 cmp g0, 0;
 nop;
 jmp.eq cont;
 mv a0, 0;
 mv a2, 1024;
 set.loop 3;     // Overwrite previous set.loop with iter=3.
                 // inst=2 remain unchanged from previous set loop.
 ld [a2]+32;
 ld [a2]+32;
 ld [a2]+32; loop_begin;    // First instruction of the loop. iter is now 3.
 ld [a2]+32; ld.normal R0; Rrot;
 ld.normal R0; rot; loop_end;
 ld.normal R0; rot;
cont:
```

The set.loop instruction may not be pipelined. In the example below, the second set.loop instruction is not legally placed.

```
 set.loop 5,1;    //set.loop for first loop
 nop;             //'op' is any instruction
 set.loop 7,1;    //set.loop for second loop is not legal
 ld a2+32; loop_begin; loop_end;    //first loop: execute 5 times
 ld a2+32; loop_begin; loop_end;    //second loop: execute 7 times
```

4.17.3 Nested hardware loops

Hardware loops can be nested within each other. VSPA supports up to 8 level of loop nesting.

If another *set.loop* and *loop_begin* instruction pair is encountered within a loop body, the outer level loop information (including instruction count and iteration count) are automatically pushed and saved onto a hardware stack, and the loop information (including instruction count and iteration count) specified by the new *set.loop* instruction will be used.

When an inner hardware loop exits from its execution, the loop information at the top of the hardware stack will be popped to retrieve the next outer loop execution, and so on.

Typically, a *set.loop* instruction is paired with a *loop_begin* instruction, with the *loop_begin* typically following the *set.loop* instruction, separated by at least one instruction.

Example:

```
 set.loop 4, 9;                        // Outer loop: 9 inst
 ld [a2]+32;
 ld [a2]+32; loop_begin;               // Start of outer loop
 ld [a2]+32; ld.normal R0;
 st [a3]+32; ld.normal R2;
 set.loop 10, 3; ld.normal R3;         // Inner loop: 3 inst
 st [a3]+32; ld.normal R4;
 st [a3]+32; ld.normal R5; loop_begin; // Start of inner loop
 st [a3]+32; ld.normal R6;
 st [a3]+32; ld.normal R7; loop_end;   // End of inner loop
 st [a3]+32; ld.normal R0; loop_end;   // End of outer loop
```

4.17.4 Early termination of a hardware loop

A loop may be terminated early using a *loop_stop* instruction. This is usually done in conjunction with a conditional *jmp* instruction which will execute the *loop_stop* instruction or skip around it based on the early termination criteria.

A *loop_stop* instruction may also be executed in a sub-routine called from within the loop body, either by a conditional *jsr* to the sub-routine or by conditionally skipping the *loop_stop* instruction within the sub-routine.

The last three instructions of the loop body must execute in consecutive order with no change of flow for the loop to be stopped correctly.

Example: In-line *loop_stop* instruction

```
   set.loop 4,LBEG,LEND;
   cmp g0,0;
   nop;
LBEG:
   jmp.ne NO_OUT; loop_begin;  // Skip the loop_stop instruction, unless g0 == 0
   sub g0,1;
   cmp g0,0;
   loop_stop;                  // This instruction is skipped unless g0 is equal to 0
NO_OUT:
   nop;
   nop;
LEND:
   nop; loop_end;
   nop;
```

Example: The *loop_stop* instruction may be executed in a sub-routine called from the loop

```
   set.loop 4,LBEG,LEND;
   cmp g0,0;
   nop;
LBEG:
   jsr.eq OUT; loop_begin;  // jsr to the loop_stop routine if g0 is equal to 0
   sub g0,1;
   cmp g0,0;
   nop;                    // OUT subroutine returns here to execute the end of the loop
   nop;
LEND:
   nop; loop_end;
   ...
OUT:
   loop_stop; rts;          // End of OUT
   nop;
   nop;
```

4.17.5 Hardware loop execution constraints

The following is a list of hardware loop execution constraints. If these execution constraints are not met, then the hardware loop execution will be non-deterministic. Therefore it is crucial that the programmers strictly follow these constraints. Some of these constraints have already been described in the preceding sections.

1. A *loop_begin* or an *rts* instruction cannot be placed in a delay slot of a prior *jmp*, *jsr* or *rts* instruction. See also Jump delay slots.

```
// The following code is illegal.
jsr.gt cont;
nop;
rts;        // Illegal: rts is inside a delay-slot of a prior jsr
```

2. A *loop_begin* instruction cannot immediately follow a *set.loop* instruction. The two instructions must be separated by at least one other instruction.

```
// The following code is illegal.
set.loop 6, 5;
loop_begin;   // Illegal: loop_begin immediately follows set loop
              //          There is no inst separating the two.
```

3. A *loop_begin* instruction cannot be followed by another *loop_begin* instruction without an intervening *set.loop* instruction.

```
// The following code is illegal.
loop_begin;
setB.VRAptr rS0 0.0;
add a2, a1;
loop_begin;   // Illegal: Two loop_begin without an intervening set loop
```

4. Two nested loops cannot end with the same last instruction. That is, the last instruction of an inner loop cannot also be the last instruction of an outer loop.

```
   // The following code is illegal.
   set.loop 6, Lstart1, Lend2;
   nop;
Lstart1:
   loop_begin;                // start of outer loop
   set.loop 6, Lstart2, Lend1;
   nop;
Lstart2:
   loop_begin;                // start of inner loop
Lend1:
   nop; loop_end; loop_end;   // end of inner and outer loops
                      // Illegal: inner and outer loops end with same inst
```

5. Each *rts* instruction inside a loop must have a calling *jsr* inside the loop.

```
   // The following code is illegal.
   set.loop 6, Lstart2, Lend2;
   nop;
Lstart2:
   loop_begin;                 // start of loop
   rts;                        // Illegal: rts inside a loop
   nop;
Lend2:
   nop; loop_end;              // end of loop

   // The following code is legal:
   set.loop 6, Lstart2, Lend2;
   nop;
Lstart2:
   loop_begin;                 // start of loop
   jsr function;
```

```
      nop;
      nop;
      jmp skipFunction;
      nop;
      nop;
   function:
      nop;                        // nop for illustration purposes only
      <function body>
      rts;                        // legal: rts inside a loop
      nop;                        // nop for illustration purposes only
      nop;                        // nop for illustration purposes only
   skipFunction:
      nop;
      nop;
      nop;
   Lend2:
      nop; loop_end;              // end of loop
```

6. A *jmp* or *jsr* instruction can be placed inside a loop, if it is not placed in the last three instructions of the loop. That is, the *loop_end* must not be in the delay slots of the *jmp* or *jsr* instruction.

```
      // The following code is illegal.
      set.loop 6, Lstart3, Lend3;
      nop;
   Lstart3:
      cmp g2, g3; loop_begin;  // start of loop
      nop;
      nop;
      jmp.eq equal;  // Illegal: jmp is placed at second to last inst of loop
   Lend3:
      nop; loop_end; // end of loop
      nop;
   equal:
```

7. A 1 or 2 instruction must not be placed within the last 3 instructions of an outer loop. That is, the *loop_end* instruction for the outer loop cannot be in the 'delay' slots of the *loop_end* for the 1 or 2 instruction loop.

```
      // The following code is illegal.
      set.loop 6, Lstart3a, Lend3a;
      nop;
   Lstart3a:
      cmp g2, g3; loop_begin;  // start of loop
      nop;
      set.loop 8, 1; // Setup 1 istruction loop
      nop;
      nop; loop_begin; loop_end;  // Illegal: 1 instruction loop is placed at second to
last inst of loop
   Lend3a:
      nop; loop_end; // end of loop
      nop;


      // The following code is illegal.
      set.loop 6, Lstart3b, Lend3b;
      nop;
   Lstart3b:
      cmp g2, g3; loop_begin;  // start of loop
      nop;
      set.loop 8, 2; // Setup 2 istruction loop
      nop;
      nop; loop_begin;
      nop; loop_end;  // Illegal: 2 instruction loop is placed at second to last inst
of loop
   Lend3b:
      nop; loop_end; // end of loop
      nop;
```

8. If a *jsr* instruction is placed inside a loop, the target of the *jsr* must not be the instruction immediately following the loop.

```
   // The following code is illegal.
   set.loop 6, Lstart4, Lend4;
   nop;
Lstart4:
   cmp g2, g3; loop_begin;  // start of loop
   nop;
   jsr.eq equal;
   nop;
   nop;
Lend4:
   nop; loop_end;   // end of loop
equal:              // Illegal: target of jmp is at the inst immediately
                    //          following the loop
```

9. If a *jmp* instruction is placed inside a loop, the target of the *jmp* can be inside or outside the loop. If the target lies outside the loop, then it must not be the instruction immediately following the loop. If a *jmp* is taken to a target outside of a loop, then a subsequent *jmp* must be taken to return to the loop body.

```
   // The following code is illegal.
   set.loop 6, Lstart4, Lend4;
   nop;
Lstart4:
   cmp g2, g3; loop_begin;  // start of loop
   nop;
   jmp.eq equal;
   nop;
   nop;
Lend4:
   nop; loop_end;   // end of loop
equal:              // Illegal: Target of jmp is at the inst immediately
                    //          following the loop
   nop;
```

10. All loops must end with a *loop_end* instruction.

```
    // The following code is illegal.
    set.loop 6, Lstart5, Lend5;
    nop;
 Lstart5:
    cmp g2, g3; loop_begin;  // start of loop
    nop;
    jmp.eq equal;
    nop;
    nop;
 Lend5:
    nop;        // Illegal: No loop_end instruction
```

11. A *loop_stop* instruction cannot be placed in the last three instructions of a loop. After execution of the *loop_stop* instruction, the last three instructions (inclusive of *loop_end*) must execute in consecutive cycles with no change of flow.

```
   // The following code is illegal.
   set.loop 6, Lstart6, Lend6;
   nop;
Lstart6:
   cmp g2, g3; loop_begin; // start of loop
   nop;
   add g0, 1;
   loop_stop;     // Illegal: loop_stop in last 3 instructions
   nop;
Lend6:
   nop; loop_end;          // end of loop

   // The last 3 instructions of a loop must execute in consecutive cycles after
   // loop_stop instruction.
   set.loop 4,Lstart7,Lend7;
```

```
            cmp g0,0;
            nop;
         Lstart7:
            jsr.eq OUT; loop_begin; // jsr to loop_stop routine if g0 is equal to 0
            sub g0,1;
            cmp g0,0;     // Illegal: out of order execution of last 3 instructions
                          //          after loop_stop
            nop;         // OUT subroutine returns here to execute the end of the loop
         Lend7:
            nop; loop_end;
            ...
         OUT:
            loop_stop; rts;       // End of OUT
            nop;
            nop;
```

4.17.6 Hardware loop legal examples

```
   // The following code is legal.
   set.loop 6, LstartOk, LendOk;
   nop;
LstartOk:
   cmp g2, g3; loop_begin;     // start of loop
   nop;
   jsr.eq equal;               // jsr not in last 3 cycles
   nop;
   nop;
LendOk:
   nop; loop_end;              // end of loop
   nop;
equal: rts;
   nop;
   nop;
```

4.17.7 set.loop instruction

The *set.loop* instruction has four formats.

- In the first format, the instruction sets the current iter_count to Iu10 and the current loop_size to Iu19.
- In the second format, the instruction sets the current iter_count using the content of an ag*X* register and the current loop_size to Iu19.
- In the third format, the instruction sets the current iter_count to Iu10.
- In the fourth format, the instruction sets the current iter_count using the content of an ag*X* register.

The valid range of ag*X* values is 0 through 65535, values from 1 to 65536 provide a loop count of 1 through 65535. An ag*X* value of 0 will produce a loop count of 65536.

For loops with known iteration counts at compile time and with iteration counts 1024 or less, the first or third formats can be used. Otherwise, the second or fourth instruction formats can be used.

Instruction Formats

```
   set.loop Iu10,Iu19;          // OpC
   set.loop agX,Iu19;        // OpC
   set.loop Iu10;              // OpB
   set.loop agX;             // OpB
   setC.loop Iu16;             // OpC
```

4.18 Control flow instructions

This group of instructions can be used to conditionally or unconditionally redirect the control flow of a program. There are three instructions in this group:

- `jmp` - conditional or unconditional jump with no return.
- `jsr` - jump to a subroutine, with return address pushed onto a 16-deep Return Address Stack (RAS). The return address of the `jsr` instruction is the program memory address of the `jsr` instruction plus 3.
- `rts` - return from subroutine. It pops the return address from the RAS and performs a jump to that program memory address.

The syntax of these instructions can be summarized in the following Table.

Table 67. Control Flow Instructions

Instructions[1,2]	Descriptions
jmp <target>;	Unconditional jump to <target>
jmp.cc <target>;	Conditional jump to <target>
jsr <target>;	Unconditional jump to subroutine at <target>
jsr.cc <target>;	Conditional jump to subroutine at <target>
rts;	Return from subroutine

1. *cc* denotes the condition upon which the jmp or jsr is based. See "Logical test instruction modifiers" table for all possible *cc*.
2. <target> can be any gX or a 16-bit immediate value.

The *cc* denotes the optional logical test condition which the jmp (or jsr) is based on. The leftmost columns of Table 70 below shows all possible conditions. The table also shows the values of the condition flags for the jump to occur. A jmp (or jsr) instruction without the *cc* specified is equivalent to a jmp.al (or jsr.al).

4.18.1 Jump delay slots

The two instructions immediately following a `jmp` (or `jsr`) instruction will always be executed, even though the `jmp` (or `jsr`) may be taken. Likewise, two instructions immediately following a `rts` instruction will always be executed, even though the `rts` is taken.

The two instructions that follow the `jmp`, `jsr` or `rts` instructions are called Jump Delay Slots. Programmers should always try to fill the jump delay slots with some useful instructions, whenever possible.

The `rts` instruction cannot be used with Format-3 macro-instructions where two half-word instructions are packed into a single macro-instruction.

VSPA has a typical three stage pipeline that is used by all instructions. It is a non-flushing pipeline, so two instructions are in the fetch and decode stages while a third is in the execute stage. If an instruction that causes a change of flow is executed, the following two instructions that have been pulled into the pipeline will also be executed - they will not be flushed.

Put another way, the two instructions that follow a jmp, jsr, or rts instruction will also be executed, immediately following the execution of the jmp/jsr/rts.

CAUTION

A `loop_begin` or `rts` cannot be placed in the jump delay slot of a prior `jmp`, `jsr` or `rts` instruction.

4.18.1.1 Register setup times

The jmp*(.cc)* gX and jsr*(.cc)* gX instructions determine the target address from the value that is in the gX register. The gX value must be valid in the cycle before the instruction is executed. This restriction exists whether the instruction is to be conditionally executed.

The gX register must be valid in the cycle before execution, but need not be valid in the following cycle, that is, only one cycle of validity is required.

4.18.2 Compare-and-jump example

The following shows an example of compare-and-jump.

```
mv.w g1, [rS0];    // move element from vector register array to g1.
cmp g1, 0;       // compare g1 with 0.
nop;             // jmp needs 1 cycle of setup after cmp
jmp.eq target;   // jump to target if g1 is equal to 0.
mv a0, 32;       // first delay slot (always executed)
mv a4, 1024;    // second delay slot (always executed)
```

4.18.3 Back-to-back conditional jumps

In general, VSPA ISA does not allow back-to-back jumps. Specifically, a `jmp` or `jsr` should not be placed in the delay slots of another prior `jmp` or `jsr`. That is, two `jmp/jsr` should be placed at least three instructions apart.

However, there is an exception to this restriction. The VSPA ISA allows back-to-back conditional jumps, provided that the following two rules are met.

1. Two or three `jmp/jsr` are placed in close proximity (less than three instructions from each other); they are all conditional.
2. At run-time, in any given three-instruction window, at most one `jmp/jsr` is actually taken.

The code in Example 1 is illegal since there are two `jmp/jsr` in close proximity (less than three instructions apart), and they are not all conditional.

```
Example 1:
    jsr.eq A;
    nop;
    jmp B; // unconditional violates rule 1
```

The code in Example 2 is legal since of the three conditional `jsr`, only one will be taken. This is guaranteed by the fact that the first `jsr` is based on a less-than condition, the second is based on an equal condition, and the third is based on a greater-than condition - only one of these three conditions can occur. That is, within that three-instruction window, only one of the three `jsr` can be taken.

```
Example 2:
    jsr.lt LT_CASE;
    jsr.eq EQ_CASE;
    jsr.gt GT_CASE;
    nop;
    nop;
```

The code in Example 3 is illegal since more than one `jmp` can possibly be taken, depending on the run-time value of g0.

```
Example 3:
    cmp g0, 4;
    nop;
    jmp.le LE_CASE;
    jmp.eq EQ_CASE; // violates rule 2
```

If g0 is equal to 4 prior to executing this code, the hardware behavior will be non-deterministic.

WARNING

If a user chooses to use back-to-back conditional jumps, it is the user's responsibility to guarantee that only one of these conditional jumps, inside any three-instruction window, is taken at run-time.

The following are some use cases of back-to-back conditional jumps.

```
Example 1:  Jump Table - only one jump is taken based on the value of an
            IP register field, TX_MODE.
    mvip g0, TX_MODE, TX_MODE_MASK;
    sr g0, g0, TX_MODE_BIT;
    cmp g0, 1;
    cmp g0, 2;
    jmp.eq CASE_1; // ipReg=1
    jmp.eq CASE_2; // ipReg=2
    cmp g0, 3;
    cmp g0, 4;
    jmp.eq CASE_3; // ipReg=3
    jmp.eq CASE_4; // ipReg=4

Example 2:  Traverse a large Jmp Table via binary search.
    mvip g0, TX_MODE, TX_MODE_MASK;
    sr g0,g0, TX_MODE_BIT;
    cmp g0, midPoint;
    nop;
    jmp.lt LT_MIDPOINT;
    jmp.eq EQ_MIDPOINT;
    jmp.gt GT_MIDPOINT;
    :
    :
    LT_MIDPOINT:
    cmp g0, quarterPoint;
    nop;
    jmp.lt LT_QTRPOINT;
    jmp.eq EQ_QTRPOINT;
    jmp.gt GT_QTRPOINT;
    :

Example 3:  Here is an example of a "switch-case" construct using back to
            back conditional instructions
    cmp g0, 1;
    cmp g0, 2;
    jmp.eq CASE1;
    cmp g0, 3;
    jmp.eq CASE2;
    cmp g0, 4;
    jmp.eq CASE3;
    nop;
    jmp.eq CASE4;
    jmp.ne DEFAULT;
    nop;
    nop;
```

4.19 Conditional instructions

The instructions shown in Table 68 and Table 69 may be executed conditionally. The *cc* value in these tables represents a logical test instruction modifier which can be appended to these instructions to restrict execution based upon a certain condition. See Table 70 for a list of all available logical test instruction modifiers.

All of these instructions require a one-cycle setup time of the condition code bits. See Conditional instruction setup time for a detailed explanation of the one-cycle setup time requirement.

Table 68. Conditional Scalar Instructions

Instruction	Type	Description
mpy*(.cc)*(.s) gZ gX,gY	opS	Multiply
div*(.cc)*(.s) gZ gX,gY	opS	Divide

Table continues on the next page...

Table 68. Conditional Scalar Instructions (continued)

Instruction	Type	Description
mod*(.cc)*(.s) gZ gX,gY	opS	Modulus
sr*(.cc)*(.s) gZ gX,gY	opS	Shift Right
add*(.cc)* gZ gX,gY	opS	Add
sub*(.cc)* gZ gX,gY	opS	Subtract
cmp*(.cc)* gX,gY	opS	Compare
sl*(.cc)* gZ gX,gY	opS	Shift Left
and*(.cc)* gZ gX,gY	opS	And
or*(.cc)* gZ gX,gY	opS	Inclusive Or
xor*(.cc)* gZ gX,gY	opS	Exclusive Or
bclr*(.cc)* gZ gX,gY	opS	Bit Clear
mv*(.cc)* gY gX	opS	Move
not*(.cc)* gY gX	opS	Not

Table 69. Conditional Jump Instructions

Instruction[1]	Description
jmp*(.cc)* <target>	Conditional jump to <target>
jsr*(.cc)* <target>	Conditional jump to subroutine at <target>

1. <target> can be any gX or a 16-bit immediate value.

4.19.1 Logical test instruction modifiers

Certain instructions can be appended with an optional logical test instruction modifier, which results in execution of the instruction being dependent upon the evaluation of the condition code flags. See Table 71 for a description of the condition code flags. The table below lists the possible logical test instruction modifiers and their corresponding condition code flags values.

Table 70. Logical test instruction modifiers

Logical Test Modifiers	Descriptions	Operation Type	Condition Code Flags[1]					
			N	Z	V	C	au_ap	au_an
al	Always	Simple	x	x	x	x	x	x
au_an[2]	AU all negative	AU OP	x	x	x	x	x	1
au_ap[2]	AU all positive	AU OP	x	x	x	x	1	x

Table continues on the next page...

Table 70. Logical test instruction modifiers (continued)

au_nan[2]	AU not all negative	AU OP	x	x	x	x	x	0
au_nap[2]	AU not all positive	AU OP	x	x	x	x	0	x
cc	Carry Clear	Simple	x	x	x	0	x	x
cs	Carry Set	Simple	x	x	x	1	x	x
eq	Equal Zero	Simple	x	1	x	x	x	x
ge	Greater or Equal not (N xor V)	Signed	-	x	-	x	x	x
gt	Greater not ((N xor V) or Z)	Signed	-	-	-	x	x	x
hi	Higher not (Z or C)	Unsigned	x	-	x	-		
hs	Higher or Same	Unsigned	x	x	x	0	x	x
le	Less Than or Equal ((N xor V) or Z)	Signed	-	-	-	x	x	x
lo	Lower	Unsigned	x	x	x	1	x	x
ls	Lower or Same (Z or C)	Unsigned	x	-	x	-	x	x
lt	Less Than (N xor V)	Signed	-	x	-	x	x	x
mi	Minus	Simple	1	x	x	x	x	x
ne	Not Equal	Simple	x	0	x	x	x	x
nv	Never	Simple	x	x	x	x	x	x
pl	Plus	Simple	0	x	x	x	x	x
vc	Overflow Clear	Simple	x	x	0	x	x	x
vs	Overflow Set	Simple	x	x	1	x	x	x

1. "x" denotes don't care; "1" denotes the flag is set; "0" denotes the flag is clear; "-" denotes multiple bits are evaluated for the logical condition shown in description.
2. Logical test instruction modifiers of operation type AU OP do not apply to conditional instructions shown in "Conditional Scalar Instruction" table.

4.19.2 Condition code flags

VSPA maintains six condition code flags in the CREG unit: *N* (negative), *Z* (zero), *V*(overflow), *C*(carry), *au_ap* (AU all positive) and *au_an* (AU all negative). Table 71 describes how these condition flags are being set. Four of these condition codes (N,Z,V,C)

are set by scalar operations and the remaining two (au_ap, au_an) are set by vector operations. By default the scalar condition codes are modified by cmp, btst and mv cc instructions only, but the VSPA core can be configured to modify them on a wide range of scalar instructions by setting system control register 4 (condition code update switch) to a 1. See Table 73 for a list of instructions which can update the scalar condition codes when this register is set.

Table 71. VSPA Condition Flags

Condition Flag	Descriptions	Compare Operation Triggered by[1]
N	This flag is set when the result of the operation carried out by a scalar instruction is Negative.	cmp/btst instruction
Z	This flag is set when the result of the operation carried out by a scalar instruction is equal to Zero.	cmp/btst instruction
V	This flag us set when the result of the operation carried out by a scalar instruction has a 2's complement overflow, that is, result sign mis-match.	cmp/btst instruction
C	This flag is set when the result of the operation carried out by a scalar instruction has a carry (borrow).	cmp/btst instruction
au_ap	For real number computations, this flag is set when all AU outputs are positive. In complex mode, this flag is set when all real elements (all even elements) are positive.	Any AU operation
au_an	For real number computations, this flag is set when all AU outputs are negative. In complex mode, this flag is set when all real elements (all even elements) are negative.	Any AU operation
au_az	For real number computations, this flag is set when all AU outputs have an unbiased exponent value of zero. For complex number calculations, this flag is set when all real elements (all even elements) have an unbiased exponent value of zero.	Any AU operation

1. See "Instructions which always modify Condition Codes" table for a list of instructions which always set the condition codes.

Note:

- Upon executing a scalar instruction, the condition codes, *N*, *Z*, *V* and *C*, will be updated based on the result of the comparison. Once updated, these condition code flags will remain unchanged until the next cmp instruction is executed.
- Upon each AU operation, the condition flags, *au_an*, *au_ap* and *au_az*, will be updated based on the AU results. Once updated, these condition code flags will remain unchanged until the next AU operation is performed.
- To affect a conditional jump based on a scalar instruction, the conditional jump must be placed at least two cycles away from the scalar instruction.
- To affect a conditional jump based on an AU operation, the conditional jump must be placed at least six cycles away from the AU instruction. If there are more AU operations after that AU operation of interest, then the conditional jump instruction must be placed exactly six cycles after that AU operation.

The instructions in Table 72 will modify the condition code bits regardless of the state of system control register 4.

Table 72. Instructions which always modify condition code flags

Instruction	Type	Description
cmp gX,I	opD	Compare 32 bit immediate
btst gX,I	opS	bit test
cmp*(.cc)* gX,gY	opS	Compare registers
mv*(.cc)* cc gX	opS	Move gX into cc
cmp(.s) gX,Is	opS	Compare 16 bit immediate
mv cc Is	opS	Move 4 bit immediate value into cc
addS.ucc	opS	If the optional field .ucc is used, then the instruction will update the condition codes based on the result of the operation.
addD.ucc	opD	
subS.ucc	opS	
subD.ucc	opD	

The instructions in Table 73 will modify the condition code bits only if system control register 4 is set to a 1.

Table 73. Instructions which optionally modify condition code flags

Instruction	Type	Description
add gX,I	opD	Add 32 bit immediate
sub gX,I	opD	Subtract 32 bit immediate
andl gX,I	opD	And 32 bit immediate
orl gX,I	opD	Inclusive Or 32 bit immediate
xorl gX,I	opD	Exclusive Or 32 bit immediate
sr(.s) gY gX,I	opS	Shift right by immediate
sl gY gX,I	opS	Shift left by immediate
and gX,Is	opS	And 16 bit immediate
or gX,Is	opS	Inclusive Or 16 bit immediate
xor gX,Is	opS	Exclusive Or 16 bit immediate
mpy*(.cc)*(.s) gZ gX,gY	opS	Multiply registers
mac(.cc) g*Z*,g*X*,g*Y*	opS	Multiply and accumulate (integer values)

Table continues on the next page...

Table 73. Instructions which optionally modify condition code flags (continued)

Instruction	Type	Description
div*(.cc)*(.s) gZ gX,gY	opS	Divide registers
mod*(.cc)*(.s) gZ gX,gY	opS	Modulus registers
sr*(.cc)*(.s) gZ gX,gY	opS	Shift Right by register
add*(.cc)* gZ gX,gY	opS	Add registers[1]
sub*(.cc)* gZ gX,gY	opS	Subtract Registers[1]
sl*(.cc)* gZ gX,gY	opS	Shift left register
and*(.cc)* gZ gX,gY	opS	And registers[1]
or*(.cc)* gZ gX,gY	opS	Inclusive or registers
xor*(.cc)* gZ gX,gY	opS	Exclusive or registers
bclr*(.cc)* gZ gX,gY	opS	Bit clear
not*(.cc)* gY gX	opS	not register
abs gY gX	opS	Absolute value
div(.s) gX,ls	opS	Divide by 16 bit immediate
mod(.s) gX,ls	opS	Modulus by 16 bit immediate
mpy(.s) gX,ls	opS	Multiply by 16 bit immediate
add(.s) gX,ls	opS	Add 16 bit immediate[1]
sub(.s) gX,ls	opS	Subtract 16 bit immediate[1]
rdiv(.s) gX,ls	opS	Divide into 16 bit immediate
rmod(.s) gX,ls	opS	Modulus into 16 bit immediate
rsub(.s) gX,ls	opS	Subtract (reverse) 16 bit immediate[1]

1. Condition codes not updated if destination is stack pointer (sp).

4.19.3 Conditional instruction setup time

Some instructions may or may not be executed depending on the result of a condition code bit or bits. These conditional instructions require the condition codes they depend on to be valid one cycle before the execution phase of the conditional instruction. When any of the instructions shown in Table 74 are used with a logical test instruction modifier (for example, '.eq', '.ne', '.cs', and so on), they require a setup time of the CC bits with respect to the execution of the instruction. See Compare-and-jump example for an example of the setup requirements of a conditional instruction relative to a compare instruction which is updating the CC bits.

Table 74. Conditional instructions setup time

Instruction	Setup[1]	Description
jsr/jmp*(.cc)* I	1	Jump to immediate address
jsr/jmp*(.cc)* gX	1	Jump to address in gX
mpy*(.cc)*(.s) gZ gX,gY	1	Multiply
div*(.cc)*(.s) gZ gX,gY	1	Divide
mod*(.cc)*(.s) gZ gX,gY	1	Modulus
sr*(.cc)*(.s) gZ gX,gY	1	Shift Right
add*(.cc)* gZ gX,gY	1	Add
sub*(.cc)* gZ gX,gY	1	Subtract
cmp*(.cc)* gX,gY	1	Compare
sl*(.cc)* gZ gX,gY	1	Shift Left
and*(.cc)* gZ gX,gY	1	And
or*(.cc)* gZ gX,gY	1	Inclusive Or
xor*(.cc)* gZ gX,gY	1	Exclusive Or
bclr*(.cc)* gZ gX,gY	1	Bit Clear
mv*(.cc)* gY gX	1	Move
not*(.cc)* gY gX	1	Not

1. Can also be considered the number of intervening instructions between cmp and conditional instruction: cmp; nop; // intervening instruction to allow proper CC setup <instr>*.cc*

The condition(s) must be valid in the cycle before execution, but need not be valid in the following cycle, that is, only one cycle of validity is required.

Note that the condition codes can be used as data by the mv instruction.The setup time for CCs applies only to their use as conditions, not as data. So, when used as data, the CCs are available for use in the cycle immediately following their update, just like gX or agX.

4.20 Numerically controlled oscillator (NCO) instructions

The NCO can be configured for generation of complex exponential sequences for efficient use in Fourier transforms and mixing operations. The vector sequence is generated according to the following expression (which uses the variables *k*, *f* and *i* to refer to nco_k, nco_freq and nco_phase, respectively)

$$\exp\{j2\pi kf\,(\mathrm{i} + \mathrm{nv} + [0, 1, 2, ..., \mathrm{v} - 1]) / 2^{32}\} \text{ where, } \mathrm{n} = 0, 1, 2, ...$$

and produces *v* complex single precision scalars the mux path described in S1mode options and detailed description. The generator is a function of the parameters described inTable 75.

Table 75. NCO parameters

Parameter	Description
nco_k	Base frequency multiplier represented as an unsigned 16-bit integer, which is typically used for scaling the base frequency for twiddle factors in Fourier transforms.
nco_freq	Base frequency represented as a signed 32-bit integer represented in one's complement form.
nco_phase	Initial phase represented as a signed 32-bit integer represented in one's complement form.
n	Discrete time index which advances by one upon each *rd S1* instruction with *S1mode=S1nco*.
v	Vector size. Three values are supported: v=1 for complex scalar vector operations, v=8 for Fourier transform butterfly twiddle factors, v=16 for general complex mixing operations.

The current state of the phase accumulator, *nco_phase* + *n* * *v*, is accessible and useful for initializing, saving and restoring the NCO phase in mixer applications. Instructions for controlling the NCO are summarized in Table 76.

Table 76. NCO parameter control instructions

Mnemonic	Family	Description
set.nco {radix2, singles, normal}, Iu16, Is32	OpD	Configures all parameters in a single instruction. The frequency multiplier and base are specified as immediate scalars. The vector size is specified using a keyword: radix2(v=8), singles(v=1), and normal(v=16). This instruction automatically clears the phase accumulator. Order of arguments is nco_k, nco_freq. **NOTE** With this instruction, the LSB of the base frequency parameter is always set to be 0. Further, only the least significant 10 bits of the frequency multiplier are set. The base frequency needs to be specified as f << 2. **NOTE** The AU precision (set.prec) must be set prior to the set.nco instruction. Changing the AU precision after the set.nco instruction may result in unexpected nco results.
mv nco_k, Iu11	OpB	Sets the least significant 11 bits of the NCO base frequency multiplier. This instruction automatically clears the phase accumulator.
mv nco_k, g*X*	OpB	Configure the nco base frequency multiplier using a general purpose register. This instruction automatically clears the phase accumulator.
add nco_k, Is11	OpB	Modify the nco base frequency multiplier by adding or subtracting an immediate scalar. This is useful when switching between FFT/DFT stages. This instruction automatically clears the phase accumulator.

Table continues on the next page...

Table 76. NCO parameter control instructions (continued)

Mnemonic	Family	Description
mv nco_freq, g*X*	OpB	Configure the nco base frequency using a general purpose register.
mv nco_phase, g*X*	OpB	Swap the contents of a general purpose register and the phase accumulator. **NOTE** • g*X* is modifed by this instruction. • The value loaded into the g*X* register is a "look-ahead" value of nco_phase. That is, the current value of nco_phase is incremented by 64 in normal mode, incremented by 32 in radix2 mode, and incremented by 1 in singles mode. The same incremented value is also the value read by the debugger when reading the nco_phase register.
mv g*X*, nco_phase	OpB	Move the contents of the phase accumulator to a general purpose register. **NOTE** The value loaded into the g*X* register is a "look-ahead" value of nco_phase. That is, the current value of nco_phase is incremented by 64 in normal mode, incremented by 32 in radix2 mode, and incremented by 1 in singles mode. The same incremented value is also the value read by the debugger when reading the nco_phase register. Note that the value read may not be accurate if the nco_phase update is ongoing.

Following is an example of how NCO is programmed:

```
set.prec  half_fixed, half_fixed, single, single, half_fixed;
set.creg  255, 8;
set.nco normal, 0x1,0;
mv g1, 536870912;
mv nco_freq, g1;
mv g0, 0;
mv nco_phase, g0;
nop;
rd S0; rd S1; rd S2; set.Smode S0real1,S1nco,S2zeros;
nop;
mad;
nop;
nop;
nop;
wr.straight;
```

Chapter 5
IPPU Architecture

5.1 IPPU overview

This chapter describes the operation and register programming of the Inter-vector Permutation Processing Unit (IPPU). The IPPU is a programmable coprocessor used in SOCs as an accelerator for data reordering in memory for use by the VCPU.

5.1.1 IPPU SOC level components

The IPPU (SOC level) components:

- IPPU core: The IPPU fetches instructions from the IPPU program memory (IPPU PRAM), and uses arbitrated single-port data memories (IPPU DMEM and VCPU DMEM) for storage during computation.
- IPPU Program memory (IPPU PRAM): IPPU instructions are stored in the IPPU PRAM. The IPPU PRAM is 32-bit wide memory. It is dynamically loaded via the DMA when the IPPU is in the idle state. This memory is used for the IPPU code.
- IPPU data memory (IPPU DMEM) and VCPU data memory (VCPU DMEM): Working data memory. It is a single-port arbitrated RAM organized into lines of 1024 bits, with each line containing 64 real or 32 complex samples. The IPPU, VCPU, and the DMA can access both the IPPU DMEM and the VCPU DMEM. Arbitration scheme provides the DMA the highest priority, the IPPU the mid level priority, and VCPU the lowest priority.
  - Addresses are in 32-bit or 16-bit word units.
  - DMEM reads are always a single complete line.
  - The smallest element VCPU can write is a 16-bit half-word. Write accesses can be a full line, a half-word, or multiple half-words.
  - The IPPU always reads a full 1024-bit line from the DMEM, and is capable of writing the full 1024-bit line or writing a single or multiple 16-bit half-word in a single write operation.
- Internal peripheral bus (IP Bus) registers: Control and status registers for the hardware components in the IPPU module, which are visible to VCPU and the host software.

5.1.2 IPPU features

The IPPU module has the following features:

- Bit reverse, digit reverse, bit reverse and digit reverse address mapping.
- De-interleaved and indirect mapping.
- FFT size 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192 and 16384 bit reversal modes.
- Up to 10-digit 2, 3, or 5 radix factors.
- Hardware-supported program control for:
  - 2 levels of nested loops.
  - 2-deep Return Address stack for subroutine calls. IPPU RS pointer is one bit. Overflow is not reported and it is the user responsibility to avoid error conditions such as underflow and overflow.
  - Conditional and unconditional jump, jsr (jump to Subroutine), and rts (return from subroutine).
  - Real-Time, Programmer provided start address.
- Low power when not running - no state change of registers minimizes dynamic power, and automatic clock gating when in idle state.
- Simple register mapped control and status from/to VCPU or host software driver.
- Limited debug support and internal register visibility.

5.2 Inter-vector permutation processing unit

5.2.1 IPPU core

The IPPU core fetches, decodes and executes instructions that reorganize data in memory for further processing by the vector digital signal processor. See IPPU Core Block Diagram.

Every clock cycle the core fetches and decodes a 32-bit instruction word. Each bit-field of the instruction controls a logical unit in the core's data path (such as the Memory Address Generator Unit). The vector data follows a path from DMEM into a Register File (RF), Data is reordered in the Register File, and out to DMEM. The IPPU organizes the samples for efficient use by the VCPU.

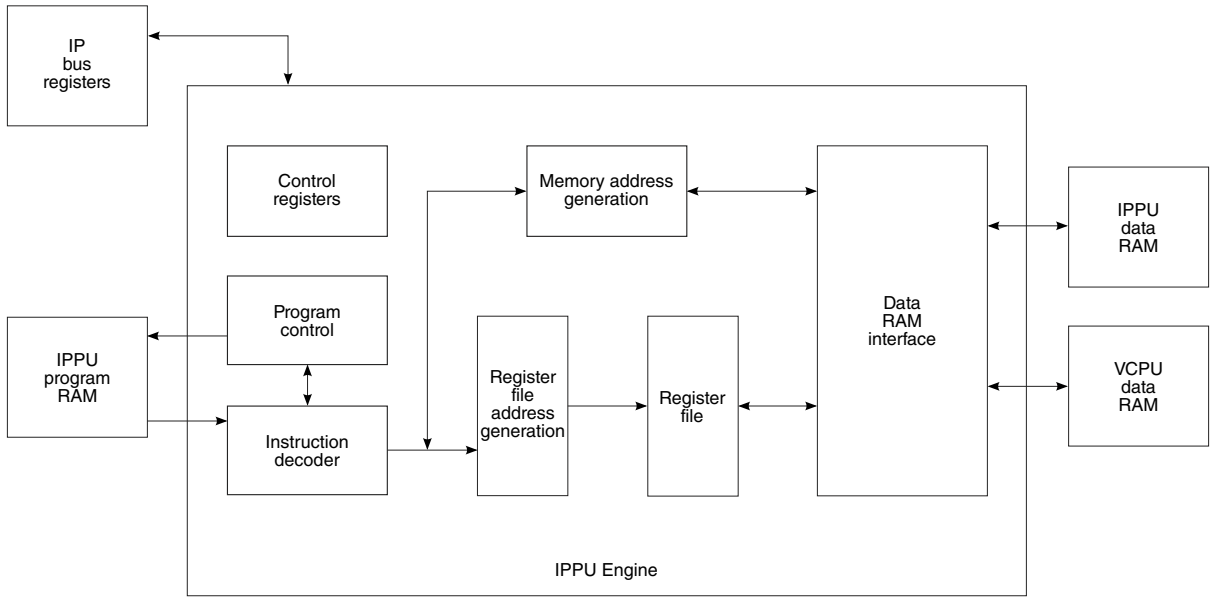


Figure 35. IPPU Core Block Diagram

Table 77. IPPU Core Components

Component	Description
Control register unit	Specifies and holds IPPU core's configuration.
Program control unit	Manages the program counter and generates addresses and control for IPPU PRAM. It also controls subroutines, jumps and hardware loops.
Instruction decoder unit	Fetches instructions from IPPU PRAM and decodes it into one or more micro-instructions (which control other units in the core data path).
Memory address generation (MAG) unit	Manages address generation for the DMEMs. • Flexible post modification modes including; auto-increment, auto-decrement, indexing by a constant, and absolute loading.
Register file address generation (RAG) unit	Manages address generation for the Register File Unit.
Register file (RF)	The register file consists of 4 1024-bit registers: r0, r1, r2 and r3.

Table continues on the next page...

Table 77. IPPU Core Components (continued)

Component	Description
	• Each register can contain either 64 real half-precision elements or 32 complex half-precision elements. • Each register in RF is 1024 bits wide. • r1/r0 concatenate to present a virtual 2048 bit register.
Data RAM interface	Generates the control, address, and data signals to/from the IPPU DMEM and the VCPU DMEM. Handles the stall signals from these memories.

5.2.2 IPPU operating states

The IPPU has 3 states of operation: *Running*, *Idle* and *Suspended*:

- Idle State:

After Reset. the IPPU enters the Idle (low-power) state. When in the Idle state, once the IPPU CONTROL register is written, the IPPU enters the Running state (note that the IPPU can start immediately, of wait for even to trigger its start. Refer to the IPPU CONTROL register for more details), loads the Start Address, and starts fetching and executing instructions from the program memory (IPPU PRAM) starting at the provided "start Address". The IPPU asserts the *ippu_busy* bit and clears the *ippu_done* and *ippu_aborted* bits in the in the IPPU STATUS register. Refer to Running State below for a description how the IPPU transition from Running state to Idle State

- Running State:

Refer to the Idle State above for a description how the IPPU enters the Running state. The IPPU exits the Running state and enters the Idle state when it encounters one of the following conditions:

1. Normal termination - Execution of the Done instruction, or
2. Abort Termination - Receiving the Abort command via the *ippu_abort* signal. The IPPU immediately stops execution of new instructions, sets the Aborted status bit in the IPPU STATUS register, and enters the Idle state.

- Suspended State:

When the IPPU is in the Running State and the *ippu_suspend* signal is asserted as result of setting the *ippu_suspend* bit in the IPPU Run Control Register, the IPPU enters the Suspended state. The IPPU suspends instruction fetch and execution. Instruction fetch and execution will resume when the *ippu_suspend* bit is cleared. The *ippu_suspend* acts as stop/continue control for the IPPU. The ippu_abort signal will cause the core to move to the Idle state.

IPPU state transitions are shown in Figure 36.

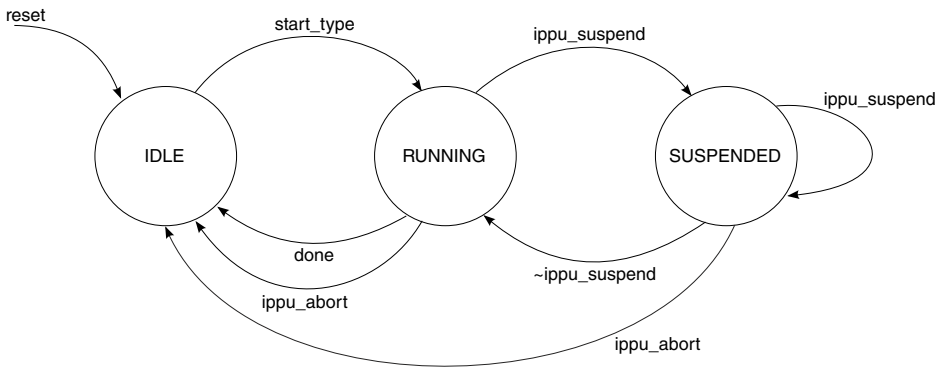


Figure 36. IPPU Operating State Diagram

The IPPU starts after reset in the Idle state. The IPPU consumes very low current in this state. Internal IPPU registers are gated OFF and no internal state change may occur.

IPPU transition from Idle state to Running state is controlled by the start_type field the IPPU CONTROL register, and thus essentially by VCPU or host software. The IPPU will remain in Idle state until the start_type field in the IPPU CONTROL register is written.

In Running state the IPPU is fetching and executing code. The IPPU software controls the transition back to Idle state by executing a special "*done*" instruction. The IPPU transition from Running state to Idle state is also controlled by setting the *ippu_abort* bit the IPPU CONTROL register. The *ippu_abort* bit is self-clearing, so the host does not need to clear it once a one was written to it.

If the IPPU is in the Running state, and the *ippu_suspend* bit in the IPPU CONTROL register is set to one by VCPU or host software, the IPPU immediately transition into the Suspended state. In the Suspended state the IPPU does not fetch instructions, suspends code execution, and does not initiate DMEM transactions. DMEM transactions that are in progress will be completed. To exit the suspended state, VCPU or host software must clear the *ippu_suspend* bit in the IPPU CONTROL register.

The IPPU reports its state to the host and VCPU software via the status bits: ippu_busy, ippu_suspended, ippu_aborted, ippu_done, and ippu_error in the IPPU STATUS registers.

5.2.3 IPPU memory access considerations

When accessing the VCPU Data RAM or the IPPU Data RAM, the DMA has highest priority, followed by the AXI slave, IPPU, FECU, and the VCPU which has the lowest priority. When more than one unit is assessing the same DMEM simultaneously, the lower priority unit/(s) is/are stalled. While these stalls should not affect the VCPU or the IPPU performance significantly, they should be taken into account when calculating the overall system performance.

The DMA has the highest priority. The DMA uses a DMEM buffer to hold an entire DMEM line's worth of data. It will take multiple clock cycles to transfer this data to the AXI bus, thus providing the lower priority units an opportunity to transfer data without starving extended period of time.

The IPPU does not support stalls on program memory (IPPU PRAM) accesses. The DMA should never access the IPPU PRAM when the IPPU is in the running state. The DMA may access the IPPU PRAM only when the IPPU is in the idle state.

5.2.4 IPPU initialization

The IPPU core fetches instructions from the IPPU PRAM, a RAM memory. Before the IPPU core can begin execution, application-specific software programs must be copied from external system memory to the IPPU program RAM via the DMA.

After an asynchronous reset assertion and negation, the IPPU enters the Idle state, waiting for the start_type in the IPPU CONTROL register to be written. The DMA is used by the host processor to load the image of IPPU PRAM. The host software must set up and enable the DMA to load the IPPU PRAM, and then poll for the transfer to complete. After IPPU PRAM is loaded, the host or the VCPU Software can write the start_type field in the IPPU CONTROL registers to transition the IPPU into the running state, where the IPPU starts fetching and executing instructions from the IPPU PRAM.

It is possible for either the host or the VCPU software to be the manager of the IPPU PRAM image loads.

The IPPU must be in the idle state before a new IPPU PRAM image is loaded or the IPPU PRAM is accessed by the DMA.

5.3 IPPU interrupts

The IPPU module does not generate interrupts directly. The IPPU generates active high done flag that may be used to generate interrupts. An IPPU-done interrupt will be generated whenever the IPPU executes a DONE instruction, and the irqen_ippu_done bit in the VSPA IRQEN register is set. The host should clear the interrupt by writing a "1" to the irq_pend_ippu_done bit in the VSPA STATUS register, or by clearing the irqen_ippu_done bit in the VSPA IRQEN register.

5.4 IPPU done to VCPU go event

When executing the done instruction, the IPPU generates a VPCU go flag that may be used to wake up the VCPU. A VCPU go event is generated whenever the IPPU executes a done instruction and the vcpu_go_en bit in the IPPUCONTROL register is

set. This will set the ippu_go status in the VCPU System Control (CONTROL) register and eventually may cause the VCPU to go. The VCPU should clear the ippu_go status in the VCPU System Control (CONTROL) register by writing a "1" to it.

Chapter 6
IPPU Instruction Set

6.1 Size definitions

Table 78 defines sizes corresponding to different hardware components in bits. Some sizes are symbolic throughout to accommodate variable VSPA line widths (for different AU counts).

Table 78. Size definitions

Size		Definition
A	18 bits	Addressing space for VCPU and IPPU DMEM
N_AU	16	number of AUs
N_elem	N_AU*2	number of 32 bit-elements per memory line
N_bits_line	N_elem*32	Number of bits per memory line
W	ceil(log2(N_elem*2)) bits	
X	ceil(log2(N_elem)) bits	
Y	ceil(log2(N_elem*32)) bits	
Z	ceil(log2(N_elem*2*2)) bits	
Ze	ceil(log2(N_elem*8*2)) bits	

6.2 Hardware definitions

Table 79. Hardware definitions

What	Name	Size	Description
DMEM address pointers	a0, a1	A	• Contains DMEM addresses offset from the start of the DMEM type (bank) (per 32-bit or 16-bit element) based on the state of the ippu_legacy_mem_addr field in IPPUCONTROL register • Fetches either (i) a 32-bit DMEM element, or (ii) a DMEM line consisting of N_elem 32-bit elements • These are general memory access pointers. The address contained in a0, a1 can point to either the VCPU's or IPPU's DMEM space. The appropriate DMEM type is specified by the ld.type and st.type operator fields. • Supports modulo range

Table continues on the next page...

Table 79. Hardware definitions (continued)

What	Name	Size	Description
Offset registers	m0, m1	A	• Used to specify offsets to post increment/decrement the pointers a0/a1 in load/store instructions
General registers	as0, as1, as2, as3	A	• General purpose storage registers. • Elements can be moved to/from these registers DMEM address registers, offset registers.
Data Register file	r0, r1	N_bits_line	• Storage for elements fetched from memory. • r1 and r0 form a continuous buffer of N_bits_line*2 and addressable in the same way. However, full or partial DMEM lines can be written only to either r0 or r1, but not across both r0 and r1.
Compare Register file	elem_mask	N_bits_line	• Storage for bit mask fetched from memory
Memory Index registers	mem_index0, mem_index1	A	• Used for indirect addressing. mem_index0 corresponds to a0, and, mem_index1 corresponds to a1. • These registers are not accessible to the user.
Data Register file pointers	r_rd_ptr, r_wr_ptr	Ze	• Read and write pointer to a 4-bit element within the data register file. • Read and write pointers are used by st.type and ld.type commands, respectively.
Memory element pointers	mem_elem_rd_ptr, mem_elem_wr_ptr	3 bits	• Pointer to a 4-bit element within a 32-bit element addressed by a0, a1.
Compare Register file pointers	elem_mask_ptr	Y	• Pointer to a bit within the compare register file.
Vectorized Index pointer	vindx_ptr	W	• Pointer into vectorized index register. • Used for vectorized index addressing (see Vectorized indirect addressing for details).

6.3 IPPU instructions summary

Table 80. IPPU instructions summary

Instruction	Cycles	Description
`ld.type [aX]+/-mX, elem_offset, ld_mode, wr_offset, latch_mode`	4	Load instructions

Table continues on the next page...

Table 80. IPPU instructions summary (continued)

Instruction	Cycles	Description
`ld.type [aX]+/-Is8, elem_offset, wr_offset, latch_mode`		
`ld.type [aX]+/-Is5, elem_offset,ld_mode, wr_offset, latch_mode`		
`ld.mask.type [aX]+/-mX`		
`ld.mask.type [aX]+/-Is9`		
`ld.index.type [aX]+/-mX`	3	Load memory index instructions
`ld.index.type [aX]+/-Is9`		
`st.type [aX]+/-mX, elem_offset, st_mode, rd_offset, write_mode`	2	Store instructions
`st.type [aX]+/-Is8, elem_offset, rd_offset, write_mode`		
`st.type [aX]+/-Is5, elem_offset, st_mode, rd_offset, write_mode`		
`set.range aY, asA, asB`	1	Set range instructions
`set.range aY, asA, Iu18`		
`set.br aX, br_mode`	2	Configure bit-reversal/digit-reversal engine instructions
`set.dr.radix digit_index, radix, Iu11`		
`set.dr.config digit_index, Iu3, Iu11`		
`mv ippu_reg, ippu_reg`	1	Move register instruction
`mv ippu_reg, Iu18`		
`ld asX, Iu6`	5	Load input argument instruction
`cmp.bit IsY`	1	Compare instruction
`jmp Iu16`	3	Jump instructions
`jsr Iu16`		
`jsr.z Iu16`		
`jsr.nz Iu16`		
`jmp.z Iu16`		
`jmp.nz Iu16`		
`rts`		
`rts.z`		
`rts.nz`		

Table continues on the next page...

Table 80. IPPU instructions summary (continued)

Instruction	Cycles	Description
`set.loop Iu1, Iu8`	2	Loop instructions **NOTE** Must have a one-cycle delay between writing any value into the as*X* register (because of a load or a move instruction) and using it as an iteration count source.
`set.loop Iu1, asX`		
`loop_begin Iu1`	1	
`loop_end Iu1`		
`done`	1	Done instruction
`clr Rx`	1	Clear register instruction
`set.mask IuY`	1	Set/clear element mask register instructions
`set.mask.all`		
`clr.mask IuY`	1	
`clr.mask.all`		
`add aX, Is8`	1	Add instructions
`add aX, mY`		
`add.cb aX, Is8`		
`add.cb aX, mY`		

6.4 Load instructions

IPPU instructions summary

Syntax:

```
ld.type [aX]+/-mX, elem_offset, ld_mode, wr_offset, latch_mode
ld.type [aX]+/-Is8, elem_offset, wr_offset, latch_mode
ld.type [aX]+/-Is5, elem_offset, ld_mode, wr_offset, latch_mode
ld.mask.type [aX]+/-mX
ld.mask.type [aX]+/-Is9
```

Description:

Load instruction fetches *N_elem* 32-bit elements from DMEM using the address and the address mapping specified. The contents are then latched, either entirely or partially, in continuous units of 32-bit elements onto a data register file line, or as 4, 8, 16, or 32-bit elements onto 4, 8, 16, or 32-element locations on the data register file.

- A 32-bit DMEM element is identified by the access pointer a*X*.
- A 4, 8 or 16-bit element within a 32-bit DMEM element is identified by the value contained in mem_elem_rd_ptr.

- A destination data register or a 4, 8, 16 or 32-bit element location within the data register file is identified by the value contained in r_wr_ptr.

When the load instruction with the ls8 offset field is used, the ld_mode cannot be specified in the instruction. In this case, the ld_mode will be provided by the last instruction that did use a ld_mode.

ld.mask.type instruction latches the entire contents of the DMEM line to the elem_mask register.

In indirect ld_mode, the DMEM address that was loaded into the mem_index0 or mem_index1 register replaces the a0 or a1 register, respectively, as the DMEM word address pointer. In indirect ld_mode, the m*X* and elem_offset fields are ignored.

The type field specifies whether the IPPU accesses the IPPU's DMEM or the VCPU's DMEM. The address pointer is the offset from the start of the DMEM type (bank).

Table 81. Load instruction arguments

Argument	Description
type	Indicates which DMEM to access. ippu vcpu
a*X*	Pointer to DMEM word location (X = 0, 1). 0 a0 1 a1
+/-	0 inc (add) 1 dec (subtract)
m*X*	Offset register specifying post-fetch (increment or decrement) offset for the DMEM access pointer a*X* (ignored in indirect mode) (X = 0, 1). 0 m0 1 m1
ls8	Signed 8-bit immediate field indicating a post-fetch (increment or decrement) offset for the DMEM access pointer a*X* (ignored in Indirect addressing). The architecture guarantees a wrap low. That is, when the DMEM access pointer aX would appear to go negative after increment or decrement, it actually goes to a high address (for example, 0 - 1 = 3ffff). **NOTE** This field CANNOT be used in conjunction with the ld_mode field.
ls5	Signed 5-bit immediate field indicating a post-fetch (increment or decrement) offset for the DMEM access pointer a*X* (ignored in indirect mode). **NOTE** This field CAN be used in conjunction with the ld_mode field.
ls9	Signed 9-bit immediate field indicating a post-fetch (increment or decrement) offset for the Compare Register file pointer (elem_mask_ptr).
elem_offset	Signed 3-bit immediate field indicating post-latch update (increment or decrement) for mem_elem_rd_ptr.

Table continues on the next page...

Table 81. Load instruction arguments (continued)

Argument	Description
ld_mode	Loading mode specifying that the address generation that can take following values: 000 normal: linear address mapping 001 br: bit-reversed address mapping 010 dr: digit-reversed address mapping 011 br_dr: digit and bit reversed mapping 100 vindirect: vectorized indirect mapping (see Vectorized indirect addressing for details) 101 ind: indirect mapping (see Indirect addressing for details) • In ld_mode == ind (indirect mode), the index pointed to by the previously instructed *ld.index.type* instruction (corresponding to the specified a*X*) is used as the DMEM address to load from. 110 ri: (relative indirect), relative indirect mapping • In ld_mode == ri (relative indirect mode), the index pointed to by the previously instructed ld.index.type instruction is added to the pointer aX to generate a memory address used as the DMEM address to load from. 111 Reserved
wr_offset	Signed 9-bit immediate field indicating a post-latch update (increment or decrement) for the data register write pointer r_wr_ptr.
latch_mode	0000 normal: Normal Mode. Latches the entire content of the read bus onto the destination register specified. With 2 AU configuration, the least significant 2 bits of the DMEM address (read) pointer are ignored; with 4 AU configuration, the least significant 3 bits of the DMEM address (read) pointer are ignored; With 8 AU configuration, the least significant 4 bits of the DMEM address (read) pointer are ignored; with 16 AU configuration, the least significant 5 bits of the DMEM address (read) pointer are ignored; with 32 AU configuration, the least significant 6 bits of the DMEM address (read) pointer are ignored; and with 64 AU configuration, the least significant 7 bits of the DMEM address (read) pointer are ignored.
	0001 e256: 256-bit Element Mode. Latches 256 bits (starting at the address pointed to by the DMEM address pointer) onto a 256-bit element location within the data register file. • The destination location is specified by the r_wr_ptr. The least significant 6 bits of the write pointer are ignored. • The least significant 3 bits of the DMEM address (read) pointer are ignored. **NOTE** This mode is inapplicable to VSPA versions with 2 AUs or less. This mode should not be used in such configurations.
	0010 e128: 128-bit Element Mode. Latches 128 bits (starting at the address pointed to by the DMEM address pointer) onto a 128-bit element location within the data register file. • The destination location is specified by the r_wr_ptr. The least significant 5 bits of the write pointer are ignored. • The least significant 2 bits of the DMEM address (read) pointer are ignored.

Table continues on the next page...

Table 81. Load instruction arguments (continued)

Argument	Description
	0011 e64: 64-bit Element Mode. Latches 64 bits (starting at the address pointed to by the DMEM address pointer) onto a 64-bit element location within the data register file. • The destination location is specified by the r_wr_ptr. The least significant 4 bits of the write pointer are ignored. • The least significant 1 bit of the DMEM address (read) pointer is ignored.
	0100 e32: 32-bit Element Mode. Latches a 32-bit element (pointed to by the DMEM address pointer) onto a 32-bit element location within the data register file. • The destination location is specified by the r_wr_ptr. The least significant 3 bits of the pointer are ignored.
	0101 e16: 16-bit Element Mode. Latches a 16-bit element (within the 32-bit element pointed to by the DMEM address pointer) onto a 16-bit element location within the data register file. • The 16-bit source element is specified by the most significant bit of mem_elem_rd_ptr. • The destination location is specified by the r_wr_ptr. The least significant 2 bits of the pointer are ignored.
	0110 e8: 8-bit Element Mode. Latches an 8-bit element (within the 32-bit element pointed to by the DMEM address pointer) onto a 8-bit element location within the data register file. • The 8-bit source element is specified by the most significant 2 bits of mem_elem_rd_ptr. • The destination location is specified by the r_wr_ptr. The least significant 1 bit of the pointer is ignored.
	0111 e4: 4-bit Element Mode. Latches a 4-bit element (within the 32-bit element pointed to by the DMEM address pointer) onto a 4-bit element location within the data register file. • The 4-bit source element is specified by mem_elem_rd_ptr. • The destination location is specified by the r_wr_ptr.
	1000 L2H: Low To Hign Mode. Latches the least significant *K* 32-bit elements of the read bus (where K = count given by the least significant *X* bits of the DMEM address pointer) onto the most significant *K* 32-bit locations of the specified data register. **NOTE** X= 2 with 2 AU configuration, or, 3 with 4 AU configuration, or 4 with 8 AU configuration, or 5 with 16 AU configuration, or 6 with 32 AU configuration, or 7 with 64 AU configuration.
	1001 H2L: Low To Low Mode. Latches the most significant *n_elem-K* 32-bit elements of the read bus (where K = count given by the least significant *X* bits of the DMEM address pointer) onto the least significant *n_elem-K* 32-bit locations of the specified data register.

Table continues on the next page...

Table 81. Load instruction arguments (continued)

Argument	Description
	NOTE X= 2 with 2 AU configuration, or, 3 with 4 AU configuration, or 4 with 8 AU configuration, or 5 with 16 AU configuration, or 6 with 32 AU configuration, or 7 with 64 AU configuration.
	1010 L2L: Low To Low Mode. Latches the least significant *K* 32-bit elements of the read bus (where K = count given by the least significant *X* bits of the DMEM address pointer) onto the least significant *K* 32-bit locations of the specified data register. **NOTE** X= 2 with 2 AU configuration, or, 3 with 4 AU configuration, or 4 with 8 AU configuration, or 5 with 16 AU configuration, or 6 with 32 AU configuration, or 7 with 64 AU configuration.
	1011 H2H: High To Hign Mode. Latches the most significant *n_elem-K* 32-bit elements of the read bus (where K = count given by the least significant *X* bits of the DMEM address pointer) onto the most significant *n_elem-K* 32-bit locations of the specified data register. **NOTE** X= 2 with 2 AU configuration, or, 3 with 4 AU configuration, or 4 with 8 AU configuration, or 5 with 16 AU configuration, or 6 with 32 AU configuration, or 7 with 64 AU configuration.
	1110 - 1111 Reserved

6.5 Load memory index instructions

IPPU instructions summary

Syntax:

```
ld.index.type [aX]+/-mX
ld.index.type [aX]+/-Is9
```

Description:

Load memory index instruction fetches a 32-bit element from DMEM specified by a0 or a1 as specified in the operand and writes the least significant *A* bits of this element to mem_index0 or mem_index1 register respectively.

The type field specifies whether the IPPU accesses the IPPU's DMEM or the VCPU's DMEM. The address pointer is the offset from the start of the DMEM type (bank).

Table 82. Load memory index instruction arguments

Argument	Description
type	Indicates which DMEM to access. ippu vcpu

Table continues on the next page...

Table 82. Load memory index instruction arguments (continued)

Argument	Description
a*X*	Pointer to DMEM location (X = 0, 1). 0 a0 1 a1 When pointer a*X*==a0 load into register mem_index0 When pointer a*X*==a1 load into register mem_index1 Note: The index register loaded X is the same X as the a*X* used in this instruction.
+/-	0 inc (add) 1 dec (subtract)
m*X*	Offset register specifying a post-fetch (increment or decrement) offset for the DMEM access pointer a*X*. 0 m0 1 m1
Is9	Signed 9-bit immediate field indicating a post-fetch (increment or decrement) offset for the DMEM access pointer a*X*.

6.6 Store instructions

IPPU instructions summary

Syntax:

```
st.type [aX]+/-mX, elem_offset, st_mode, rd_offset, write_mode
st.type [aX]+/-Is8, elem_offset, rd_offset, write_mode
st.type [aX]+/-Is5, elem_offset, st_mode, rd_offset, write_mode
```

Description:

The Store instruction stores a 16, or 32-bit element from a data register file to a 16, or 32-bit element, or from an entire data register file line to an entire DMEM line.

- The source element/line location is identified by r_rd_ptr.
- The destination line or 32-bit element location in DMEM is identified by aX.
- The 16-bit element within the 32 bit destination location is specified by mem_elem_wr_ptr.

When the instruction with the Is8 field is used, the st_mode cannot be specified in the Store instruction. In this case, st_mode used in the last executed st.mem instruction is used.

- In indirect mode, the DMEM address contained in mem_index0 (or mem_index1) is used as the DMEM pointer to write to or from, based on whether a0 or a1 is used in the aN field, respectively. The fields mX, elem_offset are ignored.

The <mem_type> field specifies whether the IPPU accesses the IPPU DMEM or the VCPU's DMEM. The address pointer is the offset from the start of the DMEM type (bank).

Table 83. Store instruction arguments

Argument	Description
type	Indicates which DMEM to access: {ippu,vcpu}
a*X*	Pointer to DMEM word location (X = 0, 1). 0 a0 1 a1
+/-	0 inc (add) 1 dec (subtract)
m*X*	A post-fetch (increment or decrement) offset specified in offset registers for the DMEM pointer (X = 0, 1) (ignored in indirect mode). 0 m0 1 m1
ls8	Signed 8-bit immediate field, indicating a post-write (increment or decrement) offset for the DMEM access pointer aN (ignored in indirect mode). This field CANNOT be used in conjunction with the st_mode field.
ls5	Signed 5-bit immediate field, indicating a post-write (increment or decrement) offset for the DMEM access pointer aN (ignored in indirect mode). This field CAN be used in conjunction with the st_mode field.
elem_offset	Signed 3-bit immediate field, indicating post-latch update (increment or decrement) for mem_elem_wr_ptr.
st_mode	Storing mode that can take following values: 000 normal: linear address mapping 001 br: bit-reversed address mapping 010 dr: digit-reversed address mapping 011 br_dr: digit and bit reversed mapping 101 indirect: indirect mapping (see Indirect addressing for details) • In st_mode == indirect mode, the index pointed to by the previously instructed ld.type indexX instruction (corresponding to the aX specified), is used as the DMEM address to store to. N 110 rind: relative indirect mapping • In st_mode == relative indirect mode, the index pointed to by the previously instructed ld.index.type instruction is added to the pointer aX to generate a memory address used as the DMEM address to load from. 100, 111 Reserved

Table continues on the next page...

Table 83. Store instruction arguments (continued)

Argument	Description
rd_offset	Signed 9-bit immediate field, indicating post-store update (increment or decrement) for the data register read pointer r_rd_ptr.
write_mode	000 normal: Writes entire data register file line (specified by the MS bit of r_rd_ptr) to the DMEM address (specified by aX).
	001 e32: Writes the 32-bit element (specified by r_rd_ptr) to the 32-bit DMEM location (specified by aX). • The least significant 3 bits of r_rd_ptr are ignored.
	010 e16: Writes the 16-bit element (specified by r_rd_ptr) to a 16-bit element location (within the 32-bit DMEM location specified by aX). • The least significant 2 bits of r_rd_ptr are ignored. • The destination 16-bit element within the 32-bit DMEM location is specified by the MS bit of mem_elem_wr_ptr.
	011, 100, 101, 111: Reserved

6.7 Set range instructions

IPPU instructions summary

Syntax:

```
set.range aY, asA, asB
set.range aY, asA, Iu19
```

Description:

The set range instruction sets the circular buffer boundaries for the DMEM address pointer a*Y*.

The start address of the circular buffer is specified via the general register as*A*.

as*B* and Iu19 indicate the number of 16-bit elements in the buffer. That is, asB and Iu19 are the circular buffer size in 16-bit element.

The end address of the circular buffer can be (indirectly) calculated either:

- indirectly via the general register as*B* indicating the number of 16-bit elements in the buffer (in this case, the circular buffer end address is "as*A* + as*B* -1", "as*B* = num elements" and "as*B* > 0"), or
- indirectly via the immediate value Iu19 indicating the number of 16-bit elements in the buffer (in this case, the circular buffer end address is "as*A* + Iu19 - 1", "Iu19 = num elements" and "Iu19 >= 1").

NOTE

The circular buffer will be disabled when:

- as*A*=0 and as*B*=0 for set.range a*Y*, as*A*, as*B*
- as*A*=0 and Iu19=0 for set.range a*Y*, as*A*, Iu19

NOTE

Addresses are to 16-bit elements and size is number of 16-bit elements.

Table 84. Set Range instruction arguments

Argument	Description
a*Y*	Specifies the pointer to which the set range instruction applies (Y = 0, 1). 0 a0 1 a1
as*A*	A general register (as0, as1, as2 or as3) which specifies the start address of the circular buffer. 00 as0 01 as1 10 as2 11 as3
as*B*	A general register (as0, as1, as2 or as3) which specifies the size of the circular buffer (number of 16-bit elements). 00 as0 01 as1 10 as2 11 as3
Iu19	A 19-bit unsigned immediate value which specifies the size of the circular buffer in number of 16-bit elements.

6.8 Configure bit-reversal, digit-reversal engine instructions

IPPU instructions summary

Syntax:

```
set.br aX, br_mode
set.dr.radix digit_index, radix, Iu11
set.dr.config digit_index, Iu3, Iu11
```

Description:

These instructions configure the bit-reversal, digit-reversal and bit-digit-reversal engines for the DMEM address pointers.

set.br, *set.dr.radix* and *set.dr.config* are actual hardware instructions.

Table 85. Configure bit-reversal, digit-reversal engine instructions

	Instruction	Description
Hardware instructions	set.br	Configures the BR engine for use in either br or brdr mode.
	set.dr.radix	Configures the radix and address increment for each prime digit corresponding to the prime factors of the DFT size.
	set.dr.config	Configures the initial values of DR address generator counter for each prime digit corresponding to the prime factors of the DFT size.

Table 86. Configure bit-reversal, digit-reversal engine instruction arguments

Argument	Description
a*X*	X = 0, 1. 0 a0 1 a1
br_mode	Bit-reversal mode. 0000 fft32 0001 fft64 0010 fft128 0011 fft256 0100 fft512 0101 fft1024 0110 fft2048 0111 fft4096 1000 fft8192 1001 fft16384 1010 - 1111 Reserved
digit_index	One of { 1..10 } and identifies one of the 10 digits used for DR.
radix	3-bit immediate value indicating the radix (this version supports 2, 3 and 5). 000 Reset radix registers when lu11 also is zero. 001 Not valid 010 2 prime radix factor 011 3 prime radix factor 100 Not valid 101 5 prime radix factor 110 Not valid 111 Not valid
lu3	3-bit immediate value indicating the counter values for address generation.
lu11	*set.dr.radix digit_index, radix, lu11* : 11-bit unsigned immediate value indicating the address increment. Value of zero together with radix=0 resets the radix registers. *set.dr.config digit_index, lu3, lu11* : 11-bit unsigned immediate value indicating the address.
num2	Number of prime factor 2 in DFT size.
num3	Number of prime factor 3 in DFT size.

Table continues on the next page...

Table 86. Configure bit-reversal, digit-reversal engine instruction arguments (continued)

Argument	Description
num5	Number of prime factor 5 in DFT size.
dr_offset	Index of elements to be accessed first in the digit-reversed array.

In the case where radix and Iu11 address increment are both zero, the radix registers are reset and equivalent to the following instructions:

- set.dr.radix 1, 0, 0;
- set.dr.radix 2, 0, 0;
- set.dr.radix 3, 0, 0;
- set.dr.radix 4, 0, 0;
- set.dr.radix 5, 0, 0;
- set.dr.radix 6, 0, 0;
- set.dr.radix 7, 0, 0;
- set.dr.radix 8, 0, 0;
- set.dr.radix 9, 0, 0;
- set.dr.radix 10, 0, 0;

6.9 Move register instruction

IPPU instructions summary

Syntax:

```
mv ippu_regA, ippu_regB
mv  ippu_regA, Iu18
```

Description:

The Move Register instruction moves the value of one register to another. It can also set the value of DMEM address pointers, offset registers and pointers to the specified 18-bit-wide immediate value field.

If the destination register width is less than the source register, then only the relevant least significant bits of immediate field are used. If the destination register width is greater than the source register, then it is padded with zeros.

Table 87. Move Register instruction arguments

Argument	Description
ippu_regA	Move destination register One of: • aX (X = 0, 1) • mX (X = 0, 1) • asX (X = 0, 1, 2, 3) • vindx_ptr • r_rd_ptr

Table continues on the next page...

Table 87. Move Register instruction arguments (continued)

Argument	Description
	• r_wr_ptr • elem_mask_ptr • mem_elem_rd_ptr • mem_elem_wr_ptr
ippu_regB	Move source register One of: • aX (X = 0, 1) • mX (X = 0, 1) • asX (X = 0, 1, 2, 3) • vindx_ptr • r_rd_ptr • r_wr_ptr • elem_mask_ptr • mem_elem_rd_ptr • mem_elem_wr_ptr
Iu18	18-bit immediate value

6.10 Load input argument instruction

IPPU instructions summary

Syntax:

```
ld asX, Iu6
```

Description:

The load input argument instruction writes the least significant A bits of a 32-bit element within a pre-defined sector in IPPU DMEM space onto the register specified. The address offset of the 32-bit element within the pre-defined memory space is specified as an input argument.

The starting address of the sector is defined by the content of the IP register IPPUARGBASEADDR. For example, if IPPUARGBASEADDR contains 0x0001 and if the instruction ld as0, 3 is executed, then the 32-bit element in IPPU_DMEM[1 + 3] is read and the least significant 18 bits are written to as0.

Table 88. Load input argument instruction arguments

Argument	Description
as*X*	Destination register (X = 0, 1, 2, 3).
Iu6	Unsigned 6-bit immediate field indicating the address offset in 32-bit (4-bytes) units (with respect to the base of the assigned IPPU DMEM sector) from where to populate the destination register. Addressing is on a 32-bit element boundary.

6.11 Compare instruction

IPPU instructions summary

Syntax:

```
cmp.bit IsY
```

Description:

The Compare instruction compares a bit in the compare register (with the location specified by elem_mask_ptr) and sets the ZERO flag accordingly. The elem_mask_ptr is post-updated, based on the immediate offset specified by IsY.

Table 89. Compare instruction arguments

Argument	Description
IsY	Signed Y-bit immediate post (incremental or decremental) offset for the elem_mask_ptr.

6.12 Jump instructions

IPPU instructions summary

Syntax:

```
jmp Iu16
jsr Iu16
jsr.z Iu16
jsr.nz Iu16
jmp.z Iu16
jmp.nz Iu16
rts
rts.z
rts.nz
```

Description:

The Jump instruction executes unconditional and conditional jumps to the destination label specified.

Table 90. Jump engine instructions

Instruction	Description
jmp jsr rts	Unconditional jump
jmp.z jsr.z rts.z	Jump is taken if ZERO flag is set.
jmp.nz jsr.nz rts.nz	Jump is taken if ZERO flag is not set.
jsr	After taking the jump, it will return when it encounters the "rts" instruction.

Each of these Jump instructions has a 2-cycle pipeline delay. Therfore, the 2 instructions immediately following any of these instructions WILL be executed BEFORE the jump is taken, for unconditional jumps or rts, and regardless of the outcome, for conditional jumps or rts.

Restrictions:

See Section Delay slot considerations for important usage restrictions.

Table 91. Jump instruction arguments

Argument	Description
Iu16	Destination program label to jump to.

6.13 Loop instructions

IPPU instructions summary

Syntax:

```
set.loop Iu1, Iu8
set.loop Iu1, asX
loop_begin Iu1
loop_end Iu1
```

Description:

The loop instruction sets, starts and ends a do-loop. The loop is executed as many times as specified by either the lower 8 bits of a general purpose register or an immediate 8-bit field in the *set.loop* instruction. Two-level loop nesting is allowed.

There is a slight difference how the loop iteration count is applied when used by either the immediate field or the asX register. When applying the iteration count via immediate field, the loop count is exactly what is programmed, with valid range 1-256. When applying the iteration count via asX register, the loop count is one more than the value in the asX register (bits 0-7). That is, for asX[7:0] values 0-255, the actual loop count will be 1-256, respectively.

The following is a list of rules to follow when using the loop instructions:

- Back-to-back single instruction loops are not permitted. There must be at least one other instruction in between two single instruction loops. Loops that are greater than one instruction can be back to back.
- A single instruction loop cannot immediately follow a *loop_begin* instruction of the same or different index.
- A single instruction loop cannot be the first instruction of a code section.
- A *loop_end* instruction cannot be specified in an instruction which immediately follows another instruction that specifies a *loop_end* instruction. There has to be at least one other instruction in between.
- Every *loop_begin* instruction must have a corresponding *set.loop* instruction specified somewhere BEFORE the *loop_begin* instruction is executed. It is possible to have multiple *loop_begin*/*loop_end* pairs reference a common loop index (and count) following a single *set.loop* instruction with the same index.
- Every *loop_begin* instruction must have a corresponding *loop_end* instruction specified somewhere in the same instruction or AFTER the *loop_begin* is executed.
- Loops of different indices can be nested. In other words, a *loop_begin* AND a *loop_end* can be inside the body of a *loop_begin*/*loop_end* pair of a different index. However, a *loop_begin*/*loop_end* pair cannot straddle a *loop_begin* without a *loop_end* or a *loop_end* without a *loop_begin*.
- There must be a one-cycle delay between writing to an as*X* register and using it as iteration count source.
- There must be atleast five instructions between 'ld.arg into asX' and using asX in a 'set.loop asX' instruction.
- A *jmp*/*jsr*/*rts* instruction cannot be specified in an instruction which also contains a *loop_end* instruction.
- Neither a *loop_begin* nor a *loop_end* instruction can be specified in the delay slots of a *jmp*/*jsr*/*rts* instruction.
- A *loop_end* instruction cannot be the target of a *jmp/jsr/rts* instruction.

Table 92. Loop instruction arguments

Argument	Description
Iu1	A 1-bit immediate value which specifies which level of loop is being configured (0, 1).
Iu8	An 8-bit immediate value which specifies the loop count. The count is a sticky parameter which is preserved for each loop index independently of the number of *loop_begin*/*loop_end* pairs referencing this index.
asX	General register, where X = 0, 1, 2, 3. Note that only the lower 8-bit of of the asX register are used to configure the loop.

6.14 Done instruction

IPPU instructions summary

Syntax:

```
done
```

Description:

The Done instruction ends the current execution session and sets the "ippu_done" flag.

6.15 Clear register instruction

IPPU instructions summary

Syntax:

```
clr r0;
clr r1;
clr r2;
clr r3;
clr r0,r1;
clr r0,r1,r2,r3;
clr <r0>|<,r1>|<,r2>|<,r3>;
```

Description:

The Clear Register instruction clears any of r0, r1,r2, r3 or all registers and sets them to all zeros. See Set/clear element mask register instructions for a separate `clear mask all` instruction.

Table 93. Clear Register instruction arguments

Argument	Description
Rx	r0, r1, r2, r3

6.16 Set/clear element mask register instructions

IPPU instructions summary

Syntax:

```
set.mask IuY
set.mask all
clr.mask IuY
clr.mask all
```

Description:

The set/clear mask instructions set one bit or all bits of the elem_mask register as follows:

- *set.mask* instruction sets the bit at the specified bit index Iu*Y* to 1 in the elem_mask register.
- *set.mask.all* instruction sets all of the bits in the elem_mask register to 1.
- *clr.mask* instruction clears (sets to 0) the bit at the specified bit index Iu*Y* in the elem_mask register.
- *clr.mask.all* instruction clears (sets to 0) all the bits in the elem_mask register.

Table 94. Set/clear element mask register instruction arguments

Argument	Description
Iu*Y*	*Y*-bit immediate field specifying which bit to set in the elem_mask register.

6.17 Add instructions

IPPU instructions summary

Syntax:

```
add aX, Is8
add aX, mY
add.cb aX, Is8
add.cb aX, mY
```

Description:

The ‘add’ instruction adds a value specified in Is8 or m*Y* to address pointer a*X* (a0 or a1).

The ‘add.cb’ instruction adds a value specified in Is8 or m*Y* to address pointer a*X* (a0 or a1), and then adjust the result to reside inside the circular buffer range as defined for this address pointer.

Table 95. Add instruction arguments

Argument	Description
a*X*	Pointer to DMEM location (X = 0, 1) 0 - a0 1 - a1
Is8	8-bit signed immediate value which specifies the amount to add to address pointer aY (a0 or a1)
m*Y*	Offset register specify the increment or decrement offset for the DMEM access pointer a*Y* (Y = 0, 1) 0 - m0 1 - m1

6.18 Advanced features/usage notes

6.18.1 Delay slot considerations

The delay slots of the `loop`, `jmp/jsr/rts` instructions place certain restrictions on the usage of loops and jumps:

- `jmp/jsr/rts` instruction cannot be specified in an instruction containing a `loop_end`.
- `rts` cannot be specified in the delay slots of any `jmp/jsr` instruction.

- `jmp/jsr` can be placed in the delay slots of other `jmp/jsr` instructions. However, caution is advised with such usage because all delay slots of each `jmp` encountered will have to be accounted for while tracking program behavior.

6.18.2 BR - Bit-reversal

The bit-reversal feature allows data to be read from Data Memory or written to Data Memory in a bit-reversed order. The bit reversal engine in the IPPU can be used to reorder bit-reversed outputs of the DIF-FFT to linear form, or used to reorder the linear inputs of the DIT-FFT to bit-reversed form.

- Radix-2 FFTs using decimation-in-frequency (DIF) methods take inputs in linear order, and generate outputs in bit-reversed order.
- Radix-2 FFTs using decimation-in-time (DIT) method take inputs in bit- reversed order, and generate outputs in linear order.

To operate the bit reversal engine, the following steps must be followed:

- Only a0 and a1 can be used as pointers for the DMEM ld operation.
- Define the range of the input/output data in DMEM for reading/writing bit-reversed data- this can be achieved by defining the `set.range` instruction on the respective DMEM read/write pointer.
- Configure the bit-reversal mode- this can be achieved with the `set.br` instruction.
- Use the `ld.type` or `st.type` instruction in br mode to read/write data. Enable the bit-reversal addressing mode is done with the set.br instruction. The Bit-Reversal (br) mode is sticky.

6.18.3 Indirect addressing

In the indirect addressing mode of operation, DMEM addresses to required elements are stored within contiguous locations in DMEM.

In the indirect addressing mode (ld_mode=indirect), the following instructions are used in conjuction with the *ld.index.type* instruction.

```
ld.type [aX]+/-mX, elem_offset, ld_mode, wr_offset, latch_mode
ld.type [aX]+/-Is5, elem_offset, ld_mode, wr_offset, latch_mode
```

The *ld.index.type* instruction fetches a DMEM element from memory pointed by a*X* and writes the least significant 18 bits to the mem_index*X* register.

When *ld.type [aX]* or *st.type [aX]* instructions are set to indirect mode (ld_mode=indirect or st_mode=indirect, respectively), the instructions use the address contained in mem_index*X* to fetch/write based on the a*X* used.

For example, let

```
VCPU.DMEM[0] = 56
VCPU.DMEM[1] = 92
```

where 56 and 92 refer to DMEM addresses in the VCPU/IPPU space. Let us assume that they refer to the IPPU DMEM space, and let

```
IPPU.DMEM[56] = 3.3 + i*4.4
IPPU.DMEM[92] = 2.2 + i*5.5
```

Consider the following instructions:

```
mv a0, 0;
ld.index.vcpu [a0] + 1;                  // STATE: mem_index = UNKNOWN
ld.index.vcpu [a0] + 0;                  // STATE: mem_index = UNKNOWN
nop;                                      // STATE: mem_index = UNKNOWN
ld.ippu [a0] + 0, 0, indirect, +8, e32;  // STATE: mem_index = 56
ld.ippu [a0] + 0, 0, indirect, +8, e32;  // STATE: mem_index = 92
nop;
nop;                                      // STATE: r0[31:00] = 3.3 + i*4.4
```

```
nop;                                        // STATE: r0[63:32] = 2.2 + i*5.5
```

The indirect mode for store instruction will operate using the same logic as above.

6.18.4 Vectorized indirect addressing

In the vectorized indirect addressing mode of operation, DMEM addresses are generated by extracting the 16-bit "index" from register R1 and adding it to address register a0 or a1.

The vector indirect addressing in the IPPU can be used to reorder de-interleaved outputs.

In the vectorized indirect addressing mode `ld.type [aX]+/ld_mode=vindirect+/latch_mode` DMEM address is calculated as follows:

```
DRAM_address = aX + (R1[vindx_ptr] >> e_adj)
Where: R1[vindx_ptr] is 16-bit R1[(vindx_ptr+1)*16 : vindx_ptr*16],
e_adj  = 0: if latch_mode=e32; 1: if latch_mode=e16; 2: if latch_mode =e8; 3: if
latch_mode=e4; and 0: otherwise.
se_ptr = R1[vindx_ptr] & se_mask; Where se_mask=0x0: if latch_mode=e32; 0x1: if
latch_mode=e16; 0x3: if latch_mode =e8, and 0x7: if latch_mode=e4.
```

The `ld.type [aX]+/ld_mode=vindirect+/latch_mode` instruction fetches a 32-bit element from DMEM memory pointed to by `DRAM_address` and writes sub_element (e32, e16, e8, or e4) as pointed to by `se_ptr` into `R0[r_wr_ptr]`.

The vindirect mode can be used in load instructions only. It is not valid in store instructions.

Any `ld.type [aX]` instruction that results in R1 as the destination will clear the vindx_ptr register.

Any `ld.type [aX]+/ld_mode=vindirect` instruction will auto-increment the vindx_ptr register.

Example code using the vindirect addressing mode for deinterleaving:

```
mv r_wr_ptr, (N_AU * 8);                    //point r_wr_ptr to R1
mv a0, as1;                              //set a0 to lookup table address
ld.ippu [a0]+4, 0, normal, 0, normal; //load a line of lookup table
nop;                                     //pipeline wait
set.loop 0, 8;
mv r_wr_ptr, 0;                          //point r_wr_ptr to LSB of R0

//in loop: read next 8-bit element and store in R0
ld.ippu [a1]+0, 0, vindirect, 2, e8; loop_begin 0; loop_end 0;
nop;                                     //pipeline wait
nop;                                     //pipeline wait
mv r_wr_ptr, (N_AU * 8);                 //point r_wr_ptr to R1 for next iteration
```

Chapter 7
DMA Controller

7.1 Direct memory access unit (DMA)

The VSPA DMA is capable of transferring data between the VSPA memories and the VSPA AXI bus interface. It moves data between VCPU DMEM, RF transceivers, and SRAM (used to hold data for symbol processing). It is also intended to load VCPU's program RAM (PRAM) and the IPPU's program RAM (IPPU PRAM).

7.1.1 DMA module operation

The VSPA DMA is an autonomous unit within VSPA, and is controlled by an array of VSPA IP registers; it can therefore be controlled by the VCPU and/or any another IP bus master with access to the VSPA IP bus slave port.

The DMA controls an AXI bus master interface, and has priority access to and control over the VCPU's DMEM. So, when a DMA operation is triggered, the DMA will steal necessary cycles from the VCPU when it needs access to DMEM. DMA accesses to VCPU PRAM are allowed while the VCPU is in operation. The DMA will steal the necessary VCPU cycles to write to the VCPU PRAM. Aside from the delay due to the stolen VCPU cycles, the VCPU's operation is unaffected. The IPPU PRAM can only be written by the DMA when the IPPU is not accessing its PRAM - it must be idled if its PRAM contents are to be replaced.

The DMA has 16 channels, and each channel is driven by a 2-entry FIFO. Each FIFO entry holds all the information required for a DMA transfer, including starting addresses, selected VSPA memories, byte count, and transfer mode. There are a set of global status registers indicating whether each channel's transfers are active, have completed, or had any type of error.

There is a common status/abort control register that, when read, shows that channels that have pending activity, and when written to a "1" will abort all pending activity for the selected channels. A single write to this common control register can deactivate any desired number of channels.

The DMA has 4 independent data movement engines, one that writes to the AXI bus, and 3 that read from the AXI bus. All of the engines operate concurrently, and the write engine is independent from the read engines. There are separate arbiters for the write engine and the read engines.

The DMA uses several sets of internal registers (buffers) to improve the timing and efficiency of operations. The DMA has 2 DMEM buffers, each equal in width to a DMEM line. There are AXI bus width data buffers that are coupled to the AXI rdata and wdata buses. There are also PRAM and IPPU PRAM buffers for writing AXI data into PRAM and IPPU PRAM.

The DMA uses the ACTIVE_THREAD_ID to generate the awthread and arthread outputs. These outputs reflect the state of the ACTIVE_THREAD_ID at the time the DMA channel was programmed. DMA transactions are marked as SUPV, and have unrestricted DMEM write permissions. If the host programs a DMA transaction, the DMA will mark it as SUPV with thread ID=0. This should grant host programmed transactions full access to any memory locations, but will identify them as thread ID=0.

7.1.2 Issuing DMA commands

An entity desiring a DMA operation programs the command/control registers. Note that this is a command based interface, with a four register set (DMA_DMEM_PRAM_ADDR, DMA_AXI_ADDR, DMA_AXI_BYTE_CNT, and DMA_XFR_CTRL) that acts as the command buffer. The interface must be programmed by writing last to the DMA_XFR_CTRL register, which transfers the four-register wide command into the channel's control FIFO (assuming it's not already full).

Note that the VCPU can write any combination of bits via its register write mask; it is not required to write all bits. In fact, it can do a write that will update no bits if the register write mask is set to 0. So, since the DMA command interface acts as a command buffer, the VCPU can issue a DMA command that exactly duplicates the last DMA command issued by the VCPU merely by writing to the DMA_XFR_CTRL register with a write mask of 0. This of course assumes the other three registers of the command buffer interface have not been written by the VCPU since the last command was launched.

The host processor can program the DMA in the same fashion as the VCPU, using registers at the same addresses. However, a separate command buffer is implemented for the host, so that the host and the VCPU do not interfere with each other when programming the DMA. Unless the host and VCPU are sharing control over the same DMA channels, there is no need for a software semaphore to control access to the DMA command interfaces. The host interface is not readable and does not provide

bit write capabilities, so whenever the host writes to the command registers, all bits must be written. So, a host routine that programs the DMA registers should not be interruptible by another host or host routine that can program the DMA registers.

7.1.3 DMA channel arbitration

The DMA has 4 independent data movement engines, one that writes to AXI slaves and 3 more that read from AXI slaves. The write engine has its own arbiter, and the 3 read engines share an arbiter. All engines can operate concurrently. The arbiters are round robin, and arbitration occurs at the completion of each AXI burst. Channel 0 is special in that it is exempt from round robin arbitration - it always wins arbitration. This makes it a good choice for priority reads and writes to AXI memory for purposes such as look-ups of data stored in large tables.

NOTE

If multiple channels are configured to read from AXI and channel 0 is configured as one of these channels, the usual round robin arbitration is disrupted and the next 2 highest priority channels will always win arbitration on the remaining read engines. As any channel completes, the next highest channel will become one of the always winning channels. When channel 0 completes the usual round robin arbitration will resume beginning with the channels currently running.

NOTE

Channels requiring external trigger will not participate in arbitration whenever their triggers are seen as negated.

The DMA has higher priority to access DMEM than the VCPU or the IPPU, so while the DMA is accessing the DMEM, the VCPU and IPPU may be stalled if they are also trying to access DMEM during the same cycle. This should not affect the VCPU too significantly, because the DMA has DMEM buffers that it uses to hold an entire DMEM line's worth of data. It will take multiple clock cycles to transfer enough data to the AXI bus, to equal the amount contained in the DMEM line buffer.

7.1.4 DMA deinterleaving engine

The VSPA DMA has a special deinterleaving (DI) engine, which may be used for storing LLRs or other interleaved data from DMEM to AXI memory, while simultaneously deinterleaving the data at a nibble (4-bit mode), byte level (8-bit mode), 16-bit mode or 32-bit mode.

The DMA has only one DI engine, so it can only allow one command involving DI in the DMA command buffer at a time. So once a DI command has been issued to the DMA, additional DI commands will be refused until the original command has fully completed. This is necessary to guarantee that the DI engine's resources will remain coherent.

7.1.4.1 DI table structures in DMEM

The DMA DI engine expects the data that is to be deinterleaved and written into AXI to be held in two tables in the DMEM. The first table, called the address table, contains the AXI destination addresses for the data to be written to AXI. The second table, called the data table, contains the data that is to be written to the AXI addresses in the address table.

The address table consists of 32-bit addresses. The data table consists of either 4, 8, 16, or 32 bit data values. The values in the data table are understood by the DMA to be 4, 8, 16, or 32 bit according to the DI_mode bits specified when the DI command is issued - this is done via the DI mode selection bits in the DMA_XFR_CTRL register.

The starting DMEM address of the address table must be aligned to a 32-bit boundary in DMEM. There are no restrictions on the starting DMEM address of the data table.

7.1.4.2 DI modes

The 8-bit DI mode expects the addresses in the AXI address table to be a series of AXI byte addresses (each 32-bits in size), and for the data in the data table to be a series of 8-bit data values.

There are two 4-bit DI modes. The address table used by the 4-bit DI modes still consists of 32-bit addresses, but they are nibble addresses as opposed to the byte addresses used by the 8-bit DI mode. Each 4-bit DI mode can store 4-bit data of one half of the AXI address space. One mode can store data into the lower 2 GB of the 4 GB AXI address space; the other mode can store data into the upper 2 GB of the 4 GB AXI address space. In order to use the 4-bit DI mode the byte address must be left shifted

by 1 bit to create the nibble addresses. If the MSB of the byte address is 0, then di_mode 010b is used. If the MSB of the byte address is 1, then di_mode 011b is used.

For example:

1. Software needs to write to address 0x8000_0000 in 4-bit De-interleaving mode. User needs to select the 011b - 4-bit de-interleave, AXI address MSB=1, as the MSB of the address is 1. Now in this case, the AXI address that needs to be filled in the DI AXI address table will be 0x80000000<<1 instead of only 0x8000_0000. The left shift will result in 0x0000_0000.
2. Software needs to write to address 0x3443_0000 in 4-bit De-interleaving mode. User needs to select the 010b - 4-bit de-interleave, AXI address MSB=0, as the MSB of the address is 0. Now in this case, the AXI address that needs to be filled in the DI AXI address table will be 0x34430000<<1 instead of only 0x3443_0000. The left shift will result in 0x6886_0000.

The 16 bit mode writes 16-bit data. The data table must be 16-bit aligned in DMEM, and all the addresses in the address table must be 16-bit aligned AXI addresses.

The 32-bit mode writes 32-bit data. The data table must be 32-bit aligned in DMEM, and all the addresses in the address table must be 32-bit aligned AXI addresses.

7.1.4.3 DI engine arbitration

The DI engine requests accesses that are arbitrated against the other channels in the same fashion as non-DI arbitration. If channel 0 is used for DI, these accesses will have highest priority until all are completed. If any channel other than 0 is selected for DI writes, standard round robin rules apply. This arbitration policy governs AXI accesses, but not DMEM accesses. The DI state machine has highest priority when reading the address and data table entries from DMEM. Since there are separate DMEM line buffers for DI addresses and DI data, DI DMEM accesses are infrequent.

7.1.4.4 DI special notes

A channel programmed for DI storage cannot be aborted - the abort command will be ignored. External trigger of a DI operation is illegal.

7.1.5 Effect of invasive debug on DMA

During debug, if the debugger requests the VSPA to halt, the DMA will also halt at the next AXI burst boundary. Single steps of the VCPU do not affect the DMA. The DMA will remain halted until a resume is executed.

7.1.6 DMA use with FIFOs

There is a special characteristic of the DMA that can allow the user to control the burst beat count when transferring data to or from a FIFO.

Ordinarily the DMA will perform 16-beat burst transfers whenever possible in order to achieve the maximum possible DMEM and AXI bus utilization efficiency. The AXI protocol requires that a burst must never cross a 4K byte address boundary. So, the only time the DMA will use a burst size of less than 16 beats is when it would cause a 4K boundary crossing, or when there is less data to be transferred in order to complete the programmed command.

The DMA calculates and obeys a "virtual" 4K boundary restriction even when it is transferring to a fixed address, that is, a FIFO. So, if the FIFO is decoded into a full 4K byte region where any access to the region is considered a FIFO access, this feature can be used to control the burst size. To use a 16 beat burst, the DMA should be programmed in fixed burst mode, with the starting address of the FIFO set to the lowest numerical address that will access the FIFO. To use a 1 beat burst, the DMA should be programmed in fixed burst mode, with the starting address of the FIFO set to the highest AXI bus width aligned numerical address that will access the FIFO. Consider the examples below:

FIFO address decode range 0x10000 - 0x10FFF (4K byte address space) AXI data bus width of 128 bits (16 bytes)

1-beat burst - set DMA AXI address to 0x10FF0 (0x11000 - 1 beat x 16bytes)

2-beat burst - set DMA AXI address to 0x10FE0 (0x11000 - 2 beats x 16bytes)

3-beat burst - set DMA AXI address to 0x10FD0 (0x11000 - 3 beats x 16bytes)

.....

.....

.....

14-beat burst - set DMA AXI address to 0x10F20 (0x11000 - 14 beats x 16bytes)

15-beat burst - set DMA AXI address to 0x10F10 (0x11000 - 15 beats x 16bytes)

16-beat burst - set DMA AXI address to 0x11F00 (0x10000 would also work)

7.1.7 DMA features not supported

The following DMA features are not supported:

- Bit/byte reordering.
- Endian conversion: little endian operation only
- PRAM accesses that start with an AXI address that is misaligned to the larger of a 64 bit boundary or the AXI data bus width
- IPPU PRAM accesses that start with an AXI address that is misaligned to the larger of a 32 bit boundary or the AXI data bus width
- VPRAM, and IPRAM share a write buffer. So if more than one DMA command targets these resources, the results will be non-deterministic. It's up to the programmer to make sure that no more than one active DMA command at a time targets any of these resources. For example, channel 2 cannot be programmed to load VPRAM if channel 1 is programmed to load IPRAM at the same time. Also, more than one channel cannot target IPRAM at the same time. The same is true of VPRAM.
- Other transfers that start at a DMEM or AXI address that is not aligned to the AXI bus width
- Hardware semaphores for reserving channels for different masters or threads; however the DMA does maintain separate control registers (at the same address) for the host and for the VCPU. So both the VCPU and the host can program DMA channels at the same time, but they should each stick to an agreed upon group of channels that are reserved for them only
- Accesses smaller than 8 bits. AXI bus protocol does not directly support 4 bit reads or writes; minimum granularity is 8 bits. If LLRs are defined to be 4 bits, then this will always require transferring an even number of 4-bit LLRs. LLR (deinterleaving) support is chip specific. So it only applies if DMA_LLR_ENG=1.
- AXI reads smaller than the width of the AXI data bus. Assuming the data bus to be 128 bits, AXI reads will appear to be a minimum of 128 bits (1 beat of 128 bits). FIFOs of 16 bits or less may get "double-popped" by bus width conversion IP. Writes are supported by byte strobes, and so may not have quite the same issues, assuming that the FIFO qualifies writes with strobes

7.1.8 Source/destination memory formatting

System memory to VCPU program RAM:

VCPU instructions must be "loosely" packed into system memory (external to VCPU), one instruction every 64 bits.

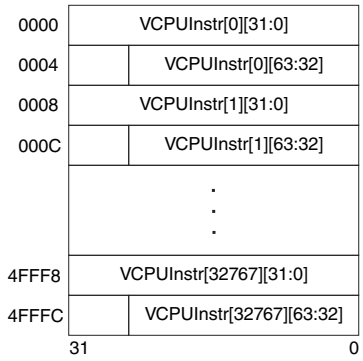


Figure 37. VCPU instruction (Rinstr) format in system memory

VCPU instructions that are fetched from system memory are packed into VCPU's program RAM (from the perspective of the VCPU program counter). Increment is by 2 due to half-word instructions.

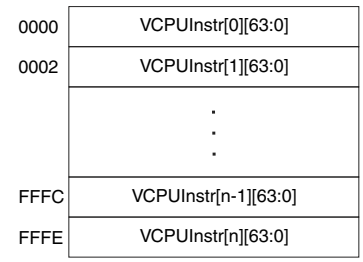


Figure 38. VCPU instruction (Rinstr) format in program RAM

System memory to IPPU program RAM (IPPU PRAM):

IPPU instructions must be "firmly" packed into system memory (external to IPPU).

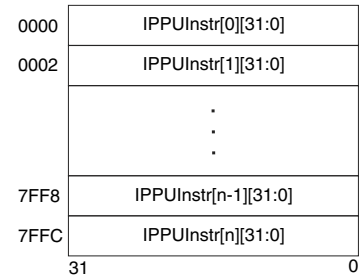


Figure 39. IPPU instruction (Rinstr) format in system memory

IPPU instructions that are fetched from system memory are packed into IPPU's program RAM (from the perspective of the IPPU program counter).

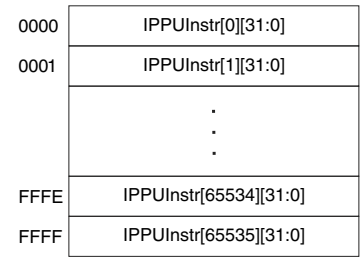


Figure 40. IPPU instruction (RUinstr) format in program IPPU PRAM

System memory to/from VCPU data memory (DMEM):

The internal DMEM can be loaded with data from external system memory, or the internal DMEM can be "dumped" to external system memory, under software control.

Data is packed into external system memory as shown in the following figure. Internal DMEM is 1024 bits wide, and 32-bit data that is copied from external system memory is packed into internal DMEM (see Figure 43, Figure 44). Using the VSPA DMA, it is the programmer's choice whether data is converted from 16-bit 2's complement to 16-bit fractional sign magnitude when it is transferred from external system memory to internal DMEM.

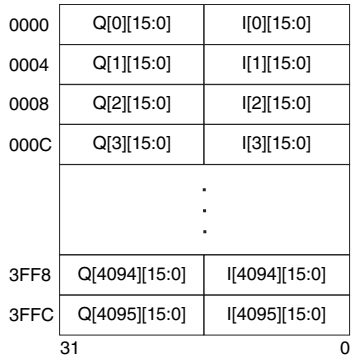


Figure 41. Complex Data Format in System Memory

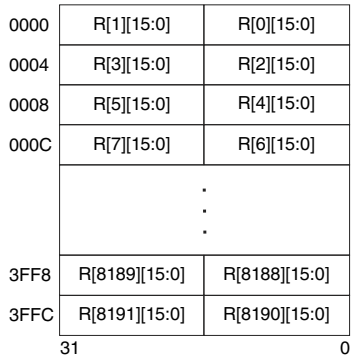


Figure 42. Real data format in system memory

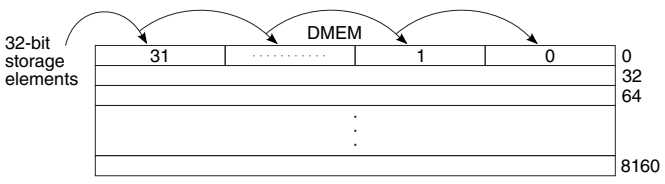


Figure 43. Data format in internal DMEM

Data that is "dumped" from internal DMEM to external system memory is packed into external system memory, as shown in Figure 43. Using the VSPA DMA, it is the programmer's choice whether each half-word is converted from sign magnitude to two's complement when it is transferred from internal DMEM to external system memory.

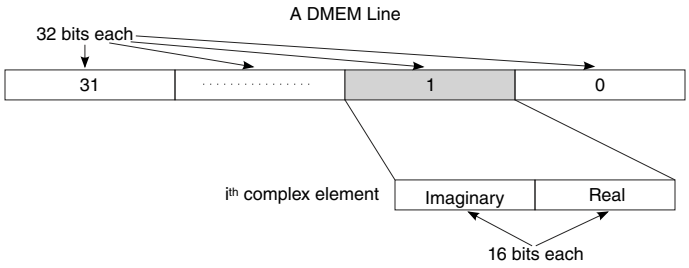


Figure 44. Data line in internal DMEM

Chapter 8
Mailboxes

8.1 Mailboxes

The VSPA platform provides several mail box facilities to facilitate communications between the VCPU and the host and between the VCPU and an external debugger. Here are their features:

1. Bi-directional messages: VCPU <-> host communication:
   - Host can send messages to the VCPU (instructions, and parameters)
   - The VCPU can send messages to host (printf functions)
   - 2x64-bit message mailboxes
   - Status flag mechanisms to indicate outgoing messages have been read
   - Status flag mechanisms to indicate incoming messages that have not been read
   - Generation of IRQ for the host/GO for VSPA
2. Bi-directional messages: VCPU <-> SOC level debug IP
   - Debug IP can send messages to the VCPU via the debug IP bus
   - The VCPU can send messages to debug IP via the debug IP bus
   - 32-bit messages
   - 64-bit messages
   - Status flag mechanisms to indicate outgoing messages have been read
   - Status flag mechanisms to indicate incoming messages that have not been read
3. Unidirectional messages: VSPA DQM -> SOC level debug IP via trace
   - VSPA sends a DQM message when writing a certain VSPA IP bus register
   - Short messages (25-bit payload)
   - > 32-bit messages (57-bit payload)

The VSPA<->host and VSPA<->Debugger mailboxes have symmetrical capabilities from the point of view of the VSPA and the other processors.

VSPA can send and receive up to two 64-bit messages to the host. The host can send and receive up to two 64-bit messages to VSPA. VSPA can send and receive 32 and 64-bit messages to the debugger, and the debugger can send and receive 32 and 64-bit messages to VSPA.

Each sender can monitor when the receiver reads each message. When VSPA is the recipient of a message, it can be set up to GO. When the host is the recipient of a message, it can be interrupted. These GO and interrupt capabilities are configured via the CONTROL and IRQEN registers respectively.

VSPA can receive a GO when it receive a message from the host (in the VSPA INBOX message), or when the host reads a messages sent via VSPA OUTBOX (note the host reads the HOST INBOX). The host can receive an interrupt when it receive a message from VSPA (in the HOST inbox message), or when VSPA reads a messages sent via the HOST OUTBOX (note VSPA reads the VSPA's INBOX).

Note that both the VCPU and the host can access both sides of the VSPA<->host mailboxes. If communications with the host is not needed or desired, the VCPU can use the host mailbox registers to communicate with other VCPU processes. Of course, only the VSPA inboxes have GO generation capability.

The VSPA DQM mailboxes allow trace port messages to be sent at will by software executing on the VCPU or the host.

Chapter 9
AXI Slave

9.1 AXI slave overview

The VSPA AXI slave interface allows AXI bus masters to directly access VSPA data memories. It also provides a flag/go system that allows the AXI masters to set status flags and trigger VSPA go events as a direct result of AXI writes to the VSPA AXI slave. This allows VSPA DMEM buffers to be read and/or written by remote processors or DMAs, and the flags can be used to communicate to the VSPA that an action has taken place, or is about to take place.

9.2 Memory map

From the perspective of the AXI slave, the memory map is as follows:

AXI start address	AXI end address	VSPA resource
AXI base address + 0x000000	AXI base address + 0x0FFFFF	VCPU DMEM byte address 0x000000 – 0x0FFFFF
AXI base address + 0x100000	AXI base address + 0x1FFFFF	IPPU DMEM byte address 0x000000 – 0x0FFFFF
AXI base address + 0x200000	AXI base address + 0x3FFFFF	64 bits of flags

9.3 Usage example

A remote VSPA (VSPA master or VSPA-M) needs to read a buffer from another VSPA (VSPA slave or VSPA-S). After the VSPA-S buffer data is fetched, VSPA-M needs to tell VSPA-S that it has consumed the buffer, so the buffer can be overwritten for another use.

The VSPA-M programs its DMA to read data from the VSPA-S using the VSPA-S AXI slave interface. If the VSPA-S buffer data resides in IPPU DMEM, with a byte offset of 0x200 from the base of IPPU DMEM, VSPA-M programs the DMA_AXI_ADDRESS register to VSPA-S AXI Base address + 0x100000 + 0x200. VSMA-M programs the DMA_DMEM_ADDRESS to the desired destination byte address in its local DMEM. VSPA-M would also program the DMA_AXI_BYTE_COUNT to indicate the number of bytes to copy, and finally it would write the DMA_XFR_CTRL register to initiate the operation.

To set the remote VSPA AXI slave GO flag, a second VSPA-M DMA command would be used. If this second command is programmed into the second FIFO entry for the same DMA channel that was used for the data movement, the second command will not begin until the previous command completes. This ensures that all the data is copied before the VSPA-S flag is set. To set AXI slave flags 33 and 32, VSPA-M would do the following:

- Initialize a 64-bit location in VSPA-M DMEM to 0x00000003_00000000. The chosen 64-bit location must also be aligned to the AXI address width of the VSPA-M DMA for the DMA to transfer it.
- Then program the VSPA-M DMA to copy 8 bytes from the chosen DMEM location to AXI location VSPA-S AXI Base address + 0x200000.
- After all the data has been read from VSPA-S, the second VSPA-M DMA command will run, causing VSPA-S flags 33 and 32 to set. If VSPA-S had enabled a go due to one or both of those flags, a VCPU GO event will occur. If there is no go enabled, the flags will set, but no VCPU GO event will be generated.

9.4 VSPA AXI slave flag system

There are 64 AXI slave flags, arranged into two 32-bit VSPA IP registers. There are also two 32-bit flag-go-enable registers, containing a bit for each flag that determines whether setting the corresponding flag causes a VSPA GO.

The flags can only be written to one from AXI, and can only be cleared by VCPU writes, write one to clear. The final beat of wdata is used to set the flags. When wlast, wready, and wvalid are all asserted, wdata[63:0] is used to set flags. Any bits that are 1's will cause the corresponding flag to set. Therefore to use the flags, an 8-byte transfer to VSPA AXI address 0x0020_0000 should be used. There is no reason to send any more data, or use any other address for flags. Sending less than 8 bytes of data will cause unintended flags to set, since the wstrb signals are ignored. Also, narrow transfers to flags of less than 64 bits will not work properly.

AXI reads will return an error response and unintelligible data.

9.5 Interface limitations

The AXI slave interface to VSPA DMEMs has a minimum write resolution of 16-bits. If an AXI master attempts to write a single byte, the DMEM will be written with 2 bytes instead. These will be aligned to the 16-bit address. For example, a write to address 0x13 will write data to bytes at addresses 0x12 and 0x13, and address 0x12 will be updated with whatever data is on the AXI wdata bus in the associated bit positions.

The AXI slave interface does not support wrapped or fixed burst types. If these burst encodings are used, addresses will still be incremented in a linear fashion, ignoring awburst and arburst encodings.

Chapter 10
Debug and Trace

10.1 Debug

The VSPA platform contains several debug features to enable and simplify both hardware validation and software debug.

Debug activities can be conducted either by an external debugger or a host processor in the system that is capable of controlling various accessible registers and of taking action based on current conditions. These activities would be controlled by accesses that would take place through the Debugger IP register interface. Note that this is a separate IP interface that is not directly accessible by the VCPU.

This section introduces the VSPA Debug architecture and provides a general overview of the top-level blocks and their respective functionality.

The VSPA debug block is comprised of three main components: debug trace unit (DTU)[30], debug event generation unit (DEGU), and debug run control (DRC). The VSPA debug block contains separate enable bits for non-invasive debug (nidbg_en), and invasive debug (idbg_en), in the GDBEN control register. These bits, along with the S0C invasive and non-invasive debug security overrides (dbg_dbgen and dbg_niden input to VSPA), can be used to independently enable/disable non-invasive (DTU) and invasive (DRC) debug, or disable VSPA debug entirely by clearing both bits.

The DEGU is comprised of shared, configurable debug resources capable of producing a variety of debug actions (both invasive and non-invasive), such as generation of a variety of trace messages, halt and resume of the VSPA, capability to act on up to a maximum of 8 cross-trigger inputs as well as generation of up to four cross-trigger outputs, based on a variety of internal VSPA debug events. The DEGU contains eight configurable sequencer-capable comparators and also houses the debug control registers, which are accessed via a dedicated debug IPbus. In addition, a set of message mailboxes exist for inter-VSPA/debug communication. Both 32-bit and 64-bit mailboxes exist from VSPA to debug and from debug to VSPA. A VSPA IPbus gateway also exists to read/write the VCPU/host IPbus registers. In addition, this gateway gives the user the choice of which identity to use for the register access (VCPU or host).

The DRC contains logic both centrally located and implemented in various parts of the VCPU in order to provide invasive run control of the engine. Once VCPU is halted in debug mode, this provides the ability for the user to single step through VCPU instructions as well as read the state of all VCPU architectural registers, including the internal jump-return and loop hardware stacks. In addition the memories may also be read and/or written. Finally, software breakpoint (SWB) functionality is implemented as part of this unit. When enabled a SWB can be used to halt the core, similar to a hardware breakpoint, as well as generate other debug events that are configured to produce many different actions. The entire VSPA_debug subblock operates in the same clock domain as the VCPU (VCPU_clk). Figure 45 depicts the high level block diagram of the debug components and how they integrate with the VCPU.

NOTE

TRACE is not available on this device.

[1] TRACE is not available on this device.

10.1.1 VSPA debug block diagram

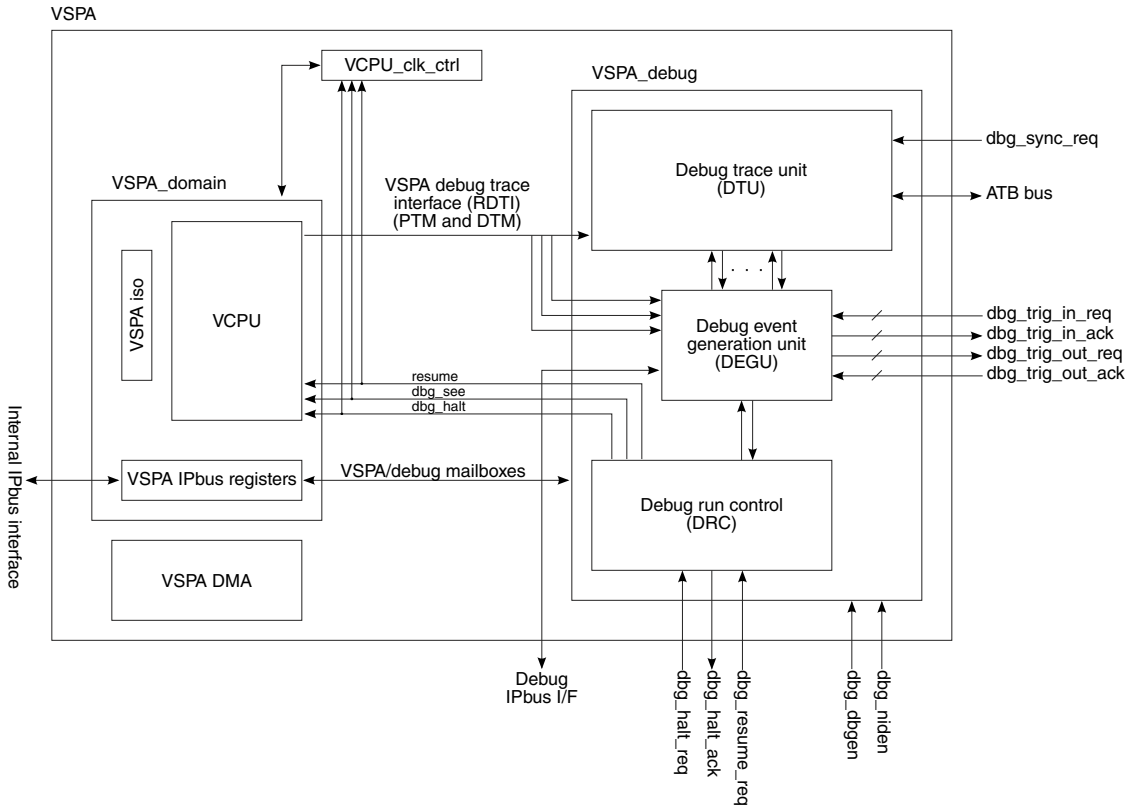


Figure 45. VSPA debug block diagram

NOTE

Debug trace unit (DTU) is not available on this device.

10.1.2 Debug functional description

10.1.2.1 Debug event generation unit (DEGU) subblock

The DEGU block is responsible for generating all non-program trace (PTM) debug events and houses all shared debug resources such as sequence-capable comparators, cross-trigger control logic, along with all debug configuration, and status registers. The DEGU contains an IPbus interface coupled to the dedicated debug IPbus, used for configuring the shared resources along with the actions taken on occurrence of the various debug events. There exists a total of upto 20 VSPA debug events as specified in the following table below:

Table 96. VSPA debug events

Event	Description
VCPU Go	One of the (many possible) VCPU Go events has occurred. Refer to VCPU GO events for all possible sources of this control signal.
VCPU Done	VCPU has executed the "done" instruction.

Table continues on the next page...

Table 96. VSPA debug events (continued)

Event	Description
SWB	Software breakpoint.
Comparator n event where *n* is 0-7	One or more of the eight comparators has triggered (if enabled and armed) due to either a match event or as a result of a different (armed and enabled) comparator causing it to trigger (if configured as part of a sequence).
Cross-Trigger in n trigger event where *n* is 0-7	One or more of the 8 Cross-trigger inputs has triggered (dbg_trig_in[n]). Each dbg_trig_in input is internally synchronized to the VSPA clock domain, and once sampled asserted high, the corresponding dbg_trig_in_ack is asserted high in response.

Each of these debug events can be configured (mapped) to cause multiple actions to occur, such as generating a cross-trigger out, start/stop Program/Data trace, generate a watchpoint message, cause VSPA to halt. As such, the DEGU unit essentially straddles the invasive and non-invasive debug boundary since it can directly or indirectly cause the DTU to generate a trace message and/or cause the DRU to halt/resume the VSPA.

10.1.2.1.1 Comparator resource

The DEGU contains eight highly configurable "sequence-capable" comparators which have the ability to act individually or can be paired with an adjacent comparator to form address ranges (windows) for added flexibility in filtering trace around areas of interest. This filtering capability can greatly reduce the amount of trace data produced, alleviating loss of trace data due to FIFO overflow. Each comparator can also be armed, either manually when it is enabled, or dynamically by any of the upto 20 possible events (including other comparators). Likewise, each can be disarmed either manually or via any of the upto 20 possible events (again including any of the comparators). In addition, any (armed and enabled) comparator can trigger any other armed and enabled comparator to activate, causing a comparator event. This provides the ability to define a sequence of events which must occur before a specific (final) desired action takes place. This feature allows the capability to filter trace around more complex sequences of events. Refer to the Debug comparator section for detailed information on the functionality of the comparator resource.

10.1.2.1.2 Cross-trigger resources

The DEGU contains 8 general purpose cross-trigger inputs and 4 cross-trigger outputs. Any of the DEGU events can be configured to cause any of the *dbg_trig_out_req[n]* (where n is 0-3) outputs to trigger (become asserted high). Likewise, any of the *dbg_trig_in_req[n]* (where n is 0-7) inputs, can be configured to cause any of the possible actions to occur, some of which are:

- Halt/resume the VSPA, by sending the appropriate control signal(s) to the DRU unit to perform the halt/resume function.

when triggered.

Each of the 8 cross-trigger inputs (*dbg_trig_in_req[n]*, where n is 0-7)is internally synchronized to the VSPA clock domain, and once sampled asserted high, will drive the corresponding ack output (*dbg_trig_in_ack[n]*) high until the request is sampled low, at which point, the ack out will be driven low (released), completing the handshake. The cross-trigger output resources behave in the same manner, that is, whenever *dbg_trig_out_req[n]* (where n is 0-3) is driven asserted high, it will remain asserted until the corresponding ack (dbg_trig_out_ack[n]) is sampled asserted high in the VSPA clock domain. At this point dbg_trig_out_req[n] will be de-asserted low (released). Note that the dbg_trig_out_ack[n] inputs are internally synchronized to VSPA clock. The dbg_trig_out_req[n] outputs must be synchronized in the receiving clock domain.

10.1.2.1.3 Debug control, configuration, and status registers

All VSPA debug configuration, control, and status registers are located in the DEGU unit, accessed via the dedicated debug IPbus. Refer to VSPA_DBG register descriptions

NOTE

Not all possible actions which can be configured for any given event necessarily has a valid use case; and may cause unpredictable behavior. It is left to the discretion of the User when configuring (mapping) events to desired actions.

10.1.2.2 Debug run control (DRC) subblock

The DRC subblock implements invasive type debug functionality, such as halting the VSPA, single stepping through VCPU instructions, and querying (reading) VSPA IPbus registers and/or internal architectural registers. In addition, the memories may also be read or written by the debug system.

10.1.2.2.1 VSPA halt and resume

The DRC provides the ability for the VSPA to be halted, entering debug mode. Halting can only occur if the idbg_en bit is set in the GDBEN register (invasive debug enabled) and the security pin, dbg_dbgen, is driven high (1). Halting can be initiated in three ways, including:

- On occurrence of any DEGU event(s) configured to cause the action of halting the VSPA.
- By writing 1 to the force_halt bit in the RCR control register via debug IPbus.
- When the external system triggers the dedicated halt request input, dbg_halt_req.

Note that the reason (or cause) of the halt is reflected in the RCSTATUS register.

Once VSPA has been halted, it can resume execution by any of the following ways:

- Temporarily by single-stepping on a VCPU instruction, by writing 1 to the single_step bit in the RCR register.
- Writing 1 to the resume bit in the RCR register.
- When the external system triggers the dedicated resume request input, dbg_resume_req.
- When any of the general purpose cross-trigger inputs, dbg_trig_in_req[n], configured to cause the action resume, are triggered by the external system.

10.1.2.2.2 VSPA internal visibility

There are two mechanisms which can be used to gain access to internal VSPA registers and memory.

- An IP gateway gives the debugger read/write access to VSPA IP bus registers via the debug IP bus.
- An architecture visibility portal provides read access to most VSPA internal architecture registers and read/write access to VSPA internal memories.

Both mechanisms require invasive debug to be enabled (GDBEN.idbg_en set) and the dbg_dbgen security pin to be driven high (write 1 to dbg_dbgen).

NOTE

There is no requirement that VSPA be halted when accessing either the IP gateway or the VSPA visibility portal. However writes may cause undesirable results, and reads may be of little value, as data could be transitioning often. For best results, it is recommended that VSPA be in halting debug mode when using these resources.

IP Gateway - The user can gain read/write access to the VSPA IP bus registers via the IP gateway. The VSPA IP bus registers appear to reside in the upper 4 KB of the debug IP bus memory map. In other words, accesses performed to the upper 4 KB (offset address 0x000 to 0x7FF) of the debug IP memory map are routed via the IP gateway to the VSPA IP bus registers. In addition, the ability exists to set the identity of debugger accesses via the IP gateway (VCPU or host) which takes effect for the IPbus register access. Refer to Debug VSP Architecture Visibility Address Pointer register (RAVAP) for details about the ip_bat bit in the RAVAP control register.

Architecture Visibility Portal - A separate VSPA architecture visibility portal is implemented via the RAVAP, RAVFD, RAVID set of registers, to provide visibility to the VSPA internal architectural registers and memories. Using this portal, accesses to the architectural registers and RF registers (R0-R7) are read-only, while accesses to the memories are read/write.

10.1.2.2.3 Software breakpoint (SWB)

The DEGUG software breakpoint event is implemented by setting the two MSBs of the VCPU instruction opcode, either at time of program compile/assembly (refer to the swbreak mnemonic), or by directly writing them in the memory via the VSPA architecture visibility portal. In either case, if invasive debug is enabled and the dbg_dbgen pin is not low (0), then the VSPA will halt upon execution of the swbreak (OpX) instruction. However, if invasive debug is not enabled even if the invasive debug security override is asserted (dbg_dbgen is driven low), the occurrence of the SWB event can still cause other non-halting actions to occur, such as arm/disarm/trigger comparator, generate cross-trigger out, trigger start/stop of program/data trace, and others as long as non-invasive debug is enabled and the non-invasive debug security override is negated (dbg_niden input is driven high).

The SWB instruction has special behavior associated with halting inside of short hardware loops, that is, loops of either 1 or 2 instructions. For the short loops, this is needed because the VCPU replays instructions from the pipeline rather than reloading them from PMEM. Therefore, using the VSPA architecture visibility portal to clear the SWB would be ineffective once VCPU has halted on the first iteration of the loop.

Because of this special behavior, in order to allow the VCPU to resume instruction execution, when a SWB is placed in a short loop and the CPU is halted during the first iteration of the loop, the SWB is cleared from the pipeline automatically. Thus, a resume command will not halt again on the next iteration of the short loop. Instead, the VCPU will run the loop to completion, and then re-enable SWB instructions. Trace of each iteration of a short loop can be accomplished by using the single step command.

The VCPU uses an internal prefetch buffer. When the VCPU is halted, its prefetch buffer may already contain the next few instructions. Changing the instruction or the SWB field of the instructions in the buffer may not be detected when the VCPU resume running. The user should not attempt to modify the instruction or the SWB field of the 4 instructions following the current PC.

10.1.2.3 Debug module comparator and sequencer

The VSPA debug module has 8 comparator sub-modules. Each comparator can be used individually, or pairs of comparators can be used together to generate a more complex event as described below. A comparator pair can consist only of an even numbered comparator n and its numerically incremented neighbor n+1 (comparator 0 can be paired with comparator 1, comparator 2 can be paired with comparator 3, and so on).

The comparator block diagram is shown in Figure 46.

Each comparator includes a 17-bit data value register, several control registers, several enable signals, four input buses with 17-bit data width plus 6-bit attribute qualifiers, muxes with mux control to select one of the input buses, Armed status bit, Seq_Trig register, and read and write ports to read/write data from/to the internal registers. The read port and the write port are connected to the debug IP bus.

Since DMEM and IPPU DMEM memories can be accessed to read or write multiple 32-bit words in each clock cycle, special support is provided so that if any (one or more) 32-bit word is accessed (read or written), and the word address matches the compare criteria, the comparator match is asserted. To do so, the comparator breaks the data value register address representation into two parts: the line address, and the word address within the line. The line address is compared against the line address on input in_a or in_b (DMEM address or IPPU DMEM address, respectively), and the selected word access strobes, in_a_strbs or in_b_strbs (dmem_elem_strb[NUM_ELEM-1:0] or ippu_dmem_elem_strb[NUM_ELEM-1:0], respectively). If both the line address and any of the strobes meets the compare criteria (for example, ==, >, <=), the comparator match is asserted.

Only an armed comparator can match (trigger). To trigger, a comparator must first be enabled, then armed, then it can be triggered.

Each comparator/sequencer can also be used as a state of a sequencer. When a comparator/sequencer is triggered, the corresponding input selected by all other comparators/sequencer action control registers is asserted. If this input is selected to arm a second comparator/sequencer, the second comparator/sequencer is armed. The second comparator will be triggered when the corresponding match condition occurs. Note that if the "always" condition is selected, the second comparator/sequencer is triggered immediately, and if the "never" condition is selected, the second comparator/sequencer is triggered only by the sequencer trigger (not by the comparator). The user can program the sequencer to create up to 8 sequential states. The sequential comparator can be triggered by a comparator match, or any other event (for example, any comparator trigger, any dbg_trig_in_req signal, the VCPU go event, and the VCPU done event)

The inputs to each comparator/sequencer are:

- in_a [16:0] - VCPU DMEM address bus to compare

- in_b [16:0] - IPPU DMEM address bus to compare
- in_c [16:0] - VCPU PMEM address bus to compare
- in_d [16:0] - VSPA IP bus address to compare
- in_e [16:0] - IPPU PMEM address bus to compare

cmp_in - compare result from the paired comparator for allowing the triggering of the comparator event in the same clock (for even numbered comparators, cmp_in is tied to 0. For odd numbered comparators, cmp_in is tied to the cmp_evt from the next lowest enumerated comparator. So, comparators 1, 3, 5, and 7 cmp_in inputs are tied to the cmp_evt outputs of comparators 0, 2, 4, and 6, respectively.

cmp_evt_in[7:0] - compare result from all other comparators (cmp_evt); used to arm, disarm, or trigger the comparator.

trig_in[7:0] - cross-trigger inputs (dbg_trig_in_req[7:0]); used to arm, disarm, or trigger the comparator.

swb - software breakpoint event; used to arm, disarm, or trigger the comparator.

VCPU_go - VCPU_go event; used to arm, disarm, or trigger the comparator.

VCPU_done - VCPU done event; used to arm, disarm, or trigger the comparator.

Outputs:

cmp_evt - Compare event asserts when the comparator matches. It can be used to generate messages, watchpoints, or start/stop program/data trace. The cmp_evt outputs from comparators 0, 2, 4, and 6 are also used as inputs to paired comparators 1, 3, 5, and 7.

Note, the VCPU and the IPPU use internal prefetch buffers. When the VCPU and IPPU are halted, their prefetch buffers may already contain the next few instructions. Setting the comparators to the VCPU/IPPU PMEM address to of the instructions already in the buffer may not be matched when the VCPU/IPPU resume running. The user should not attempt to set the comparator to any of the 4 instructions following the current PC.

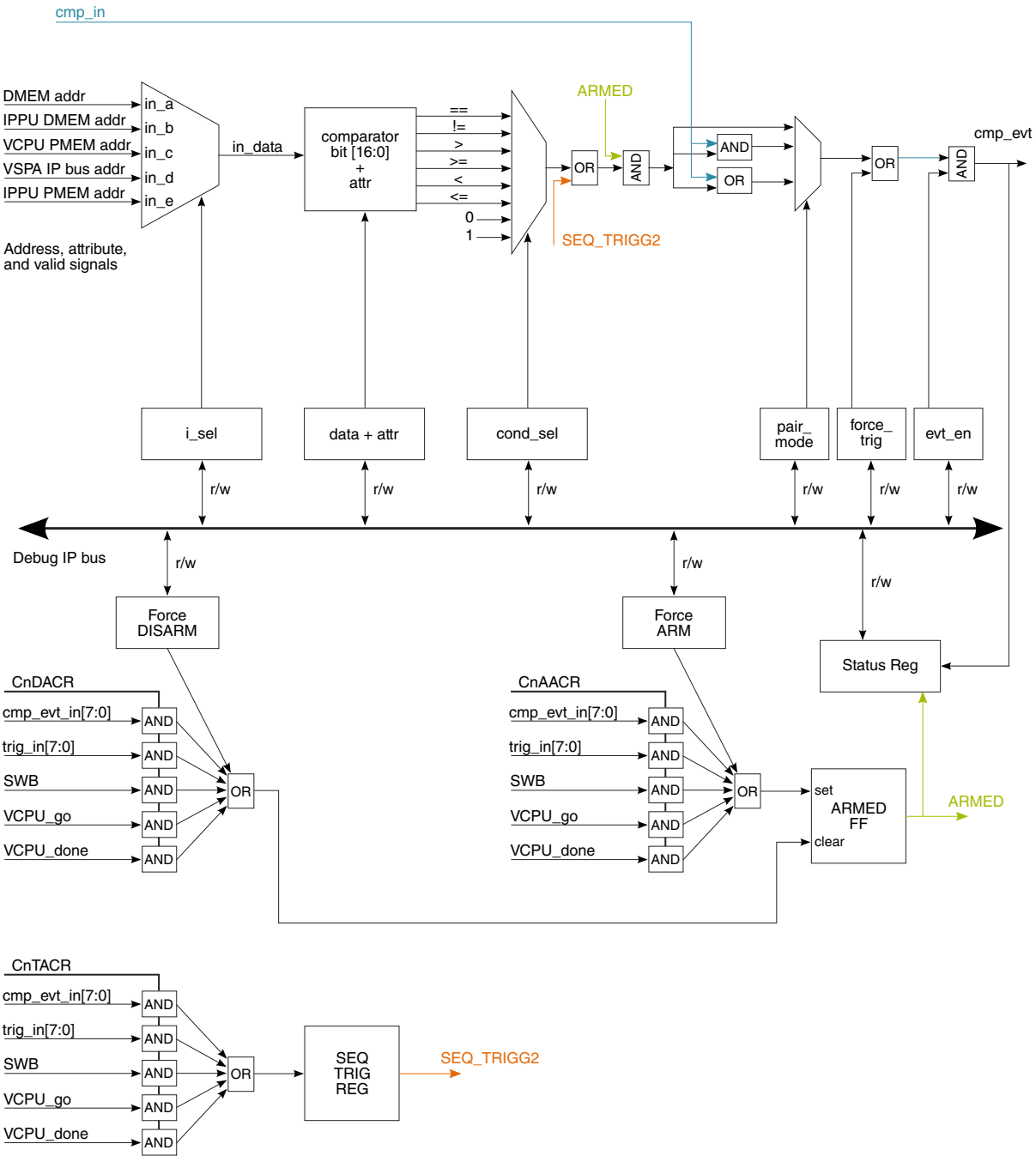


Figure 46. Debug comparator and sequencer

10.1.3 Debug using the DMA FIFOs

Unused DMA FIFO entries may be configured to "echo" or "reflect" data that is read or written by the VSPA DMA. The intention is that this data is stored in a reserved area in system memory, and then available for inspection via the host processor, or potentially available for packing and streaming out via an external interface.

If only one FIFO entry is being used on a channel, the 2nd entry can be programmed to write the data out to AXI somewhere. Since the FIFO entries are processed sequentially, the 2nd FIFO entry (AXI writes) will not begin until the 1st FIFO entry has completed - necessary if the 1st FIFO entry is AXI reads.

Chapter 11
Interrupts

11.1 Interrupts

The VSPA module generates active low interrupts. VSPA interrupts are enabled and disabled by writing to the IRQEN register.

The interrupts that the VSPA module generates are:

- *vcpu_done_irq_b:* indicates that the VCPU has completed processing a software thread.
  - This interrupt is negated by writing 1 to the corresponding flag in the VSPA STATUS register.
- *vcpu_flags0_irq_b*: indicates that the VCPU software has generated an interrupt that needs servicing from the host software.
  - This interrupt will assert (if enabled) when any of the flags in the VCPU_HOST_FLAGS0 register are set by the VCPU software.
  - The definition of the flags is software-defined.
  - This interrupt source is negated by writing a 1 to the corresponding flags in the VCPU_HOST_FLAGS0 register. When no bits are set in the VCPU_HOST_FLAGS0 register the interrupt will be negated.
- *vcpu_flags1_irq_b*: indicates that the VCPU software has generated an interrupt that needs servicing from the host software.
  - This interrupt will assert (if enabled) when any of the flags in the VCPU_HOST_FLAGS1 register are set by the VCPU software.
  - The definition of these flags is software-defined.
  - This interrupt source is negated by writing 1 to the corresponding flags in the VCPU_HOST_FLAGS1 register. When no bits are set in the VCPU_HOST_FLAGS1 register the interrupt will be negated.
- *dma_cmp_irq_b*: indicates that DMA channel completed its programmed data transfers.
  - This interrupt source is negated by writing 1 to the corresponding flags in the DMA_IRQ_STAT register. When no bits are set in the DMA_IRQ_STAT register the interrupt will be negated.
- *dma_err_irq_b:* indicates that an error occurred during a DMA data transfer or in the configuration of the DMA channel.
  - This interrupt source is negated by writing a 1 to the corresponding flags in the DMA_XFRERR_STAT and DMA_CFGERR_STAT registers. When no bits are set in the DMA_IRQ_STAT and DMA_CFGERR_STAT registers the interrupt will be negated.
- *vcpu_msg_irq_b*: indicates that the VCPU has sent a mailbox message to the host processor.
  - This interrupt source is negated by writing a 1 to the corresponding flag in the VSPA STATUS register.
- *ippu_done_irq_b*:indicates that the VSPA IPPU has completed processing a software thread.
  - This interrupt source is negated by writing a 1 to the corresponding flag is the VSPA STATUS register.

Chapter 12
Initialization

12.1 Initialization

The VCPU fetches instructions from the internal PRAM. Before the VCPU can begin execution, application-specific software programs must be copied from external system memory to internal program RAM.

After an asynchronous reset, the VCPU waits for a go bit to be set. The DMA is used by the host processor to load the boot-up image of PRAM. The host software must set up and enable the DMA to load PRAM, and then poll for the transfer to complete. After PRAM is loaded, the host can set the *host_go* bit for the first time.

After boot-up the VCPU can manage its own PRAM image loads. However, it is also possible for the host to be the manager of all PRAM image loads after boot-up. In this case, the VCPU must be in the Idle state before a new PRAM image is loaded by the host software.

Chapter 13
Forward Error Correction Unit (FECU)

13.1 FECU overview

The FECU module handles forward error correction (FEC) for applications with proprietary protocols similar to WiFi. It performs FEC for the data fields. FECU is comprised of multiple sub-modules. Each sub-module performs an operation and hands the data off to the next sub-module in the chain. Once the chain completes, the entire operation is finished, and an interrupt is generated.

Input data can be loaded into DMEM using the AXI slave, or VSPA's DMA can fetch the input data and load it into DMEM. FECU's output data is written to DMEM, and VSPA DMA can transfer it to any AXI address.

FECU is configured by FECU IP registers. See FECU register descriptions for detailed description of these registers.

13.2 FECU features

- Supports proprietary protocols similar to 802.11
- Supports up to 256QAM
- Can support multiple streams through VSPA firmware.
- Runs at 614.4 MHz
- Context save / restore for multi-user support
- Encoding operations
  - Scrambling
  - Convolutional encoding
  - Interleaving
  - LDPC tone mapping
  - LDPC Encoding
- Decoding operations
  - De-interleaving (reuse interleaver)
  - LDPC tone de-mapping
  - Viterbi Decoding
  - De-scrambling (reuse scrambler)
  - LDPC Decoding

13.3 FECU block diagram

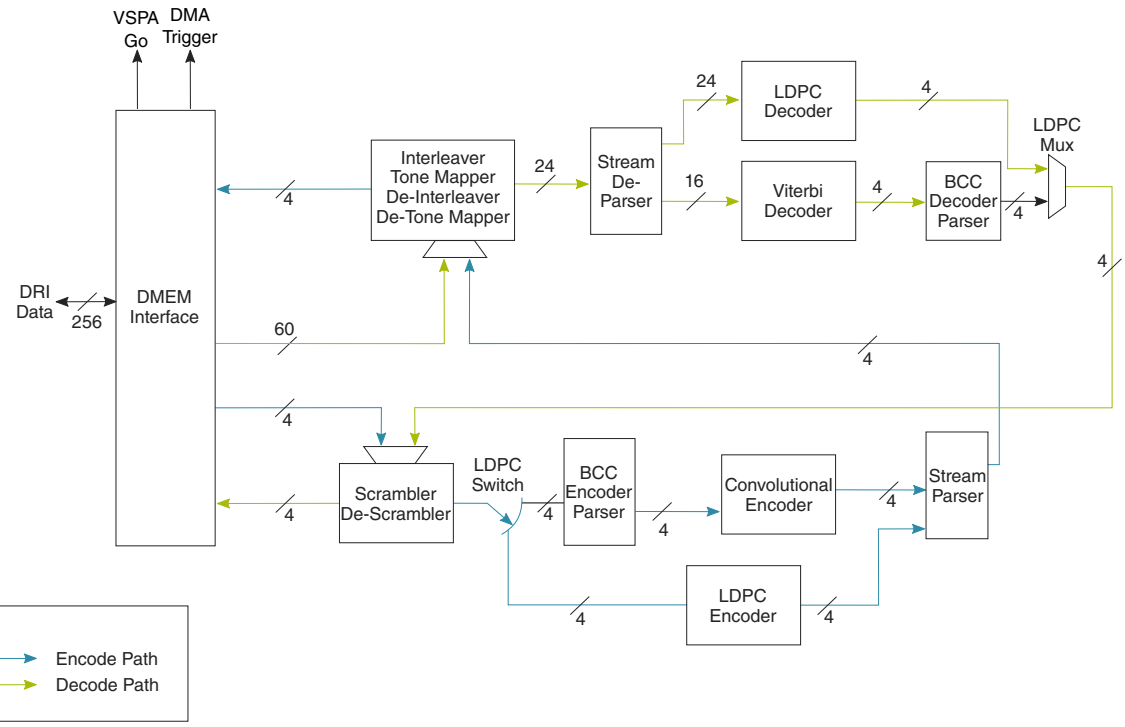


Figure 47. FECU block diagram

13.4 FECU clock generation

- FECU contains internal clock generation used to create the gated clocks for the FECU submodules.
- The FECU internal data paths run on dedicated clocks at the VSPA clock rate that are automatically enabled/disabled as needed during operation to minimize current consumption.
- Clock enable override register bits are provided for all internally-generated clocks.

13.5 FECU low power modes

- Sleep Mode
  - All clocks disabled
  - No FECU operation
  - All FECU source clocks disabled by SoC level clock control
- Idle Mode
  - Decoders not operating
  - FECU IP clock is enabled by SoC level clock control
  - FECU internal decoder clocks are disabled by the hardware automatically
- Decoder Active Mode
  - The decoder is operating.
  - IP and any required internal clocks enabled by SoC level clock control

- Internal decoder clocks are enabled by the hardware automatically.

13.6 FECU reset

- The FECU module is reset using the hard asynchronous reset for the chip.
- In addition, a software reset is provided. The software reset is enabled by writing to a register bit in the FECU_CONFIG register. It remains set until cleared by software. This reset is not required for normal operation.

13.7 FECU interrupts and VSPA go

The FECU module has one interrupt output. When FECU completes a command and the irqen_done bit in FECU CONTROL register is set it will set the irq_pend_fecu_done bit (bit 6) in VSPA STATUS register. When this bit is set and the corresponding irqen_fecu_done bit (bit 6) in IRQEN register is also set it will generate an interrupt request out of VSPA.

When FECU completes a command and vcpu_go_enable bit in FECU CONTROL register is set, it will set the bit 5 in FECU CONTROL register. This will cause VSPA to go. See VSPA CONTROL register.

The FECU command completion will also enable IPPU and DMA.

13.8 Viterbi Decoder overview

The Viterbi decoder performs the Viterbi algorithm to optimally decode convolutionally coded data. The Viterbi decoder takes its input data from the stream de-parser, and sends its output to the BCC decoder parser.

- Implements de-puncturing. Programmable de-puncture mask.
  - Supports 1/2, 2/3, 3/4, and 5/6 rates
- Constraint length 7
- 4 bit soft decision (LLR) receive input word size
- Processes 2 output bits per cycle
- 128 bit trace back
- Delays data by 384 bits, except for the last symbol.
- Uses 4 instances of 64 (d) x 128 (w) embedded trace back RAMs
- Processes 1 BCC blocks concurrently

13.9 Interleaver overview

The interleaver performs BCC interleaving and LDPC tone mapping. In does both encode and decode operations. During decode, it writes one constellation point per cycle (e.g. 64 QAM – 6 LLRs), and reads 4 LLRs per cycle. During encode, it writes 4 bits per cycle, and reads one constellation point per cycle.

- Performs BCC de-interleaving on 6 bit LLRs, and BCC interleaving on 1 bit data
- Performs LDPC tone de-mapping on 6 bit LLRs, and LDPC tone mapping on 1 bit data
- Configurable to handle 20,40,80, or 160 MHz interleaving
- Performs section 20.3.11.8 and 18.3.5.7 interleaving
- Processes 1 data streams at a time
- Uses 1 instance of 64 (d) x 240 (w) embedded RAMs
  - 160 MHz, 1024 QAM support

13.10 Convolutional Encoder overview

- Performs section "18.3.5.6 Convolutional encoder."

- Supports puncturing
  - — Configurable puncturing patterns
  - — Supports 1/2, 2/3, 3/4, and 5/6 rates
- Processes 4 input and output bits per clock cycle
- Processes 1 BCC blocks concurrently

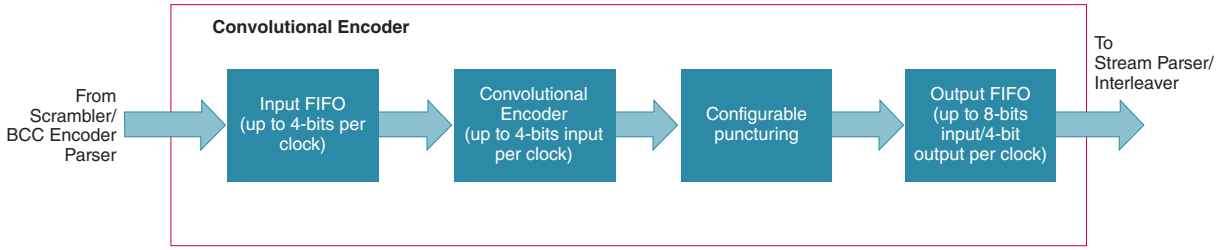


Figure 48. Convolutional Encoder

13.11 LDPC Encoder overview

The low density parity check (LDPC) encoder computes the parity check bits from the input message bits. It takes its input from the scrambler, and sends its output to the stream parser. The LDPC encoder computes the parity bits as it receives message bits. When it gets enough message bits, it waits the input and sends out the parity bits. The encoder will not send out more than coded_bits_per_symbol bits during any operation. Extra parity or repeat bits will be saved inside the encoder until the next operation.

- Performs repetition, shortening, and puncturing
- Supports 1/2, 2/3, 3/4, and 5/6 rates
- Supports 648, 1296 and 1944 block sizes
- Processes 81 bits per clock cycle

13.12 LDPC Decoder overview

The low density parity check (LDPC) decoder find the message and parity bits that satisfy the parity check matrix and are closest to the received LLRs. It takes 6 bit input LLRs from the stream de-parser, and when it gets enough LLRs, it starts the iterative decoding operation. If the decoder can't find message and parity bits that satisfy the parity check matrix, the decoder will declare a decoder failure. It if succeeds, it will send its output to the de-scrambler. The LDPC decoder only sends the message bits, and does not send parity, shortening, or repetition bits. The number of output bits will be a multiple of the LDPC block size.

- Performs de-repetition, de-shortening, and de-puncturing
- Supports 1/2, 2/3, 3/4, and 5/6 rates
- Supports 648, 1296 and 1944 block sizes
- Separate input and output buffers to allow for near 100% utilization
  - — FRAM reads and de-scramble happen in parallel with decode
- 1 sub-matrix processing engines
- Processes 1 blocks in parallel

13.13 Scrambler overview

- Performs section 18.3.5.5 and 16.2.4 scrambling and de-scrambling
- Processes 4 bits per clock cycle
- LFSR State
  - Starting scrambling state specified in IP registers
  - Starting de-scrambling state
    - Section 18.3.5.5 extracted from data
    - Section 16.2.4 self synchronizes

Chapter 14
VSPA IP Registers

14.1 Slow read registers

This version of VSPA has added slow IP register read accesses. Whenever VCPU reads a slow read register address, the data written back to the VCPU's gX register will be updated after two clock cycles. Normal (non-slow) IP register reads update the gX register after one clock cycle. All IP registers in VSPA address range 0x100 to 0x3FF (inclusive) are slow read registers. Not all registers within this range will exist on all versions of VSPA. Read result for non-existent registers is un-defined.

Note that this only applies to VCPU reads. Reads by the debugger or host do not require any additional delay. From the point of view of the debugger or host, their read behavior is the same as "normal" registers.

14.2 VSPA register descriptions

VSPA IP registers can be accessed either by a host processor through the VSPA IP bus, or by VCPU instructions (mvip). With some exceptions noted in the register field descriptions these accesses will be the same, that is reads will return the same value and writes will have the same effect. The register offset shown in all the following tables is the byte offset used by a host access, the VCPU mvip instruction requires a word index, which is equal to the byte offset / 4.

The following table serves as a key for the VSPA modules' register summary and detailed register descriptions.

Table 97. Register Conventions

Convention	Description
	Identifies reserved bits.
FIELDNAME	Idenfities an implemented bit field.
Register Field Types	
RW	Read/Write. Only software can change the value of the bit (other than a hardware reset). **NOTE** In some cases, a read-write register/bit field may have additional non-standard behavior which is described in detail in the register field description.
RO	Read only. Writing this bit has no effect.
WO	Write only.
WORZ	Write only. Always reads 0.
W1C	Write 1 to clear. Writing 0 has no effect.
Reset Values	
0	Resets to zero.
1	Resets to one.
u	Undefined at reset.

Table continues on the next page...

Table 97. Register Conventions (continued)

Convention	Description
*	See footnote for description.
Memory areas not defined should be considered reserved.	

14.2.1 VSPA_CCSR memory map

VSPA base address: 100_0000h

14.2.1.1 DMA Control and Status Registers memory map

Offset	Register	Width (In bits)	Access	Reset value
B0h	DMEM/PRAM Address (DMA_DMEM_PRAM_ADDR)	32	WO	See description
B4h	DMA AXI Address (DMA_AXI_ADDRESS)	32	WO	See description
B8h	AXI Byte Count register (DMA_AXI_BYTE_CNT)	32	WO	See description
BCh	DMA Transfer Control register (DMA_XFR_CTRL)	32	RW	See description
C0h	DMA Status/Abort Control (DMA_STAT_ABORT)	32	RW	0000_0000h
C4h	DMA IRQ Status (DMA_IRQ_STAT)	32	RW	0000_0000h
C8h	DMA Complete Status (DMA_COMP_STAT)	32	W1C	0000_0000h
CCh	DMA Transfer Error Status (DMA_XFRERR_STAT)	32	W1C	0000_0000h
D0h	DMA Configuration Error Status (DMA_CFGERR_STAT)	32	W1C	0000_0000h
D4h	DMA Transfer Running Status (DMA_XRUN_STAT)	32	RO	0000_0000h
D8h	DMA Go Status (DMA_GO_STAT)	32	W1C	0000_0000h
DCh	DMA FIFO Availability Status (DMA_FIFO_STAT)	32	RO	FFFF_FFFFh

14.2.1.2 Debug Messaging and Profiling Registers memory map

Offset	Register	Width (In bits)	Access	Reset value
98h	Cycle counter MSB register (CYC_COUNTER_MSB)	32	RW	See description
9Ch	Cycle Counter LSB Register (CYC_COUNTER_LSB)	32	RW	0000_0000h

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
600h	VCPU to DQM Trace Small Outbox register (DQM_SMALL)	32	WO	See description
620h	VCPU to Debugger 32-bit Outbox register (VCPU_DBG_OUT_32)	32	WO	0000_0000h
624h	VCPU to Debugger 64-bit MSB Outbox register (VCPU_DBG_OUT_64_MSB)	32	WO	0000_0000h
628h	VCPU to Debugger 64-bit LSB Outbox register (VCPU_DBG_OUT_64_LSB)	32	WO	0000_0000h
62Ch	Debugger to VCPU 32-bit Inbox register (VCPU_DBG_IN_32)	32	RO	0000_0000h
630h	Debugger to VCPU 64-bit MSB Inbox register (VCPU_DBG_IN_64_MSB)	32	RO	0000_0000h
634h	Debugger to VCPU 64-bit LSB Inbox register (VCPU_DBG_IN_64_LSB)	32	RO	0000_0000h
638h	VCPU to Debugger Mailbox Status register (VCPU_DBG_MBOX_STATUS)	32	RO	See description

14.2.1.3 General VCPU Control/Status Registers memory map

Offset	Register	Width (In bits)	Access	Reset value
8h	VCPU System Control register (CONTROL)	32	RW	See description
Ch	VSPA Interrupt Enable register (IRQEN)	32	RW	See description
10h	VSPA Source 1 Info (STATUS)	32	W1C	See description
30h	VSPA VCPU Illegal Opcode Address (ILLOP_STATUS)	32	RO	See description
100h	Load Register File Control register (Slow read register) (LD_RF_CONTROL)	32	RW	4501_00C4h
104h	Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_0)	32	RW	2727_2727h
108h	Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_0)	32	RW	00AA_55FFh
10Ch	Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_1)	32	RW	2A4C_086Eh

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
110h	Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_1)	32	RW	6666_6666h
114h	Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_2)	32	RW	2A4C_086Eh
118h	Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_2)	32	RW	8888_8888h
11Ch	Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_3)	32	RW	2A4C_086Eh
120h	Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_3)	32	RW	0000_0000h
124h	Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_4)	32	RW	2A4C_086Eh
128h	Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_4)	32	RW	CCCC_CCCCh
12Ch	Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_5)	32	RW	2A4C_086Eh
130h	Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_5)	32	RW	4444_4444h
134h	Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_6)	32	RW	2A4C_086Eh
138h	Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_6)	32	RW	AAAA_AAAAh
13Ch	Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_7)	32	RW	2A4C_086Eh
140h	Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_7)	32	RW	2222_2222h
400h	VCPU Mode 0 (VCPU_MODE0)	32	RO	0000_0000h
404h	VCPU Mode 1 (VCPU_MODE1)	32	RO	0002_0000h
408h	VCPU CREG 0 (VCPU_CREG0)	32	RO	0008_0000h
40Ch	VCPU CREG 1 (VCPU_CREG1)	32	RO	0000_0000h
410h	Store Unalign Vector Length (ST_UL_VEC_LEN)	32	RO	C000_0000h

14.2.1.4 IPPU Control and Status Registers memory map

Offset	Register	Width (In bits)	Access	Reset value
700h	IPPU Control register (IPPUCONTROL)	32	RW	See description
704h	IPPU Status register (IPPUSTATUS)	32	RO	See description
708h	IPPU Run Control register (IPPURC)	32	RW	See description
70Ch	IPPU Arg Base Address register (IPPUARGBASEADDR)	32	RW	See description
710h	IPPU Hardware Version (IPPUHWVER)	32	RO	0000_0000h
714h	IPPU Software Version (IPPUSWVER)	32	RW	0000_0000h

14.2.1.5 Input/Output Registers memory map

Offset	Register	Width (In bits)	Access	Reset value
70h	Platform Input (PLAT_IN_0)	32	RO	See description
80h	Platform Output (PLAT_OUT_0)	32	RW	0000_0000h
500h - 524h	General Purpose Input registers [10 registers] (GP_IN0 - GP_IN9)	32	RO	See description
580h - 5A4h	General Purpose Output registers [10 registers] (GP_OUT0 - GP_OUT9)	32	RW	0000_0000h

14.2.1.6 Thread and Protection Control and Status Registers memory map

Offset	Register	Width (In bits)	Access	Reset value
50h	Thread Control and Status (THREAD_CTRL_STAT)	32	RW	See description
54h	Protection Fault Status (PROT_FAULT_STAT)	32	W1C	0000_0000h
58h	VCPU Exception Control (EXCEPTION_CTRL)	32	RO	1000_0000h
5Ch	VCPU Exception Status (EXCEPTION_STAT)	32	W1C	0000_0000h

14.2.1.7 VCPU Go Control and Status Registers memory map

Offset	Register	Width (In bits)	Access	Reset value
28h	External Go Enable (EXT_GO_ENA)	32	RW	See description
2Ch	External Go Status (EXT_GO_STAT)	32	W1C	See description
60h - 64h	AXI Slave flags register a (AXISLV_FLAGS0 - AXISLV_FLAGS1)	32	W1C	0000_0000h
68h - 6Ch	AXI Slave Go Enable register a (AXISLV_GOEN0 - AXISLV_GOEN1)	32	RW	0000_0000h
180h	VCPU Go Address (VCPU_GO_ADDR)	32	RW	0000_0000h
184h	VCPU Go Stack (VCPU_GO_STACK)	32	RW	0000_0000h

14.2.1.8 VCPU - Host Messaging Registers memory map

Offset	Register	Width (In bits)	Access	Reset value
14h - 18h	VCPU to Host flags register a (VCPU_HOST_FLAGS0 - VCPU_HOST_FLAGS1)	32	W1C	0000_0000h
1Ch - 20h	Host to VCPU Flags register a (HOST_VCPU_FLAGS0 - HOST_VCPU_FLAGS1)	32	RW	0000_0000h
640h	VCPU to host outbox message n MSB register (VCPU_OUT_0_MSB)	32	WO	0000_0000h
644h	VCPU to host outbox message n LSB register (VCPU_OUT_0_LSB)	32	WO	0000_0000h
648h	VCPU to host outbox message n MSB register (VCPU_OUT_1_MSB)	32	WO	0000_0000h
64Ch	VCPU to host outbox message n LSB register (VCPU_OUT_1_LSB)	32	WO	0000_0000h
650h	VCPU from Host Inbox Message n MSB (VCPU_IN_0_MSB)	32	RO	0000_0000h
654h	VCPU from host inbox message n LSB register (VCPU_IN_0_LSB)	32	RO	0000_0000h
658h	VCPU from Host Inbox Message n MSB (VCPU_IN_1_MSB)	32	RO	0000_0000h
65Ch	VCPU from host inbox message n LSB register (VCPU_IN_1_LSB)	32	RO	0000_0000h
660h	VCPU to Host Mailbox Status register (VCPU_MBOX_STATUS)	32	RO	0000_0000h
680h	Host to VCPU Outbox Message n MSB register (HOST_OUT_0_MSB)	32	WO	0000_0000h
684h	Host to VCPU Outbox Message n LSB register (HOST_OUT_0_LSB)	32	WO	0000_0000h
688h	Host to VCPU Outbox Message n MSB register (HOST_OUT_1_MSB)	32	WO	0000_0000h

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
68Ch	Host to VCPU Outbox Message n LSB register (HOST_OUT_1_LSB)	32	WO	0000_0000h
690h	Host from VCPU Inbox Message n MSB (HOST_IN_0_MSB)	32	RO	0000_0000h
694h	Host from VCPU Inbox Message n LSB Register (HOST_IN_0_LSB)	32	RO	0000_0000h
698h	Host from VCPU Inbox Message n MSB (HOST_IN_1_MSB)	32	RO	0000_0000h
69Ch	Host from VCPU Inbox Message n LSB Register (HOST_IN_1_LSB)	32	RO	0000_0000h
6A0h	Host Mailbox Status Register (HOST_MBOX_STATUS)	32	RO	0000_0000h

14.2.1.9 Version and Configuration Registers memory map

Offset	Register	Width (In bits)	Access	Reset value
0h	VSPA Hardware Version (HWVERSION)	32	RO	0201_0F00h
4h	VCPU Software Version (SWVERSION)	32	RW	0000_0000h
40h	VSPA Parameters 0 (PARAM0)	32	RO	See description
44h	VSPA Parameters 1 (PARAM1)	32	RO	2510_0A0Ah
48h	VSPA Parameters 2 (PARAM2)	32	RO	See description
4Ch	VCPU DMEM Size (VCPU_DMEM_BYTES)	32	RO	0002_0000h

14.2.2 VSPA Hardware Version (HWVERSION)

Offset

Register	Offset
HWVERSION	0h

Function
VSPA hardware version

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	vspa_hw_version															
W																
Reset	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	vspa_hw_version															
W																
Reset	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 vspa_hw_version	vspa_hw_version VSPA hardware version Indicates the version of the VSPA module's hardware. The values in this register are valid immediately after reset.

14.2.3 VCPU Software Version (SWVERSION)

Offset

Register	Offset
SWVERSION	4h

Function

VCPU Software Version

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	pram_ucode_version_31_16															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	pram_ucode_version_15_0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-16 pram_ucode_version_31_16	pram_ucode_version PRAM Software Version This field can be used to indicate the version of the PRAM assembly code. VCPU software configures this register during global initialization after download of a new PRAM image. PRAM version numbers are non-zero.
15-0 pram_ucode_version_15_0	pram_ucode_version PRAM Software Version This field can be used to indicate the version of the PRAM assembly code. VCPU software configures this register during global initialization after download of a new PRAM image. PRAM version numbers are non-zero.

14.2.4 VCPU System Control register (CONTROL)

Offset

Register	Offset
CONTROL	8h

Function

NOTE

Pay close attention to the register bit field descriptions following the register figure because a field designated with read-write access in the register figure may have additional non-standard behavior described in its corresponding register bit field description.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	Reserved				host_re...	host_re...	host_ms...	host_ms...	host_re...	host_re...	host_se...	host_se...	Reserved		dma_hal...	Reserved
W									W1C	W1C	W1C	W1C				0
Reset	u	u	u	u	0	0	0	0	0	0	0	0	u	u	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	Reserved			host_vs...	host_vs...	Reserved	debug_m...	axislv_...	fecu_go	host_vs...	debug_m...	vcpu_go	ext_go	dma_go	ippu_go	host_go
W									W1C		W1C				W1C	
Reset	u	u	u	0	0	u	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-28 —	- Reserved
27 host_read_msg1_go_en	host_read_msg1_go_en This is a read/write control bit. It can be written and read at any time, by either VSPA or a host. This bit allows the generation of a VSPA_go request when the Host Read Message 1 from VCPU - Go flag (host_read_msg1_go) is set. 0b - No Go request is generated when the host_read_msg1_go is set. 1b - A Go request is generated when the host_read_msg1_go is set.
26 host_read_msg0_go_en	host_read_msg0_go_en This is a read/write control bit. It can be written and read at any time, by either VSPA or a host. This bit allows the generation of a VSPA_go request when the Host Read Message 0 from VCPU - Go flag (host_read_msg0_go) is set. 0b - No Go request is generated when the host_read_msg0_go is set. 1b - A Go request is generated when the host_read_msg0_go is set.
25 host_msg1_go_en	host_msg1_go_en This is a read/write control bit. It can be written and read at any time, by either VSPA or a host. This bit allows the generation of a VSPA_go request when the Host Sent Message 1 to VCPU - Go flag (host_sent_msg1_go) is set. 0b - No Go request is generated when the host_sent_msg1_go is set. 1b - A Go request is generated when the host_sent_msg1_go is set.
24 host_msg0_go_en	host_msg0_go_en This is a read/write control bit. It can be written and read at any time, by either VSPA or a host. This bit allows the generation of a VSPA_go request when the Host Sent Message 0 to VCPU - Go flag (host_sent_msg0_go) is set. 0b - No Go request is generated when the host_sent_msg0_go is set. 1b - A Go request is generated when the host_sent_msg0_go is set.
23 host_read_msg1_go	host_read_msg1_go **NOTE** Access to this field is non-standard and is described in detail below. Host Read Message 1 from VCPU - Go flag. This is a VCPU write-1-to-clear status bit. When set, it indicates that the host has read message 1 from the VCPU-to-Host Mailbox. This bit can only be set when the host reads the message mentioned. It can only be cleared by the VCPU writing a 1 to it. The host cannot clear it. 0b - VCPU/Host read - No go request is pending from the host mailbox

Table continues on the next page...

Table continued from the previous page...

Field	Function
	1b - VCPU/Host read - Go request was generated by the host reading a message from the mailbox.
22 host_read_msg0_go	host_read_msg0_go **NOTE** Access to this field is non-standard and is described in detail below. Host Read Message 0 from VCPU - Go flag. This is a VCPU write-1-to-clear status bit. When set, it indicates that the host has read message 0 from the VCPU-to-Host Mailbox. This bit can only be set when the host reads the message mentioned. It can only be cleared by the VCPU writing a 1 to it. The host cannot clear it. 0b - VCPU/Host read - No go request is pending from the host mailbox 1b - VCPU/Host read - Go request was generated by the host reading a message from the mailbox
21 host_sent_msg1_go	host_sent_msg1_go **NOTE** Access to this field is non-standard and is described in detail below. Host Sent Message 1 to VCPU - Go flag. This is a VCPU write-1-to-clear status bit. When set, it indicates that the host has written a message into the Host-to-VCPU Mailbox. This bit can only be set by writing the message mentioned. It can only be cleared by the VCPU writing a 1 to it. The host cannot clear it. **NOTE** VCPU must read the message before clearing this bit. Failing to follow this protocol may result in spurious VCPU_Go events. 0b - VCPU/Host read - No go request is pending from the host mailbox 1b - VCPU/Host read - Go request was generated by the host writing a message to the mailbox
20 host_sent_msg0_go	host_sent_msg0_go **NOTE** Access to this field is non-standard and is described in detail below. Host Sent Message 0 to VCPU - Go flag. This is a VCPU write-1-to-clear status bit. When set, it indicates that the host has written a message into the Host-to-VCPU Mailbox. This bit can only be set by writing the message mentioned. It can only be cleared by the VCPU writing a 1 to it. The host cannot clear it. **NOTE** VCPU must read the message before clearing this bit. Failing to follow this protocol may result in spurious VCPU_Go events.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	0b - VCPU/Host read - No go request is pending from the host mailbox 1b - VCPU/Host read - Go request was generated by the host writing a message to the mailbox
19-18 —	- Reserved
17 dma_halt_req	dma_halt_req DMA Halt Request Setting this bit makes a request to the DMA to halt operation at a clean AXI transaction boundary. This control bit must be set prior to allowing reset, isolation, or powering of the VSPA DMA circuitry. After this bit is set, the system must wait until the VSPA_STATUS register bit dma_halt_ack is also set before allowing any of the aforementioned actions. This ensures that the AXI bus is in a neutral state, and will prevent any corruption of AXI bus operation. 0b - Not requesting the DMA to halt 1b - Request for the DMA to halt operation at a clean AXI boundary
16 —	- This bit MUST be written to a 0.
15-13 —	- Reserved
12 host_vsp_flags1_go_en	host_vsp_flags1_go_en Host to VSPA flags 1 GO enable This bit enables/disables VSPA GO as results of one or more bits set in VSPA_HOST_VCPU_FLAGS1 register. If enabled, VSPA_GO request will be asserted whenever one or more bits are set in the VSPA_HOST_VCPU_FLAGS1 register. 0b - Host to VSPA flags 1 GO is disabled 1b - Host to VSPA flags 1 GO is enabled
11 host_vsp_flags0_go_en	host_vsp_flags0_go_en Host to VSPA flags 0 GO enable This bit enables/disables VSPA GO as a result of one or more bits set in VSPA_HOST_VCPU_FLAGS0 register. If enabled, VSPA_GO request will be asserted whenever one or more bits are set in the VSPA_HOST_VCPU_FLAGS0 register. 0b - Host to VSPA flags 0 GO is disabled 1b - Host to VSPA flags 0 GO is enabled
10 —	- Reserved

Table continues on the next page...

Table continued from the previous page...

Field	Function
9 debug_msg_go_en	debug_msg_go_en Debug message go enable This is a control bit. When set, it enables VCPU_Go when the debug module writes a 32-bit message or a 64-bit message LSB into the Debug-to-VCPU Mailbox. It is a read/write register bit. It can be written and read at any time. 0b - No Go request is generated when the debug module writes a message to the debug mailbox 1b - Go request is generated when the debug module writes a message to the debug mailbox
8 axislv_go	axislv_go Read only status bit. 0b - Indicates no VCPU go is pending from the AXI slave subsystem. 1b - Indicates that a VCPU go is pending due to one or more AXI slave flag bits being set, while the corresponding AXI slave go enable bits are also set.
7 fecu_go	fecu_go **NOTE** Access to this field is non-standard and is described in detail below. This is a VCPU write-1-to-clear status bit. When set, it reflects that FECU issued a VCPU GO command. It can only be set if FECU asserts the fecu_done signal AND the vcpu_go_enable bit in the FECU_CONTROL register (FECU_CONTROL[10]) is also set. It can only be cleared by VCPU writing a 1 to it. The external IPbus master (host) may not clear this bit. 0b - VCPU/Host read - No go request is pending from FECU 1b - VCPU/Host read - Go request was generated by FECU
6 host_vsp_flags_go	host_vsp_flags_go **NOTE** Access to this field is non-standard and is described in detail below. This bit is set when either condition occurs: • one or more bits are set in the VSPA_HOST_VCPU_FLAGS0 register and host_vsp_flags0_go_en is set to one, or • one or more bits are set in the VSPA_HOST_VCPU_FLAGS1 register and host_vsp_flags1_go_en is set to one This bit is cleared when both conditions occur: • all bits in the VSPA_HOST_VCPU_FLAGS0 register are zero or the host_vsp_flags0_go_en is cleared, and • all bits in the VSPA_HOST_VCPU_FLAGS1 register are zero or the host_vsp_flags1_go_en is cleared

Table continues on the next page...

Table continued from the previous page...

Field	Function
	0b - VCPU/Host read - No go request is pending from the host flags registers 1b - VCPU/Host read - Go request was generated by the host setting a bit/bits in the VSPA_HOST_VCPU_FLAGS0/1 registers, and the corresponding enable bit is set.
5 debug_msg_go	debug_msg_go **NOTE** Access to this field is non-standard and is described in detail below. Debug message go This is a VCPU write-1-to-clear status bit. When set, it indicates that the debug module has written a 32-bit message or a 64-bit message LSB into the Debug-to-VCPU Mailbox. It can only be set only by writing the messages mentioned. It can only be cleared by the VCPU writing a 1 to it. The host cannot clear it. Note: VCPU must read the message (either the 32-bit message or the 64-bit message) before clearing this bit. Failing to follow this protocol may result in spurious VCPU_Go events. 0 VCPU/Host read - No go request is pending from the debug host mailbox 1 VCPU/Host read - Go request was generated by the debug host writing a message to the mailbox 0 Host write - Ignored 1 Host write - Ignored 0 VCPU write - Ignored 1 VCPU write - Clear the status bit (if set)
4 vcpu_go	vcpu_go The VCPU program, operating in either USER or SUPV state, uses this bit to tell ITSELF to go. This allows deferred (queued) tasks to be processed repeatedly (and end with a DONE) even if there were no other pending GO requests. The VCPU SUPV or USER can write to 1, only SUPV can write to 0. When 1, a GO event will be pending. Host reads and writes have no effect. **NOTE** Since this version of VSPA is not implementing thread protection, USER mode is disabled and the SUPV bit is always held in the SUPV state.
3 ext_go	ext_go External go This is a read-only status bit. When set, it reflects that an external GO event issued a VCPU GO command (that is, 2 corresponding bits in the EXT_GO_STAT and EXT_GO_ENA registers were both set). This bit is cleared by clearing all the corresponding pairs of EXT_GO_STAT and EXT_GO_ENA register bits. To clear a corresponding pair of EXT_GO_STAT and EXT_GO_ENA register bits the user can clear either one or both bits. See for more details.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	0b - No go request is pending from an external event 1b - Go request was generated by an external event
2 dma_go	dma_go DMA unit go This is a read-only status bit. When set, it reflects that the DMA unit issued a VCPU GO command. This bit is read only and can only be cleared by clearing all the bits in the DMA_GO_STAT register. 0b - No go request is pending from the DMA unit 1b - Go request was generated by the DMA unit
1 ippu_go	ippu_go **NOTE** Access to this field is non-standard and is described in detail below. IPPU go This is a VCPU write-1-to-clear status bit. When set, it reflects that the IPPU issued a VCPU GO command. It can only be set by the IPPU completion with vcpu_go_en=1. It can only be cleared by VCPU writing a 1 to it. The external IPbus master (host) may not clear this bit. 0 VCPU/Host read - No go request is pending from the IPPU 1 VCPU/Host read - Go request was generated by the IPPU 0 Host write - Ignored 1 Host write - Ignored 0 VCPU write - Ignored 1 VCPU write - Clear the status bit (if set)
0 host_go	host_go **NOTE** Access to this field is non-standard and is described in detail below. Host go This is a combination control/status bit. It can only be set by the external IPbus master (host processor) writing a 1 to it. It can only be cleared by VCPU writing a 1 to it. The external IPbus master (host) may not clear this bit. When set, it requests that VCPU start processing instructions at PMEM address 0 (assuming VCPU is stopped). 0 VCPU/Host read - VCPU has not been requested to go by the host since it last cleared this bit 1 VCPU/Host read - A VCPU host go was requested by the host and was not yet cleared by the VCPU 0 Host write - Ignored 1 Host write - Start VCPU core processing at PMEM address 0.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	0 VCPU write - Ignored 1 VCPU write - Clear the status bit (if set)

14.2.5 VSPA Interrupt Enable register (IRQEN)

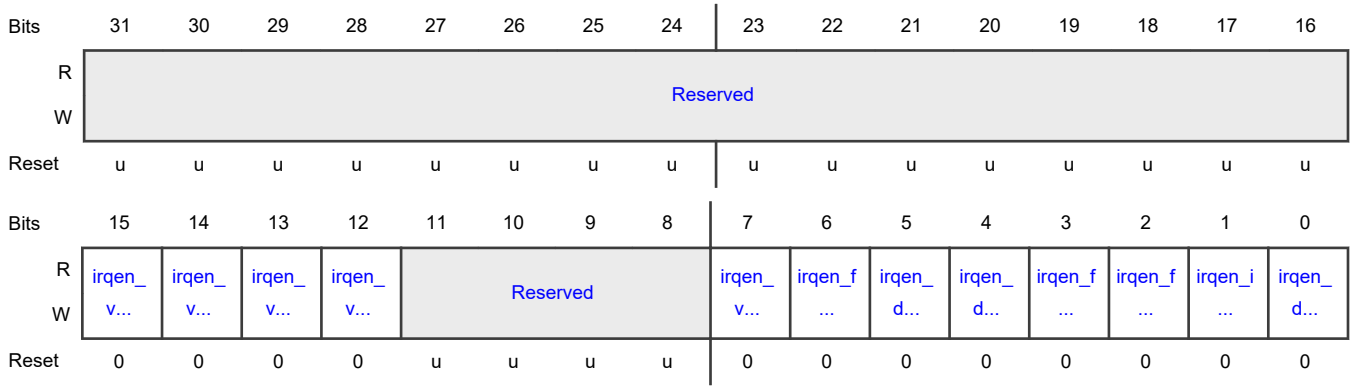
Offset

Register	Offset
IRQEN	Ch

Function

VSPA Interrupt Enable register

Diagram



Fields

Field	Function
31-16 —	- Reserved
15 irqen_vcpu_read_msg1	irqen_vcpu_read_msg1 Host to VCPU message 1 read interrupt enable. This bit enables/disables one of the host message interrupts. If enabled, the interrupt will be asserted whenever the VSPA_STATUS vcpu_read_msg1 bit is set. 0b - Host to VCPU message 1 read interrupt disabled 1b - Host to VCPU message 1 read interrupt enabled
14	irqen_vcpu_read_msg0

Table continues on the next page...

Table continued from the previous page...

Field	Function
irqen_vcpu_read_msg0	Host to VCPU message 0 read interrupt enable. This bit enables/disables one of the host message interrupts. If enabled, the interrupt will be asserted whenever the VSPA_STATUS vcpu_read_msg0 bit is set. 0b - Host to VCPU message 0 read interrupt disabled 1b - Host to VCPU message 0 read interrupt enabled
13 irqen_vcpu_sent_msg1	irqen_vcpu_sent_msg1 VCPU to Host message 1 sent interrupt enable. This bit enables/disables one of the host message interrupts. If enabled, the interrupt will be asserted whenever the VSPA_STATUS vcpu_sent_msg1 bit is set. 0b - VCPU to host message 1 sent interrupt disabled 1b - VCPU to host message 1 sent interrupt enabled
12 irqen_vcpu_sent_msg0	irqen_vcpu_sent_msg0 VCPU to Host message 0 sent interrupt enable. This bit enables/disables one of the host message interrupts. If enabled, the interrupt will be asserted whenever the VSPA_STATUS vcpu_sent_msg0 bit is set. 0b - VCPU to host message 0 sent interrupt disabled 1b - VCPU to host message 0 sent interrupt enabled
11-8 —	- Reserved.
7 irqen_vcpu_iit	irqen_vcpu_iit VCPU illegal instruction trap interrupt enable. This bit enables or disables the vspa_iit_irq_b interrupt. This is used to interrupt the host in the event when the VCPU attempts to execute an illegal instruction. 0b - VCPU Illegal instruction trap interrupt disabled 1b - VCPU Illegal instruction trap interrupt enabled
6 irqen_fecu_done	FECU done interrupt enable Enables the FECU done interrupt. When enabled, when FECU asserts the fecu_done signal AND the 'vspa_irq_enable' bit in the FECU_CONTROL register (FECU_CONTROL[12]) is also set, vspa_fecu_irq_b interrupt will be asserted. 0b - FECU done interrupt disabled 1b - FECU done interrupt enabled
5 irqen_dma_error	irqen_dma_error VSPA DMA error interrupt enable

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Enables the dma_err_irq_b interrupt. When enabled, if any DMA channel detects either an AXI transfer error or a DMA configuration error, this interrupt will be asserted. 0b - DMA error interrupt disabled 1b - DMA error interrupt enabled
4 irqen_dma_cmp	irqen_dma_cmp VSPA DMA complete interrupt enable Enables the dma_cmp_irq_b interrupt. When enabled, this interrupt will be asserted whenever any DMA_COMP_STAT status flag is set and its associated DMA_XFR_CTRL[irq_en] is also set. 0b - DMA transfer complete interrupt disabled 1b - DMA transfer complete interrupt enabled
3 irqen_flags1	irqen_flags1 Flags 1 Interrupts Enable Enables the vcpu_flags1_irq_b interrupt. This interrupt is asserted whenever any of the bits in the VCPU_HOST_FLAGS1 register are set. 0b - VCPU_HOST_FLAGS1 interrupts disabled 1b - VCPU_HOST_FLAGS1 interrupts enabled
2 irqen_flags0	irqen_flags0 Flags 0 interrupts enable Enables the vcpu_flags0_irq_b interrupt. This interrupt is asserted whenever any of the bits in the VCPU_HOST_FLAGS0 register are set. 0b - VCPU_HOST_FLAGS0 interrupts disabled 1b - VCPU_HOST_FLAGS0 interrupts enabled
1 irqen_ippu_done	irqen_ippu_done IPPU done interrupt enable Enables the IPPU done interrupt. When enabled, when the IPPU done status flag asserts, this interrupt will be asserted. 0b - IPPU done interrupt disabled 1b - IPPU done interrupt enabled
0 irqen_done	irqen_done VCPU Done interrupt enable Enables the vcpu_done_irq_b interrupt. This interrupt is asserted whenever the DONE bit in the STATUS register is set. 0b - Done interrupt disabled 1b - Done interrupt enabled

14.2.6 VSPA Source 1 Info (STATUS)

Offset

Register	Offset
STATUS	10h

Function

NOTE

Pay close attention to the register bit field descriptions following the register figure because a field designated with read-write access in the register figure may have additional non-standard behavior described in its corresponding register bit field description.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	Reserved														dma_hal...	Reserved
W																
Reset	u	u	u	u	u	u	u	u	u	u	u	u	u	u	0	u

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	vcpu_re...	vcpu_re...	vcpu_se...	vcpu_se...	Reserved			busy	vcpu_iit	irq_pen...	irq_pen...	irq_pen...	irq_pen...	irq_pen...	irq_pen...	done
W	W1C	W1C	W1C	W1C					W1C	W1C					W1C	W1C
Reset	0	0	0	0	u	u	u	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-18 —	- Reserved
17 dma_halt_ack	dma_halt_ack DMA halt acknowledge. This bit is set whenever the DMA has completed a halt request. A DMA halt request can come from either a debugger halt request, or the dma_halt_req bit in the VSPA_CONTROL register. This status bit must be set prior to allowing reset, isolation, or powering of the VSPA DMA circuitry. This ensures that the AXI bus is in a neutral state, and will prevent any corruption of AXI bus operation.. 0b - The DMA has not halted operation in response to a halt request 1b - The DMA has halted operation in response to a halt request
16	-

Table continues on the next page...

Table continued from the previous page...

Field	Function
—	Reserved
15 vcpu_read_msg1	vcpu_read_msg1 VCPU read message 1. This status bit sets when the VCPU reads message 1 (sent by the host). This can act as an acknowledgement to the host that the VCPU has consumed the message in mailbox 1. It can also be used to produce an interrupt if the irqen_vcpu_read_msg1 bit in the VSPA_IRQEN register is set. This status bit will be set only if the VCPU reads the VSPA_VCPU_IN_1_LSB register while VSPA_VCPU_MBOX_STATUS bit msg1_in_valid==1. The bit will be cleared when written to a 1 by either the host or the VCPU. 0b - The status bit was never set following reset, or was cleared by a write 1 to clear (for read) 1b - The VCPU read the VSPA_VCPU_IN_1_LSB register while VSPA_VCPU_MBOX_STATUS bit msg1_in_valid was set (for read)
14 vcpu_read_msg0	vcpu_read_msg0 VCPU read message 0. This status bit sets when the VCPU reads message 0 (sent by the host). This can act as an acknowledgement to the host that the VCPU has consumed the message in mailbox 0. It can also be used to produce an interrupt if the irqen_vcpu_read_msg0 bit in the VSPA_IRQEN register is set. This status bit will set only if the VCPU reads the VSPA_VCPU_IN_0_LSB register while VSPA_VCPU_MBOX_STATUS bit msg0_in_valid==1. The bit will be cleared when written to a 1 by either the host or the VCPU. 0b - The status bit was never set following reset, or was cleared by a write 1 to clear (for read) 1b - The VCPU read the VSPA_VCPU_IN_0_LSB register while VSPA_VCPU_MBOX_STATUS bit msg0_in_valid was set (for read)
13 vcpu_sent_msg1	vcpu_sent_msg1 VCPU sent message 1. This status bit sets when the VCPU sends message 1 to the host. It can also be used to produce an interrupt if the irqen_vcpu_sent_msg1 bit in the VSPA_IRQEN register is set. This bit will be set only if the VCPU writes the VSPA_VCPU_OUT_1_LSB register. The bit will be cleared when written to a 1 by either the host or the VCPU. **NOTE** The host must read the message before clearing this bit. Failing to follow this protocol may result in spurious interrupts. 0b - The status bit was never set following reset, or was cleared by a write 1 to clear (for read) 1b - The VCPU wrote the VSPA_VCPU_OUT_1_LSB register (for read)
12	vcpu_sent_msg0 VCPU sent message 0.

Table continues on the next page...

Table continued from the previous page...

Field	Function
vcpu_sent_msg0	This status bit sets when the VCPU sends message 0 to the host. It can also be used to produce an interrupt if the irqen_vcpu_sent_msg0 bit in the VSPA_IRQEN register is set. This bit will be set only if the VCPU writes the VSPA_VCPU_OUT_0_LSB register. The bit will be cleared when written to a 1 by either the host or the VCPU. **NOTE** The host must read the message before clearing this bit. Failing to follow this protocol may result in spurious interrupts. 0b - The status bit was never set following reset, or was cleared by a write 1 to clear (for read) 1b - The VCPU wrote the VSPA_VCPU_OUT_0_LSB register (for read)
11-9 —	- Reserved
8 busy	busy VCPU busy Indicates whether the VCPU core is idle or busy executing code. 0b - Idle 1b - Busy
7 vcpu_iit	vcpu_iit VCPU illegal instruction trap. This status bit indicates that the VCPU decoded an illegal instruction. Note that the illegal instruction is treated as an NOP. When set, the VSPA_ILLOP_STATUS register is updated to contain the address where the first illegal instruction was found. This status bit is write one to clear. When it is cleared, the VSPA_ILLOP_STATUS register is also cleared. 0b - No illegal instructions were detected by the VCPU 1b - One or more illegal instructions were detected by the VCPU
6 irq_pend_fecu_done	IRQ pending FECU done Indicates FECU done status flag was asserted (AND the 'vspa_irq_enable` bit in the FECU_CONTROL register (FECU_CONTROL[12]) is also set). This bit can only be cleared by writing one to this bit location. 0b - No IRQ pending 1b - FECU done interrupt is pending
5 irq_pend_dma_error	irq_pend_dma_error IRQ pending DMA error Indicates that the DMA unit detected an AXI bus error during a transfer of data over the AXI interface, or that an attempt was made to activate a channel with incorrectly configured parameters. This status bit cannot be directly cleared. It is simply the logical OR of all of the bits contained in the DMA_XFRERR_STAT

Table continues on the next page...

Table continued from the previous page...

Field	Function
	and DMA_CFGERR_STAT registers. It can only be cleared by clearing those underlying status register bits. 0b - No IRQ pending 1b - DMA transfer or configuration error occurred
4 irq_pend_dma_comp	irq_pend_dma_comp IRQ pending DMA complete Indicates that one or more bits are set in the DMA_IRQ_STAT register. This bit is simply the logical OR of all the bits in the DMA_IRQ_STAT register. It can only be cleared by clearing all of the bits in the DMA_IRQ_STAT register. 0b - No IRQ pending 1b - DMA transfer completed for at least one channel which has its associated DMA_IRQ_STAT bit set
3 irq_pend_flags1	irq_pend_flags1 IRQ pending VCPU_HOST_FLAGS1 Indicates that one or more bits are set in the VCPU_HOST_FLAGS1 register. This bit is simply the logical OR of all of the bits in the VCPU_HOST_FLAGS1 register. It can only be cleared by clearing all of the bits in the VCPU_HOST_FLAGS1 register. 0b - No VCPU_HOST_FLAGS1 bits set 1b - At least one bit in the VCPU_HOST_FLAGS1 register is set
2 irq_pend_flags0	irq_pend_flags0 IRQ pending VCPU_HOST_FLAGS0 Indicates that one or more bits are set in the VCPU_HOST_FLAGS0 register. This bit is simply the logical OR of all of the bits in the VCPU_HOST_FLAGS0 register. It can only be cleared by clearing all of the bits in the VCPU_HOST_FLAGS0 register. 0b - No VCPU_HOST_FLAGS0 bits set 1b - At least one bit in the VCPU_HOST_FLAGS0 register is set
1 irq_pend_ippu_done	irq_pend_ippu_done IRQ pending IPPU done Indicates that IPPU done status flag was asserted. This bit can only be cleared by writing one to this bit location. 0b - No IRQ pending 1b - IPPU done interrupt is pending
0 done	done

Table continues on the next page...

Table continued from the previous page...

Field	Function
	NOTE Access to this field is non-standard and is described in detail below. Done Indicates whether the VCPU core wishes to indicate it has completed its processing. This bit will be set automatically when the VCPU executes a "done" instruction. It can also be set anytime VCPU software writes a 1 to it. It is only cleared by a VCPU reset or the host writing a 1 to it. This bit is intended to be used to communicate status or generate an IRQ to the host processor (enabled by IRQEN register bit irqen_done). 0b - VCPU indicates processing not done 1b - VCPU indicates processing done

14.2.7 VCPU to Host flags register a (VCPU_HOST_FLAGS0 - VCPU_HOST_FLAGS1)

Offset

Register	Offset
VCPU_HOST_FLAGS0	14h
VCPU_HOST_FLAGS1	18h

Function

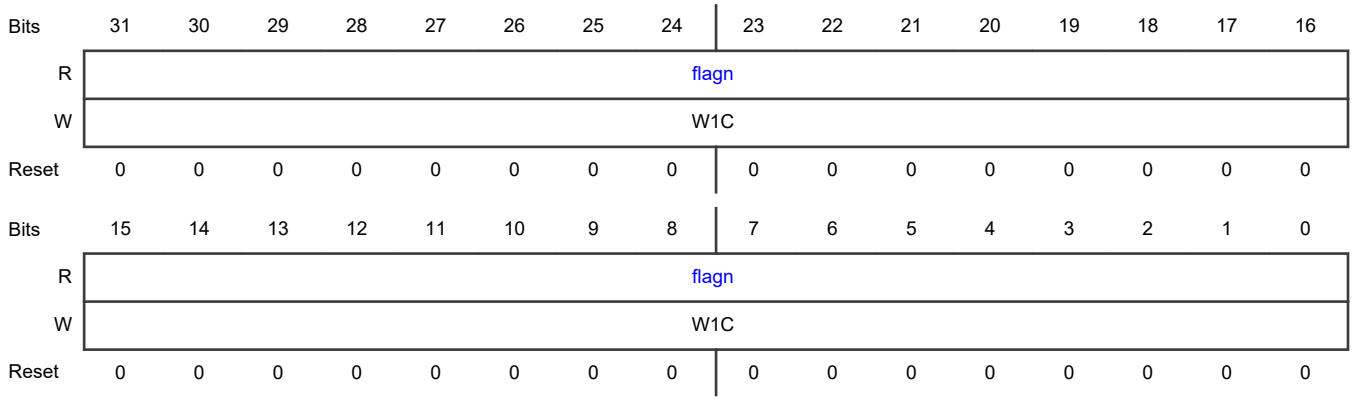
NOTE

Pay close attention to the register bit field descriptions following the register figure because a field designated with read-write access in the register figure may have additional non-standard behavior described in its corresponding register bit field description.

The VCPU_HOST_FLAGS*n* register can be used to generate host interrupts from VCPU software. If irqen_flags*n* is set in the VCPU Interrupt Enable Register, then a VCPU host group *n* event interrupt is generated when VCPU software writes any bits to a 1 in this register.

- The meaning of the 32 bits in this register is determined by the VCPU software. Each bit could be used as an independent event or the register could be used as an event code.
- Note that the event is cleared when the host writes 1 into this register.

Diagram



Fields

Field	Function
31-0 flagn	flagn **NOTE** Access to this field is non-standard and is described in detail below. Indicates that a VCPU group *n* event occurred, the nature of which is definable by VCPU software. 0 VCPU/Host read - No VCPU group *n* event occurred 1 VCPU/Host read - VCPU group *n* event occurred 0 Host write - Ignored 1 Host write - Clear VCPU group *n* event flag 0 VCPU write - Ignored 1 VCPU write - Set VCPU group *n* event flag

14.2.8 Host to VCPU Flags register a (HOST_VCPU_FLAGS0 - HOST_VCPU_FLAGS1)

Offset

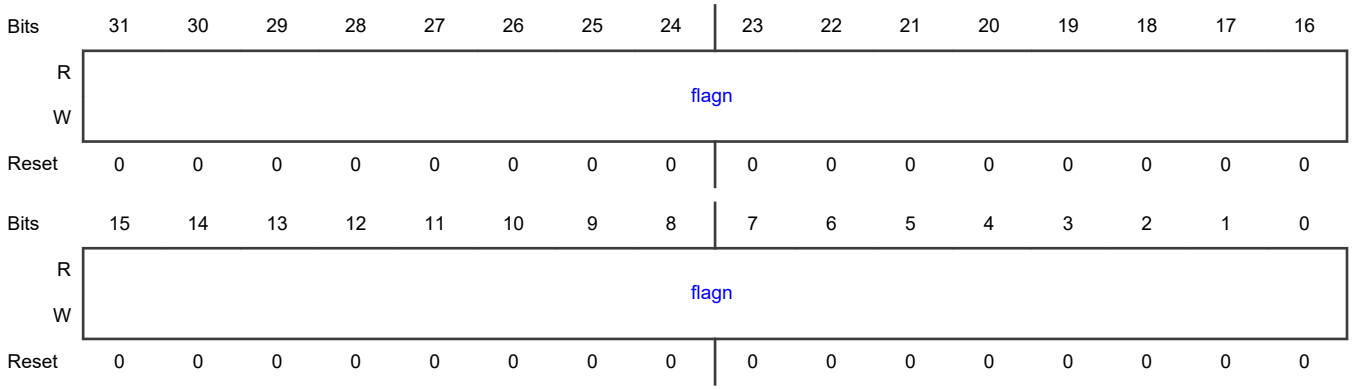
Register	Offset
HOST_VCPU_FLAGS0	1Ch
HOST_VCPU_FLAGS1	20h

Function

This register is essentially a collection of 32 single bit flags, for communications from the host to the VCPU. The host sets the bits, and the VCPU clears them, presumably to acknowledge they were seen as set. This register has an associated go enable bit, so that the VCPU can be made to go if any bit in the VSPA_HOST_VCPU_FLAGS*n* register is set.

The flags can only be set by writing a 1 from the host. They can only be cleared by writing a 1 from the VCPU, or by reset.

Diagram



Fields

Field	Function
31-0 flagn	flagn Host to VCPU flag. Read behavior is identical for both the host and the VCPU. Write behavior differs between the host and the VCPU. 0 - indicates VCPU wrote the bit to 1, or the bit was never set by the host, since the last reset. (for read behavior) 1- indicates the host wrote the bit to 1 (for read behavior) 0 - no effect (for host write behavior) 1 - sets the bit (for host write behavior) 0 - no effect (for VCPU write behavior) 1 - clears the bit (for VCPU write behavior)

14.2.9 External Go Enable (EXT_GO_ENA)

Offset

Register	Offset
EXT_GO_ENA	28h

Function

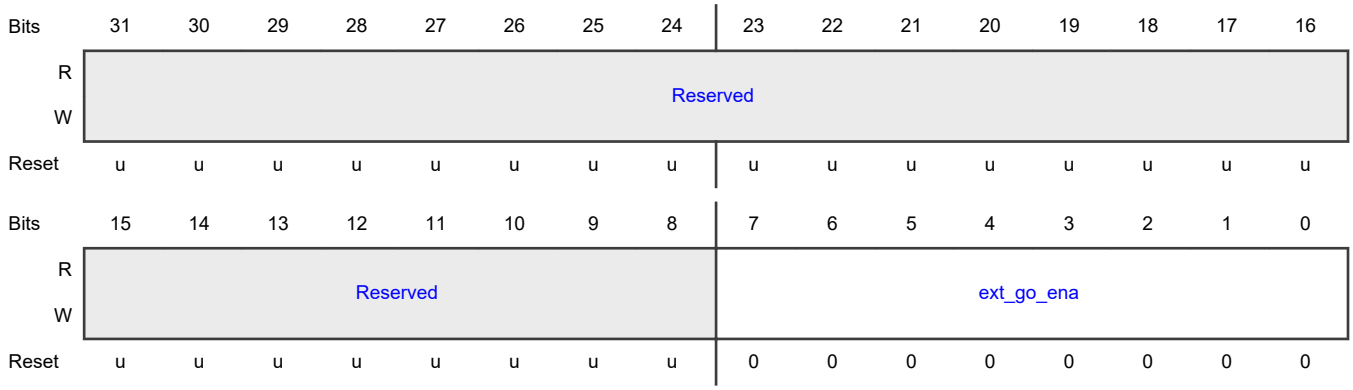
The EXT_GO_ENA register is used to control the generation of VCPU GO events in response to rising edges detected on ext_VCPU_go inputs to the VSPA platform. If a given bit *x* is set in both the EXT_GO_STAT and EXT_GO_ENA registers, then a VCPU GO will be asserted.

These register bits can be read and written by either VCPU or an IPbus host.

Once asserted, the GO event can be cleared by either:

- Clearing the EXT_GO_ENA register bit.
- Clearing the EXT_GO_STAT register bit.

Diagram



Fields

Field	Function
31-8 —	- Reserved
7-0 ext_go_ena	ext_go_ena External Go Enable Enable/disable a VCPU GO event upon detection of rising edges of the associated ext_VCPU_go inputs of the VSPA platform. 00000000b - Do not allow generation of a GO event. 00000001b - Enable generation of a GO event.

14.2.10 External Go Status (EXT_GO_STAT)

Offset

Register	Offset
EXT_GO_STAT	2Ch

Function

NOTE

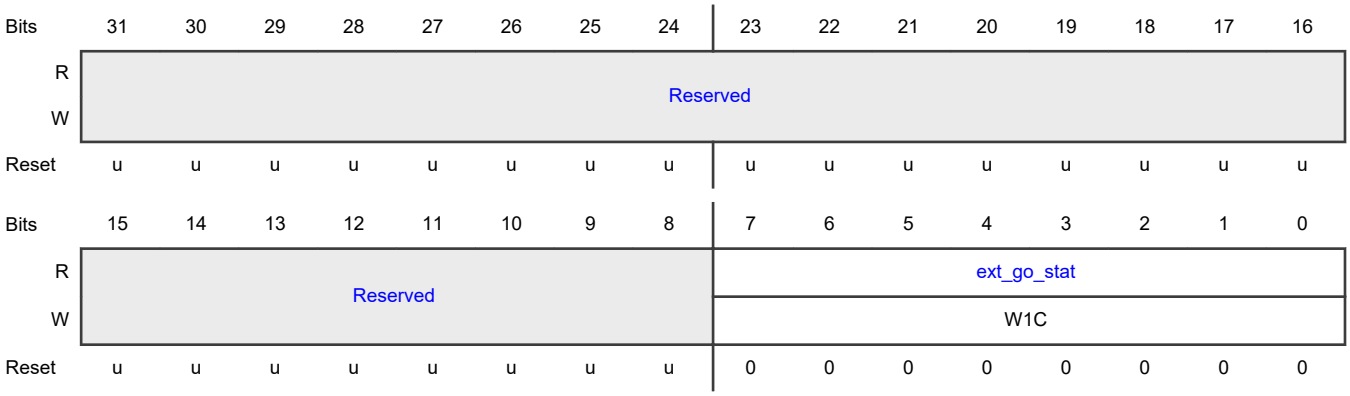
Pay close attention to the register bit field descriptions following the register figure because a field designated with read-write access in the register figure may have additional non-standard behavior described in its corresponding register bit field description.

The EXT_GO_STAT register is used to detect rising edges on the ext_VCPU_go inputs of the VSPA platform. If bits in the EXT_GO_ENA register are set, then the rising edges can also cause a VCPU GO event. If a given bit *x* is set in both the EXT_GO_STAT and EXT_GO_ENA registers, then a VCPU GO will be asserted.

These registers can be set only by rising edges on the ext_VCPU_go inputs. They can be cleared only by VCPU writing a 1 to them (write 1 to clear). The IPbus host cannot write to this register. If GO is enabled by the EXT_GO_ENA, a GO event to VCPU can be blocked or cleared by either:

- Clearing the associated EXT_GO_ENA register bit (VCPU or host must write a 0 to it).
- Clearing the associated EXT_GO_STAT register bit (VCPU must write a 1 to it).

Diagram



Fields

Field	Function
31-8 —	- Reserved
7-0 ext_go_stat	ext_go_stat **NOTE** Access to this field is non-standard and is described in detail below. External Go Stat Rising edges detected on the ext_VCPU_go input. 0 VCPU/Host read - No rising edge detected 1 VCPU/Host read - At least one rising edge was detected 0 VCPU write - Ignored 1 VCPU write - Clear status bit (if set) 0 Host write - Ignored 1 Host write - Ignored

14.2.11 VSPA VCPU Illegal Opcode Address (ILLOP_STATUS)

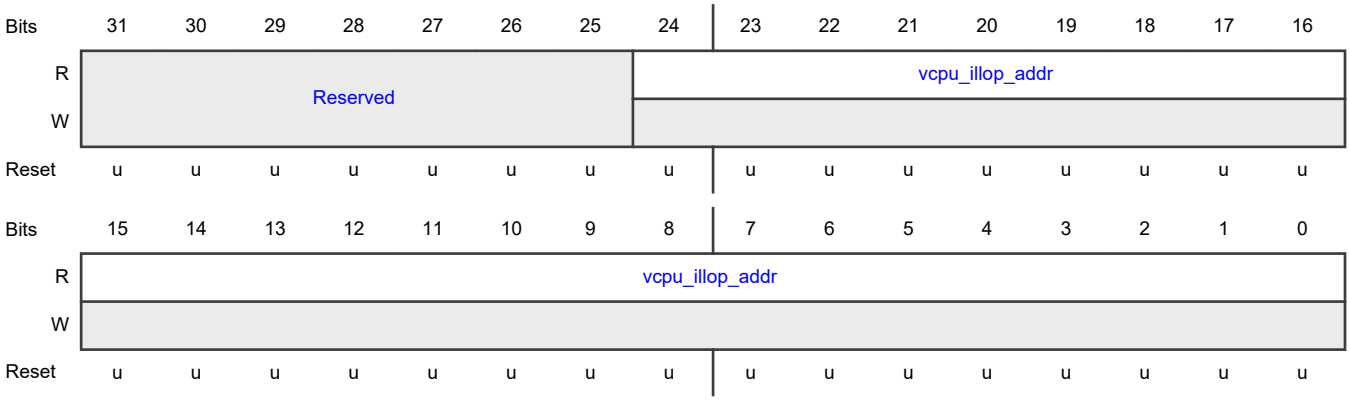
Offset

Register	Offset
ILLOP_STATUS	30h

Function

VSPA VCPU Illegal Opcode Address

Diagram



Fields

Field	Function
31-25 —	- Reserved
24-0 vcpu_illop_addr	vcpu_illop_addr VSPA VCPU Illegal Opcode Address. Indicates the address of the first illegal opcode encountered by the VCPU. The address is in units of 32-bit words. This register can only be cleared by writing a 1 to the vcpu_iit bit in the VSPA_STATUS register. Once cleared as described, the trap is rearmed. The next time the VCPU decodes an illegal opcode, it will be trapped. The VSPA_ILLOP_STATUS register and the vcpu_iit bit in the VSPA_STATUS register will both be updated.

14.2.12 VSPA Parameters 0 (PARAM0)

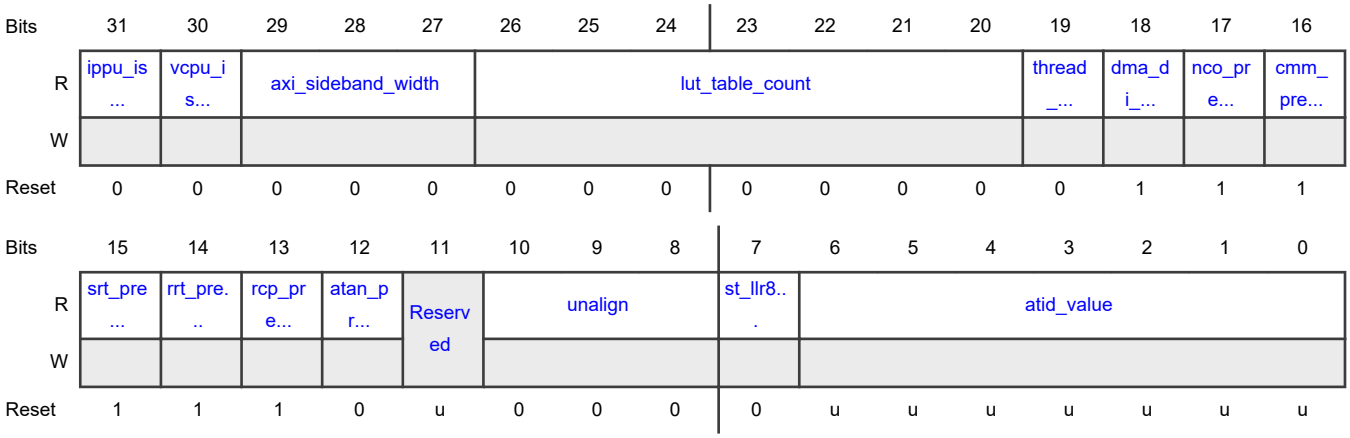
Offset

Register	Offset
PARAM0	40h

Function

PARAM0 register can be read to determine the VSPA hardware configuration which has been implemented.

Diagram



Fields

Field	Function
31 ippu_isolate	ippu_isolate IPPU isolation status Indicates whether the IPPU can operate while the VCPU is powered down. The values in this register are valid immediately after reset. 0b - The IPPU and VCPU are both powered down together. 1b - The IPPU will not be powered down, it can operate when the VCPU is powered down.
30 vcpu_isolate	vcpu_isolate VCPU isolate. This status bit indicates that the VCPU can be isolated, and possibly powered down by the system. 0b - The VCPU cannot be isolated (and cannot be powered down independently for the VSPA). 1b - The VCPU can be isolated (and possibly powered down).
29-27 axi_sideband_width	axi_sideband_width These status bits indicate the number of AXI sideband outputs the DMA uses for any given chip. The encodings are as follows: 000b - 8 awsideband and 8 arsideband outputs 001b - 1 awsideband and 1 arsideband outputs 010b - 2 awsideband and 2 arsideband outputs 011b - 3 awsideband and 3 arsideband outputs 100b - 4 awsideband and 4 arsideband outputs 101b - 5 awsideband and 5 arsideband outputs 110b - 6 awsideband and 6 arsideband outputs 111b - 7 awsideband and 7 arsideband outputs

Table continues on the next page...

Table continued from the previous page...

Field	Function
26-20 lut_table_count	lut_table_count Lookup table count These status bits indicate the number of full lookup tables supported by this instance of VSPA. Any number between 0-127 is possible, and not just the two ends. 0000000b - The VCPU has no support for lookup table functionality 0111111b - The VCPU has support for 127 full lookup tables (and 254 half tables)
19 thread_protection	thread_protection This status bit indicates whether this implementation of VSPA supports thread protection. If thread protection is supported, all the protection unit registers are implemented. If not supported, the registers do not exist, and the registers all read as 0 and the logic behaves as if the registers are always 0. 0b - thread protection is not supported 1b - thread protection is supported
18 dma_di_eng	dma_di_eng DMA de-interleaving engine. This status bit indicates whether the VSPA DMA supports de-interleaving commands. If a DMA de-interleaving command is issued to a DMA that does not support the function, a DMA configuration error will result, and the DMA command will be ignored. For further information on the DMA de-interleaving functions, refer the description of the VSPA_DMA_XFR_CTRL register. 0b - DMA does not support de-interleaving commands 1b - DMA does support de-interleaving commands
17 nco_present	nco_present This status bit indicates whether the VSPA VCPU supports numerically controlled oscillator instructions. If a VCPU NCO instruction is issued when the VCPU does not support NCO functionality, the instruction will be treated as a NOP, and will not be trapped as an illegal opcode. 0b - VCPU does not support NCO instructions 1b - VCPU does support NCO instructions
16 cmm_present	cmm_present This status bit indicates whether the VSPA VCPU supports Clustered Matrix Muxing (CMM) modes. The CMM modes are - S0group2nr, S0group2nc, S1interp2nr and S1interp2nc. If a VCPU CMM instruction is issued when the VCPU does not support CMM functionality, the instruction behavior will be undefined. 0b - VCPU does not support CMM modes 1b - VCPU supports CMM modes
15 srt_present	srt_present This status bit indicates whether the VSPA VCPU supports the square root instruction. If a VCPU srt instruction is issued when the VCPU does not support srt functionality, the instruction will be treated as a NOP, and will not be trapped as an illegal opcode.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	0b - VCPU does not support srt instructions 1b - VCPU supports srt instructions
14 rrt_present	rrt_present This status bit indicates whether the VSPA VCPU supports the reciprocal square root instruction. If a VCPU rrt instruction is issued when the VCPU does not support rrt functionality, the instruction will be treated as a NOP, and will not be trapped as an illegal opcode. 0b - VCPU does not support rrt instructions 1b - VCPU supports rrt instructions
13 rcp_present	rcp_present This status bit indicates whether the VSPA VCPU supports the reciprocal instruction. If a VCPU rcp instruction is issued when the VCPU does not support rcp functionality, the instruction will be treated as a NOP, and will not be trapped as an illegal opcode. 0b - VCPU does not support rcp instructions 1b - VCPU supports rcp instructions
12 atan_present	atan_present This status bit indicates whether the VSPA VCPU supports the arc tangent instruction. If a VCPU atan instruction is issued when the VCPU does not support atan functionality, the instruction will be treated as a NOP, and will not be trapped as an illegal opcode. 0b - VCPU does not support atan instructions 1b - VCPU supports atan instructions
11 —	- Reserved
10-8 unalign	unalign The unalign bit field is: 0 000b - No support for the st.uline instruction. Regardless of MAG pointer value, a full line will be stored, with no adjustment to the data alignment 001b - Full support for st.uline instruction. The MAG pointer can point to any 16-bit alignment and the data will be rotated into that 16-bit alignment 010b - Reduced support for st.uline instruction. The MAG pointer will be truncated to point to any 32-bit alignment and the data will be rotated into that 32-bit alignment 011b - Reduced support for st.uline instruction. The MAG pointer will be truncated to point to any 64-bit alignment and the data will be rotated into that 64-bit alignment 100b - Reduced support for st.uline instruction. The MAG pointer will be truncated to point to any 128-bit alignment and the data will be rotated into that 128-bit alignment

Table continues on the next page...

Table continued from the previous page...

Field	Function
	101b - Reduced support for st.uline instruction. The MAG pointer will be truncated to point to any 256-bit alignment and the data will be rotated into that 256-bit alignment 110b - Reduced support for st.uline instruction. The MAG pointer will be truncated to point to any 512-bit alignment and the data will be rotated into that 512-bit alignment 111b - Reduced support for st.uline instruction. The MAG pointer will be truncated to point to any 1024-bit alignment and the data will be rotated into that 1024-bit alignment
7 st_llr8_qam_enable	st_llr8_qam_enable Indicates if the Store QAM llr8 instructions are enabled 0b - VCPU does not support st.uline.llr8 instructions 1b - VCPU supports st.uline.llr8 instructions
6-0 atid_value	atid_value ATID value A VSPA instance is assigned a unique identifier. VSPA can discover its own unique identifier by reading the atid_value field in the PARAM0 register.

14.2.13 VSPA Parameters 1 (PARAM1)

Offset

Register	Offset
PARAM1	44h

Function

PARAM1 register can be read to determine the VSPA hardware configuration which has been implemented.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	rsse	axi_data_width			axi_id_width				dma_cnt							
W																
Reset	0	0	1	0	0	1	0	1	0	0	0	1	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	gp_out								gp_in							
W																
Reset	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0

Fields

Field	Function
31 rsse	rsse RSSE module present Indicates the presence of the RSSE sub-module selected by the VSPA block configuration. The values in this register are valid immediately after reset. 0b - RSSE module does not exist in this configuration 1b - RSSE module exists in this configuration
30-28 axi_data_width	axi_data_width AXI data width Indicates the width of the data channels of the external AXI buses selected by the VSPA configuration parameter AXI_DATA_WIDTH. The values in this register are valid immediately after reset 000b - 32-bit AXI data width 001b - 64-bit AXI data width 010b - 128-bit AXI data width 011b - 256-bit AXI data width 100b - 512-bit AXI data width 101b - 1024-bit AXI data width
27-24 axi_id_width	axi_id_width AXI ID width Indicates the width of the ID channels of the external AXI buses selected by the VSPA configuration parameter AXI_ID_WIDTH. The values in this register are valid immediately after reset
23-16 dma_cnt	dma_cnt DMA channel count Indicates the number of DMA channels selected by the VSPA configuration parameter DMA_CHANNEL_COUNT. The values in this register are valid immediately after reset.
15-8 gp_out	gp_out GP_OUT Register Count Indicates the number of GP Output registers selected by the VSPA configuration parameter GP_OUT_REG_COUNT. The values in this register are valid immediately after reset
7-0 gp_in	gp_in GP_IN Register Count Indicates the number of GP Input registers selected by the VSPA configuration parameter GP_IN_REG_COUNT. The values in this register are valid immediately after reset

14.2.14 VSPA Parameters 2 (PARAM2)

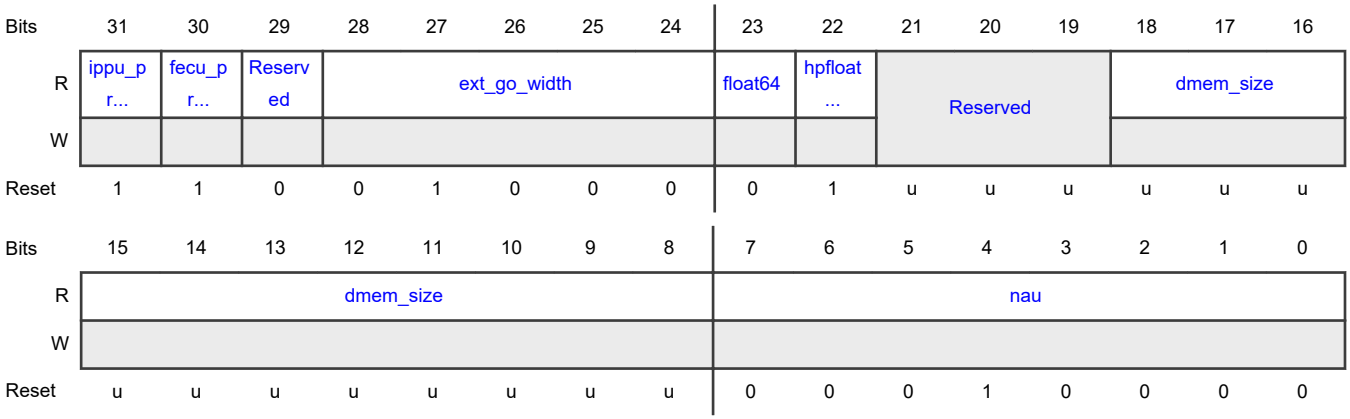
Offset

Register	Offset
PARAM2	48h

Function

PARAM2 register can be read to determine the VSPA hardware configuration which has been implemented on this VSPA instance.

Diagram



Fields

Field	Function
31 ippu_present	ippu_present IPPU present This bit indicates whether an IPPU exists in this instance of VSPA 0b - IPPU does not exist in this instance of VSPA 1b - IPPU exists in this instance of VSPA
30 fecu_present	fecu_present FECU present This bit indicates whether a FECU exists in this instance of VSPA. 0b - FECU does not exist in this instance of VSPA 1b - FECU exists in this instance of VSPA
29 —	- Reserved

Table continues on the next page...

Table continued from the previous page...

Field	Function
28-24 ext_go_width	ext_go_width External go width The value of this bit field indicates the number of ext_VCPU_go inputs that VCPU has and can respond to. Values from 1 to 31 indicate directly the number of go inputs. A value of 0 indicates there are 32 ext_VCPU_go inputs. Settings: b01000
23 float64	float64 enable Float 64 support The value of this bit indicates whether this instance of VCPU supports 64-bit floating point AU operations. This is determined by the construction of the instance in an SoC. The reset state is n, and the value of the register is a constant. 0b - 64-bit floating point AU operations are not supported 1b - 64-bit floating point AU operations are supported
22 hpfloat_present	hpfloat_present This status bit indicates whether the VSPA VCPU supports the use of half-precision floating point format. 0b - VCPU does not support half-precision floating point format 1b - VCPU supports half-precision floating point format
21-19 —	- Reserved
18-8 dmem_size	dmem_size VSPA DMEM size Indicates the location of the partition, beyond which the VCPU accesses IPPU DMEM. The memory size in bytes can be determined by multiplying the field value by 0x400.The values in this register are valid immediately after reset. **WARNING**: This bit field is deprecated, and will not be available in future revisions. It is replaced by the register VSPA_VCPU_DMEM_BYTES. Ensure that the software uses VSPA_VCPU_DMEM_BYTES for forward compatibility.
7-0 nau	nau Number of arithmetic units (AU) Indicates the number of arithmetic units implemented in this VCPU configuration. The values in this register are valid immediately after reset. 00010000b - 16AU

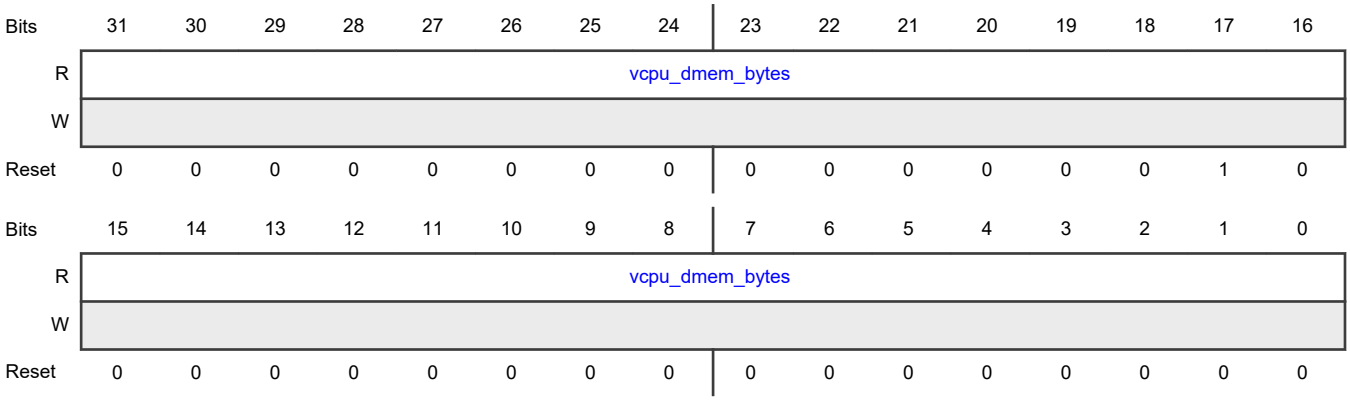
14.2.15 VCPU DMEM Size (VCPU_DMEM_BYTES)

Offset

Register	Offset
VCPU_DMEM_BYTES	4Ch

Function
VCPU DMEM Size

Diagram



Fields

Field	Function
31-0 vcpu_dmem_bytes	VSPA VCPU DMEM size in bytes. Indicates the location of VCPU DMEM partition, beyond which the VCPU and DMA access IPPU DMEM. The partition size is expressed in bytes. Note that the actual size of VCPU DMEM could be larger than the partition size; however, in most cases they will be the same. The values in this register are implementation specific, and are valid immediately after reset.

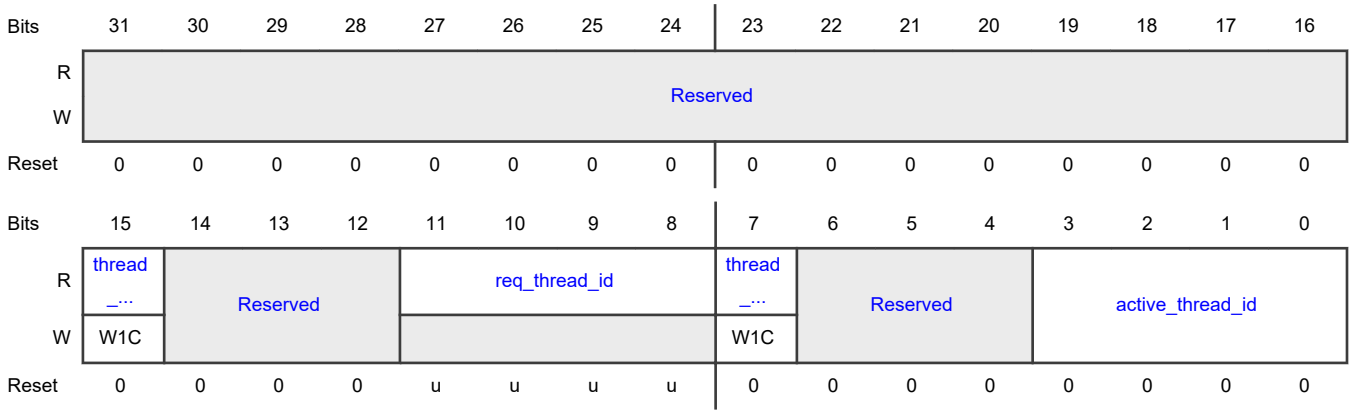
14.2.16 Thread Control and Status (THREAD_CTRL_STAT)

Offset

Register	Offset
THREAD_CTRL_STAT	50h

Function
Thread Control and Status

Diagram



Fields

Field	Function
31-16 —	- Reserved
15 thread_change_flag	thread_change_flag w1c only by VCPU SUPV or the debugger; read only by all other masters. This bit is set when a rising edge is detected on the VSPA new_thread_req input. It is intended to be used to determine that a request has been made to change to a new active thread. This request would come from an external thread scheduling system. **NOTE** This is a non-maskable GO source.
14-12 —	- Reserved
11-8 req_thread_id	req_thread_id These bits continuously reflect the synchronized state of the VSPA requested_thread_id inputs. The intended use is to determine the new thread ID being requested by an external thread scheduling system when the THREAD_CHANGE_FLAG bit is set. This bitfield is read-only.
7 thread_change_ack	thread_change_ack This bit is used by the VCPU to signal to an external thread scheduler that the previous assertion of the VSPA new_thread_req input has been acknowledged. This bit can only be written by the VCPU SUPV and the debugger; when it is written to 1, the VSPA output new_thread_ack will be asserted, conveying acknowledgment to the external thread scheduler. Writes of 0 or by any other master are ignored. This bit is self clearing; it is cleared when hardware detects the negation of the VSPA new_thread_req input. w1 to ack; When w1, asserts VSPA new_thread_ack output

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Clears when the new_thread_req pin negates Read returns the present state of the ack output.
6-4 —	- Reserved
3-0 active_thread_id	active_thread_id These bits directly control the VSPA active_thread_id outputs, and also control the internal state of the VCPU thread ID. These bits are writable by the debugger and the VCPU when it's in SUPV mode. Host writes and VCPU writes while in USER mode have no effect. Intended use is for differentiating between VSPA threads, and protecting various VSPA resources from corruption by unauthorized threads. Write changes the active_thread_id outputs. Read returns the current state.

Host/VCPU	Access
Host	RO
VCPU/Debugger	RW

NOTE

Since this version of VSPA is not implementing thread protection, USER mode is disabled and the SUPV bit is always held in the SUPV state.

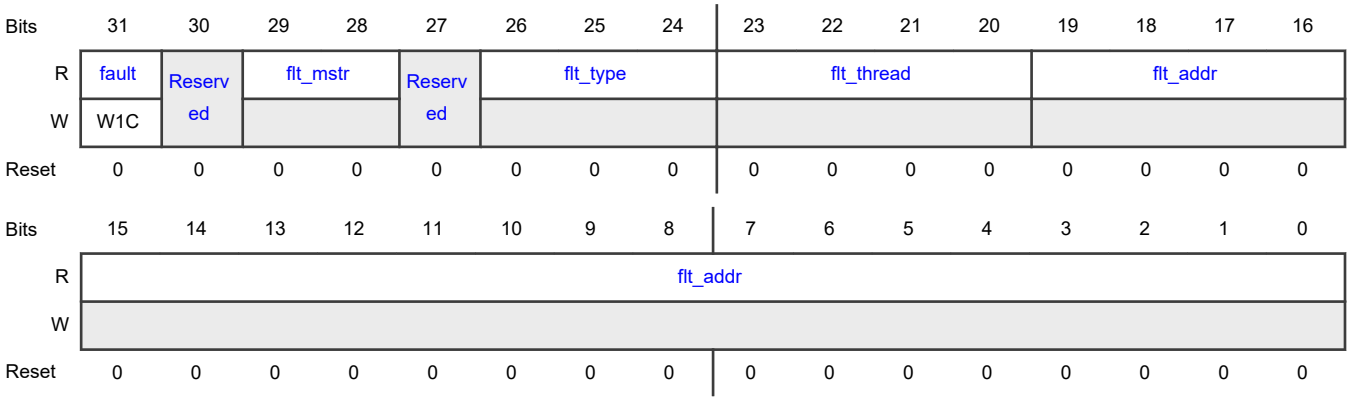
14.2.17 Protection Fault Status (PROT_FAULT_STAT)

Offset

Register	Offset
PROT_FAULT_STAT	54h

Function
Protection Fault Status

Diagram



Fields

Field	Function
31 fault	fault Indicates that a fault occurred. When this bit is set, all the other bits in this register are frozen. Subsequent faults won't change the other bits unless this bit is cleared first.
30 —	- Reserved
29-28 flt_mstr	flt_mstr Faulting master - read-only (HOST=0,VCPU=1,DMA=2,IPPU=3)
27 —	- Reserved
26-24 flt_type	flt_type Fault type - read-only (VDMEM write=1,IDMEM write=2, attempted reg. write of: IPPU=5,DMA=3,FLAG=7,GPOUT=6,SUPV ONLY reg=4,HOST wrote to VCPU only reg.=4)
23-20 flt_thread	flt_thread Thread ID that caused the fault - read-only If fault was HOST, value will be 0.
19-0 flt_addr	flt_addr Address of faulting master - read-only (VCPU=VCU_PC, IPPU=IPPU_PC, HOST=IPaddr, DMA=channel number)

14.2.18 VCPU Exception Control (EXCEPTION_CTRL)

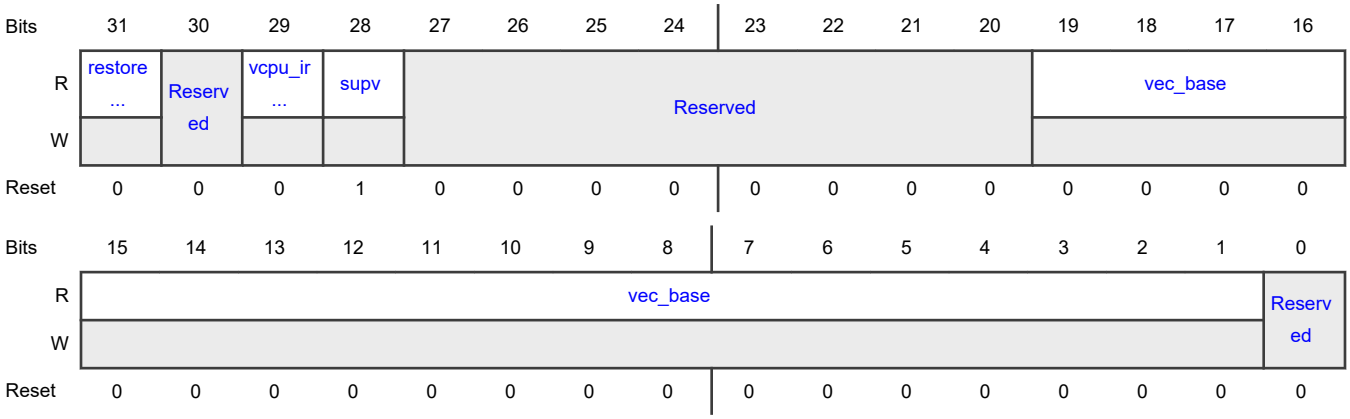
Offset

Register	Offset
EXCEPTION_CTRL	58h

Function

This register is only writable by the VCPU, as a SUPV. Other write attempts will cause IP register protection faults.

Diagram



Fields

Field	Function
31 restore_prev_vcpu_irqen	restore_prev_vcpu_irqen When written to 1, set the value of VCPU_IRQEN to PREV_VCPU_IRQEN Always reads as 0.
30 —	- Reserved
29 vcpu_irq_en	vcpu_irq_en This bit can be written by the VCPU SUPV. Other write attempts will cause IP register protection faults. It can be cleared by hardware (whenever a GO occurs or SUPV mode is entered). It can be set by hardware as an immediate response to a write of 1 to the RESTORE_PREV_VCPU_IRQEN bit, if the previous state of IRQEN was 1 when SUPV mode was last entered. This control bit enables or disables the VCPU IRQ function. 0b - The VCPU interrupt is disabled, and an assertion of VSPA input vcpu_irq will NOT cause a VCPU interrupt. 1b - The VCPU interrupt is enabled, and an assertion of VSPA input vcpu_irq WILL cause a VCPU interrupt.

Table continues on the next page...

Table continued from the previous page...

Field	Function
28 supv	supv Read: 1 if in SUPV state, 0 if in USER state. Only meaningful if VCPU is BUSY. **NOTE** Since this version of VSPA is not implementing thread protection, USER mode is disabled and the SUPV bit is always held in the SUPV state. Can only be written by VCPU when in SUPV mode; host can't write; debugger can write. Can also be set to 1 (SUPV) by processing a vcpu_irq, swi, or go. Reset value is 1.
27-20 —	- Reserved
19-1 vec_base	vec_base LSB (bit 1) is the full word aligned instruction address of the start of the JMP table which is used when processing SWI and VCPU_IRQ. With this alignment, software can mvip EXCEPTION_CTRL, SYMBOL, using a mask of 0x000FFFFF and initialize the table address (no shifting required). The JMP table base address VEC_BASE+0 is the target for the SWI. The JMP table base address VEC_BASE+8 is the target for the VCPU_IRQ.
0 —	- Reserved; always reads as 0.

14.2.19 VCPU Exception Status (EXCEPTION_STAT)

Offset

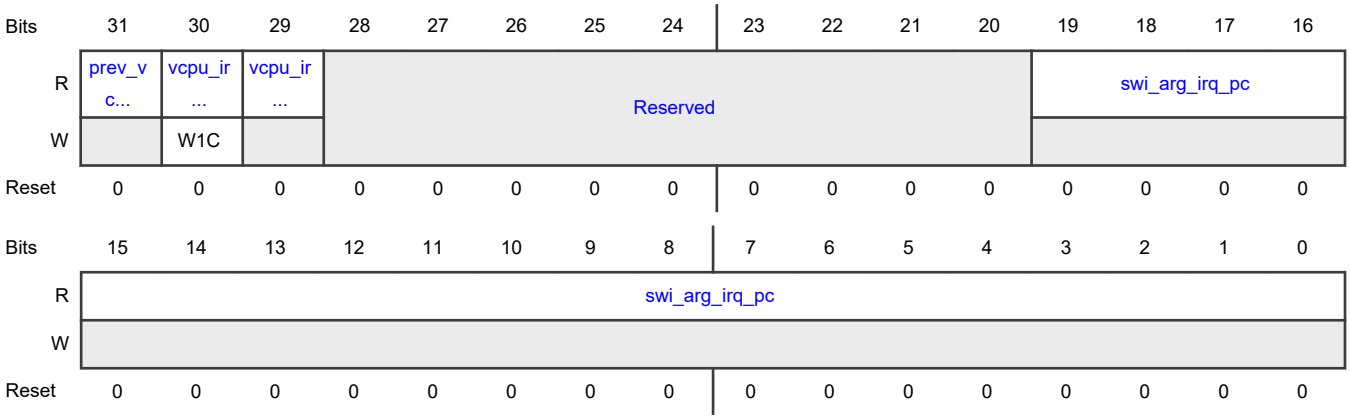
Register	Offset
EXCEPTION_STAT	5Ch

Function

NOTE

This register is only writable by the VCPU, as a SUPV. Other write attempts will cause IP register protection faults.

Diagram



Fields

Field	Function
31 prev_vcpu_irqen	prev_vcpu_irqen Read only: reflects the state of the VCPU_IRQ_EN bit when SUPV was entered last (either from SWI/VCPU_IRQ or go); writes have no effect.
30 vcpu_irq_go	vcpu_irq_go w1c when in suvp mode; else read only. Disallowed write attempts cause protection faults. This flag is set whenever the vcpu_irq input is asserted at the same time as vcpu_iqr_en (VSPA_EXCEPTION_CTRL[29]) is set. IRQ handlers must clear this bit in order to prevent a subsequent go event
29 vcpu_irq_state	vcpu_irq_state Read only: reflects the current state or the vcpu_irq input; writes have no effect.
28-20 —	- Reserved
19-0 swi_arg_irq_pc	swi_arg_irq_pc If an SWI was the cause of the current exception, a read [15:0] returns the argument provided with SWI (SWI flag is not needed since it has its own vector). Read [19:16] returns 0. If a vcpu_irq was the cause of the current exception, a read returns the PC value of the VCPU at the point the vcpu_irq was recognized. If BUSY was 0 at IRQ, this will be 0. Writes have no effect.

14.2.20 AXI Slave flags register a (AXISLV_FLAGS0 - AXISLV_FLAGS1)

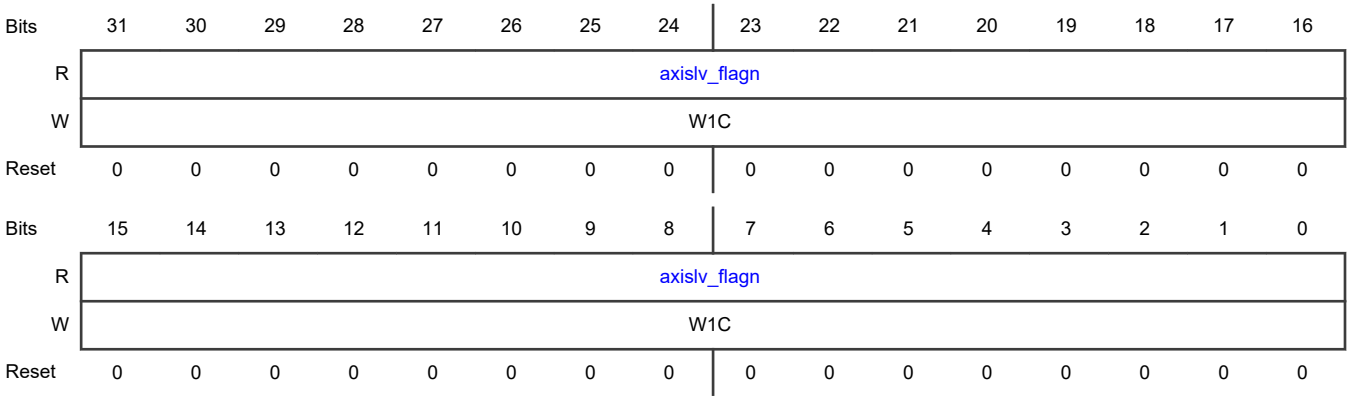
Offset

Register	Offset
AXISLV_FLAGS0	60h
AXISLV_FLAGS1	64h

Function

The register bits are w1c status bits, read from the VCPU and the host, only the VCPU can w1c.

Diagram



Fields

Field	Function
31-0 axislv_flagn	axislv_flagn (n is 31-0 for flags0, and 63-32 for flags1) The flags are set whenever the AXI slave interface is written to, with an address between 0x200000 and 0x3FFFFF. The last 64 bits of the AXI data control which flags are set by the AXI write. An AXI write of "1" will set the corresponding flag bit.

14.2.21 AXI Slave Go Enable register a (AXISLV_GOEN0 - AXISLV_GOEN1)

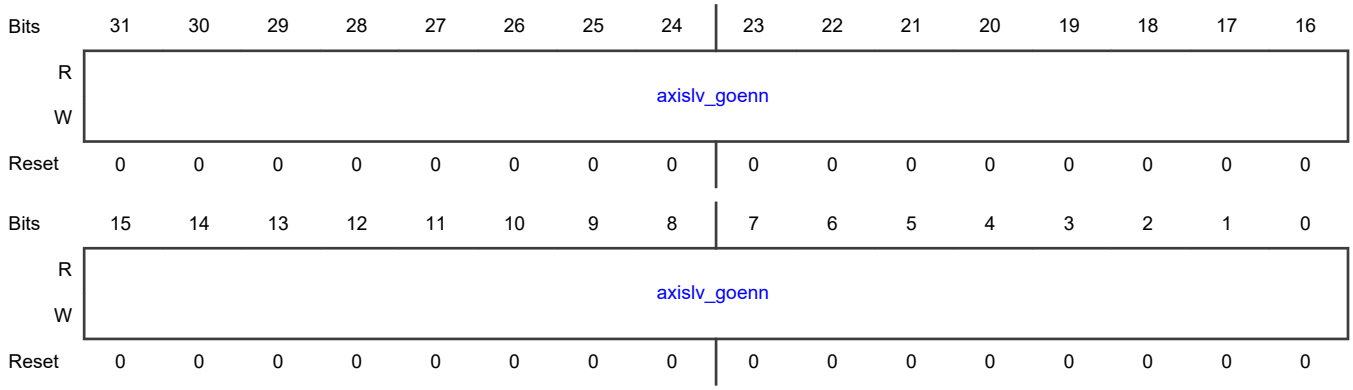
Offset

Register	Offset
AXISLV_GOEN0	68h
AXISLV_GOEN1	6Ch

Function

The bits are simple R/W bits, read and write from the VCPU and the host.

Diagram



Fields

Field	Function
31-0 axislv_goenn	axislv_goenn (n is 31-0 for goen0, and 63-32 for goen1) Enable/disable a VCPU GO event upon detection of rising edges of the associated ext_VCPU_go inputs of the VSPA platform. 00000000000000000000000000000000b - Do not allow generation of a GO event. 00000000000000000000000000000001b - Enable generation of a GO event.

14.2.22 Platform Input (PLAT_IN_0)

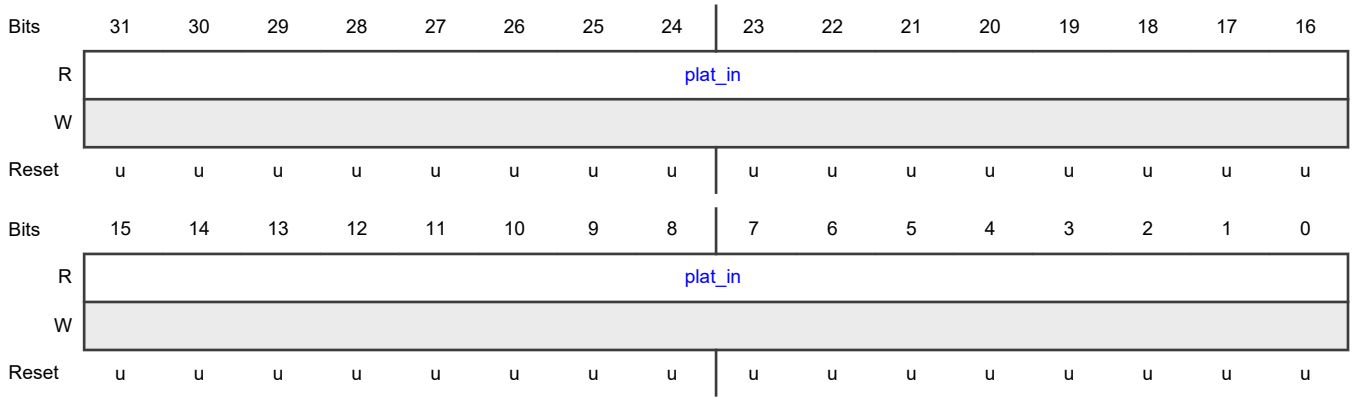
Offset

Register	Offset
PLAT_IN_0	70h

Function

This register is read only by any master, writes have no effect.

Diagram



Fields

Field	Function
31-0 plat_in	plat_in Platform integration inputs. These bits are general purpose platform-level integration inputs. They can be tied by the integration of a VSPA platform to provide visibility of the connected signals to VSPA or host software. The usage is chip-dependent.

14.2.23 Platform Output (PLAT_OUT_0)

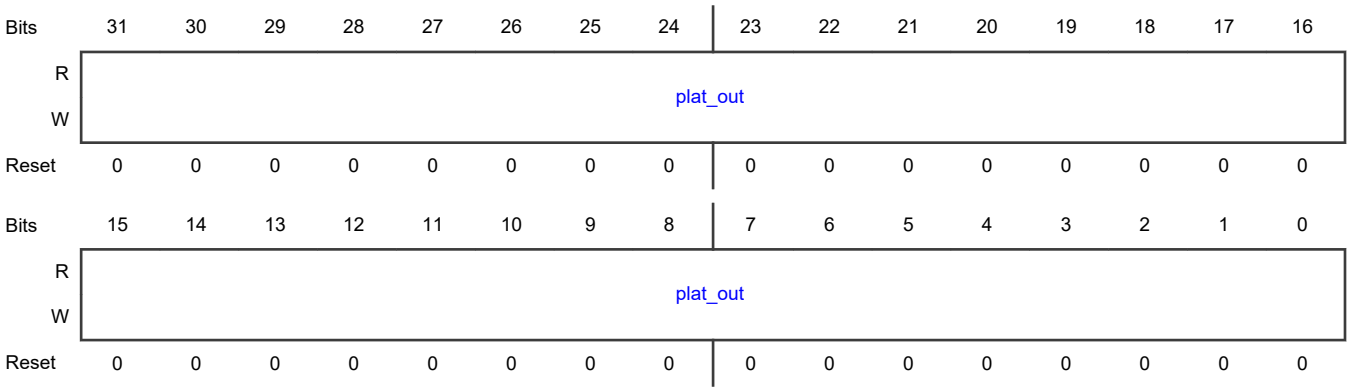
Offset

Register	Offset
PLAT_OUT_0	80h

Function

This is a read/write register. Writes possible from host, debugger, or VCPU as SUPV. VCPU user writes are ignored.

Diagram



Fields

Field	Function
31-0 plat_out	plat_out Platform integration outputs. These bits are general purpose platform-level integration outputs. They can be tied by the integration of a VSPA platform to provide control of the connected signals via VSPA or host software. The usage is chip-dependent.

14.2.24 Cycle counter MSB register (CYC_COUNTER_MSB)

Offset

Register	Offset
CYC_COUNTER_MSB	98h

Function

VSPA core has a built-in 48-bit free-running cycle counter. The cycle counter counts at the resolution of VSPA clock cycles. Out of reset, the cycle counter is disabled to save power. VCPU or the host must enable it, by writing a 1 to the msb (bit 31) of the CYC_COUNTER_MSB register, to start counting. When enabled, the cycle counter increments by one VSPA clock. The cycle counter can be used to count:

- number of clock cycles between a go event and an idle instruction
- number of clock cycles between an idle instruction and a go event
- time taken by a function in clock cycles

The VCPU can read the 48-bit register coherently, by reading the VSPA_CYC_COUNTER_MSB and the VSPA_CYC_COUNTER_LSB in two consecutive IP register accesses.

When the VCPU reads the upper 16-bits of the register, VSPA_CYC_COUNTER_MSB, the value of the lower 32 bits is captured in an internal shadow register. When the VCPU reads the lower 32-bits of the register, VSPA_CYC_COUNTER_LSB, the value read is the acquired 32 bit in a shadow register. This guarantee the 48-bit value composed from these reads is the state of the cycle counter at the time of the upper 16-bit, VSPA_CYC_COUNTER_MSB, read.

The VCPU can write the 48-bit register coherently, by writing the VSPA_CYC_COUNTER_MSB and the VSPA_CYC_COUNTER_LSB in two consecutive IP register accesses.

When the VCPU writes the upper 16-bits of the register, VSPA_CYC_COUNTER_MSB, the value of the upper 16-bit is captured in an internal shadow register. VSPA_CYC_COUNTER_MSB is not modified at this point of time. When the VCPU writes the lower 32-bit of the register, VSPA_CYC_COUNTER_LSB, the entire cycle counter 48-bit value is written at the same time. The upper 16-bit is written from the internal shadow register, simultaneously when the lower 32-bit is written. This guarantees the 48-bit value is composed from the two write accesses, the VSPA_CYC_COUNTER_MSB and the VSPA_CYC_COUNTER_LSB.

Similar to the VCPU, the host can read and write the registers coherently, and the same coherency mechanism is provided.

However, VSPA is not prevented from reading/writing the register between the host access of the upper part and the lower part. This interleaving coherency must be protected by the software. For example, host software can read the 48-bit register twice, and determine it is correct if the difference between the two read values is less than certain value. If the host write a 48-bit value, it must read it back to make sure the written value is correct.

The registers' interleaving coherency is protected only when VSPA issues the read (or write) of the upper/lower parts of the register in consecutive cycles.

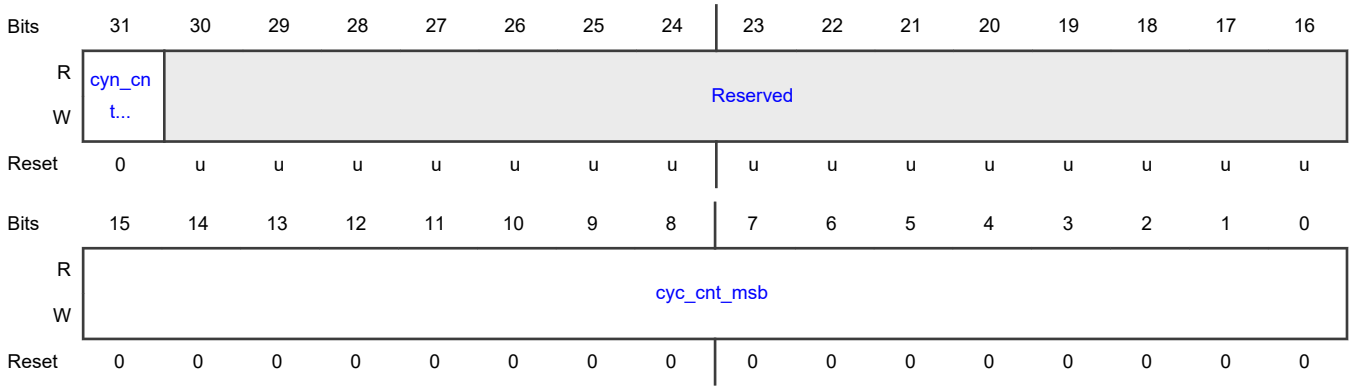
NOTE

For debug purposes, the cycle counter can be configured to halt when VSPA enters debug halted mode. For more detailed information, refer to halt_cyc_counter bit in the Debug Run Control register (RCR) register.

NOTE

The values for VSPA_CYC_COUNTER_MSB and VSPA_CYC_COUNTER_LSB registers are a snapshot when the core was stopped, but the values cannot be used with stepping to determine how much clock cycles takes for an instruction or block of code to execute.

Diagram



Fields

Field	Function
31 cyn_cnt_en	cyn_cnt_en Cycle count enable. This bit controls if the Cycle Counter is enabled (counting) or disabled (stopped). 0b - Counter is disabled. The counter is not incremented. 1b - Counter is enabled. The counter is incremented by one VSPA clock.
30-16 —	- Reserved
15-0 cyc_cnt_msb	cyc_cnt_msb Cycle count MSB This field contains the upper 16-bit, cyc_count[47:32], of the 48-bit cycle counter. Settings: Reads return 16-bit MSB, cyc_count[47:32], of the 48-bit cycle counter. Writes sets the 16-bit MSB, cyc_count[47:32], of the 48-bit cycle counter.

14.2.25 Cycle Counter LSB Register (CYC_COUNTER_LSB)

Offset

Register	Offset
CYC_COUNTER_LSB	9Ch

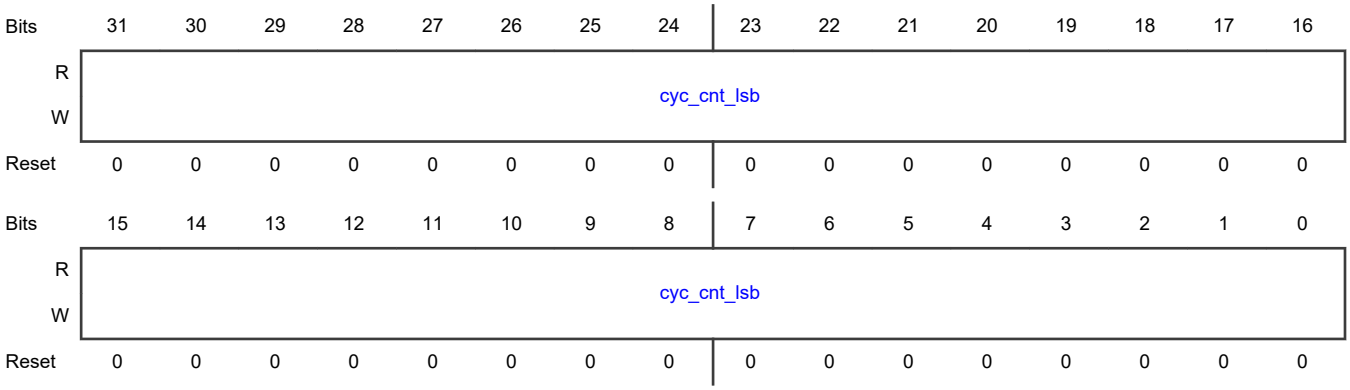
Function

VSPA core has a built-in 48-bit free-running cycle counter. See Cycle Counter MSB Register(VSPA_CYC_COUNTER_MSB) for detailed description of the cycle counter.

NOTE

The values for VSPA_CYC_COUNTER_MSB and VSPA_CYC_COUNTER_LSB registers are a snapshot when the core was stopped, but the values cannot be used with stepping to determine how much clock cycles takes for an instruction or block of code to execute.

Diagram



Fields

Field	Function
31-0 cyc_cnt_lsb	cyc_cnt_lsb Cycle count LSB. This field contains the lower 32-bit, cyc_count[31:0], of the 48-bit cycle counter. Settings: Reads return 32-bit LSB, cyc_count[31:0], of the 48-bit cycle counter. Writes sets the 32-bit LSB, cyc_count[31:0], of the 48-bit cycle counter.

14.2.26 DMEM/PRAM Address (DMA_DMEM_PRAM_ADDR)

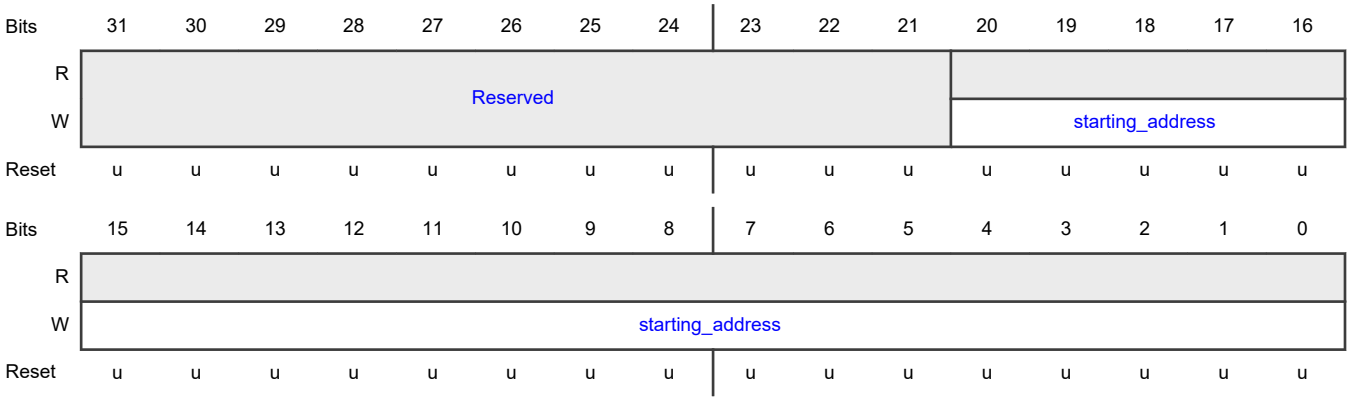
Offset

Register	Offset
DMA_DMEM_PRAM_ADDR	B0h

Function

- The DMA_DMEM_PRAM_ADDR register should be written before writing to the DMA_XFR_CTRL register.
- This register is not affected by reset and cannot be read (reads return 0s).

Diagram



Fields

Field	Function
31-21 —	- Reserved
20-0 starting_address	starting_address Starting address This bit field describes the DMEM/PRAM/IPPU PRAM byte address to be used by the transfer. For example, if reading or writing DMEM, setting this field to 0x00000 will select DMEM byte 0, 0x00006 will select DMEM byte 6. • Note that all DMEM transfers must START at a DMEM byte address that is aligned to the width of the AXI data bus. Therefore, on devices using a 32-bit data bus, the DMEM address bits 1 and 0 must be 0. On devices using a 64-bit data bus, the DMEM address bits 2, 1, and 0 must be 0. On devices using a 128-bit data bus, the DMEM address bits 3, 2, 1, and 0 must be 0. • When writing PRAM, each PRAM location is composed of 8 bytes. Therefore, this register must always be set so that bits 2, 1, and 0 are 0. Furthermore, the PMEM address specified must be aligned to the width of the AXI data bus. • When writing to IPPU PRAM, each IPPU PRAM location is considered to be composed of 4 bytes. Therefore, this register must always be set so that bits 1 and 0 are 0. Note that the starting 32-bit IPPU PRAM word is also expected to be aligned to the width of the AXI data bus. If the AXI data bus is 64 bits wide, the AXI address also needs to be 64-bit aligned. Furthermore, the IPPU PRAM starting address must be 64-bit aligned. Settings: Memory byte address in PRAM, IPPU PRAM, or DMEM, as appropriate to what the channel is configured to operate on. When the DMA is programmed for a DI storage operation, this register must be programmed to point to the DMEM byte address of the first data value of the DI data table.

14.2.27 DMA AXI Address (DMA_AXI_ADDRESS)

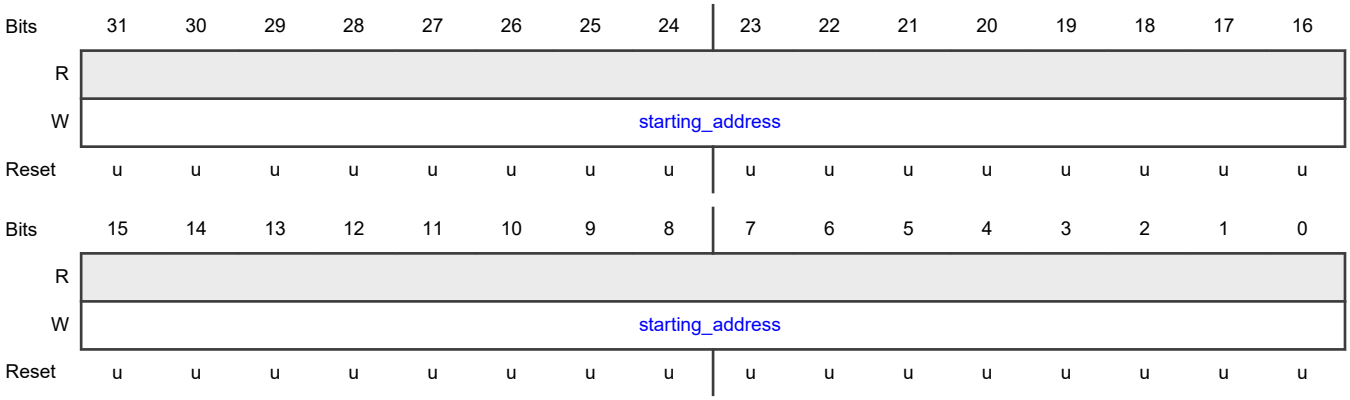
Offset

Register	Offset
DMA_AXI_ADDRESS	B4h

Function

- The DMA_AXI_ADDRESS register should be written before writing to the DMA_XFR_CTRL register.
- This register is not affected by reset and cannot be read (reads return 0s).

Diagram



Fields

Field	Function
31-0 starting_address	starting_address Starting address This field is a 32-bit field. • For transfers both to and from AXI, the LSBs of the address must be 0, allowing AXI transfers to start only on AXI bus width aligned boundaries. If the appropriate number of LSBs are not 0 in this configuration, then the configuration error status bit will be set if an attempt is made to set the associated channel enable bit, and the channel enable bit will not be set. • If transferring data into PMEM, the AXI address must be aligned to the LARGER of 64 bits or the AXI data bus width, or the configuration error status bit will be set. Settings: AXI byte address where the first transfer will begin. If the associated burst_type bit in DMA_CHAN_CTRL is set to "1", all AXI transfers will be made to this address. When the DMA is programmed for a DI storage operation, this register must be programmed to point to the DMEM byte address of the first AXI address of the DI AXI address table. This address must be aligned to a 32-bit boundary.

14.2.28 AXI Byte Count register (DMA_AXI_BYTE_CNT)

Offset

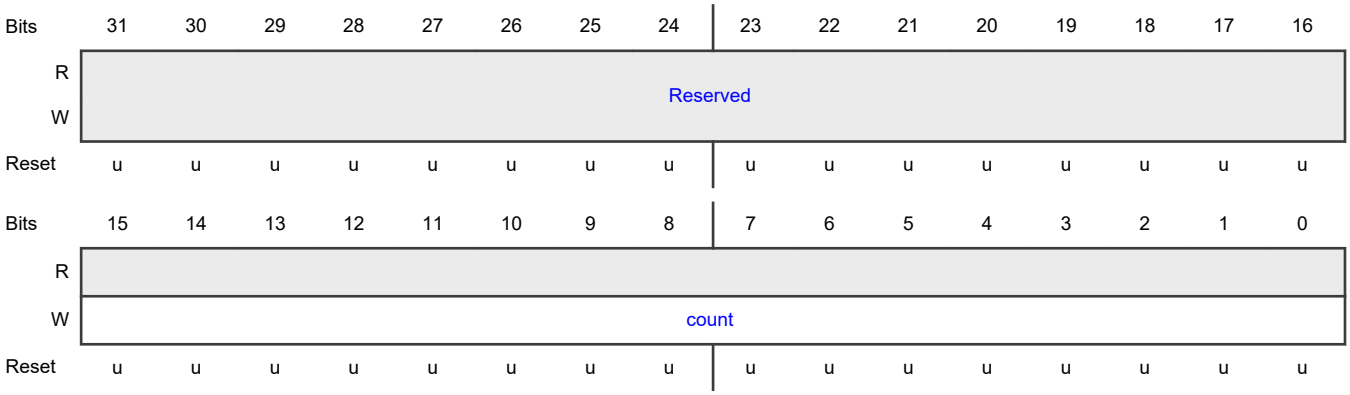
Register	Offset
DMA_AXI_BYTE_CNT	B8h

Function

- The DMA_AXI_BYTE_CNT register should be written before writing to the DMA_XFR_CTRL register.

- This register is not affected by reset and cannot be read (reads return 0s).

Diagram



Fields

Field	Function
31-16 —	- Reserved
15-0 count	count The number of bytes that get transferred. Maximum transfer size is 65536 bytes. • Note that when transferring data TO DMEM, the LSB of this register must be 0. Otherwise, the configuration error status bit will be set if an attempt is made to set the associated channel enable bit, and the channel enable bit will not be set. This is because memory write resolution is limited to minimum field sizes of 16 bits. • A configuration error will result when transferring to PRAM unless the total number of bytes transferred is an integer multiple of 8. This is required because each PRAM word is comprised of 8 bytes. • A configuration error will result when transferring to IPPU PRAM unless the total number of bytes transferred is an integer multiple of 4 bytes. Settings: 0=65536 bytes, 1=1 byte, 2=2 bytes, and so on. When the DMA is programmed for a DI storage operation, this register must be programmed to the number of data values to be stored.

14.2.29 DMA Transfer Control register (DMA_XFR_CTRL)

Offset

Register	Offset
DMA_XFR_CTRL	BCh

Function

The DMA_XFR_CTRL register must not be written until after all configuration information has first been written into the DMA_DMEM_PRAM_ADDR, DMA_AXI_ADDRESS and DMA_AXI_BYTE_CNT registers. Upon writing the DMA_XFR_CTRL register, all the command attributes from all four of these registers will be captured into the associated channel's FIFO, and the channel will be activated. Even writes from the VCPU core with no register bit mask bits set are considered writes and will activate the channel, so this register must be updated with a single write, not a series of masked writes.

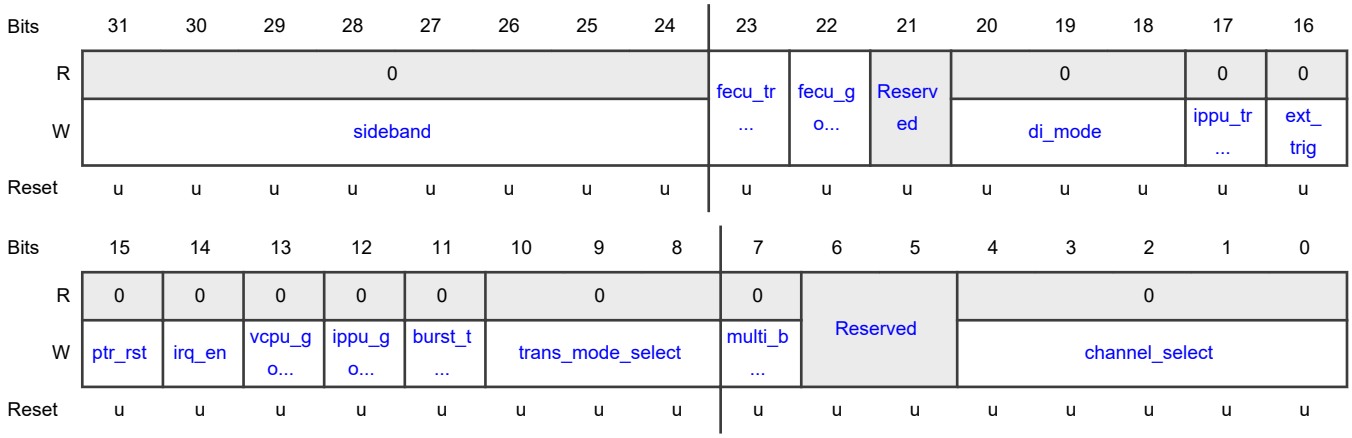
Since VCPU and a host processor can both use the DMA, there are separate sets of registers for each of them. Both sets of registers are located at the same addresses; the DMA detects whether writes come from VCPU or from the host, and updates the appropriate registers.

All bits of the registers should be written since they are not initialized by default.

Channel arbitration and priority:

- The DMA has two data movement engines, one for AXI writes and one for AXI reads. Thus, AXI read and writes can and will occur simultaneously. Each engine has its own arbitration unit.
- Each channel has a 2-entry FIFO. Commands in the FIFO will be executed in sequence (that is, the 1st command must be fully completed before the 2nd command can start).
- Channel 0 is a priority channel. If it is enabled and ready (external trigger satisfied if selected) its commands will be completed before any other channel is allowed to move any data.
- Channels other than channel 0 use round robin arbitration, arbitrating at the completion of each AXI burst. Therefore, if channels 2 and 4 are both writing to AXI, they will alternate doing one AXI burst each until one is finished. At that point the unfinished one will win every arbitration since there will be no other competition. If channels 1 and 3 were both reading from AXI (and enabled at the same time as 2 and 4), they would also alternate bursts, but their activity would be unaffected by channels 2 and 4 since those are processed by the other engine.
- If multiple channels are configured to read from AXI and channel 0 is configured as one of these channels, the usual round robin arbitration is disrupted and the next 2 highest priority channels will always win arbitration on the remaining read engines. As any channel completes, the next highest channel will become one of the always winning channels. When channel 0 completes the usual round robin arbitration will resume beginning with the channels currently running.
- The multi-burst feature affects only channels doing AXI reads. Observe and consider all the warnings accompanying the bit description. The round robin arbitration "fairness" can be distorted by the use of this feature, since it allows 4 bursts per arbitration win, compared to only 1 burst per arbitration win for channels not using multi-burst.
- The following characteristics apply to channels using an external trigger:
    1. They will only arbitrate when the external trigger input is asserted (high).
    2. They will only arbitrate when the associated ptr_rst_req/ack signals are both negated (low).
    3. If the transfers are AXI writes, the BRESP for each burst of a channel must be recieved before that channel will be allowed to be considered by the arbiter again.

Diagram



Fields

Field	Function
31-24 sideband	sideband Used to control the AXI address phase signals arsideband and awsideband, for AXI reads and writes respectively. Typical use of these bits would be for extending AXI address beyond 32 bits, or adding special security extensions. The bitfield can be anywhere from 1 to 8 bits, with the size depending on the value of parameter AXI_SIDEBAND_WIDTH, which is visible in VSPA_PARAM_0
23 fecu_trig	FECU trigger Used to control the DMA's sensitivity to FECU to DMA trigger events. If enabled, the initial DMA channel arbitration will occur only after the FECU asserts a DMA trigger event. The trigger event from FECU will only be asserted if the FECU_CONTROL register bit 8 (dma_go_enable) was set when the FECU command was placed into the FIFO. Note that the FECU to DMA trigger event is produced only for 1 cycle, when the FECU completes the FIFO command. If the FECU previously completed a command and asserted the trigger, that will not enable the DMA. The DMA waits for a trigger event that occurs after the DMA command is programmed. 0b - Ignore FECU to DMA trigger events 1b - Wait for a single (future) FECU DMA trigger event before enabling all transfers scheduled for this channel
22 fecu_go_en	FECU go enable Used to control assertion of a DMA->FECU trigger event, which can cause FECU to begin processing a scheduled command. If enabled, the DMA -> FECU event will assert immediately following the completion of the DMA command. This trigger event will only be monitored by FECU if the FECU_CONTROL register bits [1:0] (start_type) were set to 01 (wait for an external DMA trigger) when the FECU command was placed into the FIFO. Note that the DMA to FECU trigger event is asserted only for 1 cycle, after the DMA completes the FIFO command. 0b - Do not assert a DMA to FECU trigger event on completion of the DMA command. 1b - Assert a DMA to FECU trigger event on completion of the DMA command.

Table continues on the next page...

Table continued from the previous page...

Field	Function
21 —	- Reserved
20-18 di_mode	Deinterleaving mode The reserved modes listed below should be avoided if the trans_mode_select field is programmed to 0b100 (deinterleave data to AXI). Other configurations of trans_mode_select allow the DMA to ignore this field, so it can be safely set to any value in those cases. 000b - reserved 001b - 8-bit deinterleave 010b - 4-bit deinterleave, AXI address MSB=0 011b - 4-bit deinterleave, AXI address MSB=1 100b - 16-bit deinterleave 101b - 32-bit deinterleave 110b - reserved 111b - reserved
17 ippu_trig	IPPU trigger Used to control the DMA's sensitivity to IPPU done events. If enabled, initial channel arbitration will not be authorized until the IPPU signals a completion (done) event. Note that a done event is produced only for 1 cycle when the IPPU conpletes a task and enters the done state. If the IPPU is already in the done state when the DMA channel command is activated, the IPPU will need to start again and signal done again to satisfy this trigger. 0b - Ignore IPPU done events 1b - Wait for a single IPPU done event before enabling all transfers scheduled for this channel
16 ext_trig	External trigger Used to control the DMA's sensitivity to the dma_ext_trig input of the VSPA platform. If enabled, channel arbitration will not take place unless dma_ext_trig is high. **NOTE** This feature cannot be enabled in conjunction with a DI storage operation. Attempts to do so will result in a configuration error and the command will be ignored. 0b - Ignore dma_ext_trig input 1b - Wait for dma_ext_trig input before beginning each AXI burst
15 ptr_rst	Pointer reset Used for control of external FIFOs containing pointers. When enabled, the DMA will assert ptr_rst_req at the end of the channel's transfers to reset the FIFO's pointers. A ptr_rst_ack should be returned by the external device. If bit 16, ext_trig, is also set to 1, the channel will be disabled from arbitration until the handshake is completed and the ptr_rst_req and ptr_rst_ack are both low.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Note that if bit 16, ext_trig, is cleared, the channel will participate in arbitration upon transfer completion ignoring the state of ptr_rst_req and ptr_rst_ack as well as the state of the dma_ext_trig input. 0b - Ignored 1b - Upon completion of the scheduled transfers, assert ptr_rst_req
14 irq_en	IRQ enable Used to generate an IRQ to the host upon completion of the scheduled transfers. 0b - Ignored 1b - Generate IRQ upon completion of scheduled transfers
13 vcpu_go_en	VCPU Go enable Used to generate a VCPU GO upon completion of the scheduled transfers. This will set the associated dma_go status flag in the DMA_GO_STAT register. 0b - Ignored 1b - Generate VCPU GO upon completion of scheduled transfers
12 ippu_go_en	IPPU Go enable Used to generate a IPPU GO upon completion of the scheduled transfers. 0b - Ignored 1b - Generate IPPU GO upon completion of scheduled transfers
11 burst_type	Burst type Used to select between transferring to a memory or to a FIFO. 0b - Incrementing burst type 1b - Fixed burst type (all bursts go to the same AXI address)
10-8 trans_mode_select	Transfer mode select Used to select characteristics of the desired DMA transfer, such as source and target memory and format conversion.

Table 98. Transfer Mode Select Encoding

Encoding	DMA from	DMA to	Notes
000	AXI	DMEM	No format conversion
001	AXI	DMEM	Format conversion from 16-bit 2's complement -> 16-bit sign magnitude
010	AXI	PRAM	Load VCPU PRAM
011	AXI	IPPU PRAM	Load IPPU PRAM

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Table 98. Transfer Mode Select Encoding (continued)

Encoding	DMA from	DMA to	Notes
100	DMEM	AXI	Store and deinterleave data to AXI
101	u	u	Reserved
110	DMEM	AXI	No format conversion
111	DMEM	AXI	Format conversion from 16-bit sign magnitude -> 16-bit 2's complement

NOTE

While selecting any transfer mode except 000, Bit reversal mode (br32) shall be kept disabled. If Bit reversal mode is also enabled, format conversion of data will not happen for that particular transfer.

7 multi_burst	Multi-read burst enable Used to enable AXI read engine feature that produces up to 4 outstanding AXI read transactions per arbitration win for the channel. The state of this bit is ignored if the channel is configured to write to the AXI. Warnings: This feature cannot be enabled in conjunction with a DI storage operation. Attempts to do so will result in a configuration error and the command will be ignored. This feature is intended only for use with "normal" memory, not for FIFOs, so it should not be used in combination with burst_type=1 or ext_trig=1. It should also not be used for transfers targeting the VCPU or IPPU PMEM. It is only intended for AXI-> DMEM or IPPU DMEM transfers. When enabled, up to 4 outstanding AXI read address transactions will be issued in rapid succession. Some AXI interconnect or memory systems may not support this functionality. Also, other DMA channels performing AXI reads will experience a reduction in relative bus bandwidth since the multi_burst channels are allowed 4 bursts while the others are allowed only 1. 0b - Do only 1 AXI read burst per arbitration win 1b - Perform up to 4 AXI read bursts per arbitration win
6-5 —	- Reserved
4-0 channel_select	Channel select Selects the DMA channel to be used to execute the command. Note that channel 0 has arbitration priority over all other channels. If channel 0 is enabled and ready to transfer, no transfers for any other channels will be performed until all channel 0 transfers have been completed. 00000b - channel 0

Table continues on the next page...

Table continued from the previous page...

Field	Function
	00001b - channel 1 11111b - channel 31

14.2.30 DMA Status/Abort Control (DMA_STAT_ABORT)

Offset

Register	Offset
DMA_STAT_ABORT	C0h

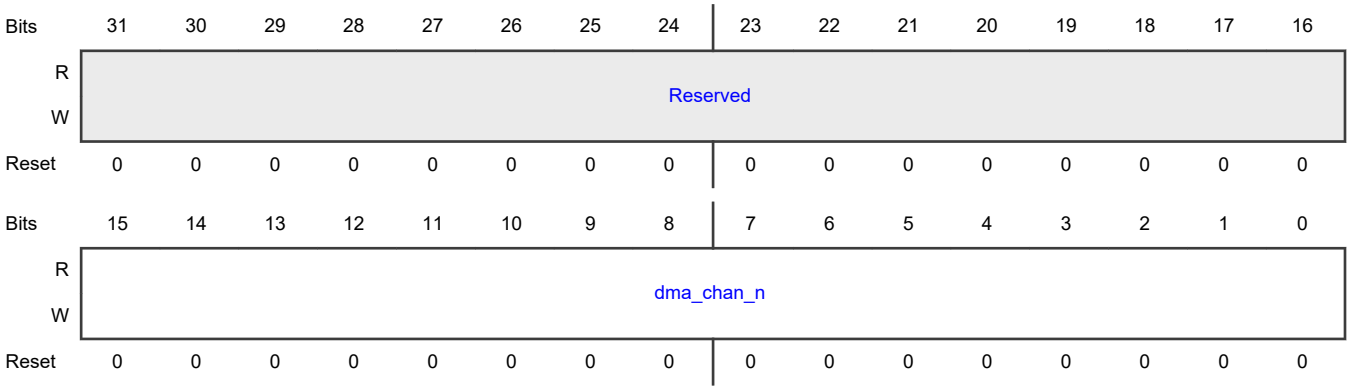
Function

NOTE

Pay close attention to the register bit field descriptions following the register figure because a field designated with read-write access in the register figure may have additional non-standard behavior described in its corresponding register bit field description.

- The DMA_STAT_ABORT register bits, when read, indicate the status of all the DMA channels, that is, whether there is current or pending activity.
- When this register is written with 1s, the associated channels are aborted or descheduled, and all pending FIFO commands are cleared.
- When a channel is aborted, the DMA_STAT_ABORT status bit will be cleared immediately, regardless of the fact that all activity may not yet have ceased. To be certain that all activity on a channel has ceased following an abort request, the corresponding bit in the DMA_XRUN_STAT must be read as a "0". No new FIFO commands should be initiated to a channel that has not completed a requested abort. Note that channels that are aborted will not set the corresponding DMA_COMP_STAT bits; the abort is treated as if the channel did not finish, so the following features will not be activated due to an abort:
  - the DMA will not trigger an IRQ for the aborted channel
  - the DMA will not trigger the IPPU even if the ippu_go_en bit was set
  - the DMA will not trigger the VCPU to go, even if the vcpu_go_en bit was set
  - the DMA will not trigger a ptr_rst_req, even if the ptr_rst bit was set
- Note that following a VCPU write to the DMA_XFR_CTRL register, these status bits do not update until one cycle after the write occurs. Therefore, an instruction to read this register should not immediately follow the instruction that writes the DMA_XFR_CTRL register.
- Channels that have been configured for DI transfers cannot be aborted. Abort commands for those channels will be ignored.

Diagram



Fields

Field	Function
31-16 —	- Reserved
15-0 dma_chan_n	dma_chan_n **NOTE** Access to this field is non-standard and is described in detail below. Read - DMA Channel n status for n=15 to 0 Each nth bit specifies if the nth DMA channel is enabled. Write 1 - Disable/abort channel n and clear its FIFO. Read: 0 Channel n Disabled. 1 Channel n Enabled. Write: 0 Do Nothing. 1 Disable/abort channel n, clear its FIFO entries.

14.2.31 DMA IRQ Status (DMA_IRQ_STAT)

Offset

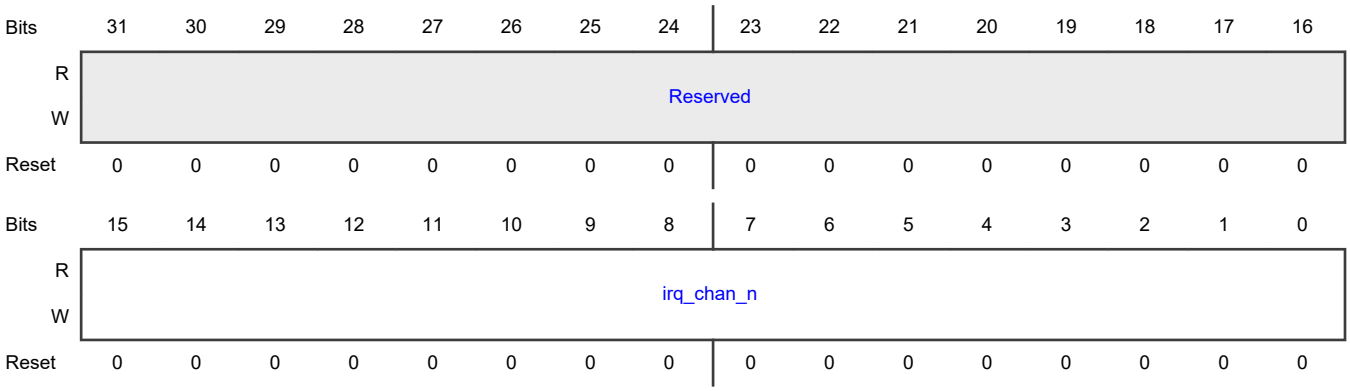
Register	Offset
DMA_IRQ_STAT	C4h

Function

NOTE

Pay close attention to the register bit field descriptions following the register figure because a field designated with read-write access in the register figure may have additional non-standard behavior described in its corresponding register bit field description.

Diagram



Fields

Field	Function
31-16 —	- Reserved
15-0 irq_chan_n	irq_chan_n **NOTE** Access to this field is non-standard and is described in detail below. IRQ channel n for n=15 to 0 Each nth bit specifies whether the nth DMA channel asserted an interrupt request upon completion of all of its scheduled transfers. These bits are write 1 to clear, and they can only be cleared by the external IPbus master, not by VCPU. 0000000000000000b - Channel n interrupt not requested 0000000000000001b - Channel n interrupt requested

14.2.32 DMA Complete Status (DMA_COMP_STAT)

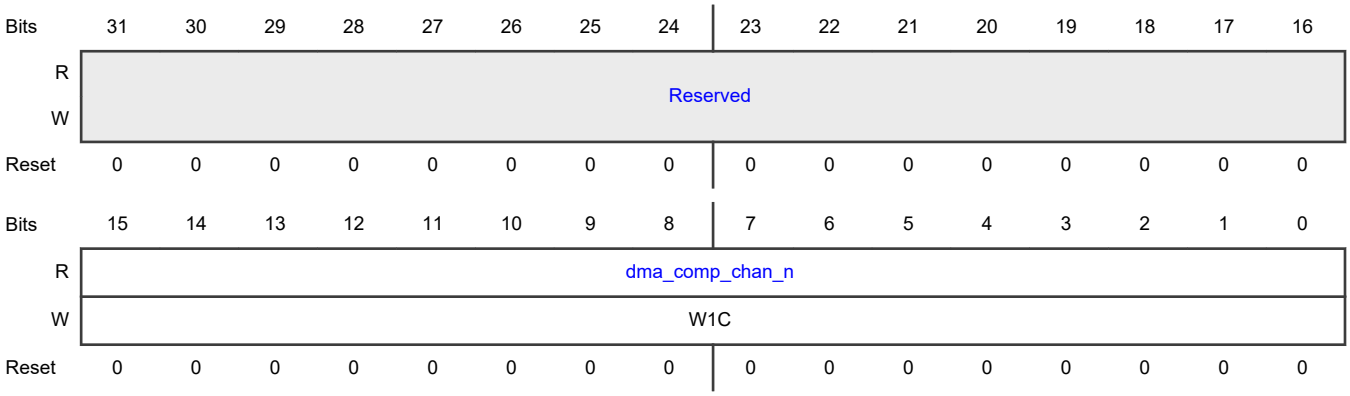
Offset

Register	Offset
DMA_COMP_STAT	C8h

Function

- The DMA_COMP_STAT register bits are cleared by writing ones to their respective bit positions.

Diagram



Fields

Field	Function
31-16 —	- Reserved
15-0 dma_comp_chan_n	dma_comp_chan_n DMA completed channel n for n=15 to 0 Each nth bit specifies whether the nth DMA channel has completed all of its scheduled transfers. 0000000000000000b - Channel n transfers have not completed. 0000000000000001b - Channel n transfers have completed.

14.2.33 DMA Transfer Error Status (DMA_XFRERR_STAT)

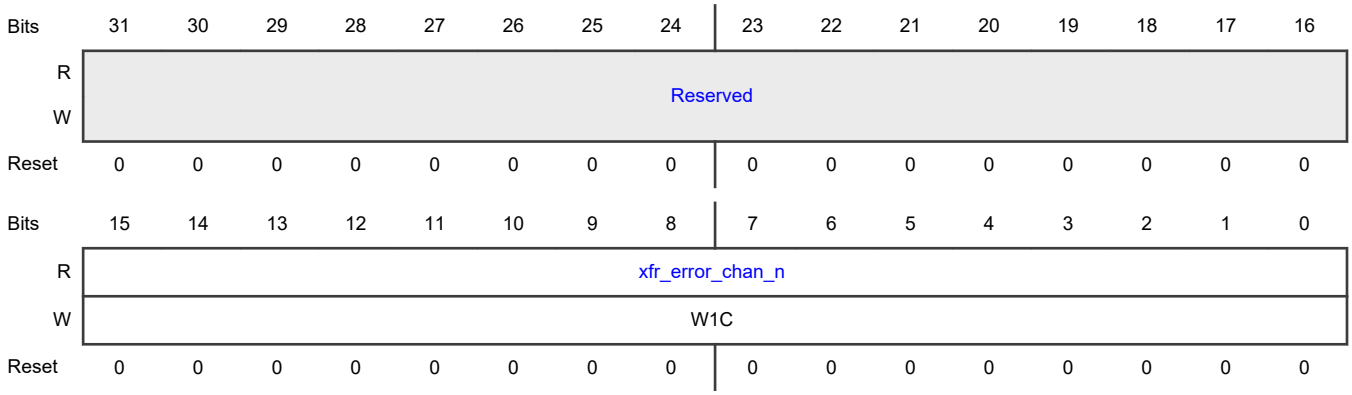
Offset

Register	Offset
DMA_XFRERR_STAT	CCh

Function

- The DMA_XFRERR_STAT register bits are cleared by writing ones to their respective bit positions.
- These register bits will set as soon as an error is detected, even if all of the transfers have not yet been completed.
- Detection of an AXI transfer error does not abort any pending transfers. All transfers will be completed whether AXI transaction errors are detected.

Diagram



Fields

Field	Function
31-16 —	- Reserved
15-0 xfr_error_chan_n	dma_comp_chan_n DMA transfer error channel n for n=15 to 0 Each nth bit indicates whether the nth DMA channel has detected AXI transaction errors during a programmed transfer. 0000000000000000b - Channel n AXI transfers have no errors, or have not completed. 0000000000000001b - Channel n AXI transfers have errors.

14.2.34 DMA Configuration Error Status (DMA_CFGERR_STAT)

Offset

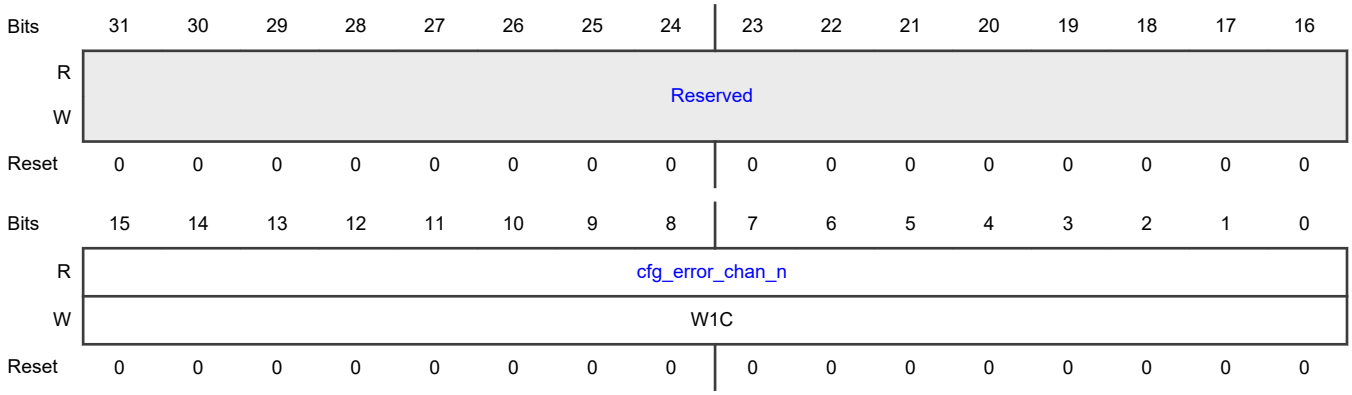
Register	Offset
DMA_CFGERR_STAT	D0h

Function

This register is used to indicate when software has attempted to activate a DMA channel that has been set up in an improper configuration.

- Note that following a VCPU write to the DMA_XFR_CTRL register, these status bits do not update until one cycle after the write occurs. Therefore, an instruction to read this register should not immediately follow the instruction that writes the DMA_XFR_CTRL register.
- If a configuration error occurs, the associated channel will not act on the invalid command, so no transfers will be scheduled.
- The DMA_CFGERR_STAT register bits are cleared by writing ones to their respective bit positions.

Diagram



Fields

Field	Function
31-16 —	- Reserved
15-0 cfg_error_chan_n	cfg_error_chan_n DMA configuration error for channel n for n=15 to 0 Each n bit indicates whether the nth DMA channel has detected an attempt to configure the associated channel while an invalid configuration was present in its associated control registers. 0000000000000000b - No attempt has been made to activate channel n with an invalid configuration. 0000000000000001b - An attempt was made to activate channel n with an invalid configuration specified by the associated channel configuration registers.

14.2.35 DMA Transfer Running Status (DMA_XRUN_STAT)

Offset

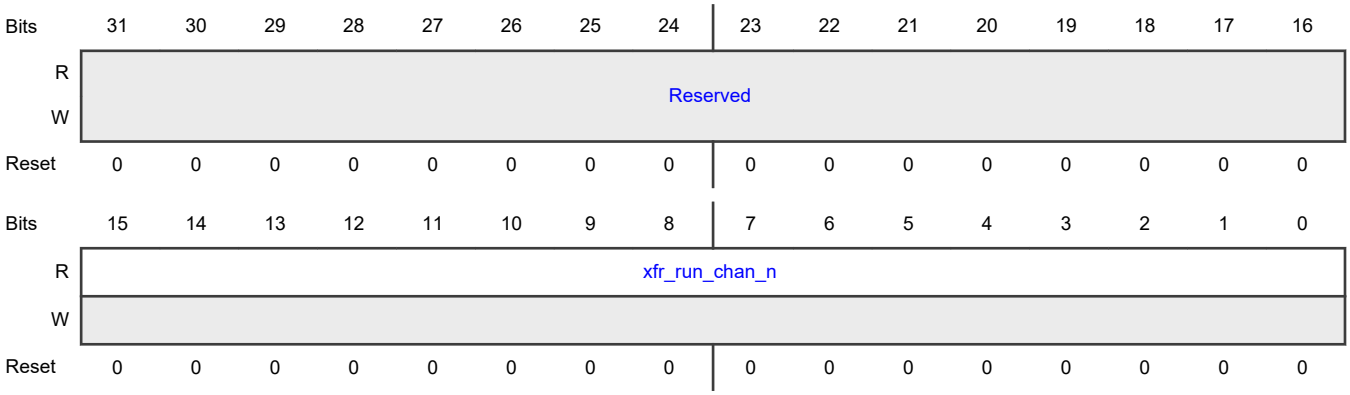
Register	Offset
DMA_XRUN_STAT	D4h

Function

- The DMA_XRUN_STAT register bits are read-only status bits. They are self clearing.
- The associated register bit will set as soon as a channel has begun transfers. When the channel has completed all transfers, the associated register bit will be cleared.

.

Diagram



Fields

Field	Function
31-16 —	- Reserved
15-0 xfr_run_chan_n	xfr_run_chan_n Transfer running for channel n for n=15 to 0 Each nth bit indicates whether the nth DMA channel is actively doing transfers. 0000000000000000b - No transfer activity is presently happening for the associated channel. 0000000000000001b - Transfer activity is presently happening for the associated channel.

14.2.36 DMA Go Status (DMA_GO_STAT)

Offset

Register	Offset
DMA_GO_STAT	D8h

Function

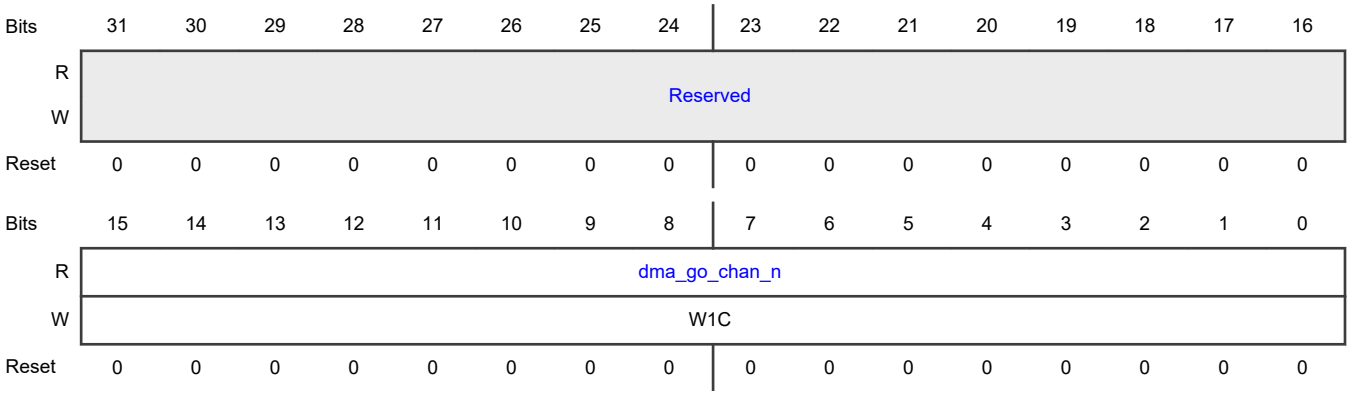
- The DMA_GO_STAT register bits are write 1 to clear status bits.

CAUTION

To prevent VSPA from missing a go event caused by these bits, they should NOT be cleared by the external IPbus master

- The associated status bit will set when the channel completes all specified transactions only if the channel's go_en bit was set in the DMA_XFR_CTRL register when the transfer was initially programmed.
- If the status bit initiating a VCPU GO is not cleared before VCPU executes the DONE instruction, VCPU will GO again.

Diagram



Fields

Field	Function
31-16 —	- Reserved
15-0 dma_go_chan_n	dma_go_chan_n **NOTE** Access to this field is non-standard and is described in detail below. VCPU go generated by channel n for n=15 to 0 Each nth bit indicates whether the nth DMA channel initiated a GO. 0 read - Channel did not initiate a GO 1 read - Channel initiated a GO 0 write - Ignored 1 write - Clear the bit (if set)

14.2.37 DMA FIFO Availability Status (DMA_FIFO_STAT)

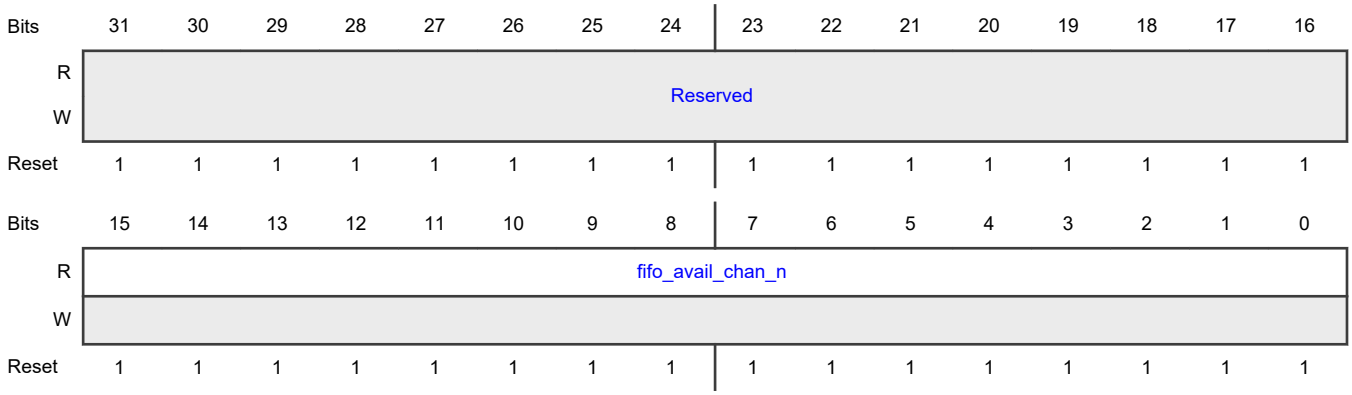
Offset

Register	Offset
DMA_FIFO_STAT	DCh

Function

- The DMA_FIFO_STAT register bits are read only status bits.
- The status register bit will set as soon as an empty FIFO entry is available for a given channel.
- Note that following a VCPU write to the DMA_XFR_CTRL register, these status bits do not update until one cycle after the write occurs. Therefore, an instruction to read this register should not immediately follow the instruction that writes the DMA_XFR_CTRL register.

Diagram



Fields

Field	Function
31-16 —	- Reserved
15-0 fifo_avail_chan_n	fifo_avail_chan_n Empty FIFO entry available for channel n for n=15 to 0 Each nth bit indicates whether the nth DMA channel has an empty FIFO entry, and is thus capable of accepting a new command. 0000000000000000b - No FIFO entry is available for the associated channel. 0000000000000001b - One or more FIFO entries are available for the associated channel.

14.2.38 Load Register File Control register (Slow read register) (LD_RF_CONTROL)

Offset

Register	Offset
LD_RF_CONTROL	100h

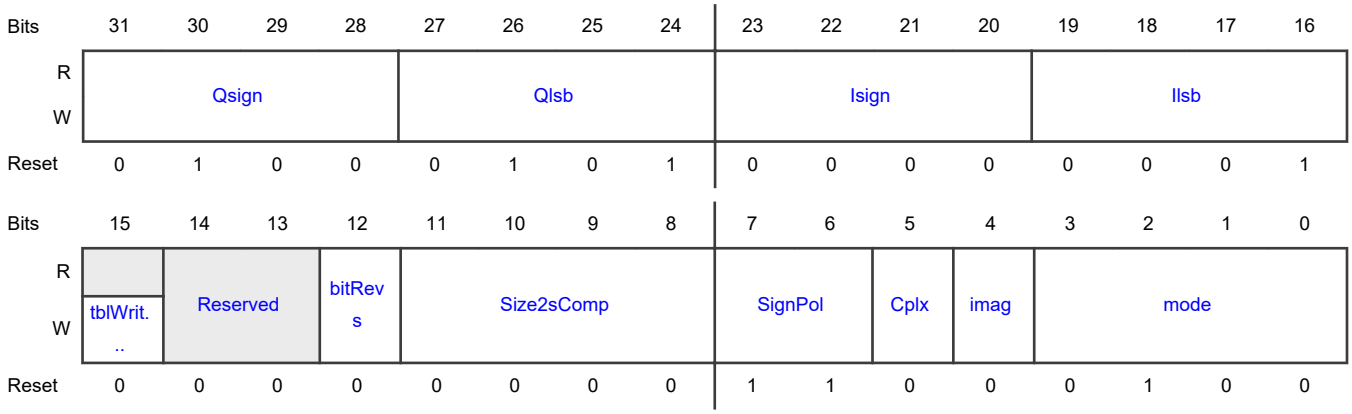
Function

This register is used to select the table mode used by the ld.qam instruction when used in QAM mode. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VSPA Instruction Set Manual.

Note that the reset state of this register corresponds to the correct setting for real mode 16QAM.

This register is a Slow read register.

Diagram



Fields

Field	Function
31-28 Qsign	Qsign Sign bit of imaginary component for QAM 256 and QAM 1024
27-24 Qlsb	Qlsb Least significant bit of imaginary component for QAM 256 and QAM 1024
23-20 Isign	Isign Sign bit of real component for QAM 256 and QAM 1024
19-16 Ilsb	Ilsb Least significant bit of real component for QAM 256 and QAM 1024
15 tblWriteEn_b	QAM Table Write Enable Writing a 0 to this bit location (data=0 and wem=1) will allow the automatic update of the LD_RF_TB_REAL_0 - LD_RF_TB_IMAG_7 registers as described in the mode field (bits 3:0) of this register. Any other write, including wem=0, will block the automatic update of the LD_RF_TB_xxx registers. This bit always reads 0.
14-13 —	- Reserved
12 bitRevs	bitRevs Controls sample reversal of the ld.2scomp instruction. This bit has no meaning for ld.qam 0b - Sample will not be bit-reversed before conversion from 2's complement value to half fixed value. 1b - Sample will be bit-reversed before conversion from 2's complement value to half fixed value.
11-8 Size2sComp	Size of input for 2s Complement Conversion

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Select the size of the value to be converted from 2s complement to half fixed. These bits have no meaning for ld.qam. 0000b - Convert 16 bit 2's complement data 1000b - Convert 8 bit 2's complement data 1010b - Convert 10 bit 2's complement data 1100b - Convert 12 bit 2's complement data
7-6 SignPol	SignPol Polarity of sign bits Bit 7 indicates the polarity of the sign bit of the imaginary (Q) component for QAM 256 and QAM 1024 Bit 6 indicates the polarity of the sign bit of the real (I) component for QAM 256 and QAM 1024 0 - non-inverted sign 1 - inverted sign
5 Cplx	Complex Generate complex values for the QAM data
4 imag	imag Imaginary Load Select Used to control which 16 bit half-word of a 32 bit modulation symbol is written to the register file after decoding when VCPU is operating in COMPLEX mode. This bit has no effect in REAL mode. 0b - real 1b - imag
3-0 mode	mode Modulation Mode Select Selects between coefficient table expansion modes corresponding to the modulation order, and therefore the number of bits from a 16 bit DMEM half-word which are decoded into a modulation symbol. For each possible mode, a specific number of coefficients are used when the ld.qam Rx, QAM instruction is executed. This allows mappings for WCDMA BPSK and 4PAM as well as LTE and 16QAM and 64QAM. Coefficients for all modes except 64QAM are stored as bit pairs in the LD_RF_REAL0 and LD_RF_IMAG0 registers. Coefficient bit pairs are LSB justified, so if only 2 pairs are used, they are bits [3:2] and [1:0]. Coefficients for 64QAM mode are stored as nibbles in the LD_RF_TB_REAL_0-7 and LD_RF_TB_IMAG_0-7 registers. Only bits 3:1 of each nibble are used to decode the modulation symbols. 1 = 2 coefficients (BPSK) 2 = 4 coefficients (4PAM) 4 = 16 coefficients (16Q/PAM) 6 = 64 coefficients (64Q/PAM) 8 = 256 coefficients (256Q/PAM)

Table continues on the next page...

Table continued from the previous page...

Field	Function
	10 = 1024 coefficients (1024Q/PAM) **NOTE** • Writing anything other than 6, 8 or 10 to the mode bits will automatically update the LD_RF_TB_REAL_0 and LD_RF_TB_IMAG_0 registers to 16QAM values. • Writing 6 to the mode bits will automatically update these registers to 64QAM values. • Writing 8 to the mode bits will automatically update these registers to 256QAM values. • Writing 10 to the mode bits will automatically update these registers to 1024QAM values. • Bit 15 of this register must also be written to a 0 to allow the above described automatic update of these registers. See the description of bit 15 for more details.

14.2.39 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_0)

Offset

Register	Offset
LD_RF_TB_REAL_0	104h

Function

This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is 2A4C_086Eh immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 25163407h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 4B3C_780Fh immediately after the mode change.

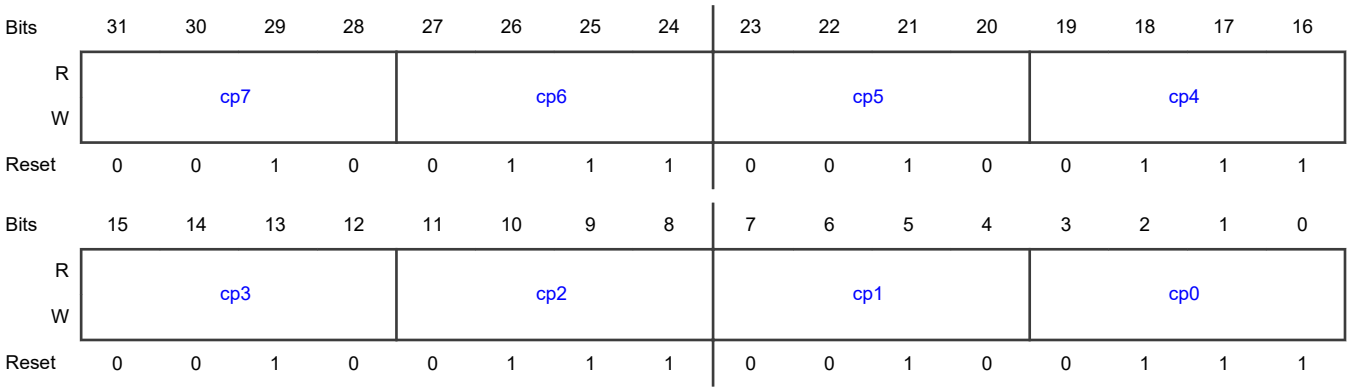
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0100	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-15 for mode=0100 X=0-7 for all other modes	00 - 1 01 - 3 10 - -1 11 - -3	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13

Table continues on the next page...

Register bit-fields	Mode==0100	Mode==0110	Mode==1000	Mode==1010
		0b111x - -7	0bx111 - 15	0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp7	cp7 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24	cp6

Table continues on the next page...

Table continued from the previous page...

Field	Function
cp6	Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp5	cp5 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp4	cp4 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp3	cp3 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp2	cp2 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp1	cp1 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp0	cp0 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.40 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_0)

Offset

Register	Offset
LD_RF_TB_IMAG_0	108h

Function

This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is EEEE_EEEEh immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 7777_7777h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 4B3C_780Fh immediately after the mode change.

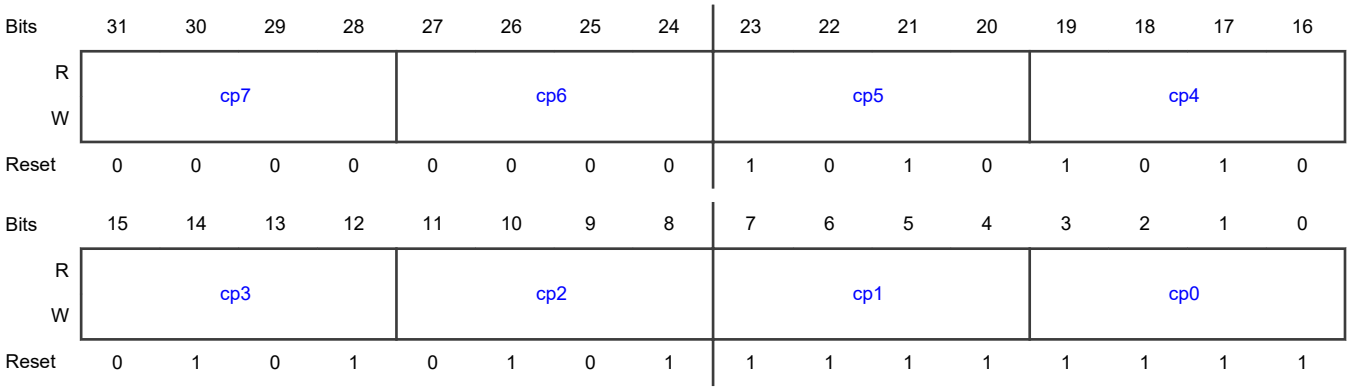
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0100	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-15 for mode=0100 X=0-7 for all other modes	00 - 1 01 - 3 10 - -1 11 - -3	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21

Table continues on the next page...

Register bit-fields	Mode==0100	Mode==0110	Mode==1000	Mode==1010
				0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp7	cp7 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp6	cp6 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp5	cp5 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp4	cp4 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp3	cp3 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	101xb - -3 110xb - -5 111xb - -7
11-8 cp2	cp2 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp1	cp1 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp0	cp0 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

Table continues on the next page...

Field	Function

14.2.41 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_1)

Offset

Register	Offset
LD_RF_TB_REAL_1	10Ch

Function

This register holds the real coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still 2A4C_086Eh immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 2516_3407h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 5A2D_691Eh immediately after the mode change.

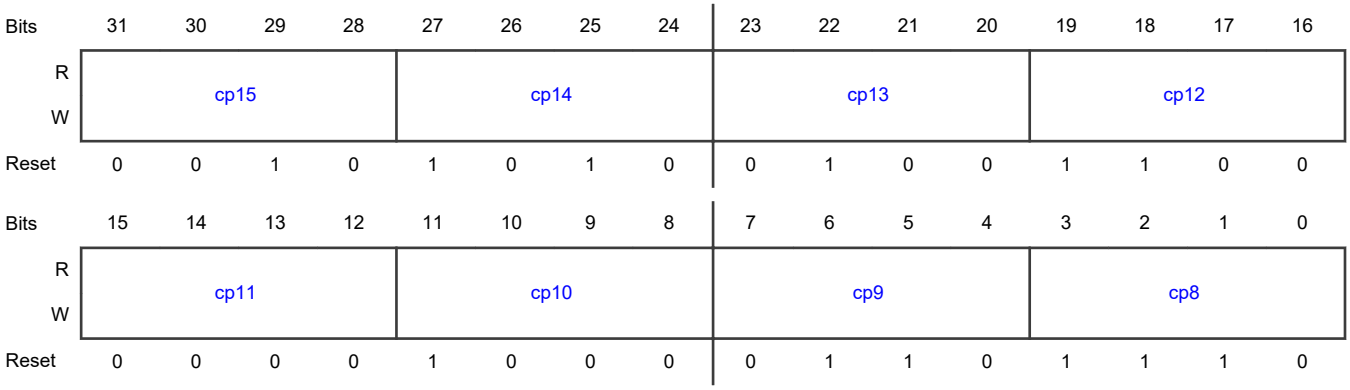
The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25

Table continues on the next page...

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
			0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp15	cp15 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp14	cp14 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	101xb - -3 110xb - -5 111xb - -7
23-20 cp13	cp13 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp12	cp12 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp11	cp11 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

Table continues on the next page...

Table continued from the previous page...

Field	Function
11-8 cp10	cp10 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp9	cp9 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp8	cp8 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.42 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_1)

Offset

Register	Offset
LD_RF_TB_IMAG_1	110h

Function

This register holds the imaginary coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

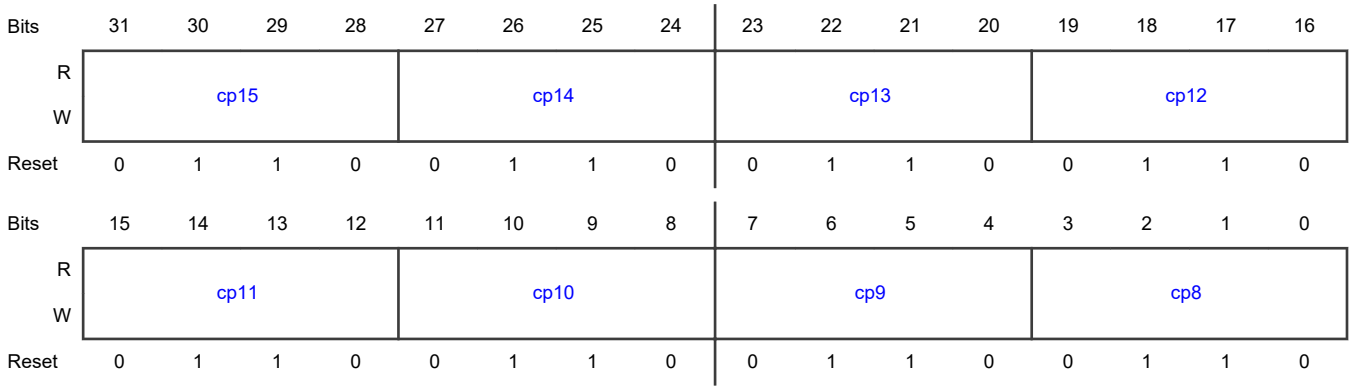
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still 6666_6666h immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 0000_0000h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 5A2D_691Eh immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp15	cp15 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp14	cp14 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp13	cp13 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp12	cp12 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp11	cp11 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp10	cp10 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp9	cp9 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp8	cp8 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.43 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_2)

Offset

Register	Offset
LD_RF_TB_REAL_2	114h

Function

This register holds the real coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

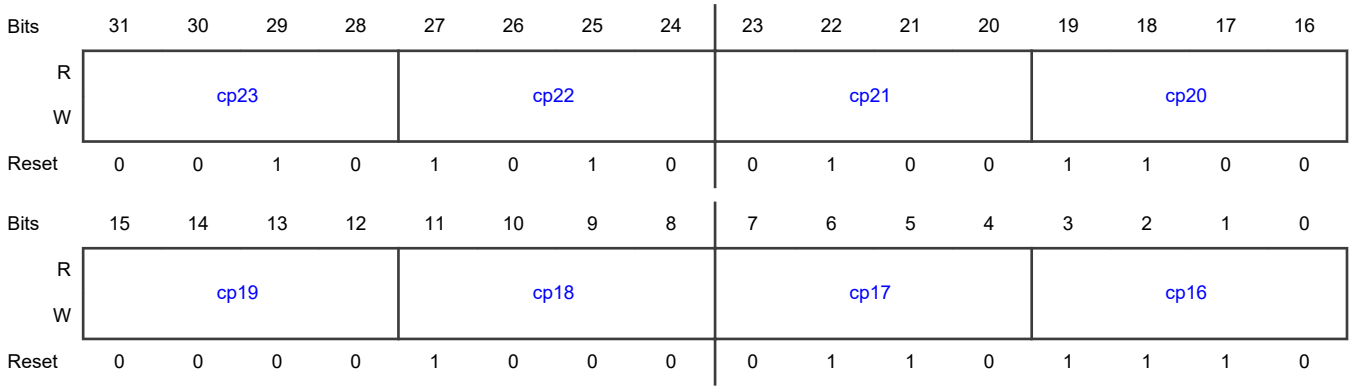
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still 2A4C_086Eh immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 2516_3407h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 2A4C_086Eh immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp23	cp23 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp22	cp22 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp21	cp21 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp20	cp20 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp19	cp19 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp18	cp18 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp17	cp17 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp16	cp16 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.44 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_2)

Offset

Register	Offset
LD_RF_TB_IMAG_2	118h

Function

This register holds the imaginary coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

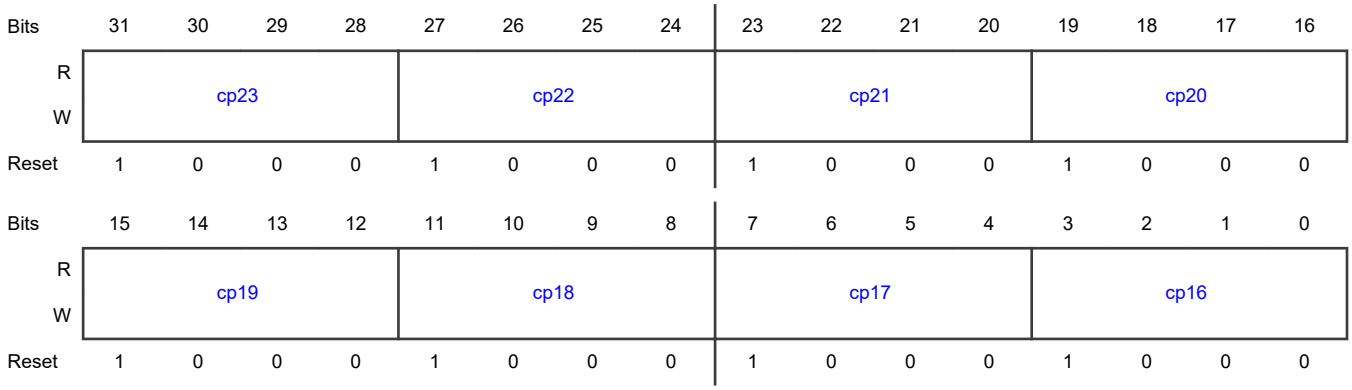
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still 8888_8888h immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 4444_4444h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 8888_8888h immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp23	cp23 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp22	cp22 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp21	cp21 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp20	cp20 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp19	cp19 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp18	cp18 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp17	cp17 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp16	cp16 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.45 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_3)

Offset

Register	Offset
LD_RF_TB_REAL_3	11Ch

Function

This register holds the real coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

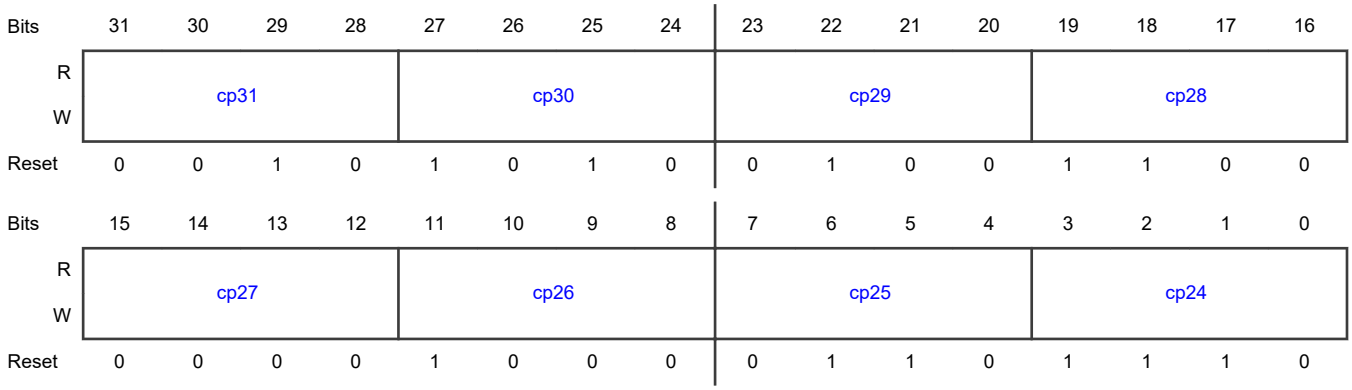
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still 2A4C_086Eh immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 2516_3407h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 2A4C_086Eh immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp31	cp31 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp30	cp30 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp29	cp29 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp28	cp28 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp27	cp27 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp26	cp26 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp25	cp25 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp24	cp24 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.46 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_3)

Offset

Register	Offset
LD_RF_TB_IMAG_3	120h

Function

This register holds the imaginary coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

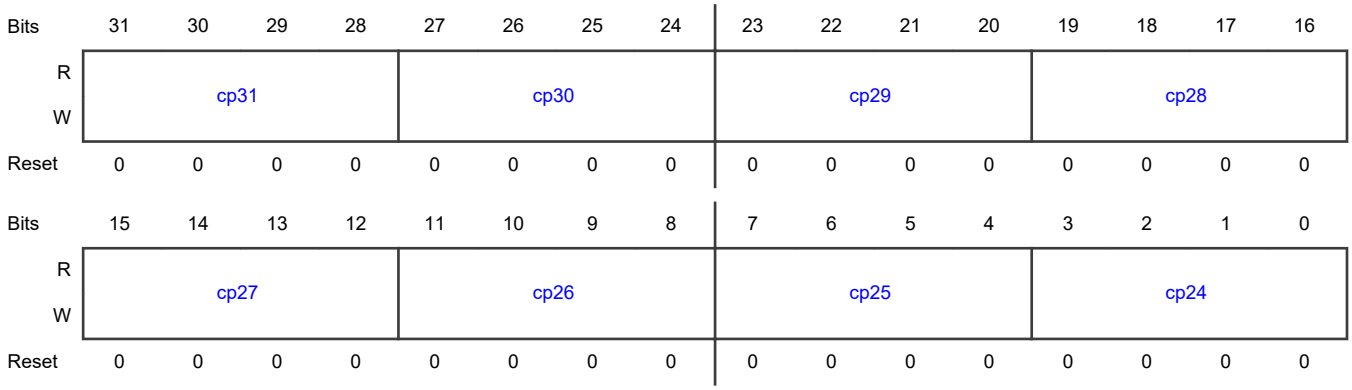
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still 0000_0000h immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 3333_3333h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 0000_0000h immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp31	cp31 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp30	cp30 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp29	cp29 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp28	cp28 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp27	cp27 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp26	cp26 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp25	cp25 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp24	cp24 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.47 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_4)

Offset

Register	Offset
LD_RF_TB_REAL_4	124h

Function

This register holds the real coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

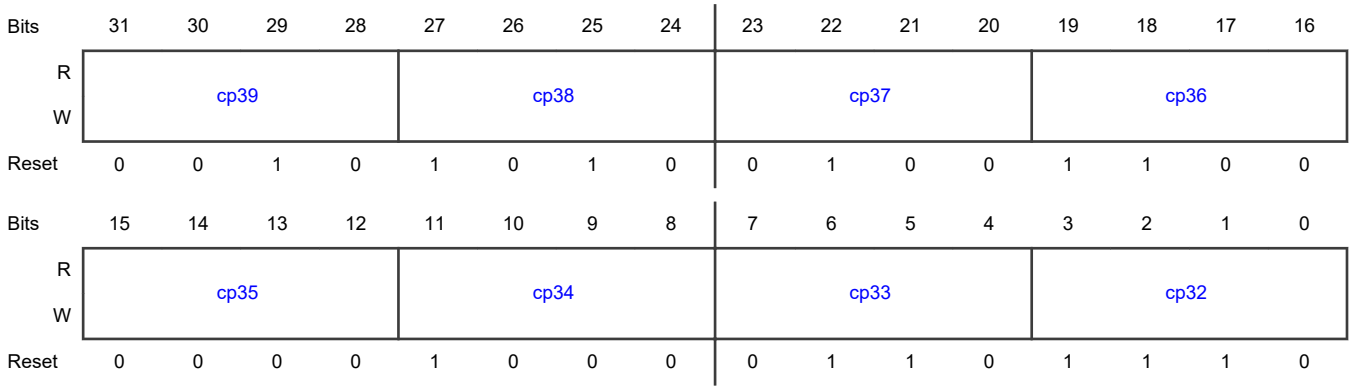
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still 2A4C_086Eh immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 2516_3407h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 2A4C_086Eh immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp39	cp39 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp38	cp38 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp37	cp37 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp36	cp36 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp35	cp35 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp34	cp34 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp33	cp33 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp32	cp32 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.48 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_4)

Offset

Register	Offset
LD_RF_TB_IMAG_4	128h

Function

This register holds the imaginary coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still CCCC_CCCCh immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 6666_6666h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is CCCC_CCCCh immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	cp39				cp38				cp37				cp36			
Reset	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	cp35				cp34				cp33				cp32			
Reset	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0

Fields

Field	Function
31-28 cp39	cp39 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp38	cp38 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp37	cp37 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp36	cp36 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp35	cp35 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp34	cp34 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp33	cp33 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp32	cp32 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.49 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_5)

Offset

Register	Offset
LD_RF_TB_REAL_5	12Ch

Function

This register holds the real coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

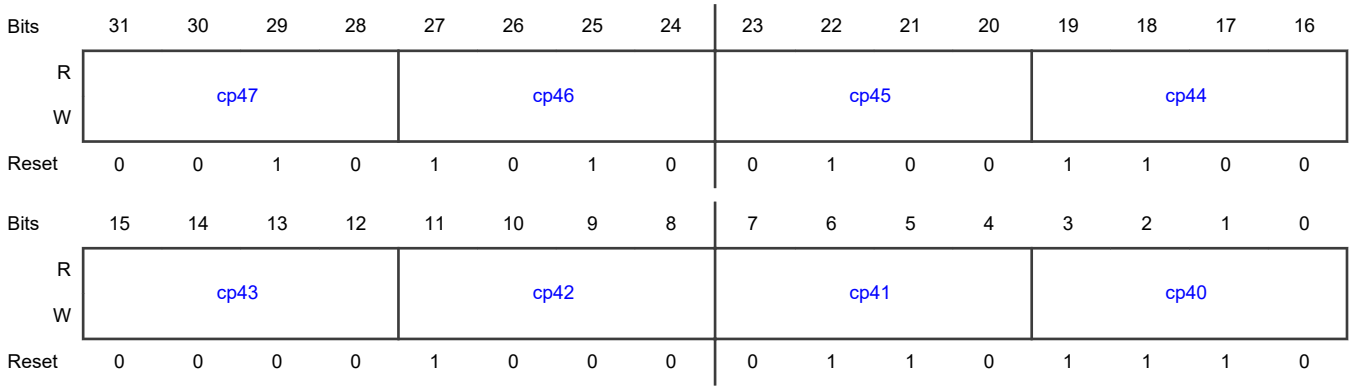
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still 2A4C_086Eh immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 2516_3407h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 2A4C_086Eh immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp47	cp47 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp46	cp46 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp45	cp45 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp44	cp44 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp43	cp43 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp42	cp42 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp41	cp41 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp40	cp40 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.50 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_5)

Offset

Register	Offset
LD_RF_TB_IMAG_5	130h

Function

This register holds the imaginary coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

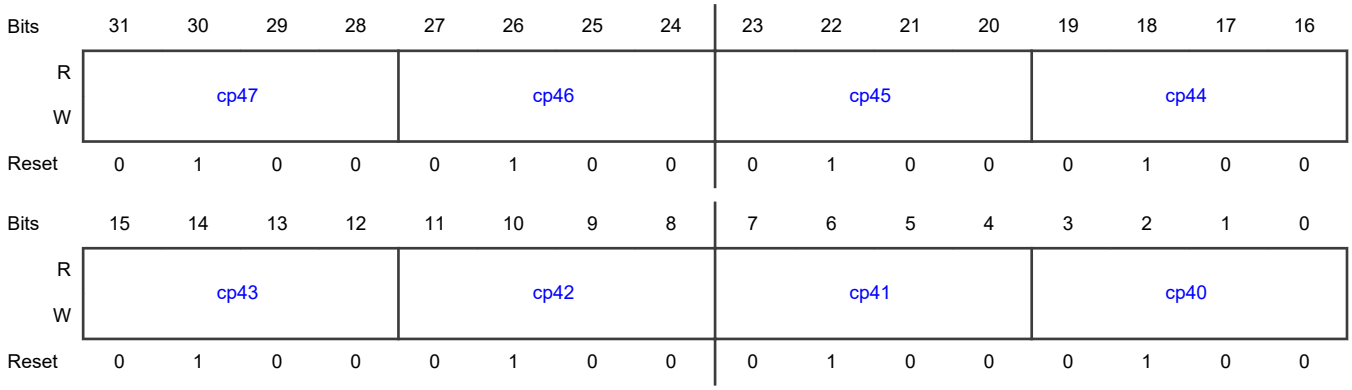
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still 4444_4444h immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 1111_1111h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 4444_4444h immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp47	cp47 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp46	cp46 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp45	cp45 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp44	cp44 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp43	cp43 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp42	cp42 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp41	cp41 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp40	cp40 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.51 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_6)

Offset

Register	Offset
LD_RF_TB_REAL_6	134h

Function

This register holds the real coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

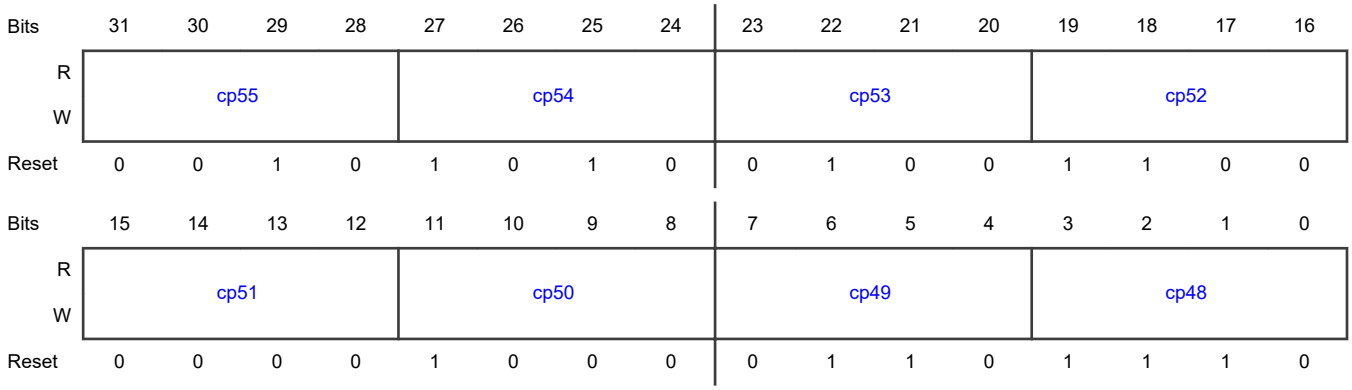
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still 2A4C_086Eh immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 2516_3407h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 2A4C_086Eh immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp55	cp55 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp54	cp54 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp53	cp53 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp52	cp52 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp51	cp51 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp50	cp50 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp49	cp49 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp48	cp48 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.52 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_6)

Offset

Register	Offset
LD_RF_TB_IMAG_6	138h

Function

This register holds the imaginary coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

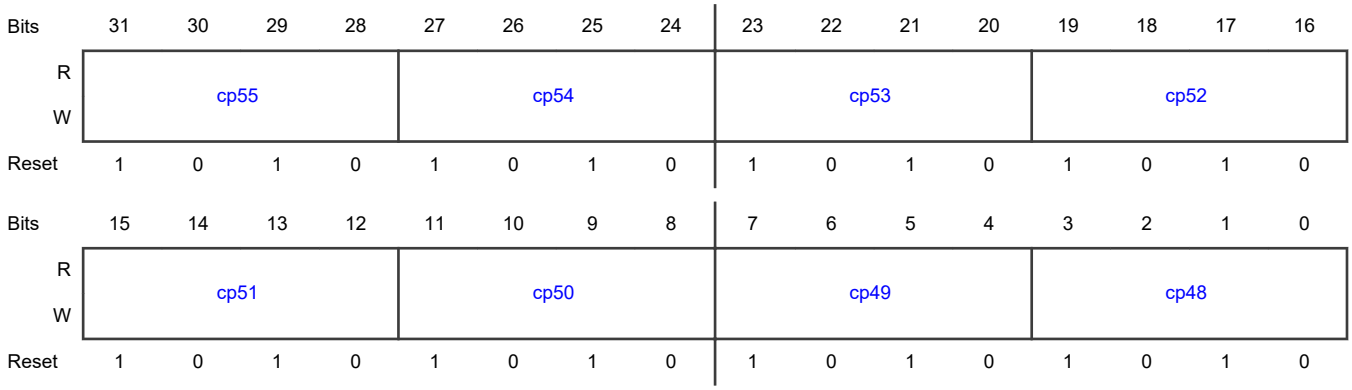
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still AAAA_AAAAh immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 5555_5555h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is AAAA_AAAAh immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp55	cp55 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp54	cp54 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp53	cp53 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp52	cp52 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp51	cp51 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp50	cp50 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp49	cp49 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp48	cp48 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.53 Load Register File Real Coefficient Table register (Slow read register) (LD_RF_TB_REAL_7)

Offset

Register	Offset
LD_RF_TB_REAL_7	13Ch

Function

This register holds the real coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

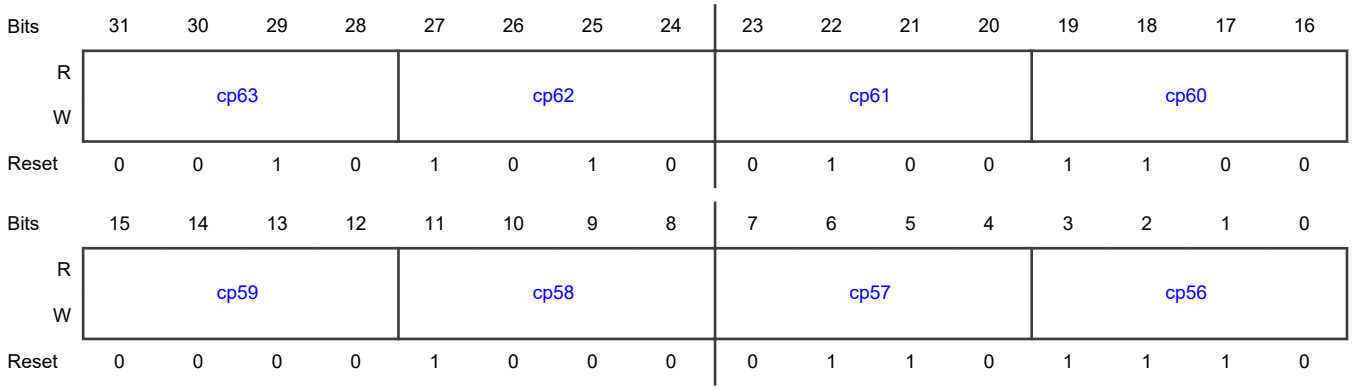
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100. When the value of mode is changed to mode==0110 the register read value is still 2A4C_086Eh immediately after the mode change, when the value of mode is changed to mode==1000 the register read value is 2516_3407h immediately after the mode change and when the value of mode is changed to mode==1010 the register read value is 2A4C_086Eh immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp63	cp63 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp62	cp62 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp61	cp61 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1

Table continues on the next page...

Table continued from the previous page...

Field	Function
	001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp60	cp60 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp59	cp59 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp58	cp58 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5

Table continues on the next page...

Table continued from the previous page...

Field	Function
	011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp57	cp57 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp56	cp56 Real coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.54 Load Register File Imaginary Coefficient Table register (Slow read register) (LD_RF_TB_IMAG_7)

Offset

Register	Offset
LD_RF_TB_IMAG_7	140h

Function

This register holds the imaginary coefficients used by the ld.qam Rx, QAM64 instruction. Up to 64 coefficients may be used, for example when working with 64QAM. The encodings of the coefficient bit groups map to coefficient values of 1, 3, 5, 7, -1, -3, -5, and -7. For more complete usage information, refer to the detailed description of the ld.qam instructions in the VCPU Programmer's Guide.

Note that the reset state of this register corresponds to the correct coefficients for real mode 64QAM.

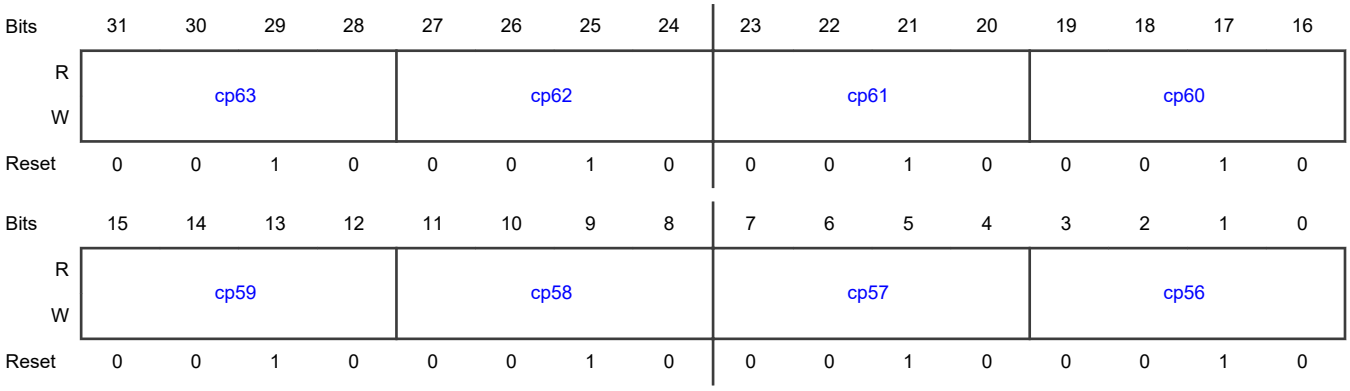
This register is a Slow read register.

The reset values shown in the register diagram below are true when the 'LD_RF_CONTROL' register bit-field mode<0100 as well as when the mode is changed to mode==0110, mode==1000, mode==1010, immediately after the mode change.

The mapping shown in the fields section below is true only when the mode==0110. See the table below for the mapping for other QAM modes.

Register bit-fields	Mode==0110	Mode==1000	Mode==1010
cpX: X=0-7	0b000x - 1 0b001x - 3 0b010x - 5 0b011x - 7 0b100x - -1 0b101x - -3 0b110x - -5 0b111x - -7	0bx000 - 1 0bx001 - 3 0bx010 - 5 0bx011 - 7 0bx100 - 9 0bx101 - 11 0bx110 - 13 0bx111 - 15	0b0000 - 1 0b0001 - 3 0b0010 - 5 0b0011 - 7 0b0100 - 9 0b0101 - 11 0b0110 - 13 0b0111 - 15 0b1000 - 17 0b1001 - 19 0b1010 - 21 0b1011 - 23 0b1100 - 25 0b1101 - 27 0b1110 - 29 0b1111 - 31

Diagram



Fields

Field	Function
31-28 cp63	cp63 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
27-24 cp62	cp62 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
23-20 cp61	cp61 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
19-16 cp60	cp60 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
15-12 cp59	cp59 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
11-8 cp58	cp58 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
7-4 cp57	cp57 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3

Table continues on the next page...

Table continued from the previous page...

Field	Function
	010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7
3-0 cp56	cp56 Imaginary coefficients. Only used if mode == 0110 in LD_RF_CONTROL register. 000xb - 1 001xb - 3 010xb - 5 011xb - 7 100xb - -1 101xb - -3 110xb - -5 111xb - -7

14.2.55 VCPU Go Address (VCPU_GO_ADDR)

Offset

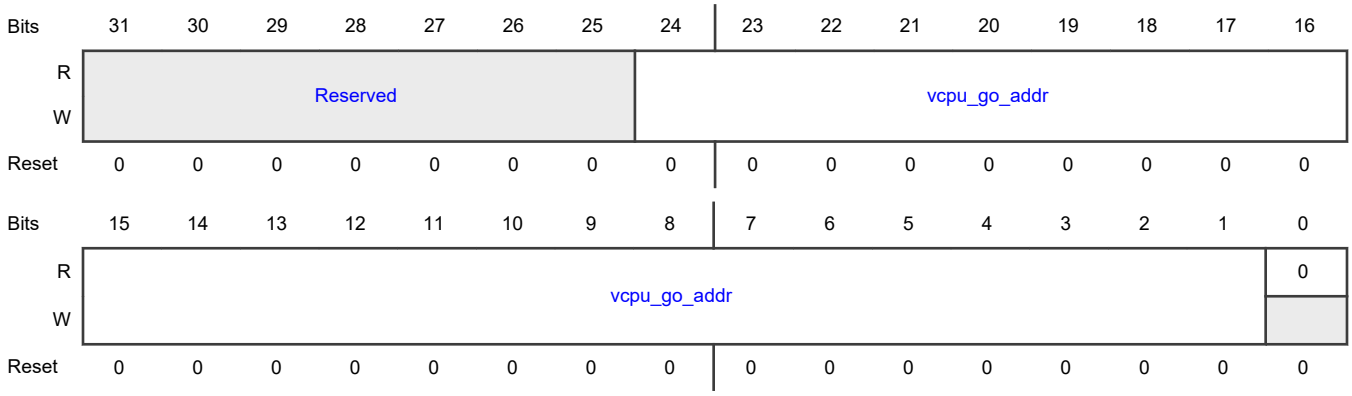
Register	Offset
VCPU_GO_ADDR	180h

Function

This register can only be written by the VCPU, writes by HOSTs are ignored.

Software can change the VCPU_GO_ADDR value at any time. For example, the startup code can write the address of the main function in this register, for subsequent go events to start execution of main.

Diagram



Fields

Field	Function
31-25 —	- Reserved
24-1 vcpu_go_addr	vcpu_go_addr The bit field width is 24 bits, and it points to an OpS sized (32-bit) instruction. Only 64-bit aligned instructions can be go starting addresses. **NOTE** This is a pointer to a 4-byte location in PMEM, the first 4 bytes of PMEM are accessed as 0x0, the next 4 bytes are accessed as 0x1, and so on.
0 —	- Reserved

14.2.56 VCPU Go Stack (VCPU_GO_STACK)

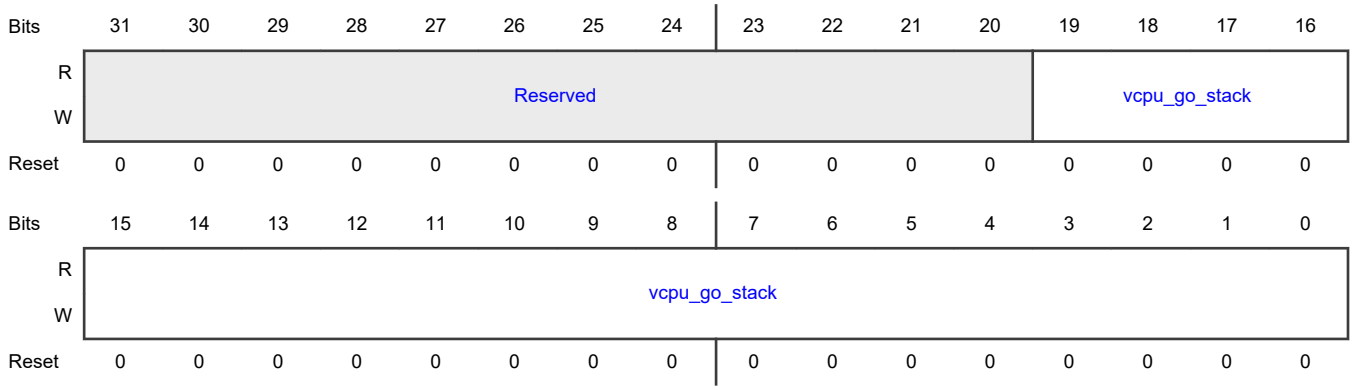
Offset

Register	Offset
VCPU_GO_STACK	184h

Function

This register can only be written by the VCPU, writes by HOSTs are ignored.

Diagram



Fields

Field	Function
31-20 —	- Reserved
19-0 vcpu_go_stack	vcpu_go_stack The bit field width is 20 bits, and it points to a half-word (16-bit) DMEM address. If this register is written to, all subsequent go events will cause the SP to be loaded with this value. If the register is not written to, go events will not cause the SP to be loaded, it will retain the value it had following the execution of the last done instruction.

14.2.57 VCPU Mode 0 (VCPU_MODE0)

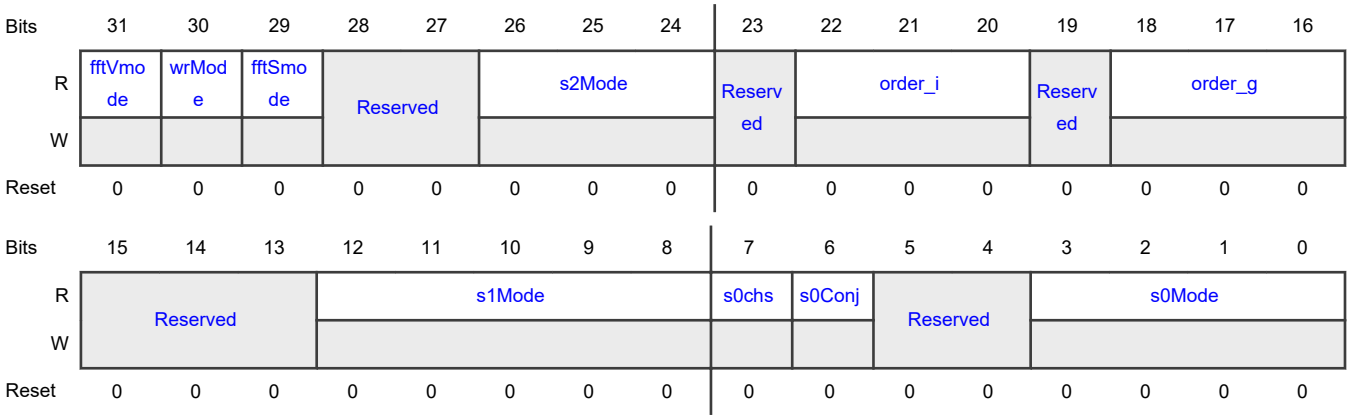
Offset

Register	Offset
VCPU_MODE0	400h

Function

This register may only be written by the VCPU, it is read only by the host.

Diagram



Fields

Field	Function
31 fftVmode	fftVmode FFT mode control bit for vector writeback
30 wrMode	wrMode Mode control bit for vector writeback
29 fftSmode	fftSmode FFT mode control bit for source mux
28-27 —	- Reserved
26-24 s2Mode	s2Mode Mode control for source mux 2
23 —	- Reserved
22-20 order_i	S1 order_i This is the exponent of base 2 in determining number of elements (n) in group, see S1mode options and detailed description - description of S1interp2nr and S1interp2nc. Value of order_i is restricted (1<2^order_i<NAU*4).
19 —	- Reserved
18-16 order_g	S0 order_g This is the exponent of base 2 in determining number of elements (n) in group, see S0mode options and detailed description - description of S0group2nr and S0group2nc. Value of order_g is restricted (1<2^order_g<NAU).

Table continues on the next page...

Table continued from the previous page...

Field	Function
15-13 —	- Reserved
12-8 s1Mode	s1Mode Mode control for source mux 1
7 s0chs	s0chs Sign control for source mux 0
6 s0Conj	s0Conj Conjugate control for source mux 0
5-4 —	- Reserved
3-0 s0Mode	s0Mode Mode control for source mux 0

14.2.58 VCPU Mode 1 (VCPU_MODE1)

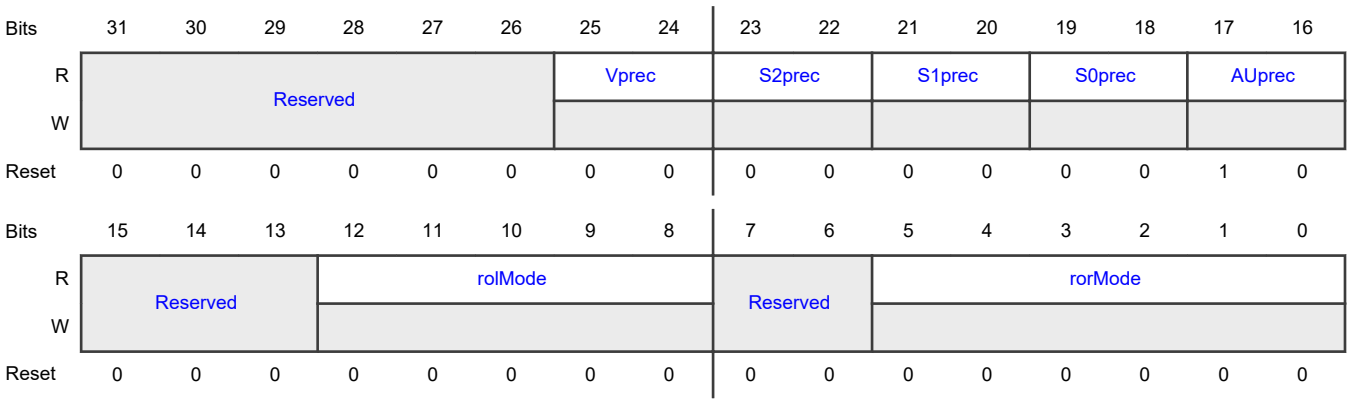
Offset

Register	Offset
VCPU_MODE1	404h

Function

This register may only be written by the VCPU, it is read only by the host.

Diagram



Fields

Field	Function
31-26 —	- Reserved
25-24 Vprec	Vprec VRA write back precision mode
23-22 S2prec	S2prec Source mux 2 precision mode
21-20 S1prec	S1prec Source mux 1 precision mode
19-18 S0prec	S0prec Source mux 0 precision mode
17-16 AUprec	AUprec Arithmetic unit precision mode
15-13 —	- Reserved
12-8 rolMode	rolMode Left rotation mode
7-6 —	- Reserved
5-0 rorMode	rorMode Right rotation mode

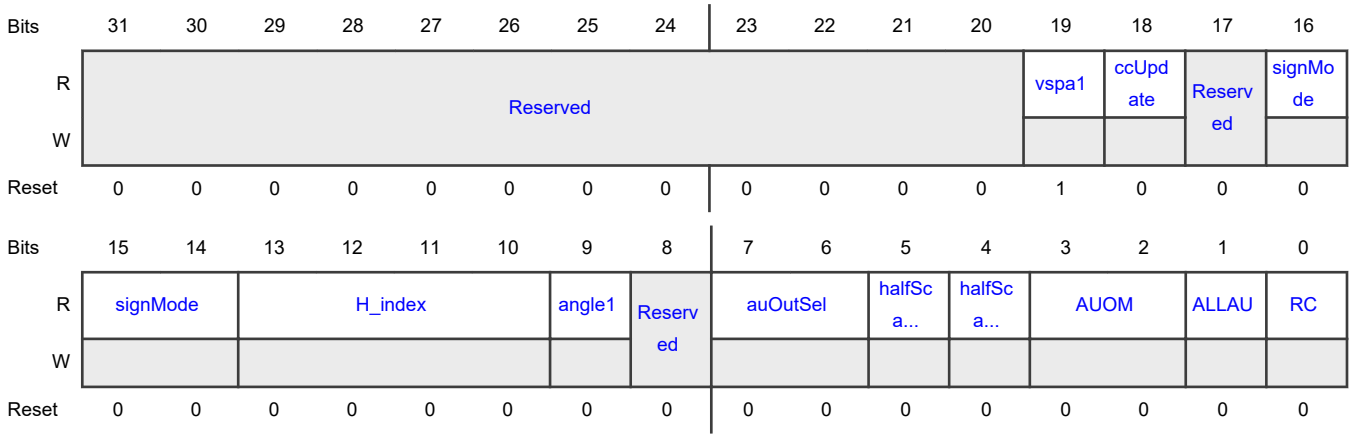
14.2.59 VCPU CREG 0 (VCPU_CREG0)

Offset

Register	Offset
VCPU_CREG0	408h

Function
This register may only be written by the VCPU, it is read only by the host.

Diagram



Fields

Field	Function
31-20 —	- Reserved
19 vspa1	vspa1 Legacy dmem addressing
18 ccUpdate	ccUpdate Condition Code update switch
17 —	- Reserved
16-14 signMode	signMode H register control
13-10 H_index	H_index H register sub-address. Read only.
9 angle1	angle ANGLE_MODULO_WRAP
8 —	- Reserved
7-6 auOutSel	auOutSel SP VAU output lane switch
5	halfScale

Table continues on the next page...

Table continued from the previous page...

Field	Function
halfScale1	S1prec control
4 halfScale	halfScale Vprec control
3-2 AUOM	AUOM VAU output mode control
1 ALLAU	ALLAU SP VAU output width switch
0 RC	RC Real/Complex mode control

14.2.60 VCPU CREG 1 (VCPU_CREG1)

Offset

Register	Offset
VCPU_CREG1	40Ch

Function

This register may only be written by the VCPU, it is read only by the host.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	Reserved												readIndex			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	readIndex				interpN				interpD				interpP			
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-20	-

Table continues on the next page...

Table continued from the previous page...

Field	Function
—	Reserved
19-12 readIndex	readIndex CReg read address
11-8 interpN	interpN Fractional interpolator numerator constant
7-4 interpD	interpD Fractional interpolator denominator constant. For more information, refer to Table 21
3-0 interpP	interpP Fractional interpolator phase

14.2.61 Store Unalign Vector Length (ST_UL_VEC_LEN)

Offset

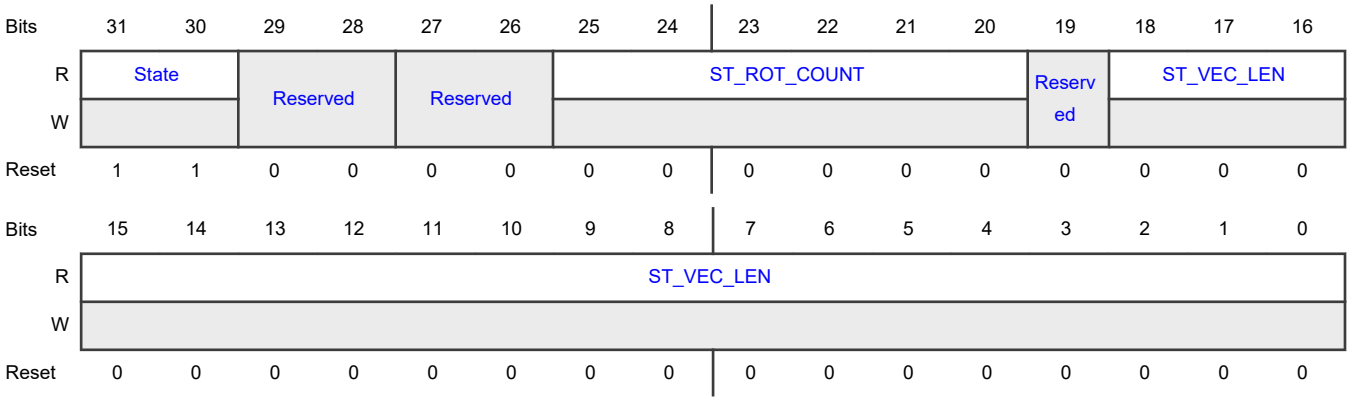
Register	Offset
ST_UL_VEC_LEN	410h

Function

See st.uline instruction.

This register may only be written by the VCPU, it is read only by the host.

Diagram



Fields

Field	Function
31-30 State	State Current state of mis-aligned store FSM **NOTE** To properly use the st.uline instruction, bits 31-19 must be written to 0's when initializing the ST_VEC_LEN field (bits 18-0) in this register. 00b - active wrap (store crossing line boundary) 01b - invalid 10b - active no wrap (full line store) 11b - idle
29-28 —	- Reserved
27-26 —	- Reserved
25-20 ST_ROT_COUNT	ST_ROT_COUNT Rotation offset into dmem line. Maximum width of field is 8 bits, actual width is dependent on AU count. Actual width = log2(NAU*4) **NOTE** To properly use the st.uline instruction, bits 31-19 must be written to 0's when initializing the ST_VEC_LEN field (bits 18-0) in this register.
19 —	- Reserved
18-0 ST_VEC_LEN	ST_VEC_LEN Total length of vector (in half-words) to be stored. This field gets updated on every st.uline instruction to reflect the remaining data length to be stored. This register may only be written by the VCPU, it is read only by the host.

14.2.62 General Purpose Input registers [10 registers] (GP_IN0 - GP_IN9)

Offset

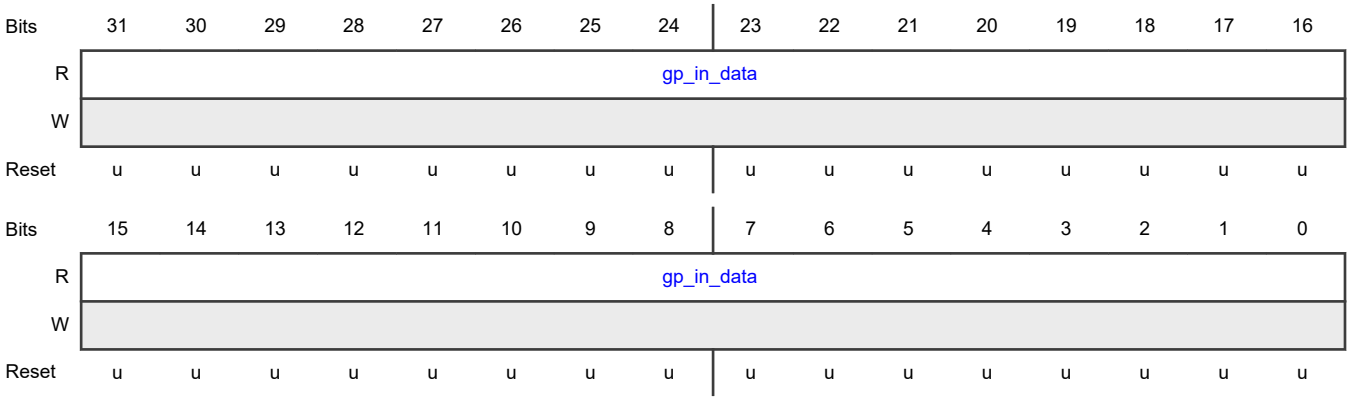
For a = 0 to 9:

Register	Offset
GP_INa	500h + (a × 4h)

Function

- GP_IN registers are read only - writes have no meaning or effect.
- 10 General Purpose Input registers are available.

Diagram



Fields

Field	Function
31-0 gp_in_data	gp_in_data Reads return the status of the associated VSPA external ports from 2 to 4 VSPA clocks in the past. The 2-4 clock delay is associated with clock gating and synchronization. These bits are general purpose inputs. They can be tied by the integration of a VSPA platform to provide visibility of the connected signals to VSPA or host software. The usage is chip dependent.

14.2.63 General Purpose Output registers [10 registers] (GP_OUT0 - GP_OUT9)

Offset

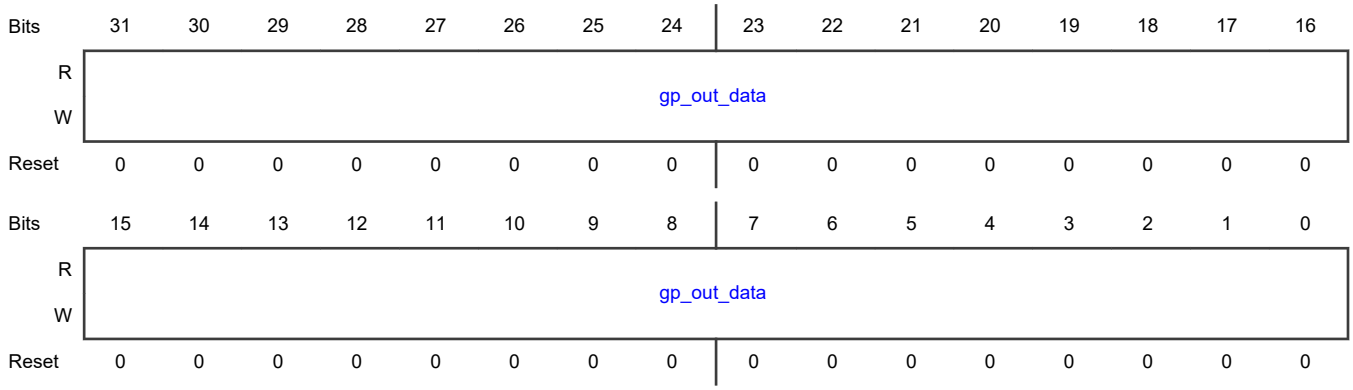
For a = 0 to 9:

Register	Offset
GP_OUTa	580h + (a × 4h)

Function

- GP_OUT registers can be written and read. Writes update the associated VSPA output ports. Reads read back the value of the associated GP_OUT register.
- 10 General Purpose Output registers are available.

Diagram



Fields

Field	Function
31-0 gp_out_data	gp_out_data gp_out_data • Writes control the state of the associated output port. • Reads return the state of the associated output port. 00000000000000000000000000000000b - output port low. 00000000000000000000000000000001b - output port high.

14.2.64 VCPU to DQM Trace Small Outbox register (DQM_SMALL)

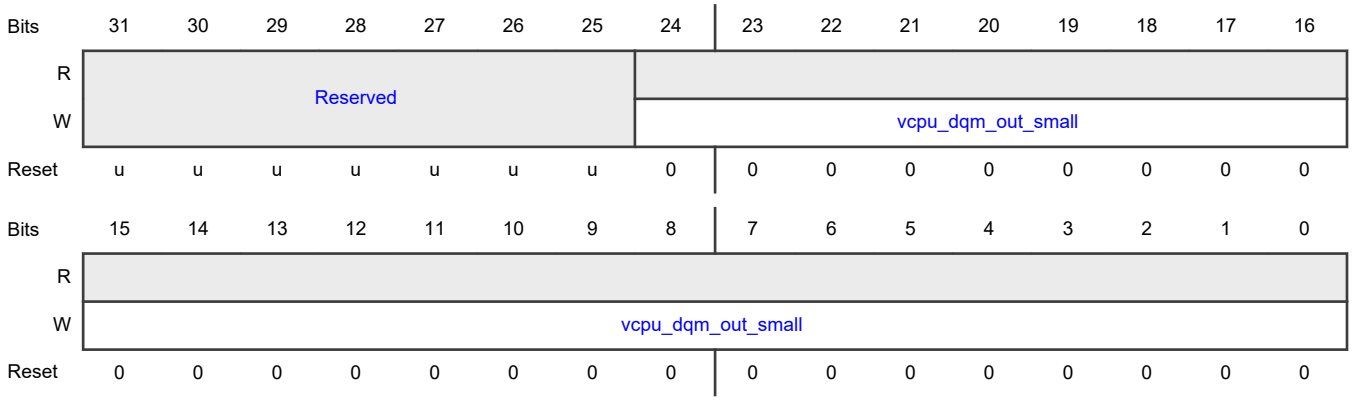
Offset

Register	Offset
DQM_SMALL	600h

Function

This register can be used by VCPU (or a host) to generate a "small" (25-bit payload) data acquisition message (DQM). Debug and DQM trace must both be enabled for the message to be sent out via VSPA's trace port.

Diagram



Fields

Field	Function
31-25 —	- Reserved
24-0 vcpu_dqm_out_small	vcpu_dqm_out_small DQM output data (small) • Writes cause the value written to be transmitted as a DQM message. • This register is write only - reads always return 0's. Settings: Data value to be transmitted as a DQM via the VSPA trace port.

14.2.65 VCPU to Debugger 32-bit Outbox register (VCPU_DBG_OUT_32)

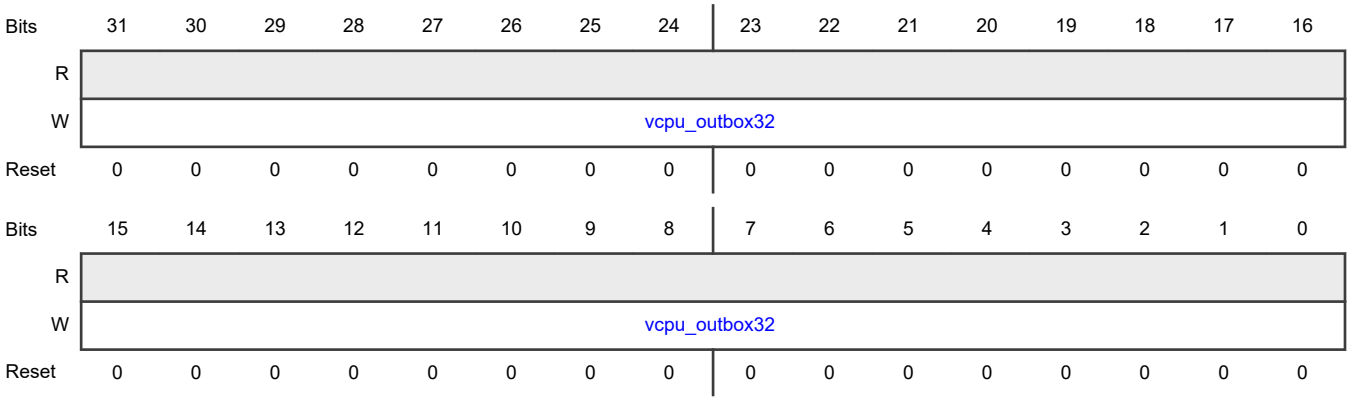
Offset

Register	Offset
VCPU_DBG_OUT_32	620h

Function

This register can be used by VCPU (or a host) to send a 32-bit message to the debugger. Writes to this register cause a 32-bit message to be forwarded to the debugger.

Diagram



Fields

Field	Function
31-0 vcpu_outbox32	vcpu_outbox32 VCPU to Debugger 32-bit outbox • Writes cause the value written to be sent to the debugger. • This register is write only - reads always return 0's. Settings: Data value to be sent to the debugger inbox.

14.2.66 VCPU to Debugger 64-bit MSB Outbox register (VCPU_DBG_OUT_64_MSB)

Offset

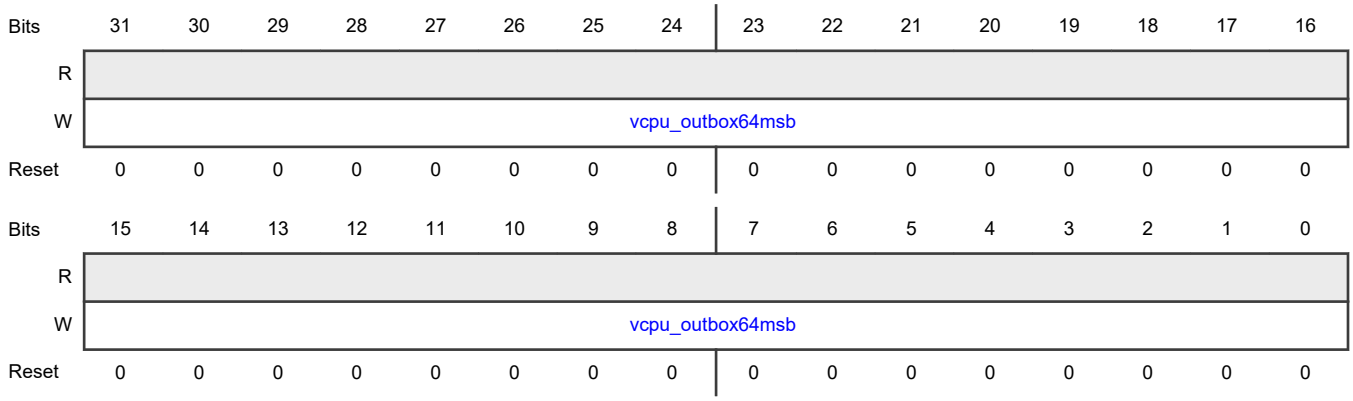
Register	Offset
VCPU_DBG_OUT_64_MSB	624h

Function

This register is part of the interface used by VCPU (or a host) to generate a 64-bit mail message directed to the debugger's 64-bit inbox.

Note that writing this register is not required to send the 64-bit mail message. The mail message will be sent following a write to the DBG_OUT_64_LSB register.

Diagram



Fields

Field	Function
31-0 vcpu_outbox64msb	vcpu_outbox64msb VCPU to Debugger 64-bit output data (MSB) • Writes update the 32 MSBs of a pending 64-bit message that will be sent to the debugger's 64-bit inbox. • The mail message will be sent in its entirety only after a write to the VCPU_DBG_OUT_64_LSB register. • Note that this register does not have to be updated for each 64-bit mail message - if it is not updated, the last value written will be reused when the mail message is sent. • This register is write only - reads always return 0's. Settings: MSB 32-bit data value to be sent to the debugger's 64-bit inbox.

14.2.67 VCPU to Debugger 64-bit LSB Outbox register (VCPU_DBG_OUT_64_LSB)

Offset

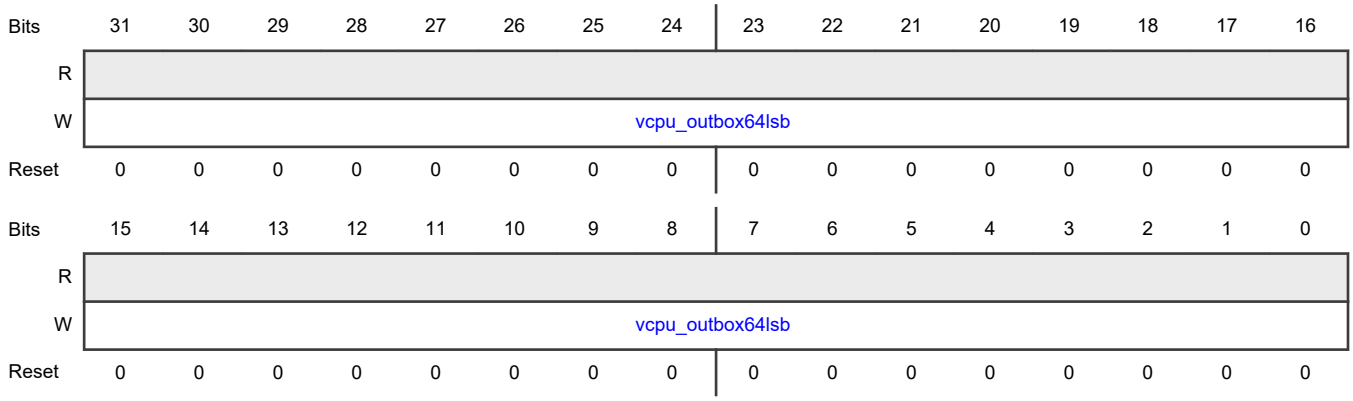
Register	Offset
VCPU_DBG_OUT_64_LSB	628h

Function

This register is part of the interface used by VCPU (or a host) to generate a 64-bit mail message directed to the debugger's 64-bit inbox.

Note that writing to this register alone will trigger the delivery of the 64-bit mail message to the debugger. The mail message will contain 32 MSBs from the VCPU_DBG_OUT_64_MSB register plus all 32 bits from this register.

Diagram



Fields

Field	Function
31-0 vcpu_outbox64lsb	vcpu_outbox64lsb VCPU to Debugger 64-bit output data (LSB) • Writes cause the immediate delivery of a 64-bit mail message to the debugger 64-bit mail inbox. The 32 LSBs of the message come from the data written to this register; the 32 MSBs come from the value contained in the VCPU_DBG_OUT_64_MSB register. • This register is write only - reads always return 0's. Settings: LSB 32-bit data value to be sent to the debugger's 64-bit inbox.

14.2.68 Debugger to VCPU 32-bit Inbox register (VCPU_DBG_IN_32)

Offset

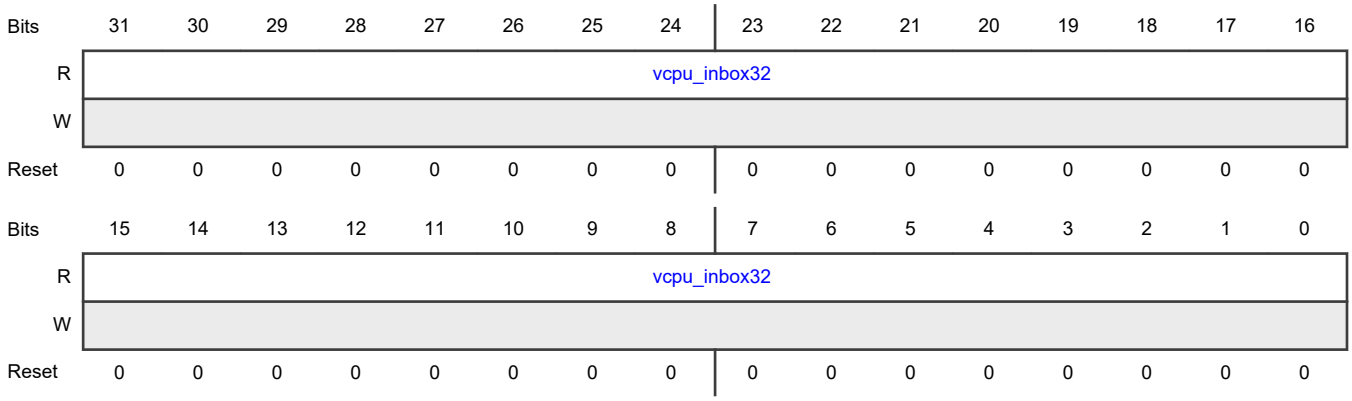
Register	Offset
VCPU_DBG_IN_32	62Ch

Function

This register is the VCPU incoming mailbox for 32-bit messages from the debugger. It can be used by VCPU (or a host) to read a 32-bit message from the debugger. The validity of the data in this register should be checked before it is read. The state of the 32-bit_msg_in_valid flag in the VCPU to Debugger mailbox status register conveys validity information.

Reads of this register will automatically clear both the 32-bit_msg_in_valid flag in the VCPU to Debugger mailbox status register (on the VSPA IP bus) and the 32-bit_msg_out_valid flag in the Debug-VCPU mailbox status register (on the debug IP bus).

Diagram



Fields

Field	Function
31-0 vcpu_inbox32	vcpu_inbox32 Debugger to VCPU 32-bit inbox • Reads the 32-bit mail message sent by the debugger. • This register is read only - writes have no effect. Settings: Reads 32-bit data value sent by the debugger outbox.

14.2.69 Debugger to VCPU 64-bit MSB Inbox register (VCPU_DBG_IN_64_MSB)

Offset

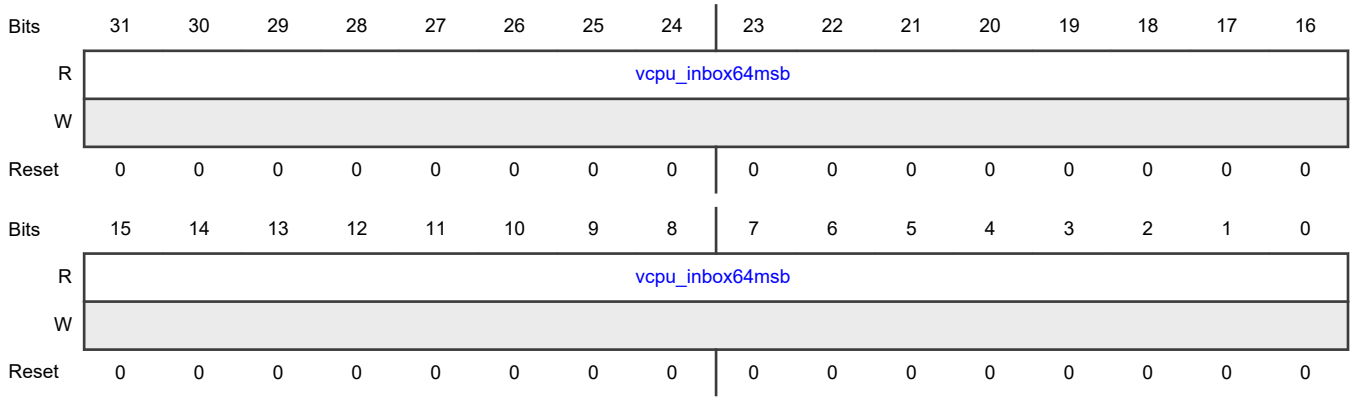
Register	Offset
VCPU_DBG_IN_64_MSB	630h

Function

This register is part of the VCPU incoming mailbox for 64-bit messages from the debugger. It can be used by VCPU (or a host) to read the 32 MSBs of a 64-bit message from the debugger. The validity of the data in this register should be checked before it is read. The state of the 64-bit_msg_in_valid flag in the VCPU to Debugger mailbox status register conveys validity information.

Reads of this register do not affect the 64-bit_msg_in_valid flag in the VCPU to Debugger mailbox status register or the 64-bit_msg_out_valid flag in the Debug-VCPU mailbox status register on the debug IP bus. Those flags are cleared by reads of the Debugger to VCPU 64-bit LSB inbox register (on the debug IP bus).

Diagram



Fields

Field	Function
31-0 vcpu_inbox64msb	vcpu_inbox64msb Debugger to VCPU 64-bit input data (MSB) • Reads the 32 MSBs of the 64-bit mail message sent by the debugger. • This register is read only - writes have no effect. Settings: Reads return 32 MSBs of the 64-bit data value sent via the debugger's 64-bit outbox.

14.2.70 Debugger to VCPU 64-bit LSB Inbox register (VCPU_DBG_IN_64_LSB)

Offset

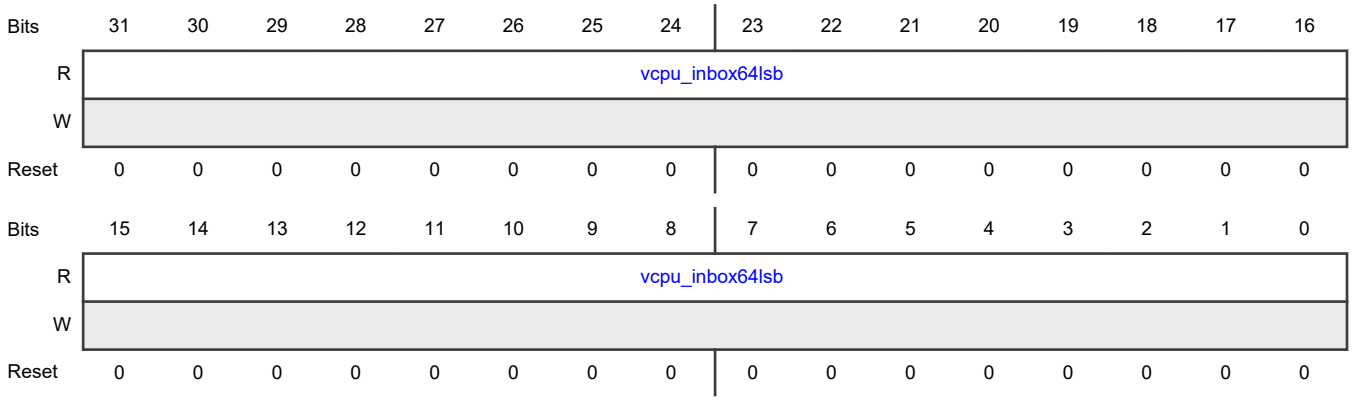
Register	Offset
VCPU_DBG_IN_64_LSB	634h

Function

This register is part of the VCPU incoming mailbox for 64-bit messages from the debugger. It can be used by VCPU (or a host) to read the 32 LSBs of a 64-bit message from the debugger. The validity of the data in this register should be checked before it is read. The state of the 64-bit_msg_in_valid flag in the VCPU to Debugger mailbox status register conveys validity information.

Reads of this register will automatically clear the 64-bit_msg_in_valid flag in the VCPU to Debugger mailbox status register (on the VSPA IP bus) and the 64-bit_msg_out_valid flag in the VCPU-Debug mailbox status register on the debug IP bus.

Diagram



Fields

Field	Function
31-0 vcpu_inbox64lsb	vcpu_inbox64lsb Debugger to VCPU 64-bit input data (LSB) • Reads the 32 LSBs of the 64-bit mail message sent by the debugger. • This register is read only - writes have no effect. Settings: Reads return 32 LSBs of the 64-bit data value sent via the debugger's 64-bit outbox.

14.2.71 VCPU to Debugger Mailbox Status register (VCPU_DBG_MBOX_STATUS)

Offset

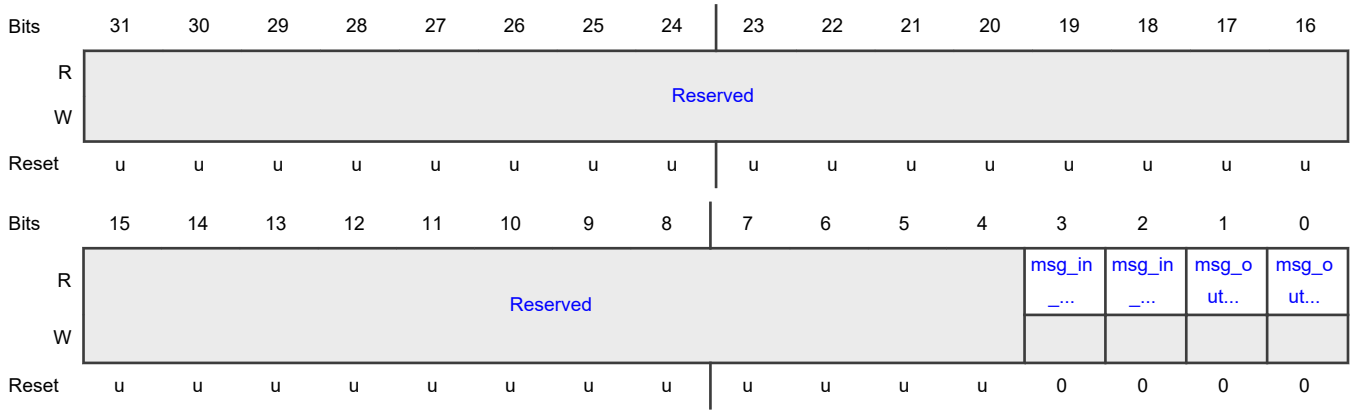
Register	Offset
VCPU_DBG_MBOX_STATUS	638h

Function

This register is used to determine the status of messages sent and received by VCPU (or a host) to and from the debugger. There are four status flags, one each for the 32 and 64-bit outgoing mailboxes and the 32 and 64-bit incoming mailboxes.

These flags are set and cleared automatically by hardware - reads and writes of this register do not affect the status flags. However, reads and writes of the mailbox data registers do control the state of the status flags.

Diagram



Fields

Field	Function
31-4 —	- Reserved
3 msg_in_valid_64bit	msg_in_valid_64bit 64-bit message inbox valid • Shows the validity/invalidity of the data in the 64-bit message inbox registers (Debugger to VCPU 64-bit MSB Inbox register and Debugger to VCPU 64-bit LSB Inbox register). 0b - Data in the 64-bit inbox registers is NOT valid. 1b - Data in the 64-bit inbox registers is valid.
2 msg_in_valid_32bit	msg_in_valid_32bit 32-bit message inbox valid • Shows the validity/invalidity of the data in the 32-bit message inbox register (Debugger to VCPU 32-bit Inbox register). 0b - Data in the 32-bit inbox register is NOT valid. 1b - Data in the 32-bit inbox register is valid.
1 msg_out_valid_64bit	msg_out_valid_64bit 64-bit message outbox valid • Shows whether a pending (unread by the debugger) message is in the 64-bit message outbox registers (VCPU to Debugger 64-bit MSB outbox register and VCPU to Debugger 64-bit LSB Outbox register). 0b - No unread data is in the 64-bit outbox registers. 1b - Data in the 64-bit outbox registers has not yet been read by the debugger.
0	msg_out_valid_32bit 32-bit message outbox valid

Table continues on the next page...

Table continued from the previous page...

Field	Function
msg_out_valid_32bit	Shows whether a pending (unread by the debugger) message is in the 32-bit message outbox register (VCPU to Debugger 32-bit Outbox register). 0b - No unread data is in the 32-bit outbox register. 1b - Data in the 32-bit outbox register has not yet been read by the debugger.

14.2.72 VCPU to host outbox message n MSB register (VCPU_OUT_0_MSB - VCPU_OUT_1_MSB)

Offset

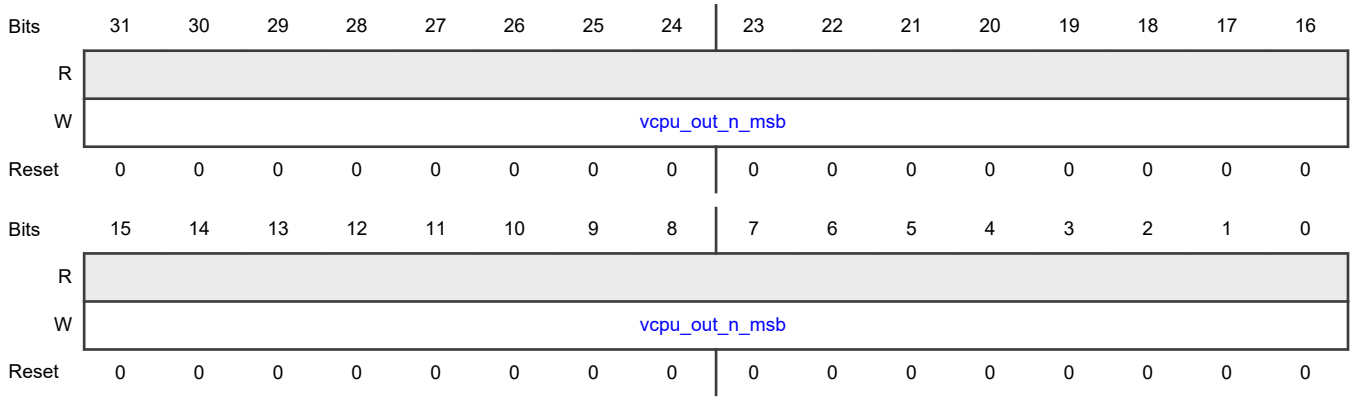
Register	Offset
VCPU_OUT_0_MSB	640h
VCPU_OUT_1_MSB	648h

Function

This register is the 32-bit MSB part of the interface used by VCPU to generate 64-bit mail message *n* directed to the host's inbox.

Note that writing this register is not required to send the 64-bit mail message. The mail message will be sent following a write to the VSPA_VCPU_OUT_*n*_LSB register. This register is write only; reads always return 0's.

Diagram



Fields

Field	Function
31-0 vcpu_out_n_msb	vcpu_out_n_msb VCPU output message *n* data (MSB). • Writes update the 32 MSBs of a pending 64-bit message *n* that will be sent to the host's 64-bit inbox.

Table continues on the next page...

Field	Function
	• The mail message will be sent in its entirety only after a write to the VSPA_VCPU_OUT_*n*_LSB register. • Note that this register does not have to be updated for each 64-bit mail message - if it is not updated, the last value written will be reused when the mail message is sent (when VSPA_VCPU_OUT_*n*_LSB is written). Settings: MSB 32-bit data value will be sent to the host's 64-bit message n inbox .

14.2.73 VCPU to host outbox message n LSB register (VCPU_OUT_0_LSB - VCPU_OUT_1_LSB)

Offset

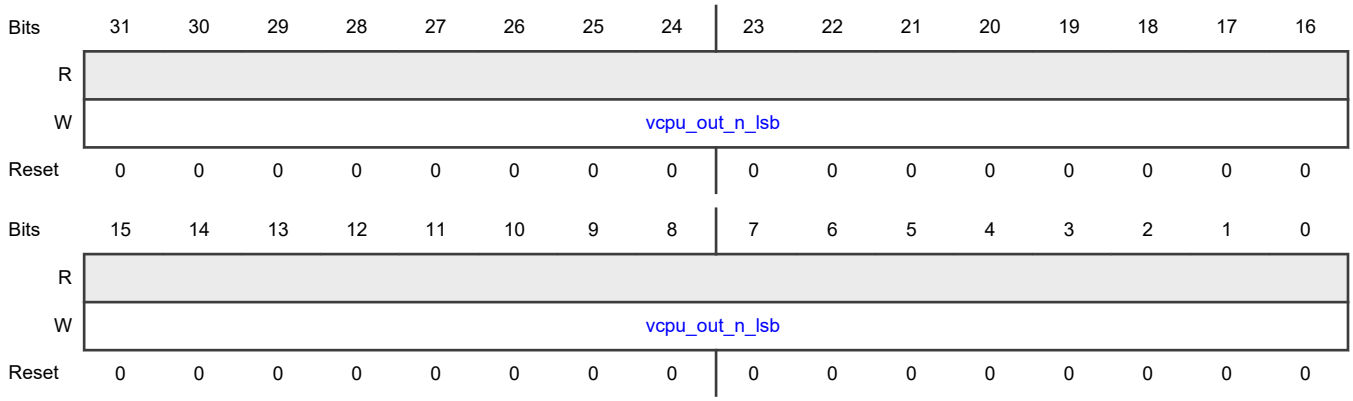
Register	Offset
VCPU_OUT_0_LSB	644h
VCPU_OUT_1_LSB	64Ch

Function

This register is the 32-bit LSB part of the interface used by VCPU to generate 64-bit mail message *n* directed to the host's inbox.

Note that writing to this register alone will trigger the delivery of the 64-bit mail message to the host. The mail message will contain 32 MSBs from the VSPA_VCPU_OUT_*n*_MSB register along with all the 32 bits from this register. This register is write only - reads always return 0's

Diagram



Fields

Field	Function
31-0 vcpu_out_n_lsb	vcpu_out_n_lsb VCPU output message *n* data (LSB)

Table continues on the next page...

Field	Function
	Writes cause the immediate delivery of a 64-bit mail message *n* to the host mail inbox. The 32 LSBs of the message come from the data written to this register; the 32 MSBs come from the value contained in the VSPA_VCPU_OUT_*n*_MSB register. Settings: LSB 32-bit data value will be sent to the host's 64-bit inbox.

14.2.74 VCPU from Host Inbox Message n MSB (VCPU_IN_0_MSB - VCPU_IN_1_MSB)

Offset

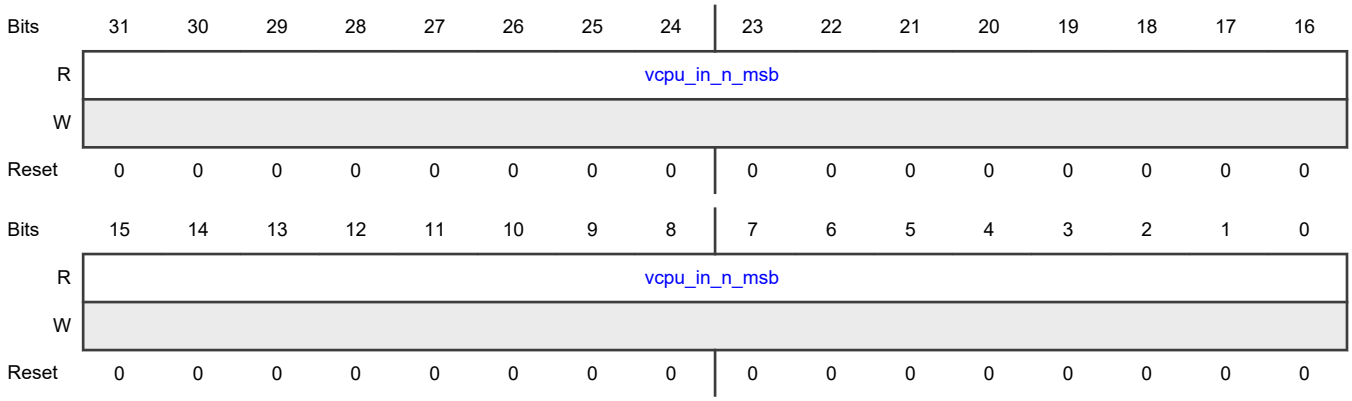
Register	Offset
VCPU_IN_0_MSB	650h
VCPU_IN_1_MSB	658h

Function

This register is the 32-bit MSB part of the incoming 64-bit mail message *n* from the host. It can be used by VCPU to read the 32 MSBs of a 64-bit message *n* from the host. The validity of the data in this register should be checked before it is read. The state of the msg_in_*n*_valid flag in the VCPU-host mailbox status register indicates validity information when set.

Reads of this register do not affect the msg_in_*n*_valid flag in the VCPU to Host Mailbox Status register (VCPU_MBOX_STATUS) or the msg_out_*n*_valid flag in the Host Mailbox Status register (HOST_MBOX_STATUS). These flags are cleared by reads of the host to VCPU 64-bit LSB inbox register. This register is read only - writes have no effect.

Diagram



Fields

Field	Function
31-0 vcpu_in_n_msb	vcpu_in_n_msb VCPU 64-bit message *n* input data (MSB). This bits are used for reading the 32 MSBs of the 64-bit mail message sent by the host. Settings: Reads return 32 MSBs of the message *n* 64-bit data value sent via the host outbox.

14.2.75 VCPU from host inbox message n LSB register (VCPU_IN_0_LSB - VCPU_IN_1_LSB)

Offset

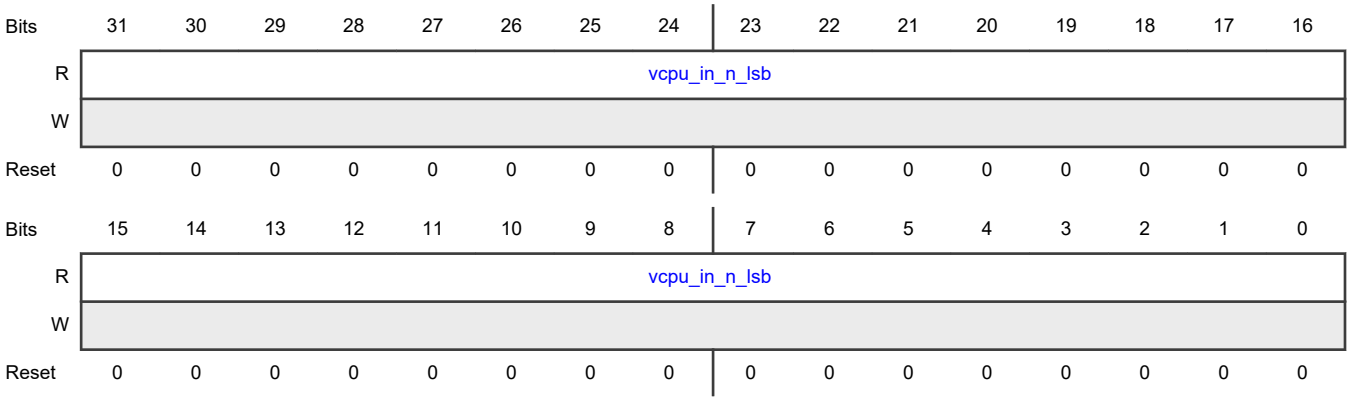
Register	Offset
VCPU_IN_0_LSB	654h
VCPU_IN_1_LSB	65Ch

Function

This register is the 32-bit LSB part of the incoming 64-bit mail message *n* from the host. It can be used by VCPU to read the 32 LSBs of a 64-bit message *n* from the host. The validity of the data in this register should be checked before it is read. The state of the msg_in_*n*_valid flag in the VCPU-host mailbox status register indicates validity information when set.

Reads of this register will automatically clear the msg_in_*n*_valid flag in the VCPU host mailbox status register (VCPU_MBOX_STATUS) and the msg_out_*n*_valid flag in the Host Mailbox Status register (HOST_MBOX_STATUS). This register is read only - writes have no effect.

Diagram



Fields

Field	Function
31-0 vcpu_in_n_lsb	vcpu_in_n_lsb VCPU 64-bit message *n* input data (LSB). the bits are used for reading the 32 LSBs of the 64-bit mail message sent by the host. Settings: Reads return 32 LSBs of the message *n* 64-bit data value sent via the host outbox.

14.2.76 VCPU to Host Mailbox Status register (VCPU_MBOX_STATUS)

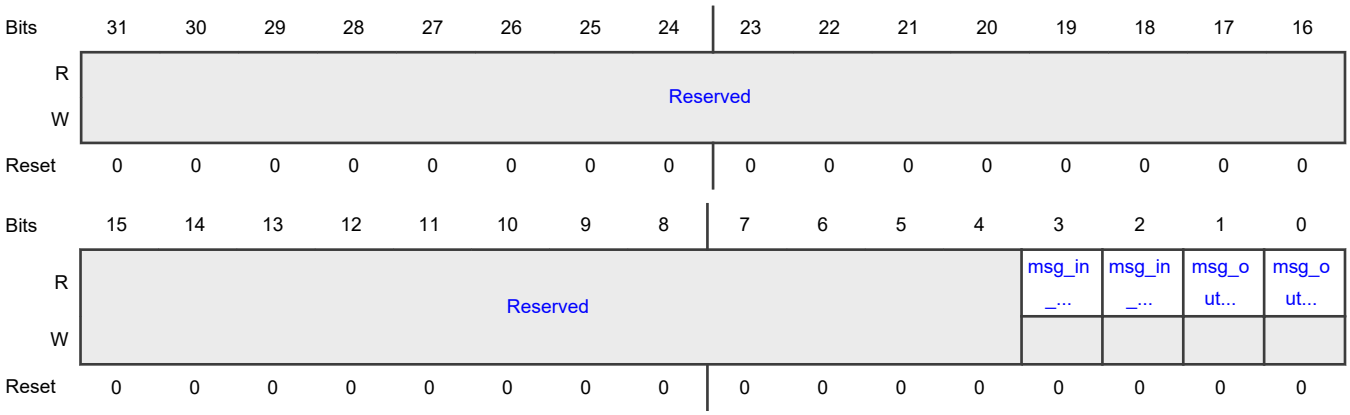
Offset

Register	Offset
VCPU_MBOX_STATUS	660h

Function

This register is used to determine the status of messages sent and received by VCPU (or a host) to and from the host. There are four status flags, one each for message 0 and message 1 outgoing mailboxes and message 0 and message 1 incoming mailboxes. These flags are set and cleared automatically by hardware - reads and writes of this register do not affect the status flags. However, reads and writes of the mailbox data registers do control the state of the status flags.

Diagram



Fields

Field	Function
31-4 —	- Reserved
3 msg_in_1_valid	msg_in_1_valid 64-bit inbox message 1 valid. Indicates the validity/invalidity of the data in the VCPU 64-bit message 1 inbox registers (VSPA_VCPU_IN_1_MSB register and VSPA_VCPU_IN_1_LSB register). 0b - Data in the 64-bit inbox registers is not valid. 1b - Data in the 64-bit inbox registers is valid.
2 msg_in_0_valid	msg_in_0_valid 64-bit inbox message 0 valid. Indicates the validity/invalidity of the data in the VCPU 64-bit message 0 inbox registers (VSPA_VCPU_IN_0_MSB register and VSPA_VCPU_IN_0_LSB register). 0b - Data in the inbox registers VSPA_VCPU_IN_0_MSB and VSPA_VCPU_IN_0_LSB is not valid. 1b - Data in the inbox registers VSPA_VCPU_IN_0_MSB and VSPA_VCPU_IN_0_LSB is valid.
1 msg_out_1_valid	msg_out_1_valid 64-bit outbox message 1 valid. Indicates if the 64-bit VCPU message outbox 1 (VSPA_VCPU_OUT_1_MSB register and VSPA_VCPU_OUT_1_LSB register) is pending (unread by the host).

Table continues on the next page...

Table continued from the previous page...

Field	Function
	0b - No data is pending to be read in the 64-bit message 1 outbox registers. 1b - Data in the 64-bit message 1 outbox registers has not yet been read by the host.
0 msg_out_0_valid	msg_out_0_valid 64-bit outbox message 0 valid. Indicates if the 64-bit VCPU message outbox 0 (VSPA_VCPU_OUT_0_MSB register and VSPA_VCPU_OUT_0_LSB register) is pending (unread by the host). 0b - No data is pending to be read in the 64-bit message 0 outbox registers. 1b - Data in the 64-bit message 0 outbox registers has not yet been read by the host.

14.2.77 Host to VCPU Outbox Message n MSB register (HOST_OUT_0_MSB - HOST_OUT_1_MSB)

Offset

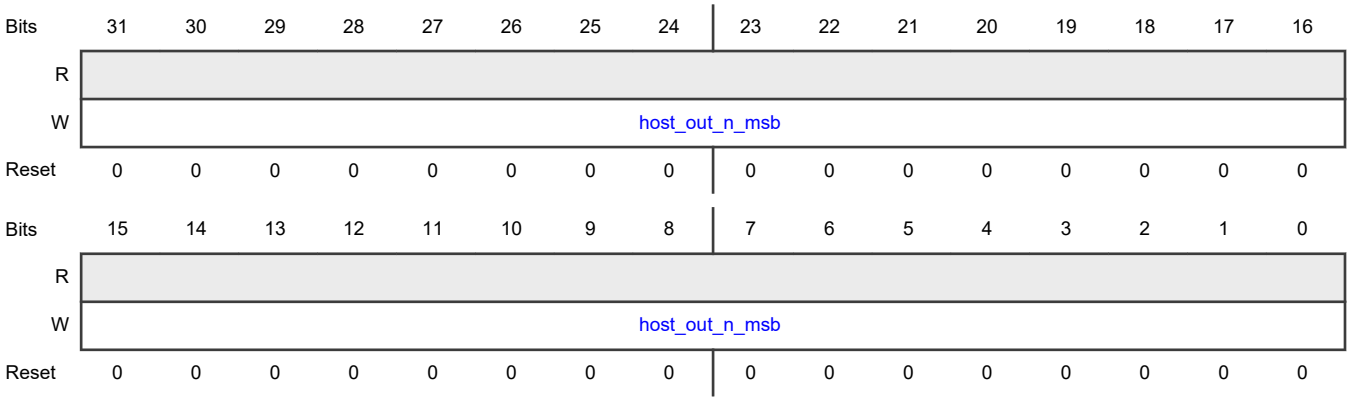
Register	Offset
HOST_OUT_0_MSB	680h
HOST_OUT_1_MSB	688h

Function

This register is the 32-bit MSB part of the interface used by the host to generate 64-bit mail message *n* directed to VCPU's inbox.

Note that writing this register is not required to send the 64-bit mail message. The mail message will be sent following a write to the VSPA_HOST_OUT_*n*_LSB register. This register is write only - reads always return 0's.

Diagram



Fields

Field	Function
31-0 host_out_n_msb	host_out_n_msb Host output message *n* data (MSB) • Writes update the 32 MSBs of a pending 64-bit message *n* that will be sent to VCPU's 64-bit inbox. • The mail message will be sent in its entirety only after a write to the VSPA_HOST_OUT_*n*_LSB register. • Note that this register does not have to be updated for each 64-bit mail message; if it is not updated, the last value written will be reused when the mail message is sent (when VSPA_HOST_OUT_*n*_LSB is written) Settings: MSB 32-bit data value to be sent to VCPU's 64-bit message *n* inbox .

14.2.78 Host to VCPU Outbox Message n LSB register (HOST_OUT_0_LSB - HOST_OUT_1_LSB)

Offset

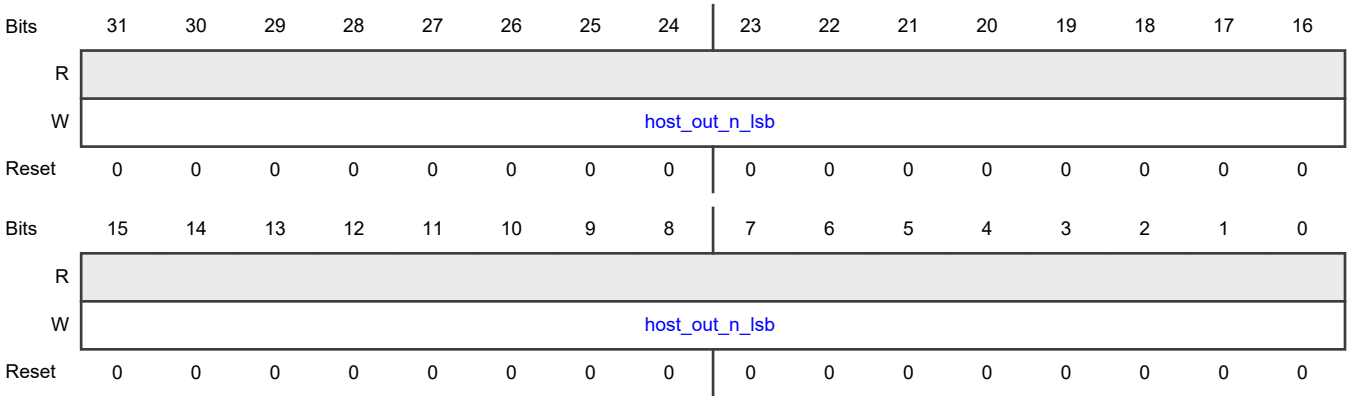
Register	Offset
HOST_OUT_0_LSB	684h
HOST_OUT_1_LSB	68Ch

Function

This register is the 32-bit LSB part of the interface used by the host to generate 64-bit mail message *n* directed to VCPU's inbox.

Note that writing to this register alone will trigger the delivery of the 64-bit mail message to VCPU. The mail message will contain 32 MSBs from the VSPA_HOST_OUT_*n*_MSB register plus all 32 bits from this register. This register is write only - reads always return 0's.

Diagram



Fields

Field	Function
31-0 host_out_n_lsb	host_out_n_lsb Host output message *n* data (LSB) Writes cause the immediate delivery of a 64-bit mail message *n* to VCPU mail inbox. The 32 LSBs of the message come from the data written to this register; the 32 MSBs come from the value contained in the VSPA_HOST_OUT_*n*_MSB register. Settings: LSB 32-bit data value to be sent to VCPU's 64-bit inbox.

14.2.79 Host from VCPU Inbox Message n MSB (HOST_IN_0_MSB - HOST_IN_1_MSB)

Offset

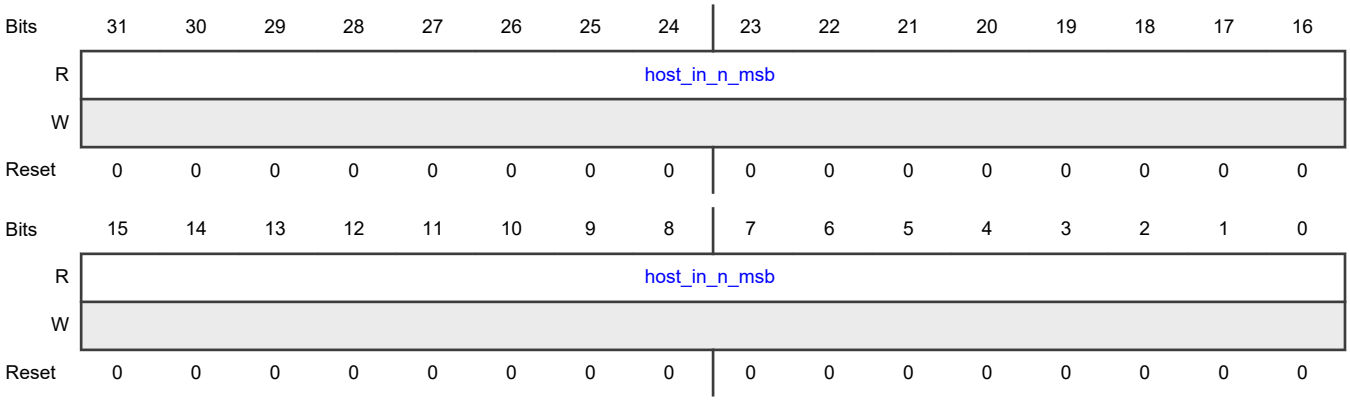
Register	Offset
HOST_IN_0_MSB	690h
HOST_IN_1_MSB	698h

Function

This register is the 32-bit MSB part of the incoming 64-bit mail message *n* from the VCPU. It can be used by the host to read the 32 MSBs of a 64-bit message *n* from the VCPU. The validity of the data in this register should be checked before it is read. The state of the msg_in_n_valid flag in the host-VCPU mailbox status register indicates validity information when set.

Reads of this register do not affect the msg_in_n_valid flag in the Host Mailbox Status register (HOST_MBOX_STATUS) or the msg_out_n_valid flag in VCPU to Host Mailbox Status Register (VCPU_MBOX_STATUS). These flags are cleared by reads of VCPU to the host 64-bit LSB inbox register.

Diagram



Fields

Field	Function
31-0 host_in_n_msb	host_in_n_msb Host 64-bit message *n* input data (MSB). Reads the 32 MSBs of the 64-bit mail message sent by VCPU. Settings: Reads return 32 MSBs of the message *n* 64-bit data value sent via VCPU outbox.

14.2.80 Host from VCPU Inbox Message n LSB Register (HOST_IN_0_LSB - HOST_IN_1_LSB)

Offset

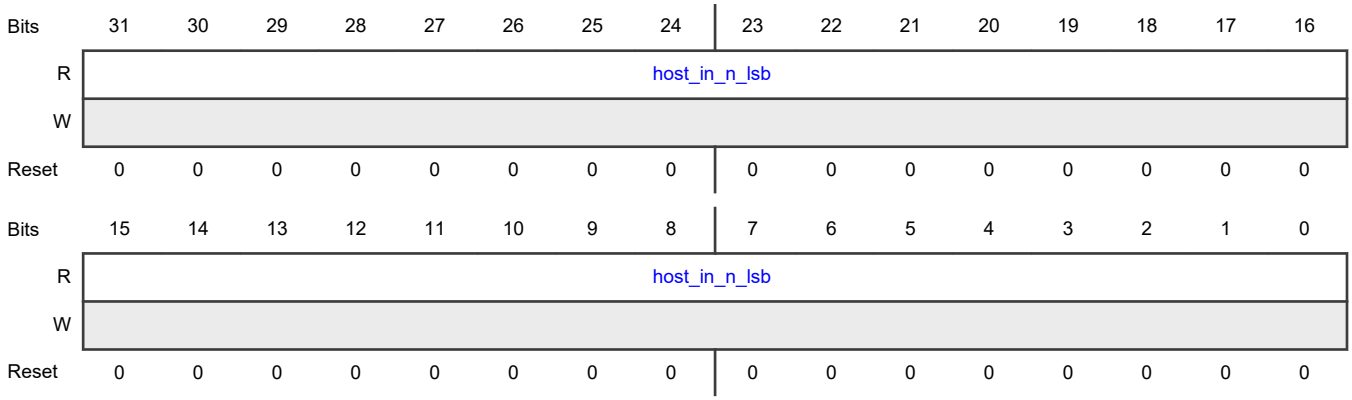
Register	Offset
HOST_IN_0_LSB	694h
HOST_IN_1_LSB	69Ch

Function

This register is the 32-bit LSB part of the incoming 64-bit mail message *n* from the VCPU. It can be used by the host to read the 32 LSBs of a 64-bit message *n* from the VCPU. The validity of the data in this register should be checked before it is read. The state of the msg_in_*n*_valid flag in the host-VCPU mailbox status register indicates validity information when set.

Reads of this register will automatically clear the msg_in_*n*_valid flag in the Host Mailbox Status register (HOST_MBOX_STATUS) and the msg_out_*n*_valid flag in VCPU to Host Mailbox Status Register (VCPU_MBOX_STATUS). This register is read only - writes have no effect.

Diagram



Fields

Field	Function
31-0 host_in_n_lsb	host_in_n_lsb Host 64-bit message *n* input data (LSB). Reads the 32 LSBs of the 64-bit mail message sent by VCPU. Settings: Reads return 32 LSBs of the message *n* 64-bit data value sent via VCPU outbox.

14.2.81 Host Mailbox Status Register (HOST_MBOX_STATUS)

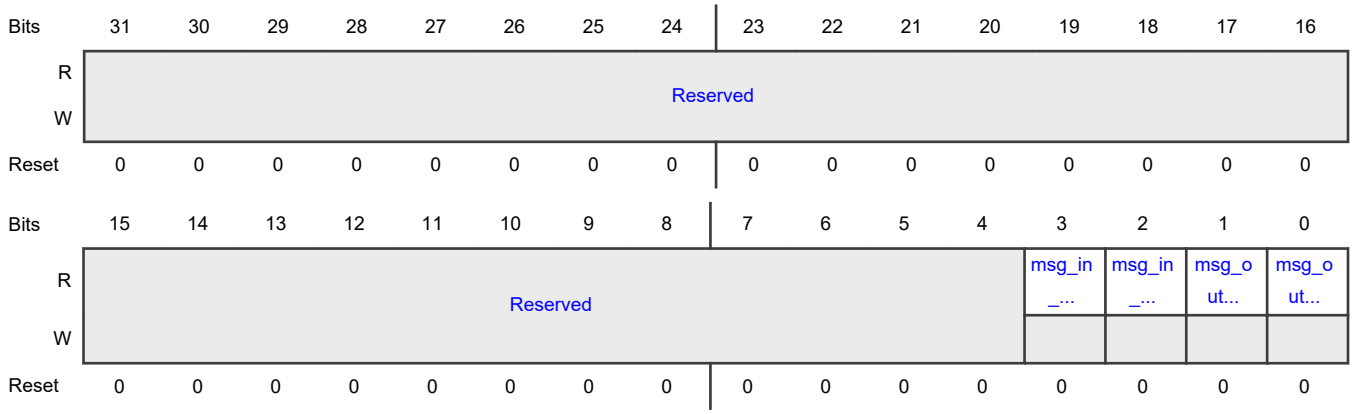
Offset

Register	Offset
HOST_MBOX_STATUS	6A0h

Function

This register is used to determine the status of messages sent and received by the host to and from VCPU. There are four status flags, one each for 64-bit message 0 and message 1 outgoing mailboxes and 64-bit message 0 and message 1 incoming mailboxes. These flags are set and cleared automatically by hardware - reads and writes of this register do not affect the status flags. However, reads and writes of the mailbox data registers do control the state of the status flags.

Diagram



Fields

Field	Function
31-4 —	- Reserved
3 msg_in_1_valid	msg_in_1_valid 64-bit inbox message 1 valid Indicates the validity/invalidity of the data in the host 64-bit message 1 inbox registers (VSPA_HOST_IN_1_MSB and VSPA_HOST_IN_1_LSB registers). 0b - Data in the 64-bit inbox registers is not valid. 1b - Data in the 64-bit inbox registers is valid.
2 msg_in_0_valid	msg_in_0_valid 64-bit inbox message 0 valid Indicates the validity/invalidity of the data in the host 64-bit message 0 inbox registers (VSPA_HOST_IN_0_MSB and VSPA_HOST_IN_0_LSB registers).

Table continues on the next page...

Table continued from the previous page...

Field	Function
	0b - Data in the inbox registers VSPA_HOST_IN_0_MSB and VSPA_HOST_IN_0_LSB is not valid. 1b - Data in the inbox registers VSPA_HOST_IN_0_MSB and VSPA_HOST_IN_0_LSB is valid
1 msg_out_1_valid	msg_out_1_valid 64-bit outbox message 1 valid Indicates if the 64-bit host message outbox 1 (VSPA_HOST_OUT_1_MSB and VSPA_HOST_OUT_1_LSB registers) is pending (unread by VCPU). 0b - No data is pending to be read in the 64-bit message 1 outbox registers. 1b - Data in the 64-bit message 1 outbox registers has not yet been read by VCPU.
0 msg_out_0_valid	msg_out_0_valid 64-bit outbox message 0 valid Indicates if the 64-bit host message outbox 0 (VSPA_HOST_OUT_0_MSB and VSPA_HOST_OUT_0_LSB registers) is pending (unread by VCPU). 0b - No data is pending to be read in the 64-bit message 0 outbox registers. 1b - Data in the 64-bit message 0 outbox registers has not yet been read by VCPU.

14.2.82 IPPU Control register (IPPUCONTROL)

Offset

Register	Offset
IPPUCONTROL	700h

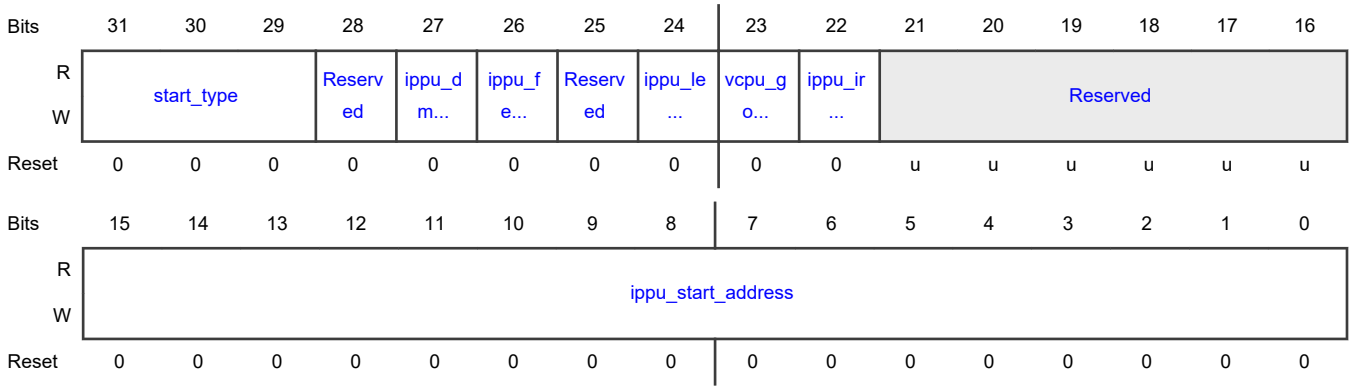
Function

The IPPU_CONTROL register must not be written until after the IPPU_ARG_BASE_ADDR register (if used) has first been written. Upon writing the IPPU_CONTROL register, all the command attributes from the IPPU_CONTROL register and the IPPU_ARG_BASE_ADDR register will be captured into the Command FIFO, and the IPPU will be activated. Note that Even writes from the VCPU core with no register bit mask bits set are considered writes and will activate the IPPU, so the IPPU_CONTROL register must be updated with a single write, not a series of masked writes.

The IPPU command fifo does not provide IPPU_ARG_BASE_ADDR register coherency between the VCPU and the host processor. Since VCPU and a host processor can both use the IPPU, and since there are NO separate sets of the IPPU_ARG_BASE_ADDR register for each of them, no coherency is provided, and the user must be causioned of such conflict.

All bits of the registers should be written since they are not initialized by default.

Diagram



Fields

Field	Function
31-29 start_type	Start type Start type 000b - Reserved 001b - dma_ippu_go_en - DMA GO command Enable to IPPU[1] 010b - fecu_ippu_go_en - FECU GO command Enable to IPPU[2] 011b - Reserved 100b - ippu_go_now[3] 101b - Reserved 110b - Reserved 111b - Reserved
28 —	- This bit is implemented, but has no effect
27 ippu_dma_trigger	ippu_dma_trigger Used to start a DMA transfer upon completion of IPPU code execution (ippu_done instruction is executed). 0b - Ignored 1b - Generate DMA start trigger upon completion of IPPU code execution (ippu_done instruction is executed).
26 ippu_fecu_trigger	ippu_fecu_trigger Used to start a FECU operation upon completion of the IPPU code execution (ippu_done instruction is executed). 0b - Ignored 1b - Generate FECU start operation upon completion of the IPPU code execution (ippu_done instruction is executed).

Table continues on the next page...

Table continued from the previous page...

Field	Function
25 —	- This bit is implemented, but has no effect
24 ippu_legacy_mem_addr	ippu_legacy_mem_addr IPPU mode: legacy_mem_addr 0b - Turn on the mode. IPPU DMEM address pointers (aX) point to 32-bit elements. 1b - Turn off the mode. IPPU DMEM address pointers (aX) point to 16-bit elements.
23 vcpu_go_en	IPPU vcpu_go_enable Enables the VCPU GO upon executing the IPPU done instruction. When set, it causes a VCPU_GO upon IPPU executing the done instruction by setting the ippu_go status bit in the VSPA_CONTROL register bit 1. Refer to VSPA_CONTROL for additional information. 0b - No Go request is generated when the IPPU executes a done instruction. 1b - Go request is generated when the IPPU executes a done instruction by setting the ippu_go status in the VSPA_CONTROL register.
22 ippu_irq_en	IPPU done interrupt enable Enables the IPPU interrupt upon executing the IPPU done instruction. When set, IPPU executes a done instruction will set the irq_pend_ippu_done status flag in the VSPA_STATUS register bit 1. This will assert the ippu_irq signal. Refer to VSPA_STATUS register for additional information. 0b - No interrupt is generated upon executing the IPPU done instruction. 1b - An interrupt is generated upon executing the IPPU done instruction
21-16 —	- Reserved
15-0 ippu_start_address	ippu_start_address ippu_start_address- IPPU Start Address IPPU operation starts when VCPU software or host software writes to the start_type field. The IPPU Loads the ippu_start_address and start code fetch and execution. The ippu_start_address field is used by the IPPU only in conjunction with the start_type field. The ippu_start_address field is not affected by reset. It can be written and read by the host software or VCPU software at any time. Settings: When writing the start_type field, this field is used as a start address for instruction fetch and execution.

1. The DMA has the capability to start the IPPU upon completion of the DMA transfer. When a DMA channel completes a transfer and the ippu_go_en bit is set (in the DMA control register), the DMA will signal the IPPU to "go".

When the DMA signals the IPPU to go and the IPPU starts the IPPU operation AND the IPPU enters the Running state. The IPPU clears the ippu_done and the 'dma_ippu_go_en' field is present at the top of the command fifo, the IPPU enters the IPPU into the Running state. The IPPU clears the ippu_done and ippu_aborted bits and sets the ippu_busy bit in the IPPU_STATUS register. The IPPU loads ippu_start_address and start code execution starting at this address. The IPPU resets the loop logic (state machine, and counters) and the return address stack, allowing clean start, even if previous reordering using execution terminated with abort, or the user neglected to clear the loop count or the Return Address Stack.

When the DMA signals the IPPU to go AND other entry/entries in the command fifo are set to 'dma_ippu_go_en', the IPPU marks the corresponding entry, so when this entry pops to the top of the command fifo, the IPPU will start code execution immediatley (consumed the dma signal to go). After poping the said entry to the top of the command fifo, the IPPU enters the Running state. The IPPU clears the ippu_done and the 'dma_ippu_go_en' field is present at the top of the command fifo, the IPPU enters the IPPU into the Running state. The IPPU clears the ippu_done and ippu_aborted bits and sets the ippu_busy bit in the IPPU_STATUS register. The IPPU Loads ippu_start_address and start code execution starting at this address.

The IPPU resets the loop logic (state machine, and counters) and the Return Address Stack, allowing clean start, even if previous Reordering Using execution terminated with abort, or the user neglected to clear the loop count or the Return Address Stack.

When the DMA signals the IPPU to go AND no entry in the command fifo is set to 'dma_ippu_go_en', the DMA signal is ignored.

2. The FECU has the capability to start the IPPU upon completion of the FECU operation. When a FECU completes an operation and the ippu_go_en bit is set(in the FECU control regisster), FECU will signal the IPPU to "go".

   When the FECU signals the IPPU to go and the IPPU starts the IPPU operation AND the IPPU enters the Running state. The IPPU clears the ippu_done and the 'fecu_ippu_go_en' field is present at the top of the comand fifo, the IPPU enters the IPPU into the Running state. The IPPU clears the ippu_done and ippu_aborted bits and sets the ippu_busy bit in the IPPU_STATUS register. The IPPU Loads ippu_start_address and start code execution starting at this address. The IPPU resets the loop logic (state machine, and counters) and the Return Address Stack, allowing clean start, even if previous Reordering Using execution terminated with abort, or the user neglected to clear the loop count or the Return Address Stack.

   When the FECU signals the IPPU to go AND other entry/entries in the command fifo are set to 'fecu_ippu_go_en', the IPPU marks the corresponding entry, so when this entry pops to the top of the command fifo, the IPPU will start code execution immediatley (consumed the fecu signal to go). Afteer poping the said entry to the top of the comand fifo, the IPPU enters the Running state. The IPPU clears the ippu_done and the 'fecu_ippu_go_en' field is present at the top of the comand fifo, the IPPU enters the IPPU into the Running state. The IPPU clears the ippu_done and ippu_aborted bits and sets the ippu_busy bit in the IPPU_STATUS register. The IPPU Loads ippu_start_address and start code execution starting at this address.

   The IPPU resets the loop logic (state machine, and counters) and the Return Address Stack, allowing clean start, even if previous Reordering Using execution terminated with abort, or the user neglected to clear the loop count or the Return Address Stack.

   When the FECU signals the IPPU to go AND no entry in the command fifo is set to 'fecu_ippu_go_en', the FECU signal is ignored.

3. Writing one into the ippu_go bit enters the IPPU into the Running state. The IPPU clears the ippu_done and ippu_aborted bits and sets the ippu_busy bit in the IPPU_STATUS register. The IPPU Loads ippu_start_address and start code execution starting at this address.

   The IPPU resets the loop logic (state machine, and counters) and the Return Address Stack, allowing clean start, even if previous Reordering Using execution terminated with abort, or the user neglected to clear the loop count or the Return Address Stack.

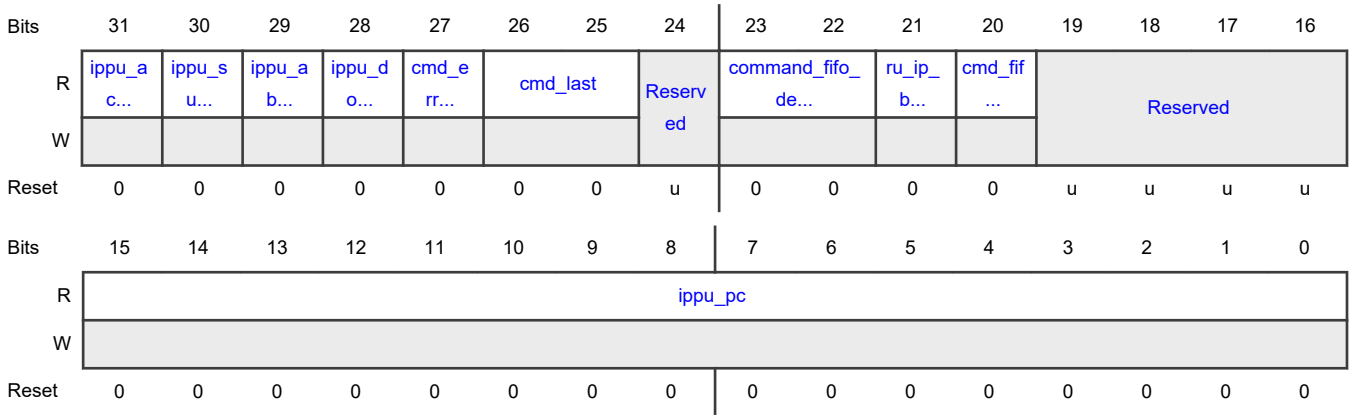
14.2.83 IPPU Status register (IPPUSTATUS)

Offset

Register	Offset
IPPUSTATUS	704h

Function
IPPU Status register

Diagram



Fields

Field	Function
31 ippu_active	ippu_active (busy_or_pending) 0b - Not busy and command fifo is empty 1b - Busy is set or there is at least one command pending in the FIFO.
30 ippu_suspended	ippu_suspended ippu_suspended - IPPU Suspended ippu_suspended indicates whether the IPPU core is suspended or not. This bit is set when the IPPU enters the Suspended state. It is cleared when the IPPU is not in the Suspended state. ippu_suspended can be read by the host software or VCPU software at any time. Writing this bit has no effect. 0b - IPPU not in Suspended state. 1b - IPPU in Suspended state.
29 ippu_aborted	ippu_aborted ippu_aborted - IPPU Aborted Indicates whether the IPPU core received the ippu_abort command while in the Running state. ippu_aborted is set when the IPPU is in the Running state and the ippu_abort bit in the IPPURC register is set. This bit will be cleared when the IPPU is activated (that is, when it is in idle state and receives an IPPU_GO which can happen on immediate start, FECU done start, DMA done start). ippu_aborted can be read by the host software or VCPU software at any time. Writing this bit has no effect. 0b - After Reset or IPPU in the Running state. 1b - ippu_abort bit was set while the IPPU was in the Running state.
28	ippu_done

Table continues on the next page...

Table continued from the previous page...

Field	Function
ippu_done	ippu_done - IPPU Done Indicates whether the IPPU core executed the done instruction. ippu_done is set when the IPPU executes the done instruction. ippu_done is cleared after reset and when the IPPU enters the Running state. ippu_done can be read by the host software or VCPU software at any time. Writing this bit has no effect. 0b - After Reset or IPPU busy 1b - IPPU executed the done instruction and in Idle sate.
27 cmd_error	command error command error - IPPU command fifo error Note: To avoid command fifo error, cmd_error, the host software or VCPU software must check the cmd_fifo_full bit is cleared before writing the IPPU Control register (IPPUCONTROL). Writing the IPPU Control register (IPPUCONTROL) when the cmd_fifo_full bit is set may result in loss of this command. 0b - After reset or no command error 1b - A write to the IPPU_CONTROL register was attempted when the command FIFO is full. Cleared by writing 1 to IPPURC[31].
26-25 cmd_last	command last source Indicates the last ippu_go command to the IPPU was received from the DMA, the IPPU or start_now by writing to the control register. 00b - After Reset or when ip_ippu_go_now is set. 01b - The fecu_ippu_go_en bit was set and the fecu_ippu_go signal was asserted. 10b - The dma_ippu_go_en bit was set and the dma_ippu_go signal was asserted. 11b - Reserved
24 —	- Reserved
23-22 command_fifo_depth	Command Fifo Depth command_fifo_depth is the number of pending or inprogress operations in the command FIFO. A maximum of 2 operations can be pending or in progress. Further attempts to start an operation will set command_error. 00b - Empty, no entries in the fifo 01b - One entry in the fifo 10b - Full, two or more entries in the fifo 11b - Reserved
21	IPPU Busy

Table continues on the next page...

Table continued from the previous page...

Field	Function
ru_ip_busy	ippu_busy indicates whether the IPPU core is Idle or busy executing code. ippu_busy is set when the IPPU enters the Running state. ippu_busy is cleared after reset and when the IPPU executes the done instruction. ippu_busy can be read by the host software or VCPU software at any time. Writing this bit has no effect. 0b - Idle state 1b - Running state
20 cmd_fifo_full	Command Fifo Full Note: The host software or VCPU software must check this bit is cleared before writing the IPPU Control register (IPPUCONTROL). Writing the IPPU Control register (IPPUCONTROL) when this bit is set may result is loss of this command. 0b - Command fifo is not full 1b - Command fifo is full; Command_depth equals 2.
19-16 —	- Reserved
15-0 ippu_pc	ippu_pc ippu_pc - IPPU Program Counter Reading this value provides approximate value of the pc at the time the IPPU_STATUS register is read. ippu_pc can be read by the host software or VCPU software at any time. Writing this field has no effect. Settings: ippu_pc provides approximate value of the pc at the time this register is read.

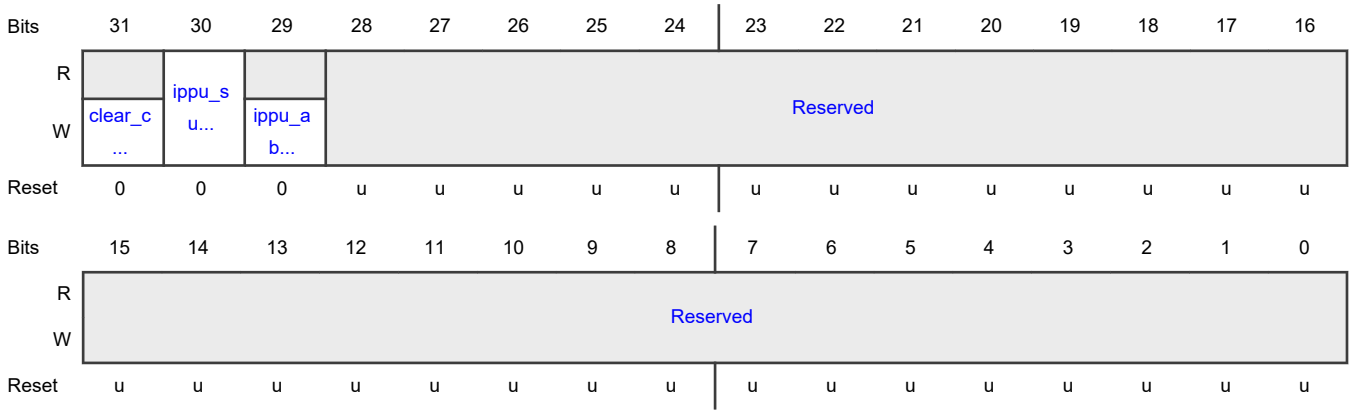
14.2.84 IPPU Run Control register (IPPURC)

Offset

Register	Offset
IPPURC	708h

Function
IPPU Run Control register

Diagram



Fields

Field	Function
31 clear_cmd_fifo_error	Clear command fifo error bit Write only bit. Reading always return zero. Writing one to this bit clears the IPPU command error bit in the status register. 0b - Do not clear 1b - Writing 1 clears the IPPU command error bit in the status register
30 ippu_suspend	IPPU suspend bit The ippu_suspend bit acts as stop/continue control for the IPPU. Setting this bit to one while the IPPU is in the Running state, will suspend the IPPU instruction fetch and execution. Instruction fetch and execution will resume when the ippu_suspend bit is cleared. The ippu_suspend bit has no effect on the IPPU while in Idle state. The ippu_suspend bit is reset to zero. It can be set, cleared and read by the host software or VCPU software at any time. 0b - IPPU runs normally 1b - While in Running state, the IPPU suspends instruction fetch and execution. No effect while in Idle state
29 ippu_abort	IPPU Abort The ippu_abort bit provides the host and the VCPU software a means to immediately terminate the IPPU operation, and all pending FIFO commands are cleared. When VCPU software or host software sets the ippu_abort bit while the IPPU is in the Running state, the IPPU immediately stops instruction fetch and execution, enters the Idle state, sets the ippu_aborted bit and clears the ippu_busy bit in the IPPU_STATUS register. The ippu_done bit is unaffected by setting the ippu_abort bit. When the abort bit is set, the IPPU clears all pending FIFO commands.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	NOTE The ippu_abort bit is reset to zero and is self clearing. Reading this bit will always return zero. 0b - Do nothing. 1b - While in Running state, the IPPU aborts instruction fetch and execution and enters the Idle state. No effect while in Idle state.
28-0 —	- Reserved

14.2.85 IPPU Arg Base Address register (IPPUARGBASEADDR)

Offset

Register	Offset
IPPUARGBASEADDR	70Ch

Function

NOTE

- The IPPU_ARG_BASE_ADDR register, if used, should be written before writing to the IPPU_CONTROL register.
- Reading this register returns the last written value, and not necessarily the active value.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	Reserved													ippu_arg_base_addr		
Reset	u	u	u	u	u	u	u	u	u	u	u	u	u	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	ippu_arg_base_addr															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-19 —	- Reserved
18-0 ippu_arg_base_addr	ippu_arg_base_addr ippu_arg_base_addr - IPPU Argument Base Address This field defines the starting address of a memory sector which is passed to the IPPU for argument transfer. After reset, this field is reset to zero (0x00000), and normally does not require to be written. The user software of VCPU software can change the address. The address is a 32 bit word address. Only the proper amount of the least significant bits (that is, 18 bits when the ippu_legacy_mem_addr is clear and 19 bits when ippu_legacy_mem_addr is set) are used by the IPPU. The rest of the bits are ignored. The ippu_arg_base_addr field is reset to zero. It can be read and written by the host software or VCPU software at any time. Settings: when the IPPU is accessing its arguments, this field is used as a start address for the argument memory space.

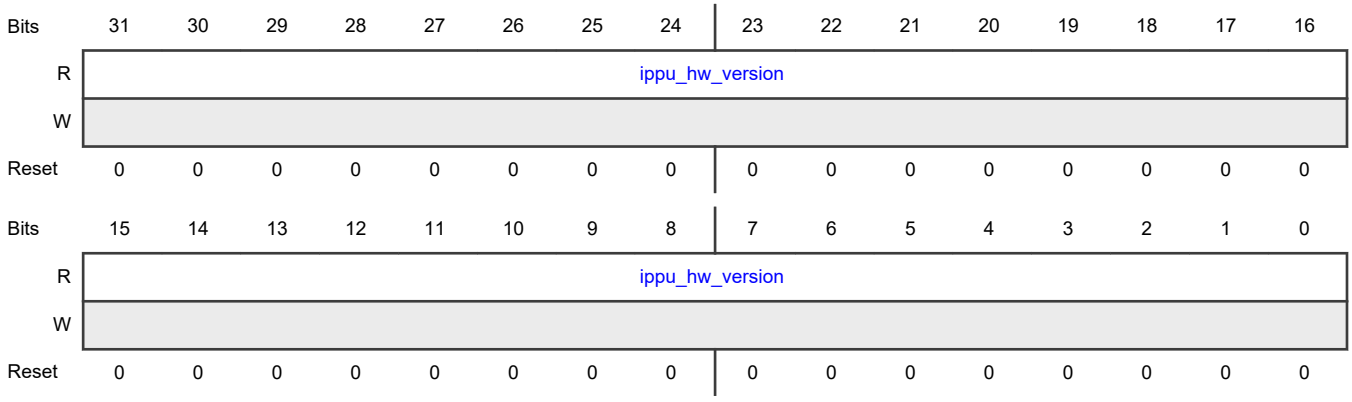
14.2.86 IPPU Hardware Version (IPPUHWVER)

Offset

Register	Offset
IPPUHWVER	710h

Function
IPPU Hardware Version

Diagram



Fields

Field	Function
31-0 ippu_hw_version	ippu_hw_version IPPU Hardware Version This field indicates the IPPU hardware version. ippu_hw_version can be read by the host software or VCPU software at any time. Writing this field has no effect.

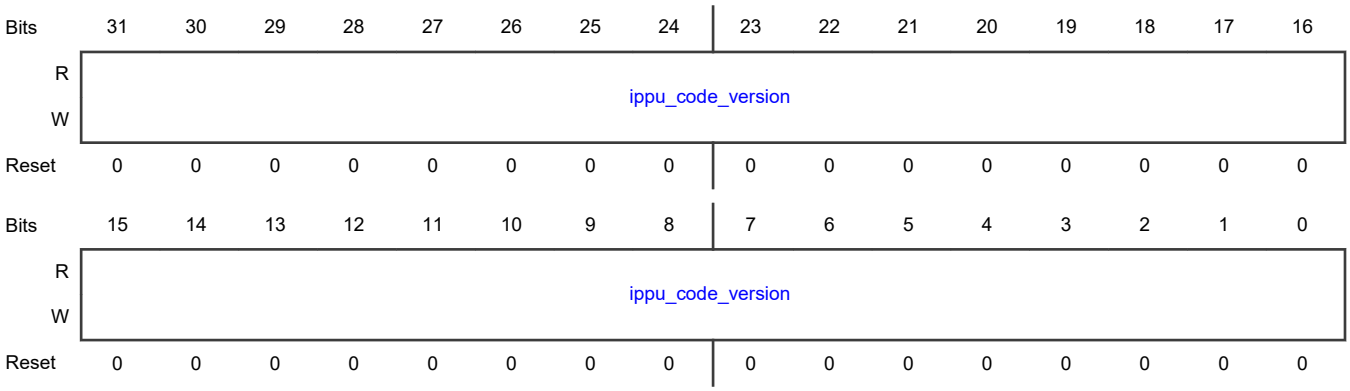
14.2.87 IPPU Software Version (IPPUSWVER)

Offset

Register	Offset
IPPUSWVER	714h

Function
IPPU Software Version

Diagram



Fields

Field	Function
31-0 ippu_code_version	ippu_code_version IPPU software code version This field can be used to indicate the version of the IPPU PRAM assembly code. • IPPU software configures this register during global initialization after download of a new IPPUPRAM image. • IPPUPRAM version numbers are non-zero. • This register does not affect the HW operation on the IPPU. It provides a place holder for the programmer.

Chapter 15
Debug registers

15.1 VSPA_DBG register descriptions

The VSP Debugger registers are defined in terms of byte addresses. The lowest two bits of the address are ignored so an access to byte address 0x01 returns the same data as an access to 0x00. Host writes to these registers are always assumed to be 32-bit accesses, so it is strongly suggested that all accesses to the registers be 32-bit aligned.

These registers can be reset in more than one way:

- The chip is powered up (power on reset)
- The debugger reset signal is asserted. Note that this resets only the debugger registers.

The memory map is segmented into several different regions as shown and described in the following table.

Addresses 0-7FF are a gateway to access the VSPA IP Bus memory-mapped registers via the VSPA Debug IP Bus. These registers are directly mapped into the lower portion of the debug memory map (address 0-7FF) and are read/written by performing accesses directly to those lower addresses. See VSPA register descriptions for details on the VSPA IP Bus registers. Registers at address 0x800 and above are only accessible by the VSPA Debug IP bus and are not accessible to the VCPU.

15.1.1 VSPA_DBG memory map

VSPA_DBG base address: 4_6000h

Offset	Register	Width (In bits)	Access	Reset value
800h	Global Debug Enable register (GDBEN)	32	RW	See description
804h	Debug Run Control register (RCR)	32	RW	See description
808h	Debug Run Control Status register (RCSTATUS)	32	RO	See description
83Ch	Debug Halt Action Control register (HACR)	32	RW	See description
840h	Debug Resume Action Control register (RACR)	32	RW	See description
870h	Debug VSP Architecture Visibility Address Pointer register (RAVAP)	32	RW	See description
874h	Debug VSP Architecture Visibility Fixed Data register (RAVFD)	32	RW	F001_F001h
878h	Debug VSP Architecture Visibility Incrementing Data register (RAVID)	32	RW	F001_F001h
87Ch	Debug Verification register (DVR)	32	RO	See description
880h - 88Ch	Debug Cross Trigger Out a Action Control registers (CTO0ACR - CTO3ACR)	32	RW	See description

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
900h	Debug Comparator Control and Status register (DC0CS)	32	RW	See description
904h	Debug Comparator a Data register (DC0D)	32	RW	See description
908h	Debug Comparator a Arm Action Control registers (C0AACR)	32	RW	See description
90Ch	Debug Comparator a Disarm Action Control registers (C0DACR)	32	RW	See description
910h	Debug Comparator a Trigger Action Control registers (C0TACR)	32	RW	See description
914h	Debug Comparator Control and Status register (DC1CS)	32	RW	See description
918h	Debug Comparator a Data register (DC1D)	32	RW	See description
91Ch	Debug Comparator a Arm Action Control registers (C1AACR)	32	RW	See description
920h	Debug Comparator a Disarm Action Control registers (C1DACR)	32	RW	See description
924h	Debug Comparator a Trigger Action Control registers (C1TACR)	32	RW	See description
928h	Debug Comparator Control and Status register (DC2CS)	32	RW	See description
92Ch	Debug Comparator a Data register (DC2D)	32	RW	See description
930h	Debug Comparator a Arm Action Control registers (C2AACR)	32	RW	See description
934h	Debug Comparator a Disarm Action Control registers (C2DACR)	32	RW	See description
938h	Debug Comparator a Trigger Action Control registers (C2TACR)	32	RW	See description
93Ch	Debug Comparator Control and Status register (DC3CS)	32	RW	See description
940h	Debug Comparator a Data register (DC3D)	32	RW	See description
944h	Debug Comparator a Arm Action Control registers (C3AACR)	32	RW	See description

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
948h	Debug Comparator a Disarm Action Control registers (C3DACR)	32	RW	See description
94Ch	Debug Comparator a Trigger Action Control registers (C3TACR)	32	RW	See description
950h	Debug Comparator Control and Status register (DC4CS)	32	RW	See description
954h	Debug Comparator a Data register (DC4D)	32	RW	See description
958h	Debug Comparator a Arm Action Control registers (C4AACR)	32	RW	See description
95Ch	Debug Comparator a Disarm Action Control registers (C4DACR)	32	RW	See description
960h	Debug Comparator a Trigger Action Control registers (C4TACR)	32	RW	See description
964h	Debug Comparator Control and Status register (DC5CS)	32	RW	See description
968h	Debug Comparator a Data register (DC5D)	32	RW	See description
96Ch	Debug Comparator a Arm Action Control registers (C5AACR)	32	RW	See description
970h	Debug Comparator a Disarm Action Control registers (C5DACR)	32	RW	See description
974h	Debug Comparator a Trigger Action Control registers (C5TACR)	32	RW	See description
978h	Debug Comparator Control and Status register (DC6CS)	32	RW	See description
97Ch	Debug Comparator a Data register (DC6D)	32	RW	See description
980h	Debug Comparator a Arm Action Control registers (C6AACR)	32	RW	See description
984h	Debug Comparator a Disarm Action Control registers (C6DACR)	32	RW	See description
988h	Debug Comparator a Trigger Action Control registers (C6TACR)	32	RW	See description
98Ch	Debug Comparator Control and Status register (DC7CS)	32	RW	See description

Table continues on the next page...

Table continued from the previous page...

Offset	Register	Width (In bits)	Access	Reset value
990h	Debug Comparator a Data register (DC7D)	32	RW	See description
994h	Debug Comparator a Arm Action Control registers (C7AACR)	32	RW	See description
998h	Debug Comparator a Disarm Action Control registers (C7DACR)	32	RW	See description
99Ch	Debug Comparator a Trigger Action Control registers (C7TACR)	32	RW	See description
E20h	Debug to VSP 32-bit Outbox register (OUT_32)	32	WO	0000_0000h
E24h	Debug to VSP 64-bit MSB Outbox register (OUT_64_MSB)	32	WO	0000_0000h
E28h	Debug to VSP 64-bit LSB Outbox register (OUT_64_LSB)	32	WO	0000_0000h
E2Ch	VSP to Debugger 32-bit Inbox register (IN_32)	32	RO	0000_0000h
E30h	VSP to Debugger 64-bit MSB Inbox register (IN_64_MSB)	32	RO	0000_0000h
E34h	VSP to Debugger 64-bit LSB Inbox register (IN_64_LSB)	32	RO	0000_0000h
E38h	Debugger to VSP Mailbox Status register (MBOX_STATUS)	32	RO	See description
F00h	Debug Parameter 0 Register (PARAM_0)	32	RO	See description
FD0h	Peripheral ID4 register (PIDR4)	32	RO	0000_0000h
FD4h	Peripheral ID5 register (PIDR5)	32	RO	0000_0000h
FD8h	Peripheral ID6 register (PIDR6)	32	RO	0000_0000h
FDCh	Peripheral ID7 register (PIDR7)	32	RO	0000_0000h
FE0h	Peripheral ID0 register (PIDR0)	32	RO	0000_0000h
FE4h	Peripheral ID1 register (PIDR1)	32	RO	0000_00E0h
FE8h	Peripheral ID2 register (PIDR2)	32	RO	0000_00F8h
FECh	Peripheral ID3 register (PIDR3)	32	RO	0000_00F0h
FF0h	Component ID0 register (CIDR0)	32	RO	0000_000Dh
FF4h	Component ID1 register (CIDR1)	32	RO	0000_00F0h
FF8h	Component ID2 register (CIDR2)	32	RO	0000_0005h
FFCh	Component ID3 register (CIDR3)	32	RO	0000_00B1h

15.1.2 Global Debug Enable register (GDBEN)

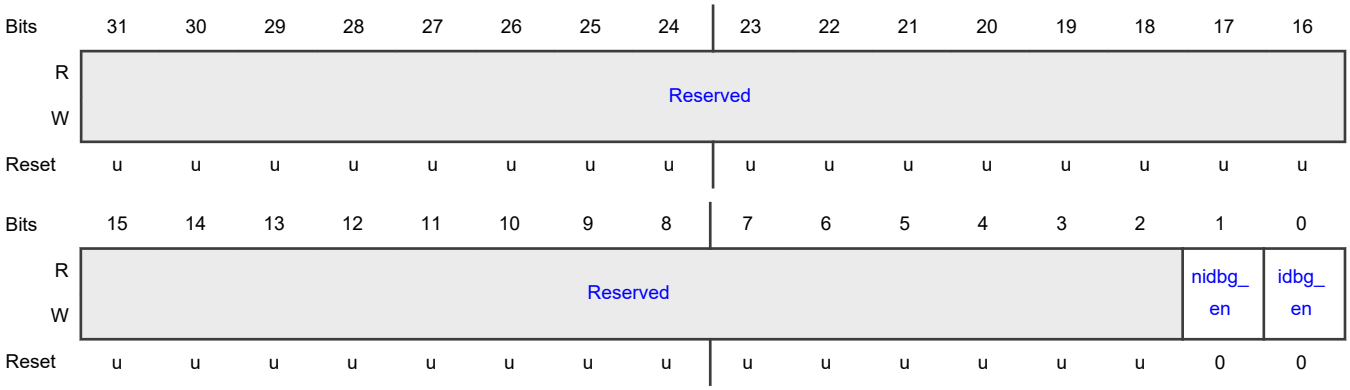
Offset

Register	Offset
GDBEN	800h

Function

This 32-bit read-write register is the overall (top-level) Global VSP Debug Enable control register.

Diagram



Fields

Field	Function
31-2 —	- Reserved
1 nidbg_en	nidbg_en Non-invasive debug mode enable This bit enables/disables non-invasive debug mode. Note: if this bit is zero, then trace is disabled, irrespective of the configuration of the other trace and action control registers. This enable bit will be overridden if the dbg_niden input is negated (driven to "0") by the external system due to security related issues. So trace can only be generated if both this bit AND the dbg_niden input are "1". 0b - Non-Invasive debug mode disabled (no trace Messages will be produced). 1b - Non-Invasive debug mode enabled (trace Messages will be output on occurrence of appropriate debug events - if configured to produce a trace message) if the dbg_niden input is driven to a "1".
0 idbg_en	idbg_en Invasive debug mode enable This is the enable for invasive (halting) debug mode.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Note: if this bit is zero, VSP cannot be halted irrespective of the configuration of the HACR register. This enable bit will be overridden if the dbg_dbgen input is negated (driven to "0") by the external system due to security related issues. So HALT can occur only if both this bit AND the dbg_dbgen input are "1". 0b - Halting debug mode is disabled (VSP cannot be halted). 1b - Halting debug mode is enabled (VSP can be halted and enter debug mode) if the dbg_dbgen input is driven to a "1".

15.1.3 Debug Run Control register (RCR)

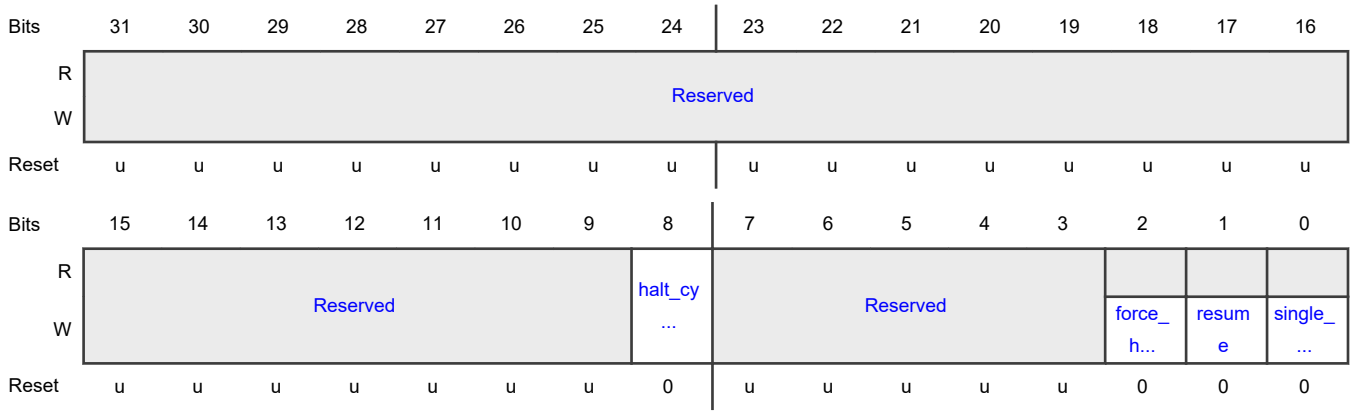
Offset

Register	Offset
RCR	804h

Function

This 32-bit read-write register is used for controlling invasive debug functions for halting all engines within the VSP (entering debug mode), single step (SS) a single VSP instruction at a time, and resume (exit from debug mode) via software. Note that in order for the VSP engine to halt and enter debug mode, the system must wait for the VSP engine to complete its current instruction and potentially wait for suspension of the DMA and IPPU units if either was actively processing at the time of the write to the Halt bit.

Diagram



Fields

Field	Function
31-9 —	- Reserved

Table continues on the next page...

Table continued from the previous page...

Field	Function
8 halt_cyc_counter	halt_cyc_counter This bit controls halting the cycle counter, VSPA_CYC_COUNTER_MSB/VSPA_CYC_COUNTER_LSB registers, when in halted state. 0b - Cycle counter continues counting when VSPA is in the halted state 1b - Cycle counter halts counting when VSPA is in the halted state
7-3 —	- Reserved
2 force_halt	force_halt Force halt - Cause VSP to enter halting debug mode When this bit is written to '1', VSP engine, IPPU engine, and DMA engines are halted in debug mode as soon as possible. Once all the units are halted, the dedicated output dbg_halt_ack will be asserted high. This command bit is write only - it will always read as "0". Settings: Write-only; Always reads as '0'.
1 resume	resume Resume normal operation - exit halting debug mode When this bit is written to '1', VSP exits halting debug mode and resumes execution where it left off prior to the halt. This command bit is write only - it will always read as "0". Settings: Write-only; Always reads as '0'.
0 single_step	single_step single step - execute a single instruction, then halt again When this bit is written to '1', VSP temporarily exits halting debug mode (with dbg_halt_ack de-asserting low), and executes a single VSP instruction (advances all VSP instructions in the VSP pipeline by one clock). Once the instruction completes, the VSP engine is again halted and halting debug mode is once again entered upon suspension of both the DMA and the IPPU units (at which point dbg_halt_ack will be asserted high). Note that single stepping through VSP 'done' instruction is not allowed. When the single stepping and the next instruction are done, VSP must receive a 'resume' command. Note that once the VCPU is halted, trace messages are not guarnteed to be accurate when resuming or single stepping, as debugging can be done in the halted mode. This command bit is write only - it will always read as "0". Settings: Write-only; Always reads as '0'.

15.1.4 Debug Run Control Status register (RCSTATUS)

Offset

Register	Offset
RCSTATUS	808h

Function

This 32-bit read-only status register is used to determine the reason for VSP being halted. Any or all these flags could potentially be asserted, indicating what caused the halt. These flags will self-clear upon exit from halting debug mode (resume).

WARNING

The halted flag (bit 13) will set after all the engines (VSP, IPPU, and DMA) have halted. The other status flags can set before the halt has completely halted all the engines. This is especially important when the debugger wishes to read VSP registers or access VSP memories via the architectural visibility portal. The portal can only work reliably when the engines are halted.

Note that this register can act as an "accumulator" during a halting debug session. After initially halting for one reason, other reasons for halting can also occur, and those reasons will accumulate in this register while halted or while single stepping.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	vcpu_go	vcpu_do...	swb	iit	prot_fa...	irq_inp...	Reserved		ctinn							
W																
Reset	0	0	0	0	0	0	u	u	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	dbg_hal...	forced_...	halted	Reserved					cmpn							
W																
Reset	0	0	0	u	u	u	u	u	0	0	0	0	0	0	0	0

Fields

Field	Function
31 vcpu_go	vcpu_go VCPU Go event When '1', this flag indicates that VSPA was halted due to occurrence of VCPU "Go" event. Note that this flag self-clears on exit from halting debug mode. 0b - VCPU "Go" NOT reason for VSPA being halted. 1b - VCPU "Go" one reason for VSPA being halted.
30	vcpu_done

Table continues on the next page...

Table continued from the previous page...

Field	Function
vcpu_done	VCPU done instruction When '1', this flag indicates that VSPA was halted due to execution of VCPU "Done" instruction. Note that this bit self-clears on exit from halting debug mode. 0b - VCPU "Done" NOT reason for VSPA being halted. 1b - VCPU "Done" one reason for VSPA being halted.
29 swb	swb Software breakpoint When '1', this flag indicates that VSPA was halted due to occurrence of Software Breakpoint. Note that this flag self-clears on exit from halting debug mode. 0b - Software Breakpoint NOT reason for VSPA being halted. 1b - Software Breakpoint one reason for VSPA being halted.
28 iit	iit VCPU illegal instruction When '1', this flag indicates that VSPA was halted due to execution of VCPU illegal instruction. Note that this bit self-clears on exit from halting debug mode. 0b - VCPU “illegal instruction” NOT reason for VSPA being halted. 1b - VCPU "illegal instruction" one reason for VSPA being halted.
27 prot_fault	prot_fault VCPU protection fault When '1', this flag indicates that VSPA encountered a protection fault. Note that this bit self-clears on exit from halting debug mode. 0b - VCPU "protection fault" NOT reason for VSPA being halted. 1b - VCPU "protection fault" one reason for VSPA being halted.
26 irq_input	irq_input VCPU interrupt request When '1', This flag indicates that VCPU was interrupted as result of the input signal “vcpu_irq” assertion. Note that this bit self-clears on exit from halting debug mode. 0b - VCPU “interrupt request” NOT reason for VSPA being halted. 1b - VCPU "interrupt request" one reason for VSPA being halted.
25-24 —	- Reserved
23-16	ctinn

Table continues on the next page...

Table continued from the previous page...

Field	Function
ctinn	dbg_trig_in_req[n] input triggered where n is 7-0 When '1', This flag indicates that VSPA was halted due to assertion of the dbg_trig_in_req[n] input. Note that this flag self-clears on exit from halting debug mode. 00000000b - dbg_trig_in_req[n] not a reason for VSPA being halted. 00000001b - dbg_trig_in_req[n] one reason for VSPA being halted.
15 dbg_halt_req	dbg_halt_req dbg_halt_req dedicated input triggered When '1', This flag indicates that VSPA was halted due to assertion of the dedicated halt request input. Note that this flag self-clears on exit from halting debug mode. 0b - dbg_halt_req not a reason for VSPA being halted. 1b - dbg_halt_req one reason for VSPA being halted.
14 forced_halt	forced_halt Debugger forced halt When '1', This flag indicates that the debugger set the force_halt control bit in the RCR register. Note that this flag self-clears on exit from halting debug mode. 0b - VSPA not halted due to debugger forced halt. 1b - VSPA halted due to debugger forced halt.
13 halted	halted Halted When '1', This flag indicates that the VCPU, IPPU, and DMA engines have all halted. The other bits in the register may be asserted before halt has completed, so it is important to observe the state of this flag to know when the halt has actually completed. This is especially important when the debugger wishes to read VSPA registers or access VSPA memories via the architectural visibility portal. The portal can only work reliably when all the engines are halted. Note that this bit self-clears on exit from halting debug mode. It also clears temporarily every time the single_step bit in the Debug Run Control Register (RCR) is set. After the single step command completes and VSPA has halted again, the halted flag will be set again. It should also be noted that this bit will be set only after the VCPU, IPPU, and the DMA engines have all halted. So, other status bits in this register may indicate a halt source prior to the completion of the halting of all the engines. The halted bit may also clear up to one cycle after the other status bits are cleared. **NOTE** This is self-clearing bit. Writing a one sets the bit. The bit will clear automatically and it will always read as zero. 0b - Halt action (if applicable) has not completed. 1b - Halt action has completed - all engines are halted.

Table continues on the next page...

Table continued from the previous page...

Field	Function
12-8 —	- Reserved
7-0 cmpn	cmpn Comparator n event occurred where n is 7-0 When '1', This flag indicates that VSPA was halted due to occurrence of Comparator n event. Note that this bit self-clears on exit from halting debug mode. 00000000b - Comparator n not reason for VSPA being halted. 00000001b - Comparator n one reason for VSPA being halted.

15.1.5 Debug Halt Action Control register (HACR)

Offset

Register	Offset
HACR	83Ch

Function

This 32-bit read-write register configures which events enable the action of halting VSPA. Note that the dedicated dbg_halt_req input is not configurable in this register. It will always halt VSPA when triggered.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	vcpu_go	vcpu_do...	swb	vcpu_il...	Reserved				ctinn							
Reset	0	0	0	0	u	u	u	u	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	Reserved								cmpn							
Reset	u	u	u	u	u	u	u	u	0	0	0	0	0	0	0	0

Fields

Field	Function
31 vcpu_go	vcpu_go Halt on VCPU Go

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Controls whether VSPA will be halted on "Go" events. 0b - Disable VSPA halt on occurrence of a VCPU "Go" event. 1b - Enable VSPA halt on occurrence of a VCPU "Go" event.
30 vcpu_done	vcpu_done Halt on VCPU Done Controls whether VSPa will be halted on "Done" events. 0b - Disable VSPA halt on occurrence of a VCPU "Done" event. 1b - Enable VSPA halt on occurrence of a VCPU "Done" event.
29 swb	swb Halt on VCPU software breakpoint Controls whether VSPA will be halted on Software Breakpoint events. 0b - Disable VSPA halt on occurrence of software breakpoint. 1b - Enable VSPA halt on occurrence of software breakpoint.
28 vcpu_illop	vcpu_illop Halt on illegal instruction trap. Controls whether VSPA will be halted on illegal instruction trap events. 0b - Disable VSPA halt on occurrence of a VCPU illegal instruction trap event. 1b - Enable VSPA halt on occurrence of a VCPU illegal instruction trap event.
27-24 —	- Reseved
23-16 ctinn	ctinn Halt on VSPA cross trigger input n where n is 7-0 Controls whether VSPA will be halted on dbg_trig_in_req[n] events. 00000000b - Disable VSPA halt on occurrence of cross trigger in n. 00000001b - Enable VSPA halt on occurrence of cross trigger in n.
15-8 —	- Reserved
7-0 cmpn	cmpn Halt on VSPA comparator n match where n is 7-0 Controls whether or not VSPA will be halted on comparator n match events. 00000000b - Disable VSPA halt on occurrence of comparator n match. 00000001b - Enable VSPA halt on occurrence of comparator n match.

15.1.6 Debug Resume Action Control register (RACR)

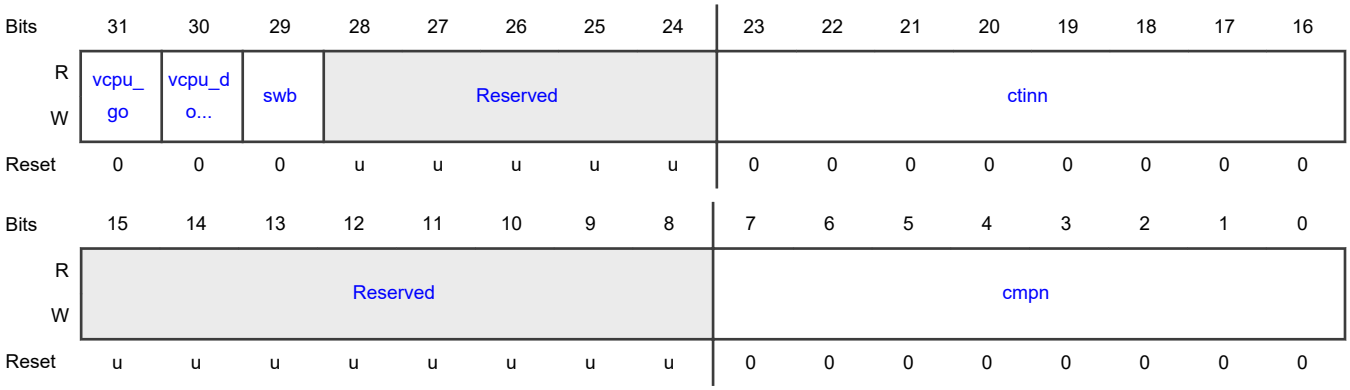
Offset

Register	Offset
RACR	840h

Function

This 32-bit read-write register configures which events are capable of causing VSPA to resume execution if currently halted (dbg_halt_ack output is asserted high). Note that the dedicated dbg_resume_req input is not configurable in this register. It will always resume VSPA when triggered.

Diagram



Fields

Field	Function
31 vcpu_go	vcpu_go Resume on VCPU Go Controls whether VSPA will resume (if halted) on "Go" events. 0b - Disable VSPA resume on occurrence of a VCPU "Go" event. 1b - Enable VSPA resume on occurrence of a VCPU "Go" event.
30 vcpu_done	vcpu_done Resume on VCPU Done Controls whether VSPA will resume (if halted) on "Done" events. 0b - Disable VSPA resume on occurrence of a VSP "Done" event. 1b - Enable VSPA resume on occurrence of a VSP "Done" event.
29 swb	swb Resume on VCPU software breakpoint Controls whether VSPA will resume (if halted) on Software Breakpoint events.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	0b - Disable VSPA resume on occurrence of software breakpoint. 1b - Enable VSPA resume on occurrence of software breakpoint.
28-24 —	- Reserved
23-16 ctinn	ctinn Resume on VSPA cross trigger input n where n is 7-0 Controls whether VSPA will resume (if halted) on dbg_trig_in_req[n] events. 00000000b - Disable VSPA resume on occurrence of cross trigger in n. 00000001b - Enable VSPA resume on occurrence of cross trigger in n.
15-8 —	- Reserved
7-0 cmpn	cmpn Resume on VSPA comparator n match where n is 7-0 Controls whether VSPA will resume (if halted) on comparator n match events. 00000000b - Disable VSPA Resume on occurrence of comparator n match. 00000001b - Enable VSPA Resume on occurrence of comparator n match.

15.1.7 Debug VSP Architecture Visibility Address Pointer register (RAVAP)

Offset

Register	Offset
RAVAP	870h

Function

This 32-bit read-write register is used for accessing VSP internal memory and architectural registers during halting debug mode, providing visibility to the register/memory state of the machine. This register provides the index of the item to be read/written, Refer to Table 99 for details. Note that the architectural registers and register file registers (R0-R7) can only be read, whereas the internal memories can be both read and written. Unless otherwise specified, the data returned for the VSP Architectural Register Portal are LSB justified in the data[31:0] field of the RAVFD or RAVID register.

Note that this visibility portal is disabled whenever invasive debug is disabled. So it is disabled when either the global invasive debug enable bit (idbg_en in the global debug enable register) is "0", OR the VSP input port dbg_dbgen is driven to "0".

Note also that a separate gateway exists for accessing the VSP IPbus registers. These registers are directly mapped into the lower portion of the debug memory map and are read/written by performing accesses directly to those lower addresses. Refer to VSPA register descriptions for details on the VSP IPbus registers.

NOTE

The VSP, IPPU, and DMA engines must be halted (or inactive) for program memory or register accesses via this resource to be successful. If a memory read or write is attempted while any of these units are active and NOT halted, the result may be non-deterministic. Data memory (DMEM and IPPU-DMEM) accesses will wait state the debug-IP bus until a free memory access cycle allows the debugger access to complete.

NOTE

A special precaution must be taken when using a debugger to access VCPU program memory while the VCPU is halted. When single stepping or resuming, the VCPU logic expects the state of the program memory's read data pins to be exactly as they were left prior to the halt.

If the debugger halts the VCPU and then reads or writes to VCPU program memory, the state of the memory read data pins can be altered. So prior to executing a single step or resuming, the debugger must re-read the same address as the VCPU last issued.

Visibility into the fetch PC is provided specifically for this purpose.

So, after a halt, if the debugger accesses VCPU program memory it must restore the read data state as follows:

1. Read the VCPU fetch PC
   - Write RAVAP to 0x00000000.
   - Read RAVFD and save the result into a variable (let's call the variable FPCADDR).
2. Read the VCPU program memory address
   - Write RAVAP to FPCADDR + 0x03000000.
   - Read RAVFD. The data read can be ignored.

After the restoration sequence above is completed, it is safe to single step or resume VCPU operation.

Table 99. VSP/IPPU Architectural Access Modes

a_mode[27:24]	Description
0	VSP Architectural Register list. Refer to Table 100. **NOTE** Data is LSB justified.
1	VSP DMEM Memory. a_index addresses 32-bit words Note return data format: data[31:0] =DMEM[a_index].
2	IPPU DMEM Memory. a_index addresses 32-bit words Note return data format: data[31:0] =DMEM[a_index].
3	VSP PRAM Memory. a_index addresses 32-bit words.

Table continues on the next page...

Table 99. VSP/IPPU Architectural Access Modes (continued)

a_mode[27:24]	Description
	PRAM is 64-bits wide. Two accesses are required to read/write the full 64 bits of data. When a_index[2] = 0, the access reads/writes 32 LSBs of PRAM data. When a_index[2] = 1, the access reads/writes the 32 MSBs of PRAM data. Writes to the PRAM must be made in pairs. The first write of the pair must be done with a_index[2]=0, and the second write must be done while a_index[2]=1. This procedure is necessary since the actual PRAM has only one write strobe, and when it's asserted all 64 bits of the PRAM are written. So, the first write's data is stored in a 32 bit buffer, and the second write triggers an actual PMEM write with data that is the concatenation of the buffered bits from the first write plus the raw data from the second write.
4	Reserved
5	IPPU PRAM Memory. a_index addresses 32-bit words. Note return data format: data[31:0] =IPPU PRAM [a_index].
6	VSP Register File (RF) R0-R7 registers. a_index addresses a 32-bit word starting with least significant element at R0. Note return data format: data[31:0] =Rn[a_index] (where n = a_index/32)
7	H Register.
8	IPPU Architectural Register list. Refer to Table 101. **NOTE** Data is LSB justified.
9	IPPU Register File (RF) R0-R1 registers. a_index addresses a 32-bit word starting with least significant element at R0. Note: return data format: data[31:0] =Rn[a_index] (where n = a_index/32)
10	Reserved

Table 100. VSP Architectural Register list (a_mode = 0)

a_index[18:2]	VSP Architectural Register
0	Prefetch PC. This is the address to PMEM.
1	Exec PC. This is the PC value of the instruction currently being executed.
2-13	General purpose registers G0 - G11.

Table continues on the next page...

Table 100. VSP Architectural Register list (a_mode = 0) (continued)

a_index[18:2]	VSP Architectural Register
14	MAG hardware stack pointer (SP) value.
15-18	MAG general purpose address registers a0 - a3.
19-20	MAG a2.min, MAG a2.max.
21-22	MAG a3.min, MAG a3.max.
23-25	MAG rr_mod_max, rr_mod_min, rr_mod_size.
26-41	MAG address storage registers as0 - as15.
42	MAG bit reversal, digit reversal mode register (br_dr_mode) and flip mode register (flip_mode). Note return data format: data[31:0] ={27'b0,br_dr_mode[1:0], flip_mode[2:0]}. data[31:0] ={29'b0,flip_mode[2:0]}.
43	CREG bits: real/complex, lite, AuOut (soft bits, hard bits, threshold bits). Note return data format: data[31:0] ={28'b0, mode, lite_mode, auout[1:0]}.
44	CREG bits (cc_update, previous cc bits, current cc bits). Note return data format: data[31:0] ={cc_update,19'b0,prev_cc[3:0], 4'b0, cc[3:0]}.
45	HW Loop depth pointer (for loops of size > 2).
46-53	HW Loop End Addresses 1-8 (top of loop stack is 1).
54-61	HW Loop Instruction Count 1-8 (top of loop stack is 1).
62-69	HW Loop Iteration Count 1-8 (top of loop stack is 1).
70	HW Loop (size 1 or 2) iteration count, in_loop_sz1 flag, in_loop_sz2 flag. Note return data format: data[31:0] ={14'b0, iter_count_loop1_or_2[15:0], in_loop1, in_loop2}.
71	Return Stack (RTS) depth pointer. Bit 31 is overflow, bit 30 is underflow.
72-87	Return Stack rts_storage 1-16 (top of loop stack is 1).
88	Source operands registers (S2mode, S1mode, S0mode, S0conj, S0chs). Note return data format: data[31:0] ={16'b0, S2mode[3:0], S1mode[3:0], 2'b0, S0conj, S0chs, S0mode[3:0]}.

Table continues on the next page...

Table 100. VSP Architectural Register list (a_mode = 0) (continued)

a_index[18:2]	VSP Architectural Register
89	RF rot_mode register. data[31:0] ={19'b0, lt_mode[4:0], 2'b0, rt_mode[5:0]
90	NCO 'k' parameter "next" value (not the current value).
91	NCO 'freq' parameter "next" value (not the current value).
92	NCO 'phase' "next" value (not the current value).
93	Reserved
94	Reserved
95	Reserved
96	Reserved
97	RAG S0 ptr register.
98	RAG S0 incr register.
99	Reserved
100	Reserved
101	Reserved
102	Reserved
103	RAG S1 ptr register.
104	RAG S1 incr register.
105	Reserved
106	Reserved
107	Reserved
108	Reserved
109	RAG S2 ptr register.
110	RAG S2 incr register.
111	Reserved
112	Reserved
113	Reserved

Table continues on the next page...

Table 100. VSP Architectural Register list (a_mode = 0) (continued)

a_index[18:2]	VSP Architectural Register
114	Reserved
115	RAG au ptr register.
116	RAG au incr register.
117	Reserved
118	Reserved
119	Reserved
120	Reserved
121	RAG mem ptr register.
122	RAG mem incr register.
123	min a0
124	max a0
125	min a1
126	max a1

Table 101. IPPU Architectural Register list (a_mode = 8)

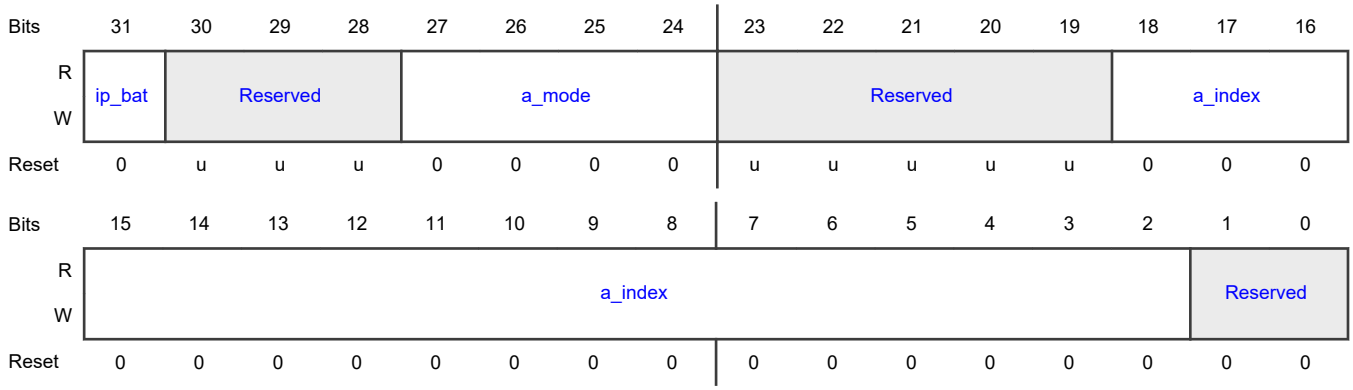
a_index[18:2]	VSP Architectural Register
0	Reserved
1	a0 - MAG general purpose address registers a0.
2	a1 - MAG general purpose address registers a1.
3	as0 - MAG address storage register as0.
4	as1 - MAG address storage register as1.
5	as2 - MAG address storage register as2.
6	as3 - MAG address storage register as3.
7	m0 - DMEM offset register 0 to access pointers in load/store instructions.
8	m1 - DMEM offset register 0 to access pointers in load/store instructions.
9	r_rd_ptr - Read pointer to a 4-bit element within the data register file.

Table continues on the next page...

Table 101. IPPU Architectural Register list (a_mode = 8) (continued)

a_index[18:2]	VSP Architectural Register
10	r_wr_ptr - Write pointer to a 4-bit element within the data register file.
11	elem_mask_ptr - Pointer to a bit within the mask register file.
12	mem_elem_rd_ptr - Pointer to a 4-bit element within a 32-bit element addressed by a0 or a1.
13	mem_elem_wr_ptr - Pointer to a 4-bit element within a 32-bit element addressed by a0 or a1.
14	a0_range_start - MAG a0 minimum value while in cyclic buffer mode.
15	a0_range_end - MAG a0 maximum value while in cyclic buffer mode.
16	a1_range_start - MAG a1 minimum value while in cyclic buffer mode.
17	a1_range_end - MAG a1 maximum value while in cyclic buffer mode.
18	z_flag - Zero flag used for conditional jmp/jsr/rts instructions.
19	Prefetch PC. This is the address to PMEM.
20	Exec IR. This is the IR (Instruction Register) value of the instruction currently being executed.
21	Exec PC. This is the PC value of the instruction currently being executed.
22	vindx_ptr - Vectorized Index pointer
23	mem_index0 - Memory Index register associated with a0
24	mem_index1 - Memory Index register associated with a1

Diagram



Fields

Field	Function
31 ip_bat	ip_bat VSP IPbus gateway - IP bridge access type. This bit specifies which identity should be used for the access to the VSP IPbus (host or VSP). Note the VSP IP Bus gateway is separate from the architecture visibility portal accessed via the a_mode and a_index fields of this register. The VSP IPbus bridge gateway is directly mapped into the lower portion of the debug IPbus memory map. This bit in no way applies to the architecture visibility portal. 0b - Access is executed as if it were from the host. 1b - Access is executed as if it were from VSP.
30-28 —	- Reserved
27-24 a_mode	a_mode Run control visibility - access mode. These bits select between the VSP internal memories and the architectural registers. Note that access to the RF registers, R0-R7 (a_mode = 6), are read-only. Also, they can only be read successfully if the VSP engine is in the halted state. Reads while VSP is not halted return unpredictable data. Note also that program memory access modes (PRAM, IPPU PRAM) will only function properly when the associated engine is halted or inactive. If accesses are attempted they may or may not be successful. This is because the engines control their program memory unless they are halted or not running. 0000b - VSP architectural registers mode - registers selected via the a_index field. 0001b - VSP DMEM mode. 0010b - VSP IPPU DMEM mode. 0011b - VSP PRAM mode. 0100b - Reserved. 0101b - VSP IPPU PRAM mode. 0110b - VSP RF register mode (R0-R7). 0111b - VSP H register mode. 1000b - IPPU architectural registers mode - registers selected via the a_index field. 1001b - IPPU RF register mode (R0-R1). 1010b - Reserved.
23-19 —	- Reserved
18-2 a_index	a_index Run control visibility - access index.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	These bits provide the index to access the data object based on the mode selected in the a_mode field (either architecture registers or one of the internal memories). Note that the two LSBs are always "0", restricting access to full 32-bit words only. Refer to Table 100 for the index value definitions for VSP register access. Refer to Table 101 for the index value definitions for IPPU register access. Settings: Index value for a given data object, depending on mode selected by a_mode field.
1-0 —	- Reserved

15.1.8 Debug VSP Architecture Visibility Fixed Data register (RAVFD)

Offset

Register	Offset
RAVFD	874h

Function

This 32-bit read-write register is used for accessing VSP internal memory and architectural registers during halting debug mode, providing visibility to the register/memory state of the machine. This register provides the data to/from the item to be read/written, depending on the configuration of RAVAP register. Note that the architectural registers can only be read, whereas the internal memories can be both read and written. Read/Write access to this register will NOT cause the RAVAP register to auto-increment.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	data															
Reset	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	data															
Reset	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1

Fields

Field	Function
31-0 data	data Run Control Visibility - Fixed Addressing Mode Data Register.

Table continues on the next page...

Field	Function
	Data to be written or read (depending on access type to this register) to the data object at address configured by the RAVAP register. Note accesses to this register will NOT cause the RAVAP to auto-increment. Settings: Data to read/write, depending on configuration of RAVAP register.

15.1.9 Debug VSP Architecture Visibility Incrementing Data register (RAVID)

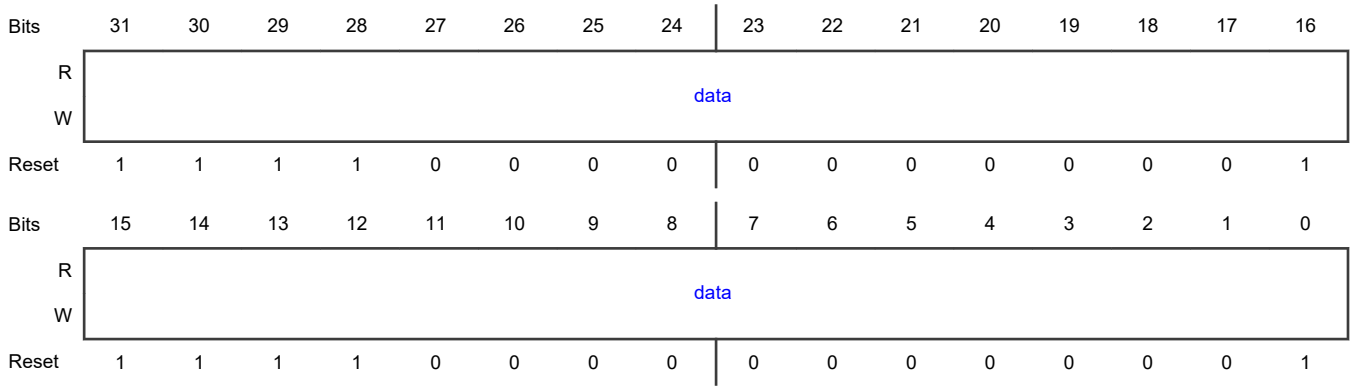
Offset

Register	Offset
RAVID	878h

Function

This 32-bit read-write register is used for accessing VSP internal memory and architectural registers during halting debug mode, providing visibility to the register/memory state of the machine. This register provides the data to/from the item to be read/written, depending on the configuration of RAVAP register. Note that the architectural registers can only be read, whereas the internal memories can be both read and written. Read/Write access to this register will cause the RAVAP register to auto-increment.

Diagram



Fields

Field	Function
31-0 data	data Run control visibility - Incrementing addressing mode data register. Data to be written or read (depending on access type to this register) to the data object at address configured by the RAVAP register. Note accesses to this register will cause the RAVAP to auto-increment. Settings: Data to read/write, depending on configuration of RAVAP register.

15.1.10 Debug Verification register (DVR)

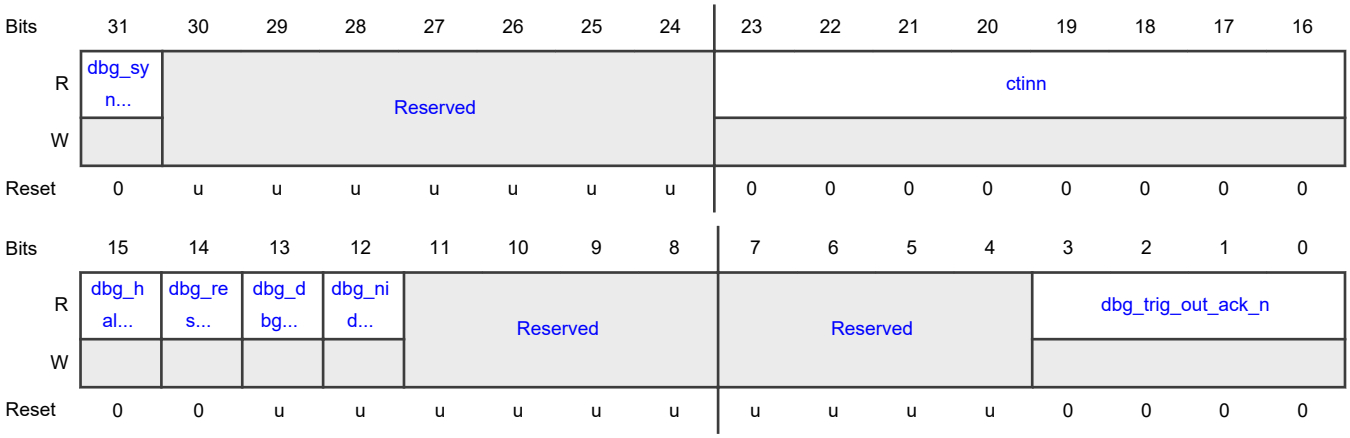
Offset

Register	Offset
DVR	87Ch

Function

This 32-bit read-only register is used for verifying the connectivity of the VSP debug inputs. The bits in this register simply reflect the value driven on the corresponding VSP debug block input pins.

Diagram



Fields

Field	Function
31 dbg_sync_req	dbg_sync_req Verification - Value driven on debug sync request pin. This bit reflects the value driven on the dbg_sync_req debug input pin for integration verification purposes. 0b - dbg_sync_req input driven low. 1b - dbg_sync_req input driven high.
30-24 —	- Reserved
23-16 ctinn	ctinn Verification - Value driven on cross-trigger in request input n pin where n is 7-0. This bit reflects the value driven on the dbg_trig_in_req[n] input pin for integration verification purposes. 00000000b - dbg_trig_in_req[n] input driven low. 00000001b - dbg_trig_in_req[n] input driven high.

Table continues on the next page...

Table continued from the previous page...

Field	Function
15 dbg_halt_req	dbg_halt_req Verification - Value driven on dbg_halt_req pin. This bit reflects the value driven on the dbg_halt_r*eq* input pin for integration verification purposes. 0b - dbg_halt_req input driven low. 1b - dbg_halt_req input driven high.
14 dbg_resume_req	dbg_resume_req Verification - Value driven on dbg_resume_req pin. This bit reflects the value driven on the dbg_resume_req input pin for integration verification purposes. 0b - dbg_resume_req input driven low. 1b - dbg_resume_req input driven high.
13 dbg_dbgen	dbg_dbgen Verification - Value driven on dbg_dbgen pin This bit reflects the value driven on the dbg_dbgen input pin for integration verification purposes. 0b - dbg_dbgen input driven low. 1b - dbg_dbgen input driven high.
12 dbg_niden	dbg_niden Verification - Value driven on dbg_niden pin This bit reflects the value driven on the dbg_niden input pin for integration verification purposes. 0b - dbg_niden input driven low. 1b - dbg_niden input driven high.
11-8 —	- Reserved
7-4 —	- Reserved
3-0 dbg_trig_out_ack_n	dbg_trig_out_ack_n Verification - Value driven on Cross-Trigger Out Ack input n pin where n is 3-0. This bit reflects the value driven on the dbg_trig_out_ack[n] input pin for integration verification purposes. 0000b - dbg_trig_out_ack[n] input driven low. 0001b - dbg_trig_out_ack[n] input driven high.

15.1.11 Debug Cross Trigger Out a Action Control registers (CTO0ACR - CTO3ACR)

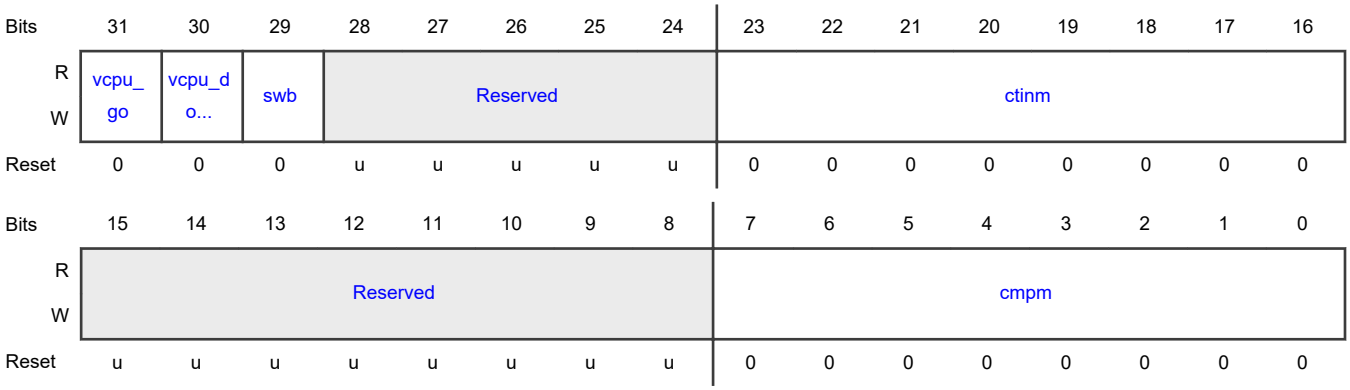
Offset

Register	Offset
CTO0ACR	880h
CTO1ACR	884h
CTO2ACR	888h
CTO3ACR	88Ch

Function

These 32-bit read-write registers configure which events enable the action of triggering cross-trigger out n, where n is 0 -3 (*dbg_trig_out_req[n]*) output. Note that when triggered on occurrence of the specific event(s) enabled in this register, the *dbg_trig_out_req[n]* output will remain asserted until the corresponding acknowledge input (*dbg_trig_out_ack[n]*) is received. At this point the cross-trigger will be negated (driven low).

Diagram



Fields

Field	Function
31 vcpu_go	vcpu_go Enable Cross-Trigger Out n triggered on a VCPU "Go" event *dbg_trig_out[n]*triggered high on occurrence of a VCPU "Go" event. Disabled when CTOnACR = 0x0000000. 0b - *dbg_trig_out[n]* NOT generated on occurrence of a VCPU "Go" event. 1b - *dbg_trig_out[n]* generated on occurrence of a VCPU "Go" event.
30 vcpu_done	vcpu_done Enable Cross-Trigger Out n triggered on execution of a VCPU "Done" instruction *dbg_trig_out[n]* triggered high on execution of a VCPU "Done" instruction.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Disabled when CTOnACR = 0x0000000. 0b - *dbg_trig_out[n]* NOT generated on execution of a VCPU "Done" instruction. 1b - *dbg_trig_out[n]* generated on execution of a VCPU "Done" instruction.
29 swb	swb Enable Cross-Trigger Out n triggered on Software Breakpoint event *dbg_trig_out[n]* triggered high on occurrence of Software Breakpoint event. Disabled when CTOnACR = 0x0000000. 0b - *dbg_trig_out[n]* NOT generated on occurrence of Software Breakpoint event. 1b - *dbg_trig_out[n]* generated on occurrence of Software Breakpoint event.
28-24 —	- Reserved
23-16 ctinm	ctinm Enable Cross-Trigger Out n triggered on occurrence of Cross-Trigger In m event where m is 7-0 dbg_trig_out[m] triggered high on occurrence of event dbg_trig_in_req[m]. Disabled when CTOnACR = 0x0000000. 00000000b - dbg_trig_out[n] NOT generated on occurrence of dbg_trig_in_req[m] event. 00000001b - dbg_trig_out[n] generated on occurrence of dbg_trig_in_req[m] event.
15-8 —	- Reserved
7-0 cmpm	cmpm Enable Cross-Trigger Out n triggered on occurrence of Comparator m match event where m is 7-0 dbg_trig_out[n] triggered high on occurrence of Comparator m match event. Disabled when CTOnACR = 0x0000000. 00000000b - *dbg_trig_out[n]* NOT generated on occurrence of Comparator m match event. 00000001b - *dbg_trig_out[n]* generated on occurrence of Comparator m match event.

15.1.12 Debug Comparator Control and Status register (DC0CS - DC7CS)

Offset

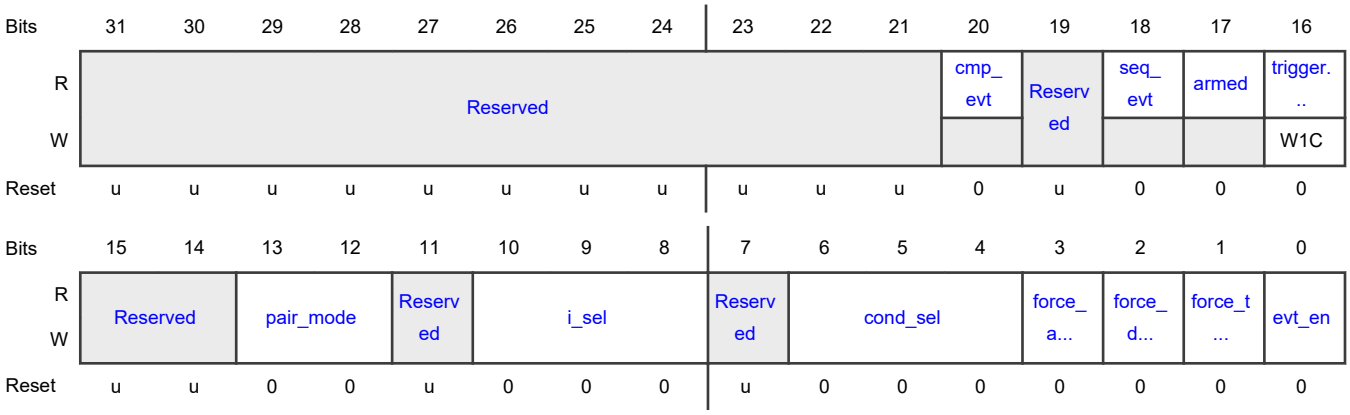
For a = 0 to 7:

Register	Offset
DCaCS	900h + (a × 14h)

Function

This 32-bit register is used to configure comparator n as well as provide status information on the state of the comparator (triggered, armed, sequential trigger state, state of comparator output).

Diagram



Fields

Field	Function
31-21 —	- Reserved
20 cmp_evt	cmp_evt Compare state status bit. This read-only status bit indicates the current state of the comparator output signal *cmp_evt*. 0b - The comparator output signal *cmp_evt* is cleared. 1b - The comparator output signal *cmp_evt* is asserted.
19 —	- Reserved
18 seq_evt	seq_evt Comparator sequential trigger status bit. This read-only status bit indicates the state of the sequential trigger bit in the comparator (seqtrig). The seqtrig bit is set on occurrence of an event (comparator trigger, trig_in, SWB, VSP "Go", VSP "Done") and the corresponding trigger enable bit is set to 1. The comparator can be triggered when either a Sequential Trigger or comparator match is detected. Note that the comparator must also be armed and enabled to be triggered. 0b - The comparator is not triggered. 1b - The comparator is triggered.
17 armed	armed Comparator armed status bit.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	This read-only status bit indicates the ARMed state of the comparator. The comparator can only be armed when either one or both of the following two conditions were met: 1) the force_arm bit in the corresponding Debug Comparator Control and Status Register (DCnCS) was set 2) the corresponding bit in the ARM Action Control Register (CnAACR) is set, and the corresponding activation event was active 0b - Comparator is not armed. 1b - Comparator is armed.
16 triggered	triggered Comparator triggered status bit. Anytime the comparator is triggered and the output signal cmp_evt is asserted high, this status bit is set to one. This bit remains asserted (sticky 1) until a value of "1" is written (W1C - write 1 to clear). The triggered bit is always readable. 0b - Comparator was not triggered since this bit was last cleared (or reset). 1b - Comparator was triggered one or more times since this bit was last cleared (or reset).
15-14 —	- Reserved
13-12 pair_mode	pair_mode Comparator pair mode configuration bits. These bits are used to configure whether the comparator is to be used individually, or paired with an adjacent comparator. If paired, the cmp_evt outputs are logically combined using either an "AND" or "OR" function. Only certain sets of comparators can be paired: The legal combinations are 0 and 1, 2 and 3, 4 and 5, 6 and 7. The pair_mode bits of the even numbered comparators should always be set to 0. The desired pairing mode should be programmed into the odd numbered comparator, and that comparator's output is the paired output. 00b - *comp_out* is the condition selected by *cond_sel* field. 01b - *comp_out* is the condition selected by *cond_sel* field AND *cmp_in* selected by *i_sel* field. 10b - *comp_out* is the condition selected by *cond_sel* field OR *cmp_in* selected by *i_sel* field. 11b - Reserved.
11 —	- Reserved
10-8 i_sel	i_sel Comparator input select configuration bits.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	These bits are used to select the inputs (address bus) for in_data values. Note - When the comparator is used to generate DTM trace messages, i_sel must be set to '001 - Select VSP DMEM'. 000b - Select none (constant zero). 001b - Select VSP DMEM. 010b - Select IPPU DMEM. 011b - Select VSP PMEM. 100b - Select VSP Peripheral Bus -IPbus. 101b - Select IPPU PMEM. 110-111b - Reserved.
7 —	- Reserved
6-4 cond_sel	cond_sel Comparator condition selects configuration bits. These bits are used to configure the condition under which a match occurs between the data_in (and attributes) input and the comp_data field in the corresponding comparator data register. 000b - Select none (comparator off). 001b - *data_in == data_reg*. 010b - *data_in != data_reg*. 011b - *data_in > data_reg*. 100b - *data_in >= data_reg*. 101b - *data_in < data_reg*. 110b - *data_in <= data_reg*. 111b - Reserved.
3 force_arm	force_arm Force comparator arm bit. When the force Arm bit is set to one, the comparator ARMED register is asserted (driven to 1). This allows the comparator to trigger on the next input event (cmp_evt_in, trig_in, and SWB).Note that the Event Enable bit must also be set to one to enable the comparator. 0b - Do not force the ARMED register to one. 1b - Force the ARMED register to one.
2 force_disarm	force_disarm Force comparator disarm bit.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	When the Force Disarm bit is set to one, the comparator ARMED register is negated (driven to 0). This disables the comparator from triggering on any of the next input events (cmp_evt_in, trig_in, and SWB). If both the Force Arm bit and Force Disarm bits are set at the same time, the ARMED register is cleared. 0b - Do NOT force the ARMED register to zero. 1b - Force the ARMED register to zero.
1 force_trig	force_trig Force comparator trigger bit. When the force trigger bit is set to one, the comparator output *cmp_evt* is asserted (driven to 1). The Event Enable bit must also be set to one to enable the comparator. The state of all other control bits and input signals are ignored. 0b - Do not force the comparator output signal *cmp_evt*. 1b - Assert (drive to 1) the comparator output signal *cmp_evt*.
0 evt_en	evt_en Comparator event enable bit. This is the main On/Off control of the comparator. When it is zero, the comparator is disabled and does not generate a comparator event. Note that the comparator register is fully accessible for both read and write regardless of the setting of this bit. 0b - Comparator is disabled. The comparator output *cmp_evt* is de-asserted and driven to 0. 1b - Comparator is enabled. A comparator event may be generated and output *cmp_evt* asserted when conditions defined by this comparator are met.

15.1.13 Debug Comparator a Data register (DC0D - DC7D)

Offset

For a = 0 to 7:

Register	Offset
DCaD	904h + (a × 14h)

Function

The comparator n (where n = 0-7) data registers each contain 17-bit data to be compared against the selected input data bus, and enables for attribute bit values (bits 29-24).

The comparator is considered matched (and the output asserts) when all of the following conditions are met:

1. the selected input data (according to the Condition Select field of the DCnCS register) matches the comp_data field.
2. a) the selected input access is a read and the rd attribute bit is set OR

   b) the selected input access is a write and the wr attribute bit is set
3. Any of the following a) - d) is true

a. The selected input is DMEM AND

there is a DMEM access from the FECU and the fecu attribute bit is set OR

there is a DMEM access from the IPPU and the IPPU attribute bit is set OR

there is a DMEM access from the VSP and the VSP attribute bit is set OR

there is a DMEM access from the DMA and the dma_host attribute bit is set

b. The selected input is IPPU DMEM AND

there is a DMEM access from the FECU and the fecu attribute bit is set OR

there is an IPPU DMEM access from the IPPU and the IPPU attribute bit is set OR

there is an IPPU DMEM access from the VSP and the VSP attribute bit is set OR

there is an IPPU DMEM access from the DMA and the dma_host attribute bit is set

c. The selected input is PMEM AND

there is a PMEM (PRAM) access from the VSP and the VSP attribute bit is set

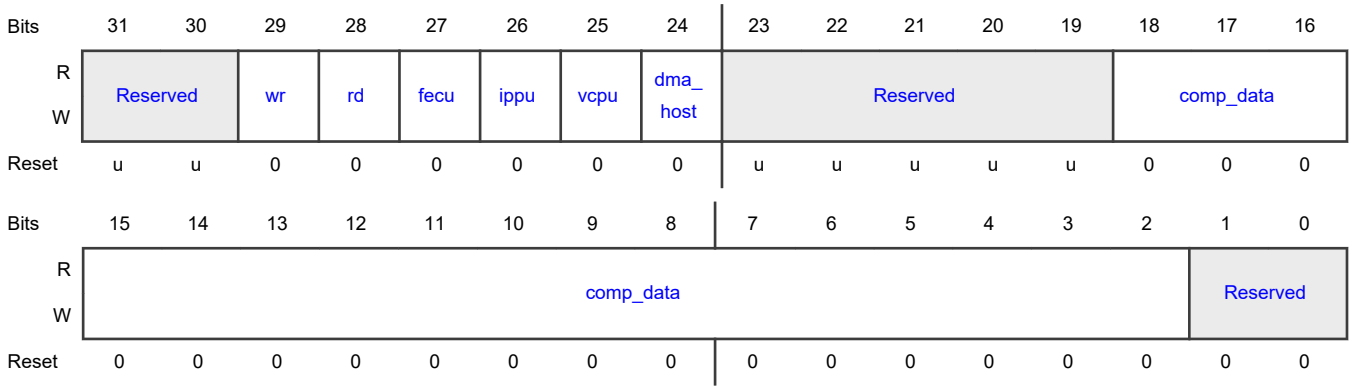
d. The selected input is VSP Peripheral Bus AND

there is a VSP peripheral bus access from the VSP and the VSP attribute bit is set OR

there is a VSP peripheral bus access from the host and the dma_host attribute bit is set

Note that the comparator will never match unless at least one attribute from the set {rd, wr} is set, and at least one attribute from the set {fecu, IPPU, VSP, dma_host}

Diagram



Fields

Field	Function
31-30 —	- Reserved
29 wr	wr Write access attribute bit. When the wr bit is set to one and there is a write transaction on the selected bus, the comparator is enabled to match the input data with the compare data field.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	0b - Do NOT enable match when a write access occurs. 1b - Enable match when a write access occurs.
28 rd	rd Read access attribute bit. When the rd bit is set to one and there is a read transaction on the selected bus, the comparator is enabled to match the input data with the compare data field. 0b - Do NOT enable match when a read access occurs. 1b - Enable match when a read access occurs.
27 fecu	fecu FECU access attribute bit. When the fecu bit is set to one and there is a FECU transaction on the selected bus, the comparator is enabled to match the input data with the compare data field. Note that this bit should only be set when DMEM or IPPU DMEM is being chosen by the i_sel bit field of the comparator control/status register. 0b - Do not enable match when an FECU access occurs. 1b - Enable match when an FECU access occurs.
26 ippu	ippu IPPU access attribute bit. When the ippu bit is set to one and there is an IPPU transaction on the selected bus, the comparator is enabled to match the input data with the compare data field. Note that this bit should only be set when DMEM or IPPU DMEM is being chosen by the i_sel bit field of the comparator control/status register. 0b - Do not enable match when an IPPU access occurs. 1b - Enable match when an IPPU access occurs.
25 vcpu	vcpu VCPU access attribute bit. When the vcpu bit is set to one and there is a VCPU engine transaction on the selected bus, the comparator is enabled to match the input data with the compare data field. 0b - Do not enable match when a VCPU engine access occurs. 1b - Enable match when a VCPU engine access occurs.
24 dma_host	dma_host DMA or host attribute bit. When the dma_host bit is set to one and there is a DMA or host transaction on the selected bus, the comparator is enabled to match the input data with the compare data field.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Note that the meaning of this bit depends on what bus is being chosen by the i_sel bit field of the comparator control/status register. It matches a DMA access to DMEM or IPPU DMEM. It matches a host access to the VSP IP Bus. 0b - Do not enable match when a DMA or host access occurs. 1b - Enable match when a DMA or host access occurs.
23-19 —	- Reserved
18-2 comp_data	comp_data Compare data value. The comp_data field contains 17-bit data to be compared against the selected input data bus. This field is aligned so as to make the address appear to be a byte address if one thinks of the LSB of this field as being bit 0 of the register. 17-bit compare data value.
1-0 —	- Reserved

15.1.14 Debug Comparator a Arm Action Control registers (C0AACR - C7AACR)

Offset

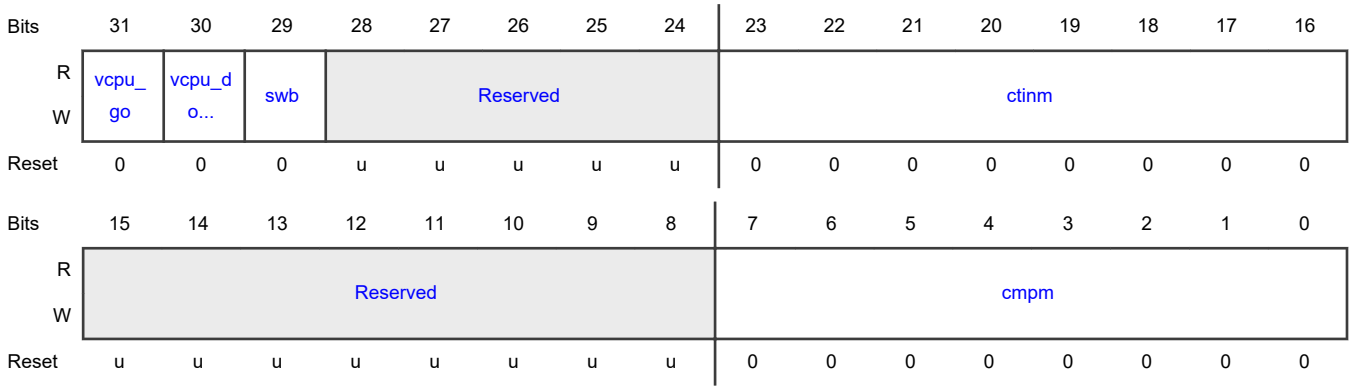
For a = 0 to 7:

Register	Offset
CaAACR	908h + (a × 14h)

Function

These 32-bit read-write registers configure which events enable the action of arming debug comparator n, where n is 0-7. Note that once enabled, a comparator must then be armed before it can trigger (assert) upon match of a pre-programmed set of attributes. The bits in this register enable which events are capable of arming debug comparator n. Refer to Debug module comparator and sequencer for detailed information on the VSPA debug sequencer comparator resource.

Diagram



Fields

Field	Function
31 vcpu_go	vcpu_go Enable arm of comparator n on a VCPU "Go" event Select whether comparator n will be armed as a result of a VCPU "Go" event. Note that comparator n must first be enabled prior to arming. 0b - Comparator n not armed on occurrence of a VCPU "Go" event. 1b - Comparator n armed on occurrence of a VCPU "Go" event.
30 vcpu_done	vcpu_done Enable arm of comparator n on execution of a VCPU "Done" instruction Select whether comparator n will be armed as a result of the execution of a VCPU "Done" instruction. Note that comparator n must first be enabled prior to arming. 0b - Comparator n not armed on execution of a VCPU "Done" instruction. 1b - Comparator n armed on execution of a VCPU "Done" instruction.
29 swb	swb Enable arm of comparator n on software breakpoint event Select whether comparator n will be armed as a result of the execution of an instruction tagged with a SWB. Note that comparator n must first be enabled prior to arming. 0b - Comparator n not armed on occurrence of Software Breakpoint event. 1b - Comparator n armed on occurrence of Software Breakpoint event.
28-24 —	- Reserved
23-16 ctinm	ctinm Enable arming of comparator n on occurrence of cross-trigger in m event where m is 7-0.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Select whether comparator n will be armed as a result of the detection of a dbg_trig_in_req[m] event. Note that comparator n must first be enabled prior to arming. 00000000b - Comparator n not armed on occurrence of *dbg_trig_in_req[m]* event. 00000001b - Comparator n armed on occurrence of *dbg_trig_in_req[m]* event.
15-8 —	- Reserved
7-0 cmpm	cmpm Enable arming of comparator n on occurrence of comparator m match event where m is 7-0 Select whether comparator n will be armed as a result of the detection of a comparator m match event. Note that comparator n must first be enabled prior to arming. 00000000b - Comparator n not armed on occurrence of Comparator m match event. 00000001b - Comparator n armed on occurrence of Comparator m match event.

15.1.15 Debug Comparator a Disarm Action Control registers (C0DACR - C7DACR)

Offset

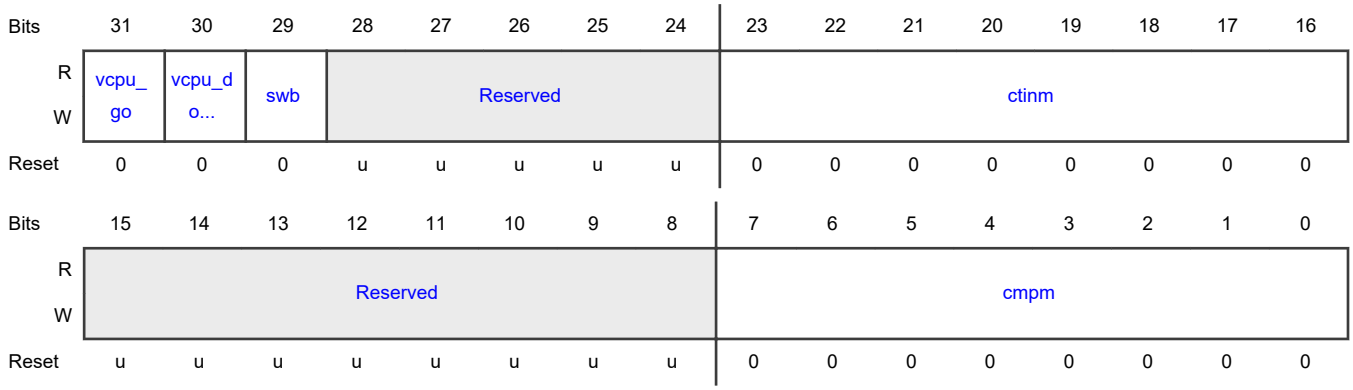
For a = 0 to 7:

Register	Offset
CaDACR	90Ch + (a × 14h)

Function

These 32-bit read-write registers configure which events enable the action of disarming debug comparator n, where n is 0-7. Note that a comparator must be enabled and armed for a disarm action to take effect. Once disarmed, a comparator cannot be triggered (even if it's match criteria is met).The bits in this register enable which events are capable of disarming debug comparator n. Refer to Debug module comparator and sequencer for detailed information on the VSPA debug Sequencer Comparator resource.

Diagram



Fields

Field	Function
31 vcpu_go	vcpu_go Enable Disarm of Comparator n on a VCPU "Go" event Select whether comparator n will be disarmed as a result of a VCPU "Go" event. 0b - Comparator n not disarmed on occurrence of a VCPU "Go" event. 1b - Comparator n disarmed on occurrence of a VCPU "Go" event.
30 vcpu_done	vcpu_done Enable Disarm of Comparator n on execution of a VCPU "Done" instruction Select whether comparator n will be disarmed as a result of the execution of a VCPU "Done" instruction. 0b - Comparator n not disarmed on execution of a VCPU "Done" instruction. 1b - Comparator n disarmed on execution of a VCPU "Done" instruction.
29 swb	swb Enable Disarm of Comparator n on Software Breakpoint event Select whether comparator n will be disarmed as a result of the execution of an instruction tagged with a SWB. 0b - Comparator n not disarmed on occurrence of Software Breakpoint event. 1b - Comparator n disarmed on occurrence of Software Breakpoint event.
28-24 —	- Reserved
23-16 ctinm	ctinm Enable disarming of comparator n on occurrence of cross-trigger in m event where m is 8-0 Select whether comparator n will be disarmed as a result of the detection of a dbg_trig_in_req[m] event. 00000000b - Comparator n not disarmed on occurrence of *dbg_trig_in_req[m]* event.

Table continues on the next page...

Table continued from the previous page...

Field	Function
	00000001b - Comparator n disarmed on occurrence of *dbg_trig_in_req[m]* event.
15-8 —	- Reserved
7-0 cmpm	cmpm Enable disarming of comparator n on occurrence of comparator m match event where m is 7-0 Select whether comparator n will be disarmed as a result of the detection of a Comparator m match event. 00000000b - Comparator n not disarmed on occurrence of Comparator m match event. 00000001b - Comparator n disarmed on occurrence of Comparator m match event.

15.1.16 Debug Comparator a Trigger Action Control registers (C0TACR - C7TACR)

Offset

For a = 0 to 7:

Register	Offset
CaTACR	910h + (a × 14h)

Function

These 32-bit read-write registers configure which events enable the action of triggering debug comparator n, where n is 0-7. Note that once enabled, a comparator must then be armed before it can trigger (assert) upon match of a pre-programmed set of attributes and/or on occurrence of other debug events. The bits in this register enable which events are capable of triggering debug comparator n. Refer to Debug module comparator and sequencer for detailed information on the VSPA debug Sequencer Comparator resource.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	vcpu_go	vcpu_do...	swb	Reserved					ctinm							
Reset	0	0	0	u	u	u	u	u	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	Reserved								cmpm							
Reset	u	u	u	u	u	u	u	u	0	0	0	0	0	0	0	0

Fields

Field	Function
31 vcpu_go	vcpu_go Enable Trigger of Comparator n on a VCPU "Go" event Select whether comparator n will be triggered as a result of a VCPU "Go" event. 0b - Comparator n not triggered on occurrence of a VCPU "Go" event. 1b - Comparator n triggered on occurrence of a VCPU "Go" event.
30 vcpu_done	vcpu_done Enable Trigger of Comparator n on execution of a VCPU "Done" instruction Select whether comparator n will be triggered as a result of the execution of a VCPU "Done" instruction. 0b - Comparator n not triggered on execution of a VCPU "Done" instruction. 1b - Comparator n triggered on execution of a VCPU "Done" instruction.
29 swb	swb Enable Trigger of Comparator n on Software Breakpoint event Select whether comparator n will be triggered as a result of the execution of an instruction tagged with a SWB. 0b - Comparator n not triggered on occurrence of Software Breakpoint event. 1b - Comparator n triggered on occurrence of Software Breakpoint event.
28-24 —	- Reserved
23-16 ctinm	ctinm Enable triggering of comparator n on occurrence of cross-trigger in m event where m is 7-0 Select whether comparator n will be triggered as a result of the detection of a dbg_trig_in_req[m] event. 00000000b - Comparator n not triggered on occurrence of *dbg_trig_in_req[m]* event. 00000001b - Comparator n triggered on occurrence of *dbg_trig_in_req[m]* event.
15-8 —	- Reserved
7-0 cmpm	cmpm Enable triggering of comparator n on occurrence of comparator m match event where m is 7-0 Select whether comparator n will be triggered as a result of the detection of a Comparator m match event. 00000000b - Comparator n not triggered on occurrence of Comparator m match event. 00000001b - Comparator n triggered on occurrence of Comparator m match event.

15.1.17 Debug to VSP 32-bit Outbox register (OUT_32)

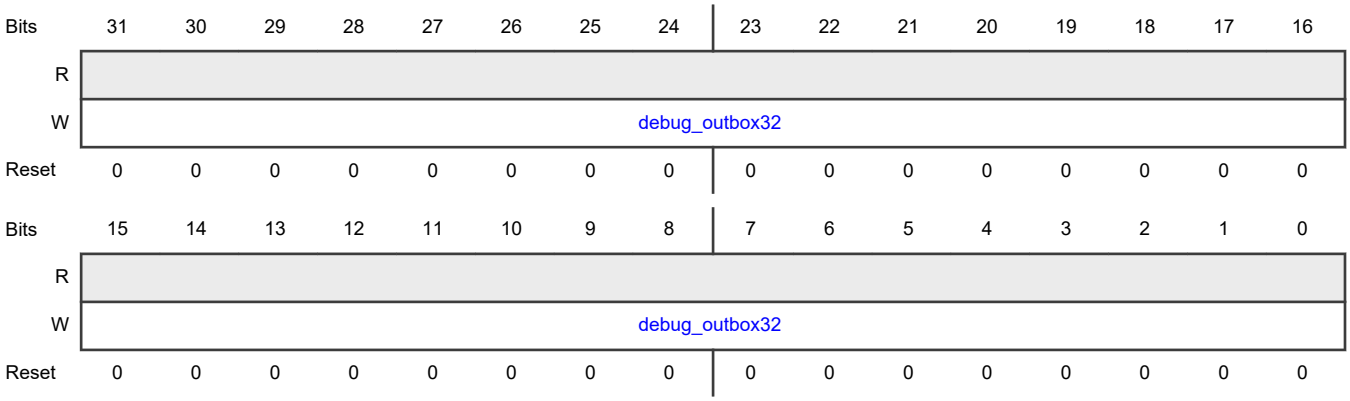
Offset

Register	Offset
OUT_32	E20h

Function

This register can be used by the debug unit to send a 32-bit message to VSP. Writes to this register cause a 32-bit message to be forwarded to VSP.

Diagram



Fields

Field	Function
31-0 debug_outbox32	debug_outbox32 Debug to VSP 32-bit Outbox Writes cause the value written to be sent to VSP. This register is write only - reads always return 0's. Data value to be sent to VSP inbox.

15.1.18 Debug to VSP 64-bit MSB Outbox register (OUT_64_MSB)

Offset

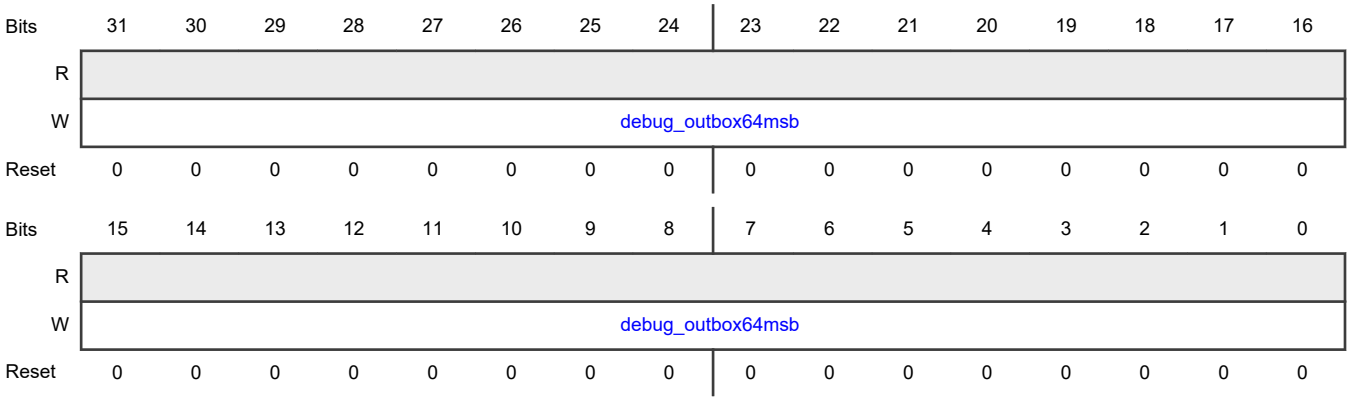
Register	Offset
OUT_64_MSB	E24h

Function

This register is part of the interface used by the debug unit to generate a 64-bit mail message directed to VSP's 64-bit inbox.

Note that writing this register is not required to send the 64-bit mail message. The mail message will be sent following a write to the DBG_OUT_64_LSB register.

Diagram



Fields

Field	Function
31-0 debug_outbox64msb	debug_outbox64msb Debug 64-bit output data (MSB) Writes update the 32 MSBs of a pending 64-bit message that will be sent to the VSP's 64-bit inbox. The mail message will be sent in its entirety only after a write to the DBG_OUT_64_LSB register. Note that this register does not have to be updated for each 64-bit mail message - if it is not updated, the last value written will be reused when the mail message is sent. This register is write only - reads always return 0's. Settings: MSB 32-bit data value to be sent to VSP's 64-bit inbox.

15.1.19 Debug to VSP 64-bit LSB Outbox register (OUT_64_LSB)

Offset

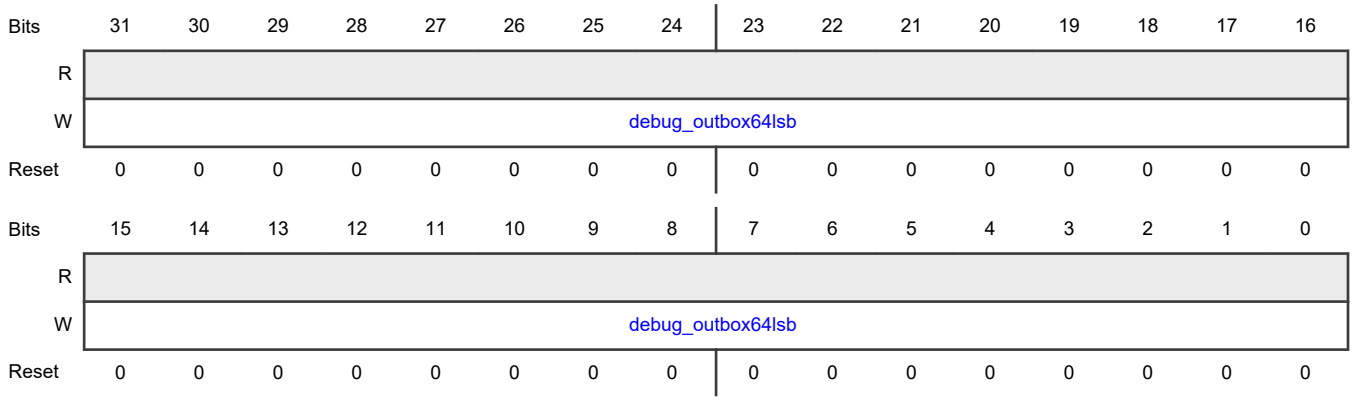
Register	Offset
OUT_64_LSB	E28h

Function

This register is part of the interface used by the debug unit to generate a 64-bit mail message directed to VSP's 64-bit inbox.

Note that writing to this register alone will trigger the delivery of the 64-bit mail message to VSP. The mail message will contain 32 MSBs from the DBG_OUT_64_MSB register plus all 32 bits from this register.

Diagram



Fields

Field	Function
31-0 debug_outbox64lsb	debug_outbox64lsb Debug 64-bit output data (LSB) Writes cause the immediate delivery of a 64-bit mail message to the VSP 64-bit mail inbox. The 32 LSBs of the message come from the data written to this register; the 32 MSBs come for the value contained in the DBG_OUT_64_MSB register. This register is write only - reads always return 0's. Settings: LSB 32-bit data value to be sent to VSP's 64-bit inbox.

15.1.20 VSP to Debugger 32-bit Inbox register (IN_32)

Offset

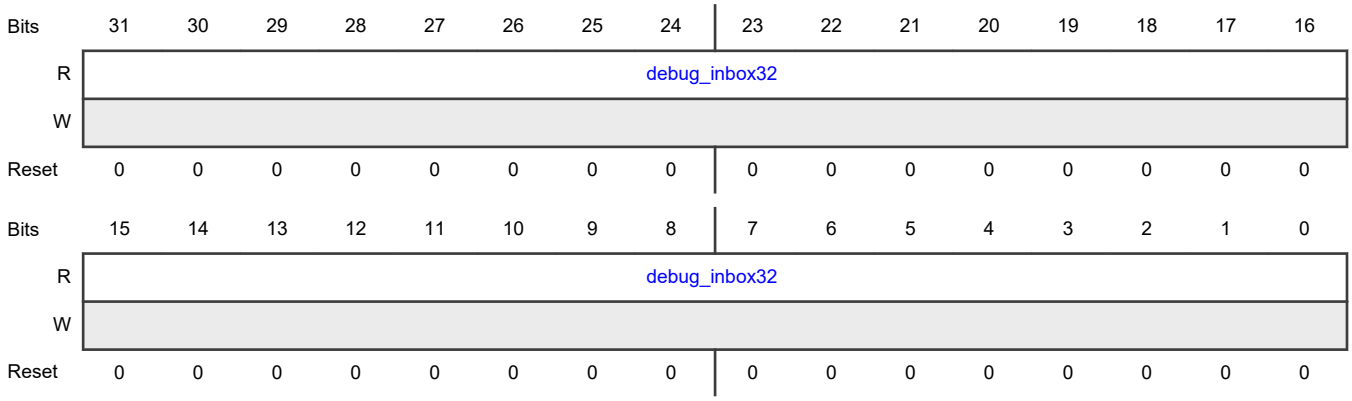
Register	Offset
IN_32	E2Ch

Function

This register is the Debug unit incoming mailbox for 32-bit messages from VSP. It can be used by the debug unit to read a 32-bit message from the VSP. The validity of the data in this register should be checked before it is read. The state of the 32-bit_msg_in_valid flag in the Debug-VSP Mailbox Status register conveys validity information.

Reads of this register will automatically clear both the 32-bit_msg_in_valid flag in the Debug-VSP Mailbox Status register (on the debug IP bus) and the 32-bit_msg_out_valid flag in the VSP-Debug Mailbox Status register (on the VSP IP bus).

Diagram



Fields

Field	Function
31-0 debug_inbox32	debug_inbox32 VSP to Debugger 32-bit Inbox Reads the 32-bit mail message sent by VSP. This register is read only - writes have no effect. Settings: Reads 32-bit data value sent by VSP outbox.

15.1.21 VSP to Debugger 64-bit MSB Inbox register (IN_64_MSB)

Offset

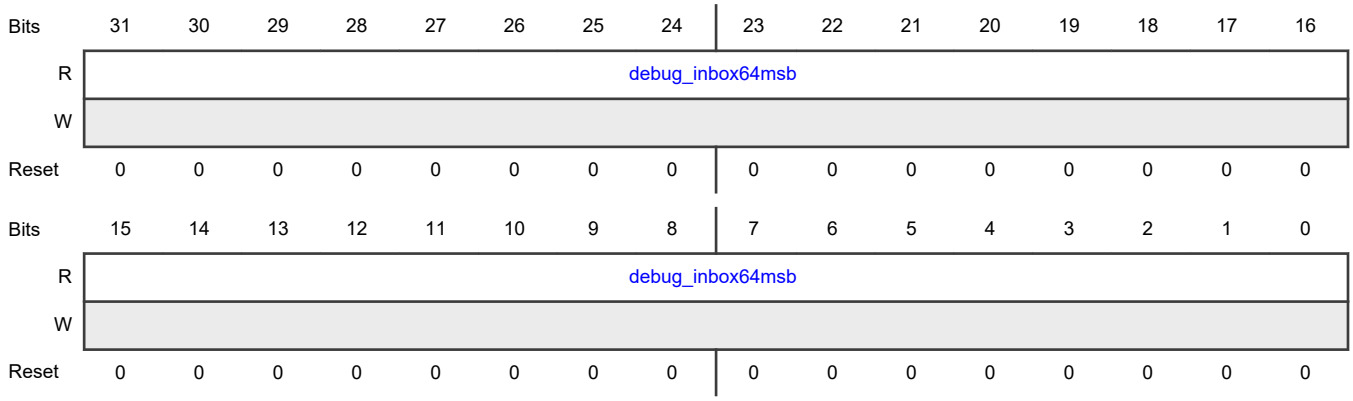
Register	Offset
IN_64_MSB	E30h

Function

This register is part of the Debug unit incoming mailbox for 64-bit messages from VSP. It can be used by the debugger to read the 32 MSBs of a 64-bit message from VSP. The validity of the data in this register should be checked before it is read. The state of the 64-bit_msg_in_valid flag in the Debug-VSP Mailbox Status register conveys validity information.

Reads of this register do not affect the 64-bit_msg_in_valid flag in the Debug-VSP Mailbox Status register or the 64-bit_msg_out_valid flag in the VSP-Debug Mailbox Status register on the VSP IP bus. Those flags are cleared by reads of the VSP to Debug 64-bit LSB Inbox Register.

Diagram



Fields

Field	Function
31-0 debug_inbox64msb	debug_inbox64msb Debug 64-bit input data (MSB) Reads the 32 MSBs of the 64-bit mail message sent by VSP. This register is read only - writes have no effect. Settings: Reads return 32 MSBs of the 64-bit data value sent via VSP's 64-bit outbox.

15.1.22 VSP to Debugger 64-bit LSB Inbox register (IN_64_LSB)

Offset

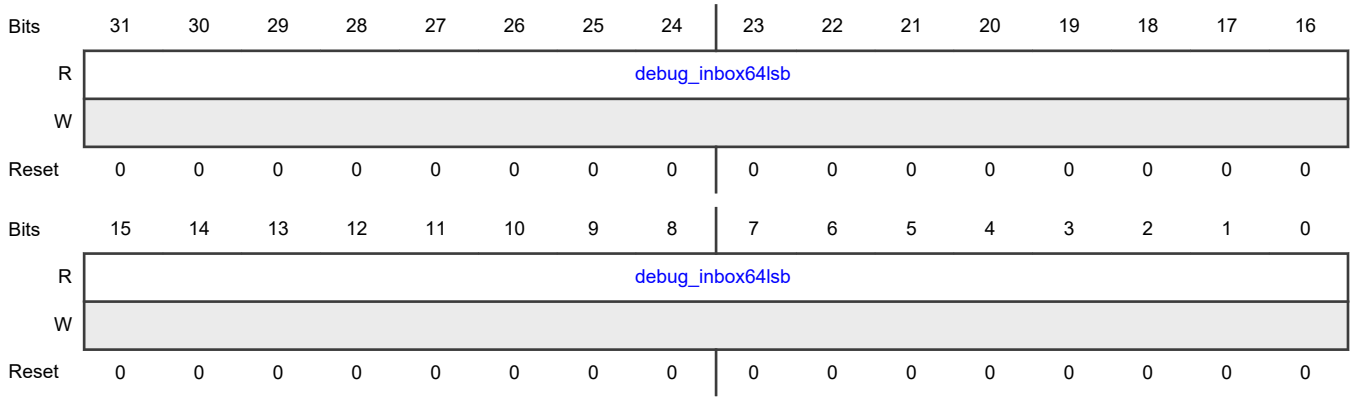
Register	Offset
IN_64_LSB	E34h

Function

This register is part of the Debug unit incoming mailbox for 64-bit messages from VSP. It can be used by the debug unit to read the 32 LSBs of a 64-bit message from VSP. The validity of the data in this register should be checked before it is read. The state of the 64-bit_msg_in_valid flag in the Debug-VSP Mailbox Status register conveys validity information.

Reads of this register will automatically clear the 64-bit_msg_in_valid flag in the Debug-VSP Mailbox Status register and the 64-bit_msg_out_valid flag in the VSP-Debug Mailbox Status register on the VSP IP bus.

Diagram



Fields

Field	Function
31-0 debug_inbox64lsb	debug_inbox64lsb Debug 64-bit input data (LSB) Reads the 32 LSBs of the 64-bit mail message sent by VSP. This register is read only - writes have no effect. Settings: Reads return 32 LSBs of the 64-bit data value sent via VSP's 64-bit outbox.

15.1.23 Debugger to VSP Mailbox Status register (MBOX_STATUS)

Offset

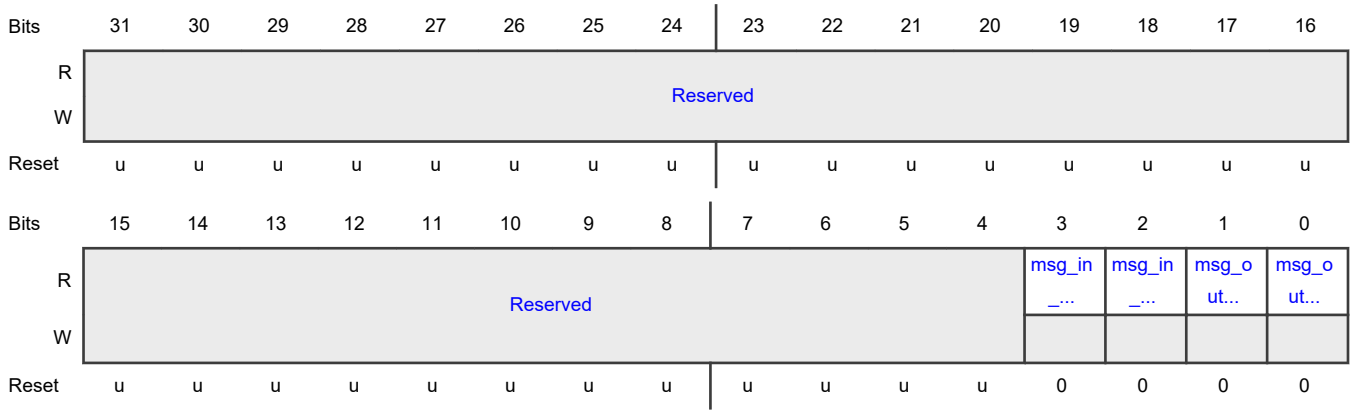
Register	Offset
MBOX_STATUS	E38h

Function

This register is used to determine the status of messages sent and received by the Debug unit to and from VSP. There are four status flags, one each for the 32 and 64-bit outgoing mailboxes and the 32 and 64-bit incoming mailboxes.

These flags are set and cleared automatically by hardware - reads and writes of this register do not affect the status flags. However, reads and writes of the mailbox data registers do control the state of the status flags.

Diagram



Fields

Field	Function
31-4 —	- Reserved
3 msg_in_valid_64bit	msg_in_valid_64bit 64-bit message inbox valid Shows the validity/invalidity of the data in the Debug 64-bit message inbox registers (VSP to Debugger 64-bit MSB Inbox register and VSP to Debugger 64-bit LSB Inbox register). 0b - Data in the 64-bit inbox registers is NOT valid. 1b - Data in the 64-bit inbox registers is valid.
2 msg_in_valid_32bit	msg_in_valid_32bit 32-bit message inbox valid Shows the validity/invalidity of the data in the debugger 32-bit message inbox register (VSP to Debugger 32-bit Inbox register). 0b - Data in the 32-bit inbox register is NOT valid. 1b - Data in the 32-bit inbox register is valid.
1 msg_out_valid_64bit	msg_out_valid_64bit 64-bit message outbox valid Shows whether a pending (unread by the debugger) message is in the debug 64-bit message outbox registers (Debugger to VSP 64-bit MSB outbox register and Debugger to VSP 64-bit LSB Outbox register). 0b - No unread data is in the 64-bit outbox registers. 1b - Data in the 64-bit outbox registers has not yet been read by VSP.
0 msg_out_valid_32bit	msg_out_valid_32bit 32-bit message outbox valid

Table continues on the next page...

Table continued from the previous page...

Field	Function
	Shows whether a pending (unread by the debugger) message is in the Debug 32-bit message outbox register (Debugger to VSP 32-bit Outbox register). 0b - No unread data is in the 32-bit outbox register. 1b - Data in the 32-bit outbox register has not yet been read by VSP.

15.1.24 Debug Parameter 0 Register (PARAM_0)

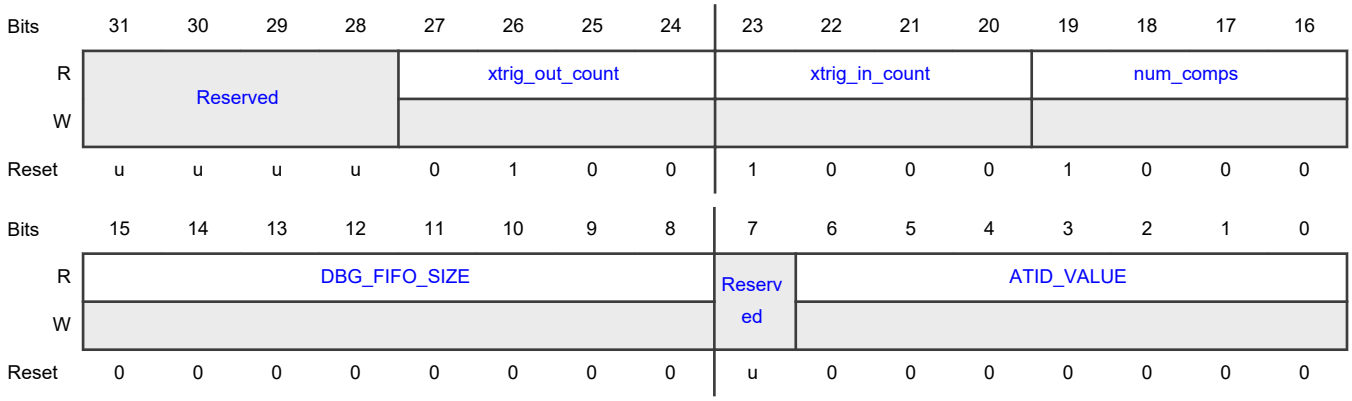
Offset

Register	Offset
PARAM_0	F00h

Function

Debug Parameter 0 Register

Diagram



Fields

Field	Function
31-28 —	- Reserved
27-24 xtrig_out_count	xtrig_out_count Cross trigger output count. These status bits indicate the number of cross trigger outputs supported by this instance of VSPA. Settings: Reads return the number of cross trigger outputs supported by this instance of VSPA. Writes have no affect.

Table continues on the next page...

Table continued from the previous page...

Field	Function
23-20 xtrig_in_count	xtrig_in_count Cross trigger input count. These status bits indicate the number of cross trigger inputs supported by this instance of VSPA. Settings: Reads return the number of cross trigger inputs supported by this instance of VSPA. Writes have no affect.
19-16 num_comps	num_comps Number of debug comparators (comparator count). Debug comparator count.These status bits indicate the number of debug comparators supported by this instance of VSPA. Settings: Reads return the number of debug comparators supported by this instance of VSPA. Writes have no affect.
15-8 DBG_FIFO_SIZE	DBG_FIFO_SIZE Number of entries in the Debug FIFO (debug FIFO size). These status bits indicate the number of entries in the debug FIFO in this instance of VSPA. Settings: Reads return the Number of entries in the Debug FIFO in this instance of VSPA. Writes have no affect. 00000000b - 0 entries 00000101b - 32 entries 00000110b - 64 entries 11111111b - Illegal
7 —	- Reserved
6-0 ATID_VALUE	ATID_VALUE These status bits indicate the value this VSPA places on the ATID lines when a trace message is sent on the ATB. Settings: Reads return the value this VSPA places on the ATID lines when a trace message is sent on the ATB. Writes have no affect.

15.1.25 Peripheral ID4 register (PIDR4)

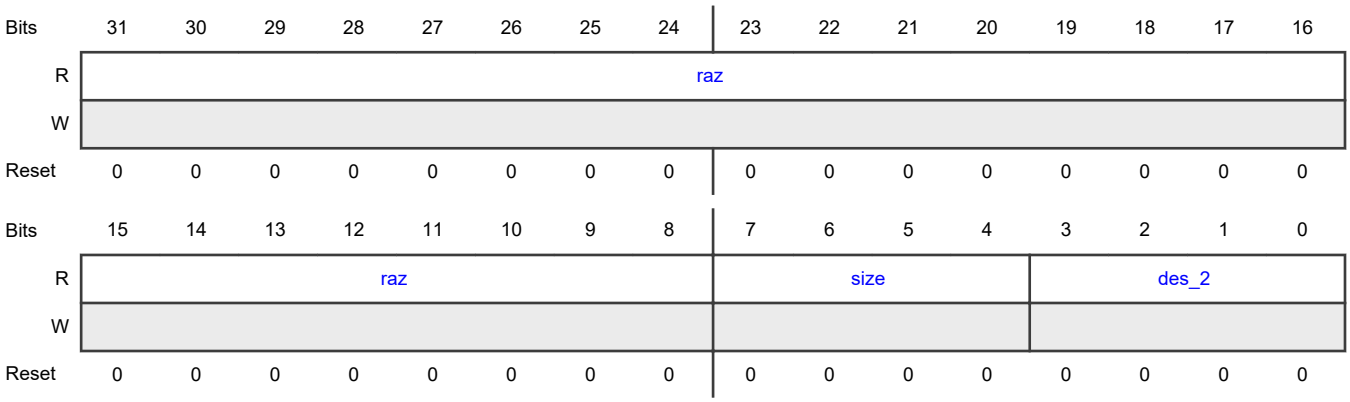
Offset

Register	Offset
PIDR4	FD0h

Function

Peripheral ID4 register is part of a group of peripheral identification registers which uniquely identify the component. The peripheral ID registers can be read by a debugger and used to identify the component to gain access to the components programmer's model and capabilities. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-8 raz	raz Reads As Zero
7-4 size	size 4KByte Count Value Indicates the contiguous size of the memory window used by the component in powers of 2 from the standard 4KB (0x0=4KB, 0x1=8KB, 0x2=16KB, 0x3=32KB, and so on) VSP 4KByte Count = 0x0 = 4KB
3-0 des_2	des_2 JEP106 Continuation Code DES_2, DES_1, DES_0 indicates the designer of the component. DES_2 is the JTAG106 Continuation Code (for NXP, DES_2=0x0)

15.1.26 Peripheral ID5 register (PIDR5)

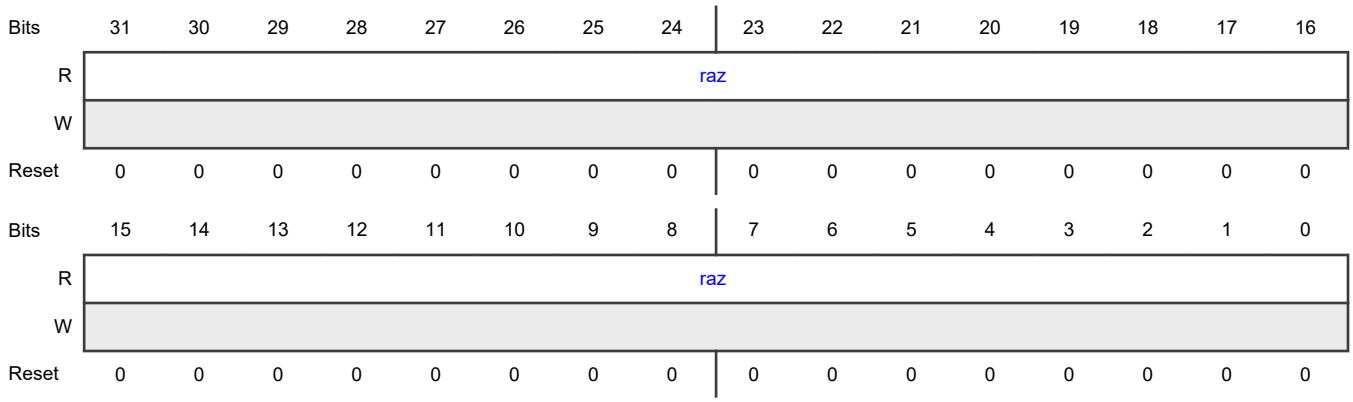
Offset

Register	Offset
PIDR5	FD4h

Function

Peripheral ID5 register is part of a group of peripheral identification registers which uniquely identify the component. The peripheral ID registers can be read by a debugger and used to identify the component to gain access to the components programmer's model and capabilities. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-0 raz	raz Reads As Zero

15.1.27 Peripheral ID6 register (PIDR6)

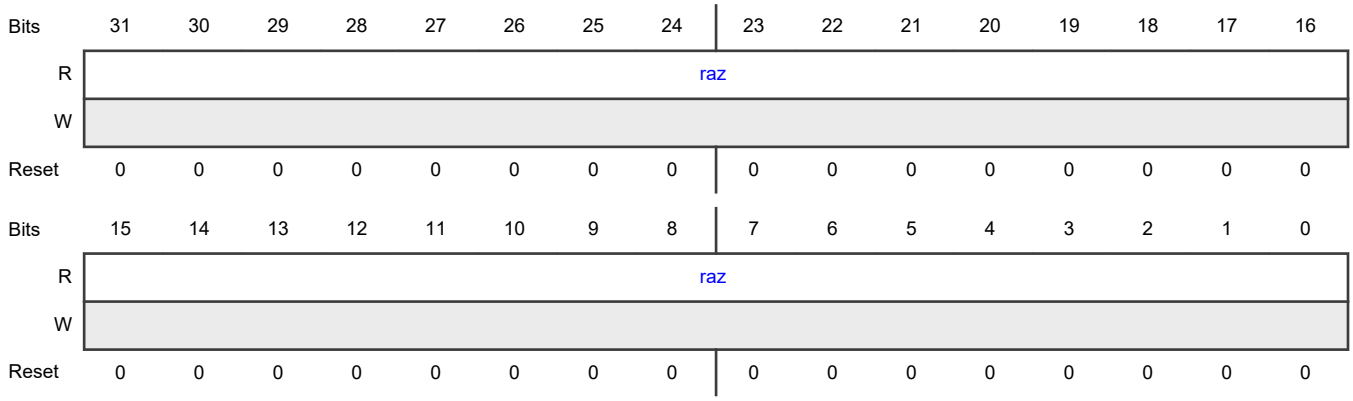
Offset

Register	Offset
PIDR6	FD8h

Function

Peripheral ID6 register is part of a group of peripheral identification registers which uniquely identify the component. The peripheral ID registers can be read by a debugger and used to identify the component to gain access to the components programmer's model and capabilities. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-0 raz	raz Reads As Zero

15.1.28 Peripheral ID7 register (PIDR7)

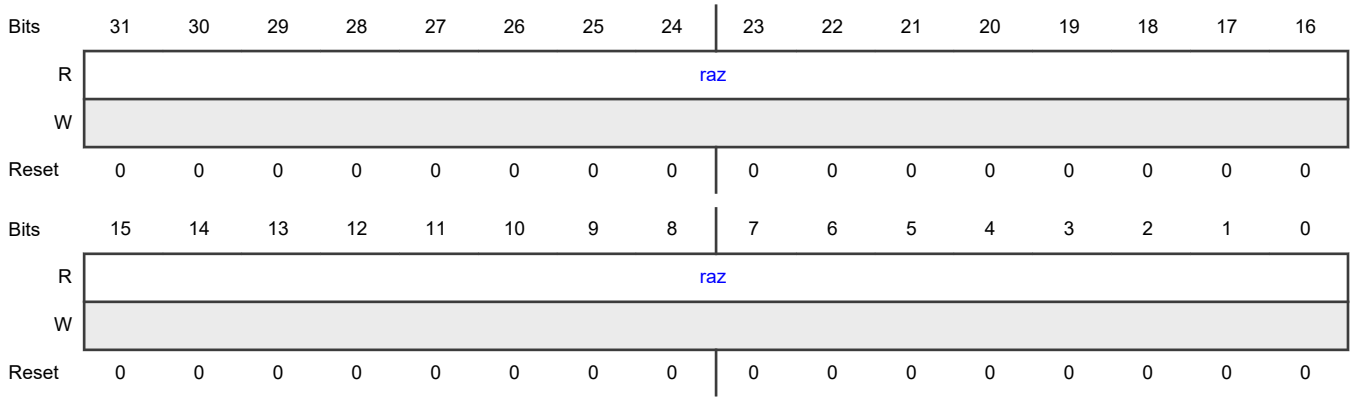
Offset

Register	Offset
PIDR7	FDCh

Function

Peripheral ID7 register is part of a group of peripheral identification registers which uniquely identify the component. The peripheral ID registers can be read by a debugger and used to identify the component to gain access to the components programmer's model and capabilities. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-0 raz	raz Reads As Zero

15.1.29 Peripheral ID0 register (PIDR0)

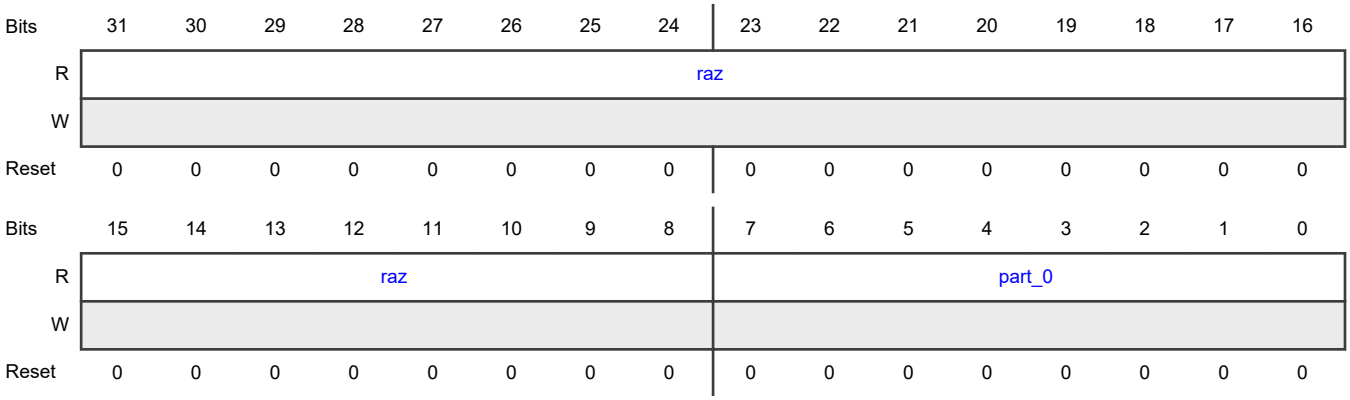
Offset

Register	Offset
PIDR0	FE0h

Function

Peripheral ID0 register is part of a group of peripheral identification registers which uniquely identify the component. The peripheral ID registers can be read by a debugger and used to identify the component to gain access to the components programmer's model and capabilities. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-8 raz	raz Reads As Zero
7-0 part_0	part_0 Part Number bits[7:0] {PART_1, PART_0} = PART[11:0] - indicates the component part number selected by the designer. PART[11:0] = 0x000 indicates a VSP.

15.1.30 Peripheral ID1 register (PIDR1)

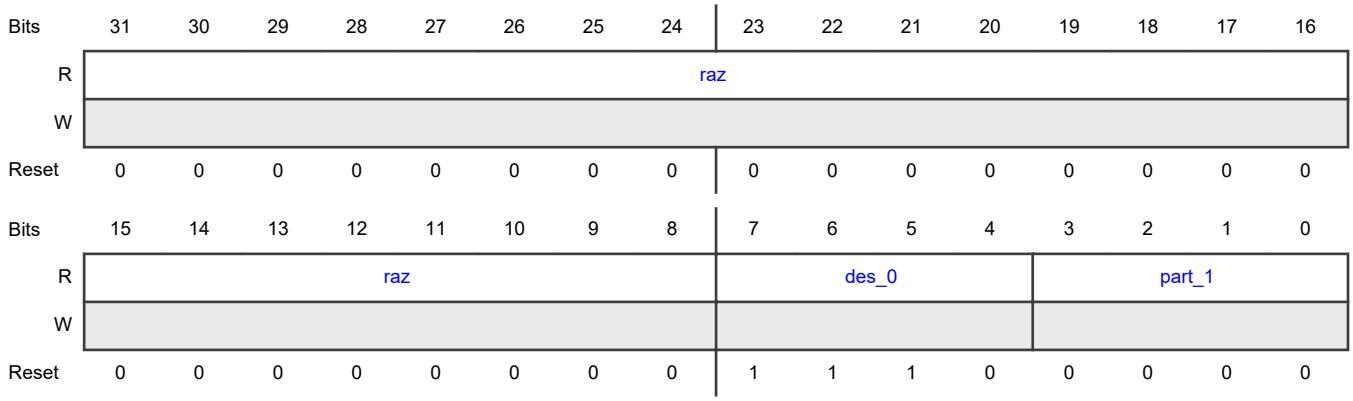
Offset

Register	Offset
PIDR1	FE4h

Function

Peripheral ID1 register is part of a group of peripheral identification registers which uniquely identify the component. The peripheral ID registers can be read by a debugger and used to identify the component to gain access to the components programmer's model and capabilities. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-8 raz	raz Reads As Zero
7-4 des_0	des_0 JEP106 Identification Code bits[3:0] The concatenated {DES_2, DES_1, DES_0} indicates the designer of the component. DES_0 are the upper bits of the JEP106 identification code (for NXP, DES_0=0xE)
3-0 part_1	part_1 Part Number bits[11:8] {PART_1, PART_0} = PART[11:0] - indicates the component part number selected by the designer. PART[11:0] == 0x0 indicates a VSP.

15.1.31 Peripheral ID2 register (PIDR2)

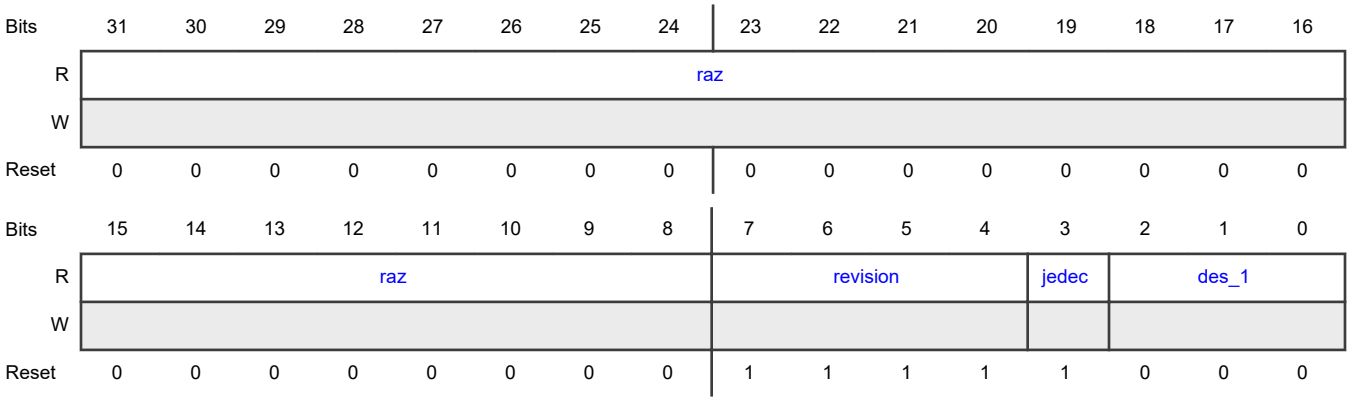
Offset

Register	Offset
PIDR2	FE8h

Function

Peripheral ID2 register is part of a group of peripheral identification registers which uniquely identify the component. The peripheral ID registers can be read by a debugger and used to identify the component to gain access to the components programmer's model and capabilities. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-8 raz	raz Reads As Zero
7-4 revision	revision Revision The revision is hard-coded to 4'b1111 which should be ignored. Refer to the VSP HWVERSION register for the VSP version details.
3 jedec	jedec Indicates a JEDEC assigned values are used (always 1'b1).
2-0 des_1	des_1 JEP106 Identification Code bits[6:4] The concatenated {DES_2, DES_1, DES_0} indicates the designer of the component. DES_1 are the upper bits of the JEP106 identification code (for NXP, DES_1=0x0)

15.1.32 Peripheral ID3 register (PIDR3)

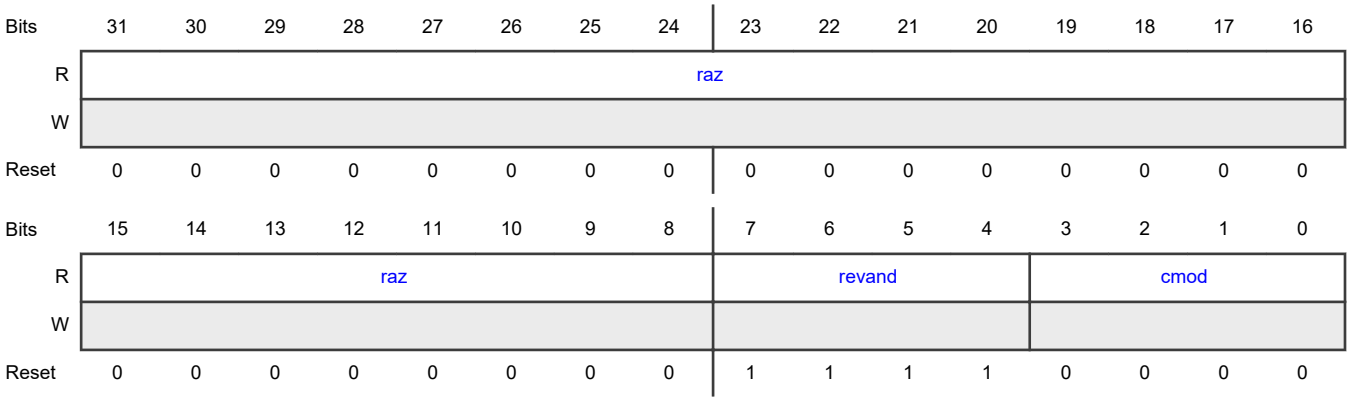
Offset

Register	Offset
PIDR3	FECh

Function

Peripheral ID3 register is part of a group of peripheral identification registers which uniquely identify the component. The peripheral ID registers can be read by a debugger and used to identify the component to gain access to the components programmer's model and capabilities. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-8 raz	raz Reads As Zero
7-4 revand	revand RevAnd The minor revision is hard-coded to 4'b1111 which should be ignored. Refer to the VSP HWVERSION register for the VSP version details.
3-0 cmod	cmod Customer Modified Currently hard-coded to 4'b0000.

15.1.33 Component ID0 register (CIDR0)

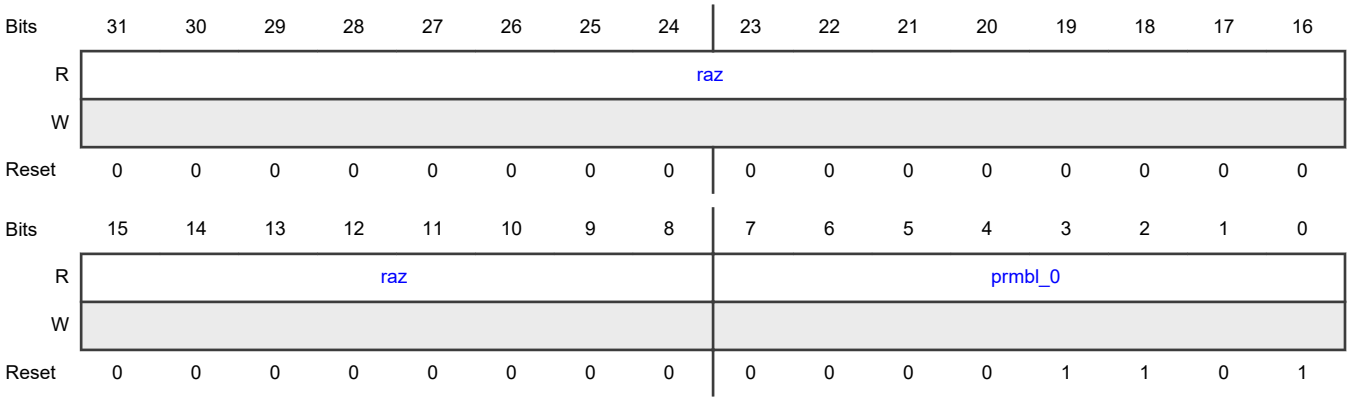
Offset

Register	Offset
CIDR0	FF0h

Function

Component ID0 register is part of a group of component identification registers which identify the component class. The component class can be read by a debugger and used to identify the type of component. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-8 raz	raz Reads As Zero
7-0 prmbl_0	prmbl_0 Preamble Hard-coded to 0x0D

15.1.34 Component ID1 register (CIDR1)

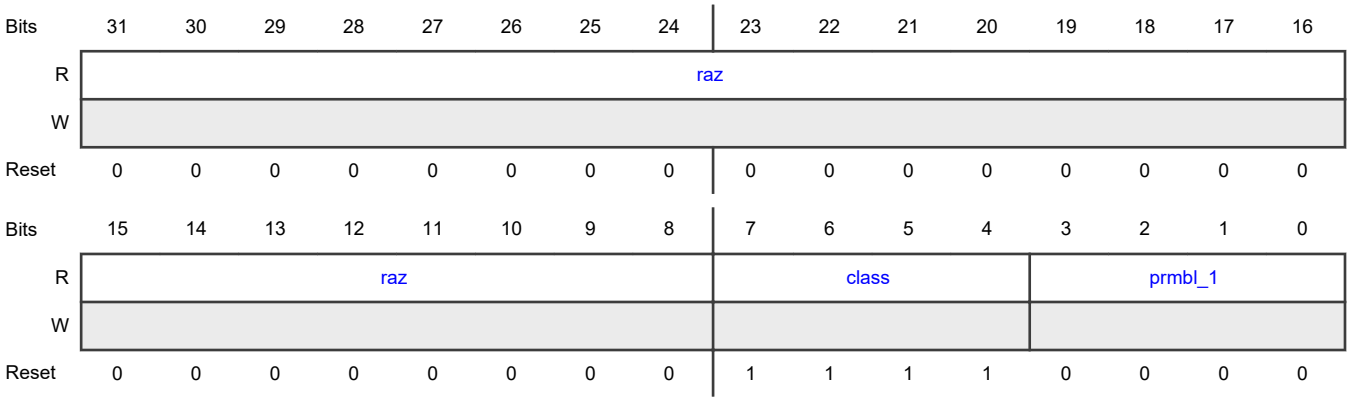
Offset

Register	Offset
CIDR1	FF4h

Function

Component ID1 register is part of a group of component identification registers which identify the component class. The component class can be read by a debugger and used to identify the type of component. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-8 raz	raz Reads As Zero
7-4 class	class Component Class Hard-coded to 0xF (indicates a CoreLink, PrimeCell, or System component with no standardized register layout)
3-0 prmbl_1	prmbl_1 Preamble Hard-coded to 0x0

15.1.35 Component ID2 register (CIDR2)

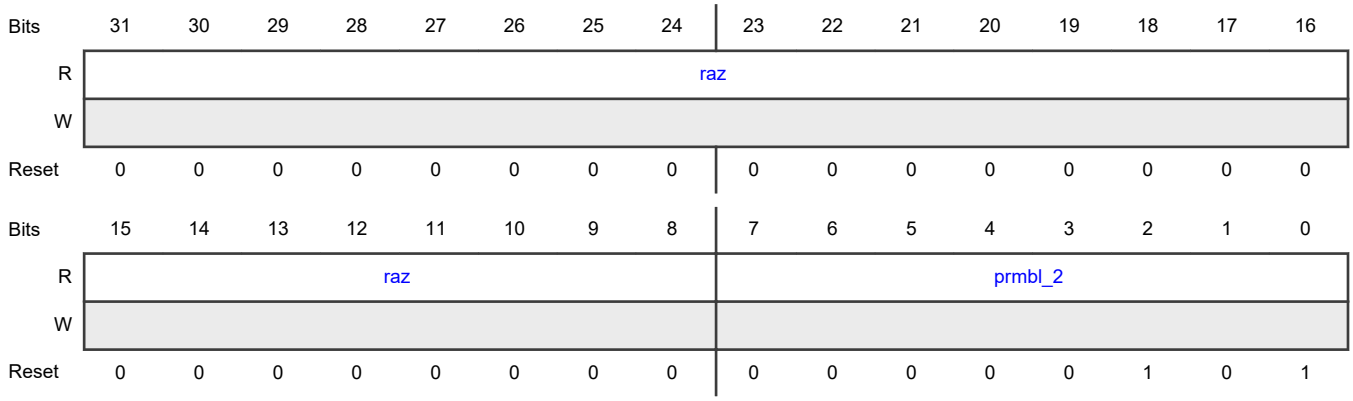
Offset

Register	Offset
CIDR2	FF8h

Function

Component ID2 register is part of a group of component identification registers which identify the component class. The component class can be read by a debugger and used to identify the type of component. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-8 raz	raz Reads As Zero
7-0 prmbl_2	prmbl_2 Preamble Hard-coded to 0x05

15.1.36 Component ID3 register (CIDR3)

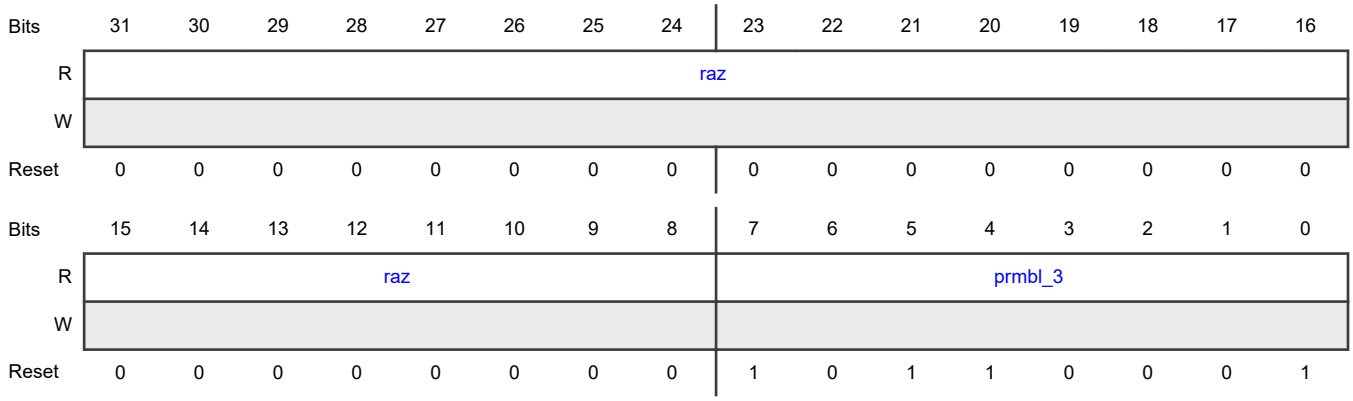
Offset

Register	Offset
CIDR3	FFCh

Function

Component ID3 register is part of a group of component identification registers which identify the component class. The component class can be read by a debugger and used to identify the type of component. Refer to the CoreSight Architecture Specification or ARM Debug Interface v5 Architecture Specification for details.

Diagram



Fields

Field	Function
31-8 raz	raz Reads As Zero
7-0 prmbl_3	prmbl_3 Preamble Hard-coded to 0xB1

Chapter 16
FECU IP Registers

16.1 FECU IP Registers

FECU registers are a part of the VSPA memory map and reside in the address space starting at 0x300 and ending at 0x384. See chapter VSPA IP Registers for VSPA memory map and register definition.

All FECU registers are slow read registers. See Slow read registers for a description of their read behavior.

16.2 FECU register descriptions

16.2.1 FECU memory map

FECU base address: 100_0000h

Offset	Register	Width (In bits)	Access	Reset value
300h	FECU Configuration register (FECU_CONFIG)	32	RW	See description
304h	FECU Symbol size register (FECU_SIZES)	32	RW	0000_0000h
308h	FECU Number of padding bits register (FECU_NUM_PAD)	32	RW	See description
30Ch	FECU Binary Convolutional Code (BCC) puncture mask register (FECU_BCC_PUNC_MASK)	32	RW	0000_0003h
310h	FECU Binary Convolutional Code (BCC) configuration register (FECU_BCC_CONFIG)	32	RW	See description
314h	FECU LDPC configuration register (FECU_LDPC_CONFIG)	32	RW	See description
318h	FECU LDPC repeat, parity, and shortening sizes register (FECU_LDPC_SIZES)	32	RW	See description
31Ch	FECU LDPC blocks with an extra shortening bit register (FECU_LDPC_EXTRA_SHORT)	32	RW	0000_0000h
320h	FECU LDPC blocks with an extra puncturing or repetition bit register (FECU_LDPC_EXTRA_REP)	32	RW	0000_0000h
324h	FECU Bypass register (FECU_BYPASS)	32	RW	See description
328h	FECU Scrambler / De-scrambler configuration register (FECU_SC_CONFIG)	32	RW	See description
32Ch	FECU DMEM Read count register (FECU_DMEM_READ_COUNT)	32	RW	See description
330h	FECU DMEM Source address register (FECU_DMEM_SRC_ADR)	32	RW	See description

Table continues on the next page...

Table continued from the previous page...

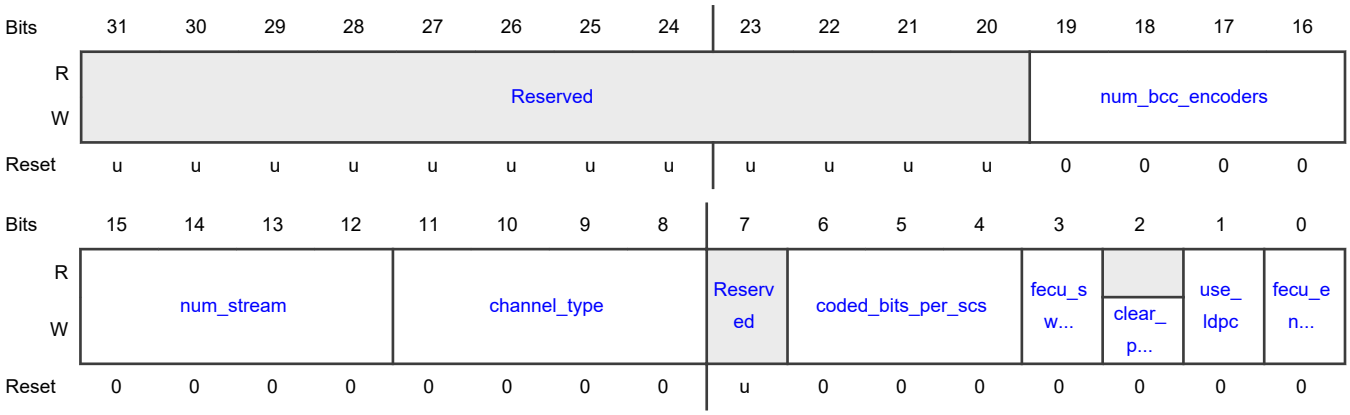
Offset	Register	Width (In bits)	Access	Reset value
334h	FECU DMEM Destination address register (FECU_DMEM_DST_ADR)	32	RW	See description
338h	FECU DMEM 2nd address register (FECU_DMEM_2ND_ADR)	32	RW	See description
33Ch	FECU DMEM 3rd address register (FECU_DMEM_3RD_ADR)	32	RW	See description
340h	FECU DMEM 4th address register (FECU_DMEM_4TH_ADR)	32	RW	See description
344h	FECU DMEM 5th address register (FECU_DMEM_5TH_ADR)	32	RW	See description
348h	FECU DMEM 6th address register (FECU_DMEM_6TH_ADR)	32	RW	See description
34Ch	FECU DMEM 7th address register (FECU_DMEM_7TH_ADR)	32	RW	See description
350h	FECU DMEM 8th address register (FECU_DMEM_8TH_ADR)	32	RW	See description
354h	FECU Save and restore configuration register (FECU_SAVE_RESTORE)	32	RW	See description
358h	FECU Control register (FECU_CONTROL)	32	RW	See description
364h	FECU Status register (FECU_STATUS)	32	W1C	See description
368h	FECU DMEM Write count register (FECU_DMEM_WRITE_COUNT)	32	RO	See description
36Ch	FECU LDPC encoder block sizes register (FECU_LDPC_ENC_BLOCK)	32	RO	See description
370h	FECU LDPC encoder status register (FECU_LDPC_ENC_STATUS)	32	RO	See description
374h	FECU LDPC decoder block sizes and counts register (FECU_LDPC_DEC_BLOCK)	32	RO	0000_0000h
378h	FECU LDPC decoder status register (FECU_LDPC_DEC_STATUS)	32	W1C	See description
380h	FECU Hardware parameters / capabilities of FECU (FECU_HW_PARAMS)	32	RO	See description
384h	FECU Hardware parameters / capabilities of the LDPC encoder and decoder in FECU (FECU_LDPC_HW_PARAMS)	32	RO	0101_1B00h

16.2.2 FECU Configuration register (FECU_CONFIG)

Offset

Register	Offset
FECU_CONFIG	300h

Diagram



Fields

Field	Function
31-20 —	- Reserved
19-16 num_bcc_encoders	num_bcc_encoders The number of BCC encoders or decoders used. Must be <= max_num_bcc_encoders.
15-12 num_stream	num_stream The number of spatial streams. Must be <= max_num_streams.
11-8 channel_type	channel_type The RF bandwidth of the channel. 0000b - 20MHZ_11a 0001b - 20MHZ__11ac 0010b - 40MHZ_11ac 0011b - 80MHZ_11ac 1000b - RU26_11ax 1001b - RU52_11ax 1010b - RU106_11ax

Table continues on the next page...

Table continued from the previous page...

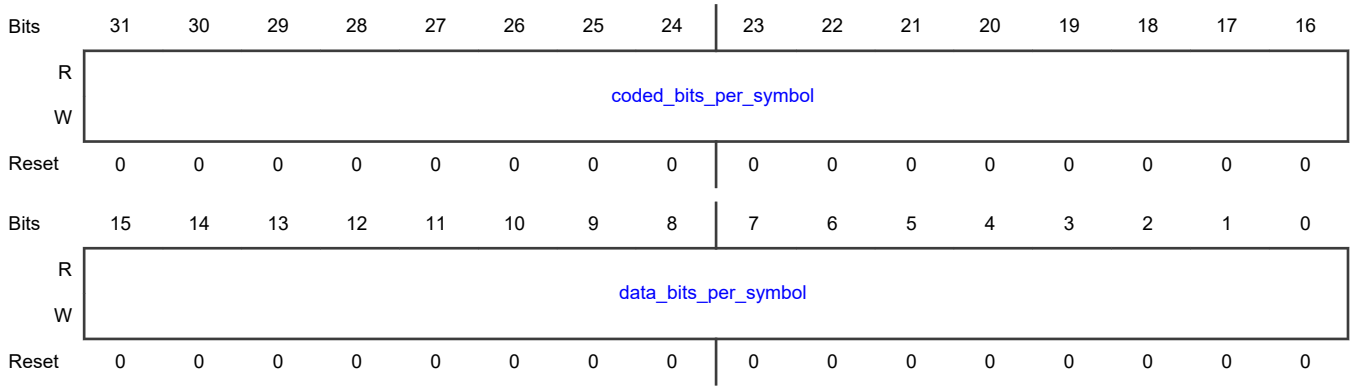
Field	Function
	1011b - RU242_11ax 1100b - RU484_11ax 1101b - RU996_11ax
7 —	- Reserved
6-4 coded_bits_per_scs	coded_bits_per_scs The number of bits per sub-carrier per symbol. Also, the number of bits in a constellation point. 000b - BPSK 001b - QPSK 010b - 16 QAM 011b - 64 QAM 100b - 256 QAM
3 fecu_sw_reset	fecu_sw_reset When high the FECU block is held in reset, and all pending commands are cleared. In order to reset FECU, this bit must remain high for 20 clock cycles.
2 clear_pending	clear_pending Writing a one to this bit will clear all pending operations in the command FIFO. This bit always reads as 0.
1 use_ldpc	use_ldpc 0b - Use BCC (convolutional / Viterbi) 1b - Use LDPC encoding / decoding
0 fecu_encode	fecu_encode 0b - FECU is decoding 1b - FECU is encoding

16.2.3 FECU Symbol size register (FECU_SIZES)

Offset

Register	Offset
FECU_SIZES	304h

Diagram



Fields

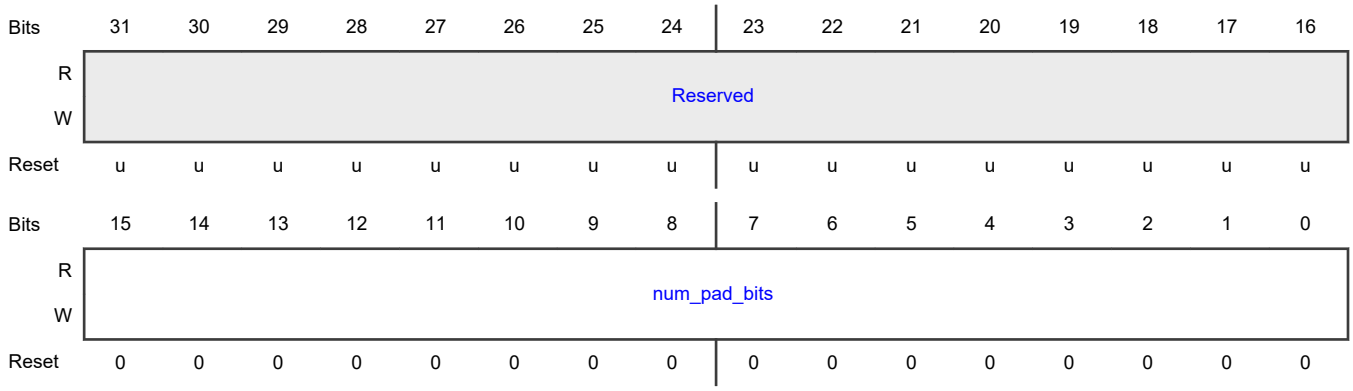
Field	Function
31-16 coded_bits_per_symbol	coded_bits_per_symbol The number of bits per symbol after the size is increased by encoding. Only used for LDPC encoding. Specifies the maximum number of bits the LDPC encoder will output.
15-0 data_bits_per_symbol	data_bits_per_symbol The number of bits per symbol before the size is increased by encoding.

16.2.4 FECU Number of padding bits register (FECU_NUM_PAD)

Offset

Register	Offset
FECU_NUM_PAD	308h

Diagram



Fields

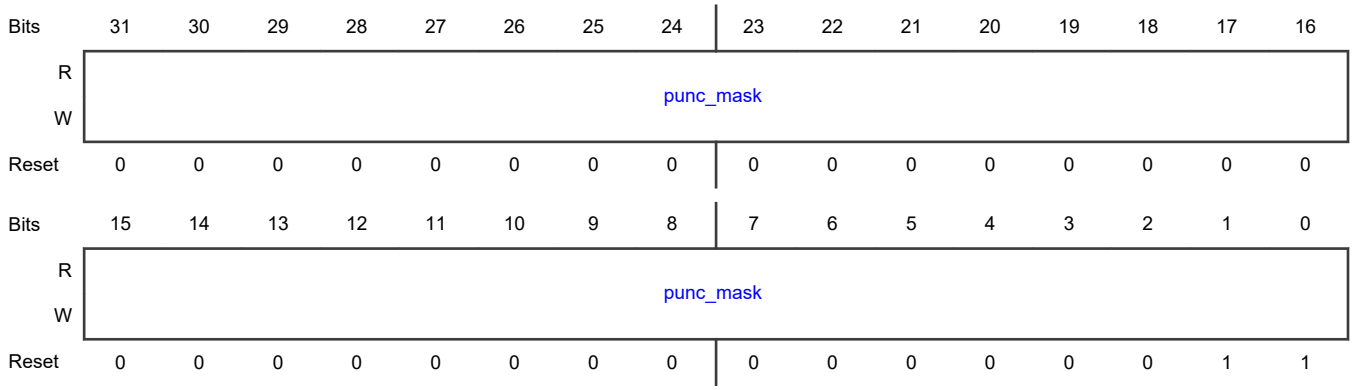
Field	Function
31-16 —	-
15-0 num_pad_bits	num_pad_bits The number of bits after the tail bits. This value is ignored unless final_symbol is set. Only used for BCC.

16.2.5 FECU Binary Convolutional Code (BCC) puncture mask register (FECU_BCC_PUNC_MASK)

Offset

Register	Offset
FECU_BCC_PUNC_MASK	30Ch

Diagram



Fields

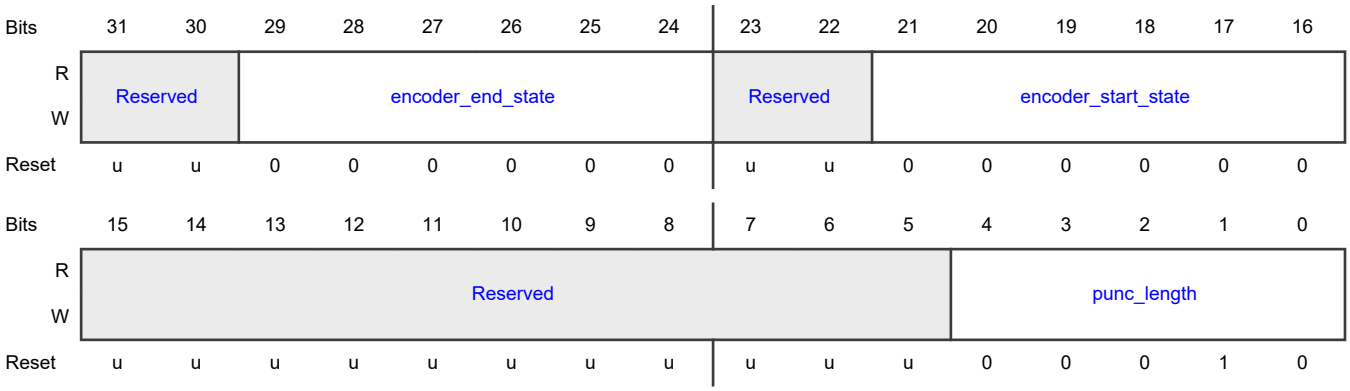
Field	Function
31-0 punc_mask	punc_mask The puncture mask used for BCC encoding and decoding.

16.2.6 FECU Binary Convolutional Code (BCC) configuration register (FECU_BCC_CONFIG)

Offset

Register	Offset
FECU_BCC_CONFIG	310h

Diagram



Fields

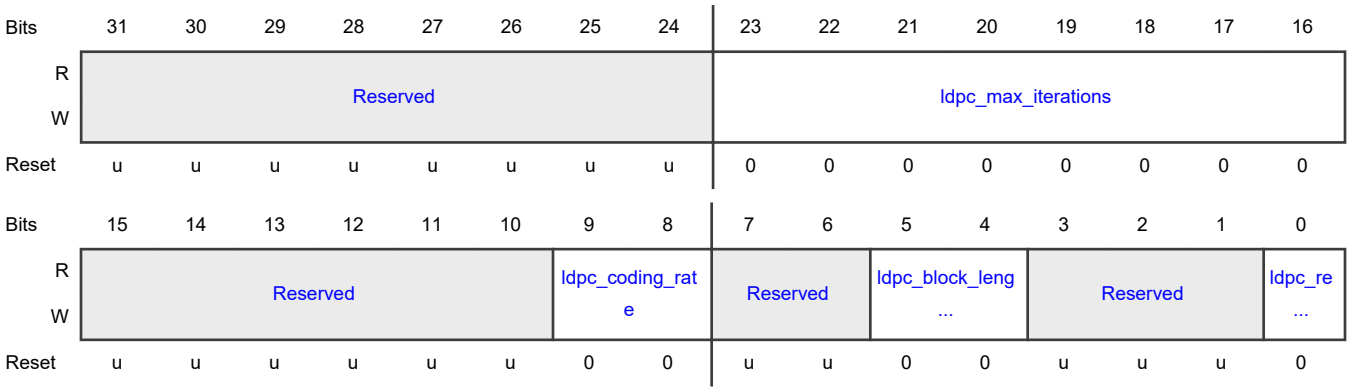
Field	Function
31-30 —	- Reserved
29-24 encoder_end_state	encoder_end_state The ending state of the BCC encoder. Used for both encoding and decoding.
23-22 —	- Reserved
21-16 encoder_start_state	encoder_start_state The starting state of the BCC encoder. Used for both encoding and decoding.
15-5 —	- Reserved
4-0 punc_length	punc_length The length of the valid bits in punc_mask.

16.2.7 FECU LDPC configuration register (FECU_LDPC_CONFIG)

Offset

Register	Offset
FECU_LDPC_CONFIG	314h

Diagram



Fields

Field	Function
31-24 —	- Reserved
23-16 ldpc_max_iterations	ldpc_max_iterations The maximum number of LDPC decoder iterations allowed before a decoder failure is declared. Setting this to 0 will allow 256 iterations.
15-10 —	- Reserved
9-8 ldpc_coding_rate	ldpc_coding_rate Specifies the coding rate used for LDPC. 00b - 1/2 01b - 2/3 10b - 3/4 11b - 5/6
7-6 —	- Reserved
5-4	ldpc_block_length

Table continues on the next page...

Table continued from the previous page...

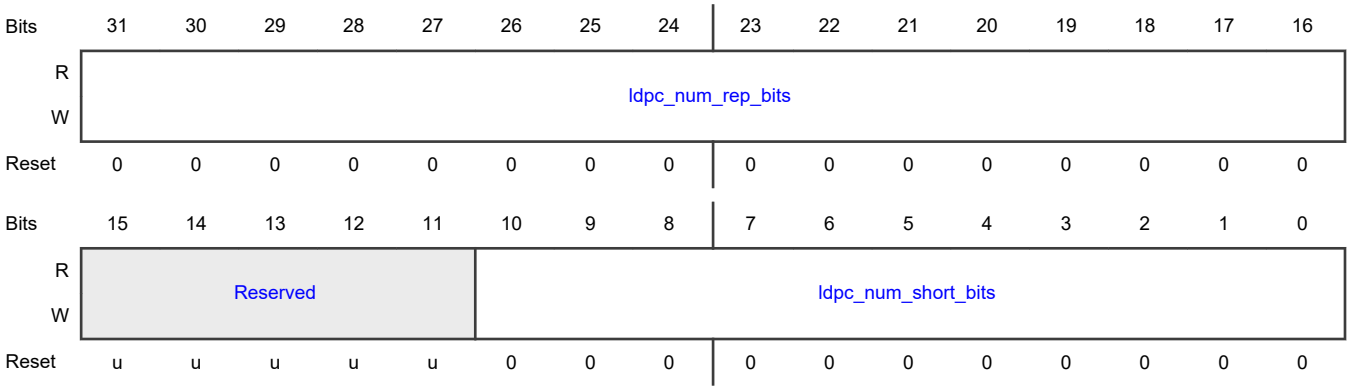
Field	Function
ldpc_block_length	Specifies the length of the LDPC codeword. 00b - 648 01b - 1296 10b - 1944 11b - Reserved
3-1 —	- Reserved
0 ldpc_repeat	ldpc_repeat Used for both encoding and decoding. 0b - Use puncturing 1b - Use repetition when processing LDPC blocks

16.2.8 FECU LDPC repeat, parity, and shortening sizes register (FECU_LDPC_SIZES)

Offset

Register	Offset
FECU_LDPC_SIZES	318h

Diagram



Fields

Field	Function
31-16	ldpc_num_rep_bits The number of bits to be repeated or punctured per block, based on the ldpc_repeat bit.

Table continues on the next page...

Table continued from the previous page...

Field	Function
ldpc_num_rep_bits	
15-11 —	- Reserved
10-0 ldpc_num_short_bits	ldpc_num_short_bits The number of shortening bits to add before encoding, or remove after decoding per block.

16.2.9 FECU LDPC blocks with an extra shortening bit register (FECU_LDPC_EXTRA_SHORT)

Offset

Register	Offset
FECU_LDPC_EXTRA_SHORT	31Ch

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	ldpc_extra_short															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	ldpc_extra_short															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 ldpc_extra_short	ldpc_extra_short The number of LDPC blocks that have 1 extra shortening bit.

16.2.10 FECU LDPC blocks with an extra puncturing or repetition bit register (FECU_LDPC_EXTRA_REP)

Offset

Register	Offset
FECU_LDPC_EXTRA_REP	320h

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	ldpc_extra_rep															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	ldpc_extra_rep															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fields

Field	Function
31-0 ldpc_extra_rep	ldpc_extra_rep The number of LDPC blocks that have 1 extra puncture or repeat bit.

16.2.11 FECU Bypass register (FECU_BYPASS)

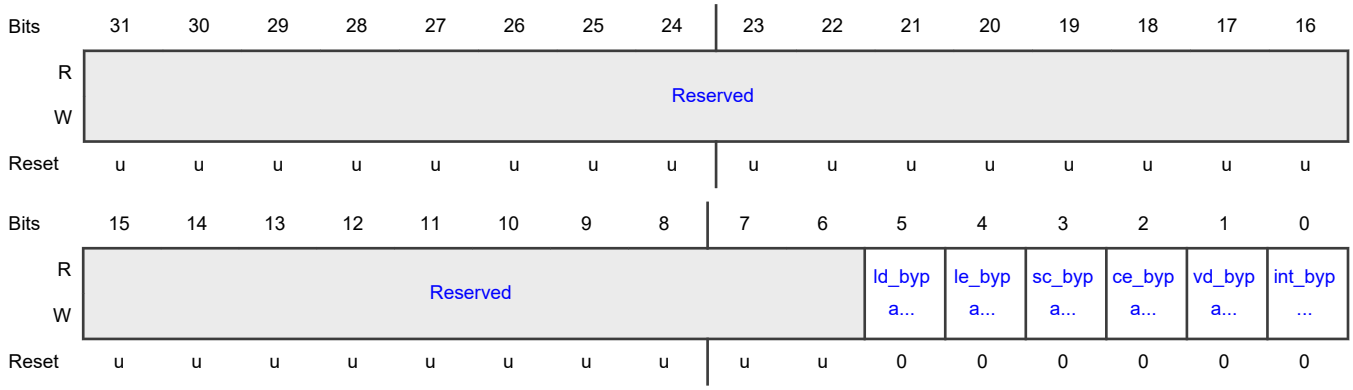
Offset

Register	Offset
FECU_BYPASS	324h

Function

Control bits used to skip some of the FECU operations.

Diagram



Fields

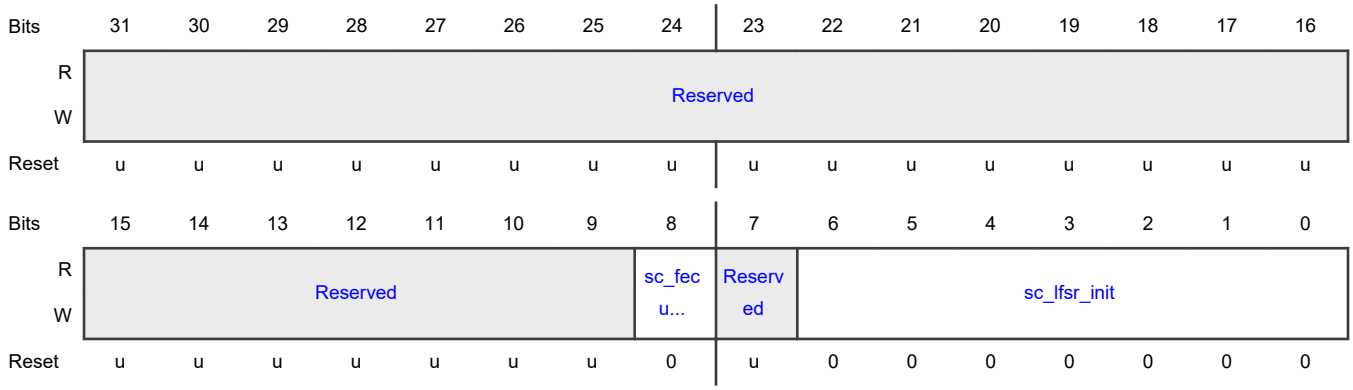
Field	Function
31-6 —	- Reserved
5 ld_bypass	ld_bypass When high, the LDPC decoder is bypassed.
4 le_bypass	le_bypass When high, the LDPC encoder is bypassed.
3 sc_bypass	sc_bypass When high, the scrambler is bypassed.
2 ce_bypass	ce_bypass When high, the convolutional encoder is bypassed.
1 vd_bypass	vd_bypass When high, the Viterbi decoder is bypassed.
0 int_bypass	int_bypass When high, the interleaver is bypassed.

16.2.12 FECU Scrambler / De-scrambler configuration register (FECU_SC_CONFIG)

Offset

Register	Offset
FECU_SC_CONFIG	328h

Diagram



Fields

Field	Function
31-9 —	- Reserved
8 sc_fecu_802b	sc_fecu_802b When set, the scrambler will process according to 802.11b
7 —	- Reserved
6-0 sc_lfsr_init	sc_lfsr_init The initial state of the scrambler LFSR. Only used during encoding.

16.2.13 FECU DMEM Read count register (FECU_DMEM_READ_COUNT)

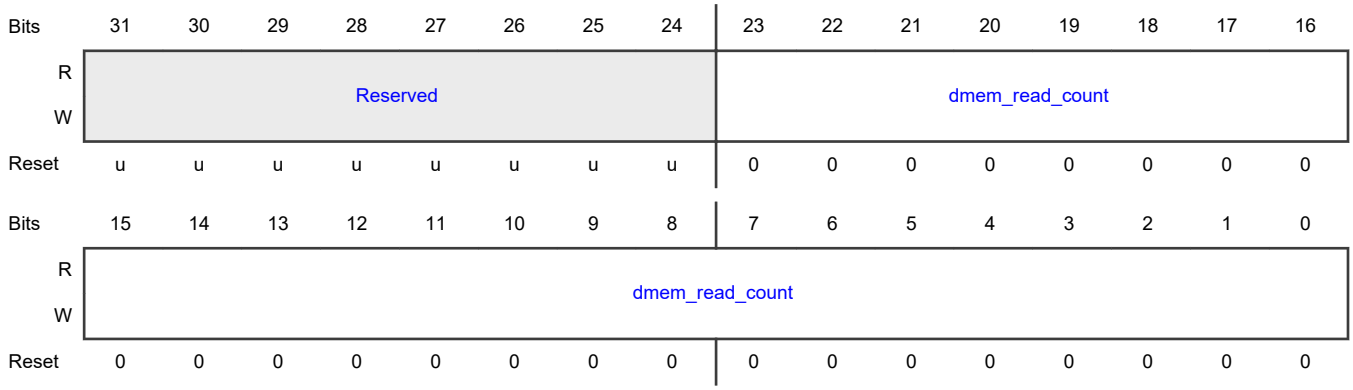
Offset

Register	Offset
FECU_DMEM_READ_COUNT	32Ch

Function

The number of items FECU should read from DMEM.

Diagram



Fields

Field	Function
31-24 —	- Reserved
23-0 dmem_read_count	dmem_read_count Number of LLRs (decode) or bits (encode) to read from DMEM.

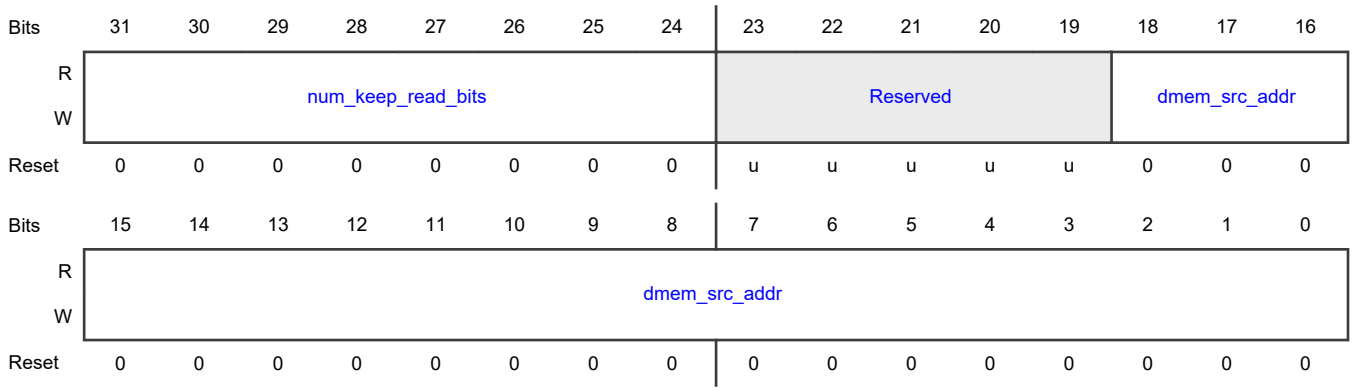
16.2.14 FECU DMEM Source address register (FECU_DMEM_SRC_ADR)

Offset

Register	Offset
FECU_DMEM_SRC_ADR	330h

Function

Diagram



Fields

Field	Function
31-24 num_keep_read_bits	num_keep_read_bits The number of bits that are read from DMEM, but not sent out to FECU for processing during this symbol. These num_keep_read_bits, will be sent out before any new data in the next symbol, if first_symbol=0. When first_symbol=1, keep bits from the previous symbol are discarded, and no keep bits are sent to FECU before new data. However, when first_symbol=1, num_keep_read_bits will be read and saved for the next symbol. The maximum value allowed is only AXI_DATA_WIDTH, even though 8 bits are allocated for this bit field.
23-19 —	- Reserved
18-0 dmem_src_addr	dmem_src_addr The source half-word (16 bit) address data is read from DMEM.

16.2.15 FECU DMEM Destination address register (FECU_DMEM_DST_ADR)

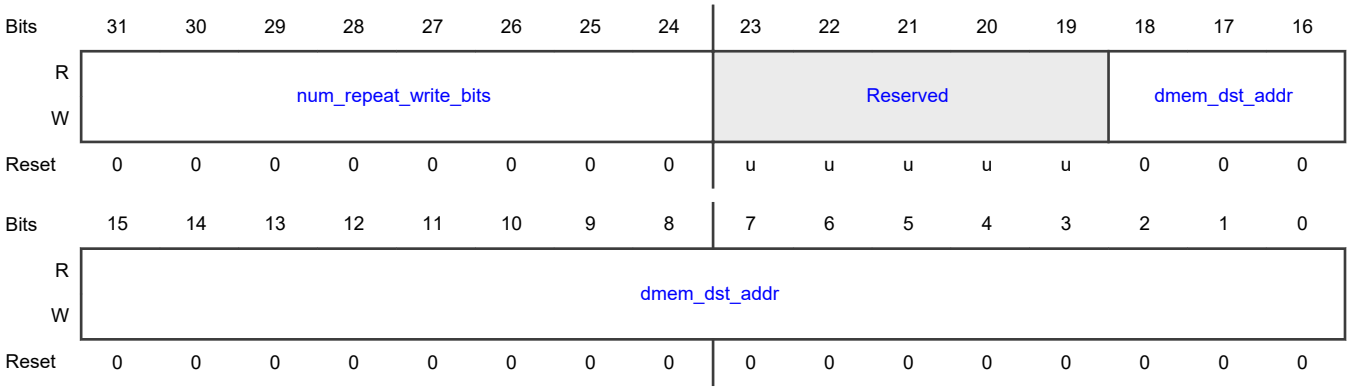
Offset

Register	Offset
FECU_DMEM_DST_ADR	334h

Function

The destination half-word (16 bit) address in DMEM.

Diagram



Fields

Field	Function
31-24 num_repeat_write_bits	num_repeat_write_bits The number of bits to repeat. These bits are written before new bits coming from FECU. They are the last bits from the previous operation. They are written starting at address dmem_dst_addr. The maximum value allowed is only AXI_DATA_WIDTH, even though 8 bits are allocated for this bit field.
23-19 —	- Reserved
18-0 dmem_dst_addr	dmem_dst_addr The half-word (16 bit) address data is written to DMEM.

16.2.16 FECU DMEM 2nd address register (FECU_DMEM_2ND_ADR)

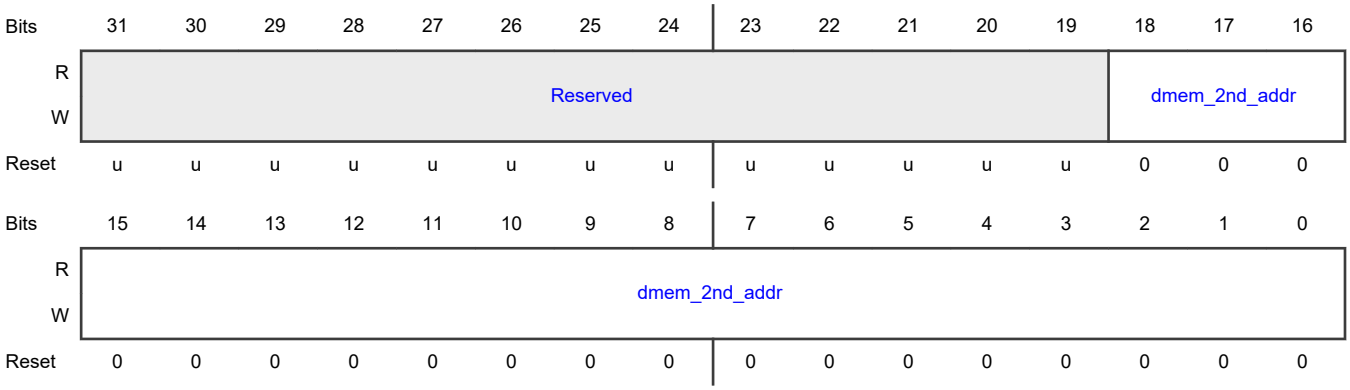
Offset

Register	Offset
FECU_DMEM_2ND_ADR	338h

Function

The half-word (16 bit) DMEM address of the 2nd stream.

Diagram



Fields

Field	Function
31-19 —	- Reserved
18-0 dmem_2nd_addr	dmem_2nd_addr The half-word (16 bit) address used for the 2nd stream. Read address when decoding, write when encoding. Only implemented when FECU_MAX_NSS >= 2.

16.2.17 FECU DMEM 3rd address register (FECU_DMEM_3RD_ADR)

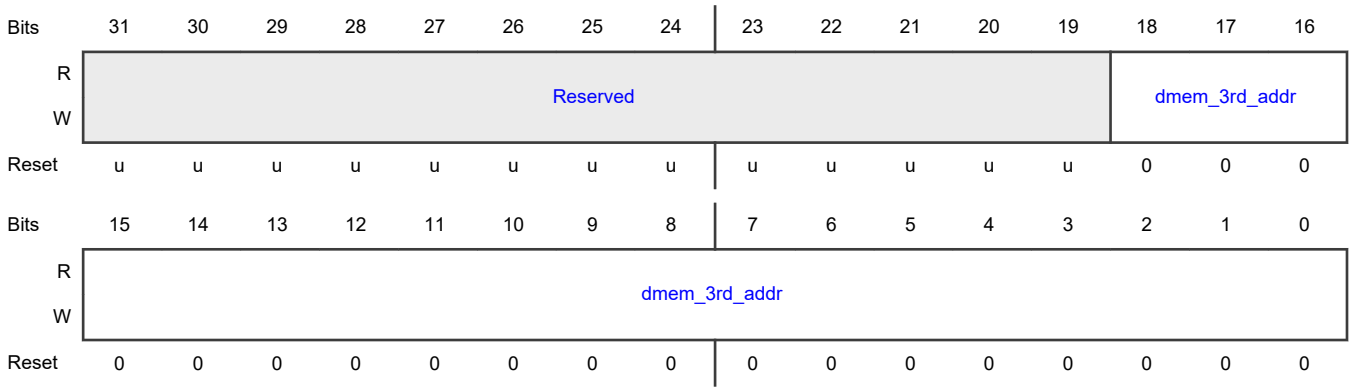
Offset

Register	Offset
FECU_DMEM_3RD_ADR	33Ch

Function

The half-word (16 bit) DMEM address of the 3rd stream.

Diagram



Fields

Field	Function
31-19 —	- Reserved
18-0	dmem_3rd_addr

Table continues on the next page...

Table continued from the previous page...

Field	Function
dmem_3rd_addr	The half-word (16 bit) address used for the 3rd stream. Read address when decoding, write when encoding. Only implemented when FECU_MAX_NSS >= 3.

16.2.18 FECU DMEM 4th address register (FECU_DMEM_4TH_ADR)

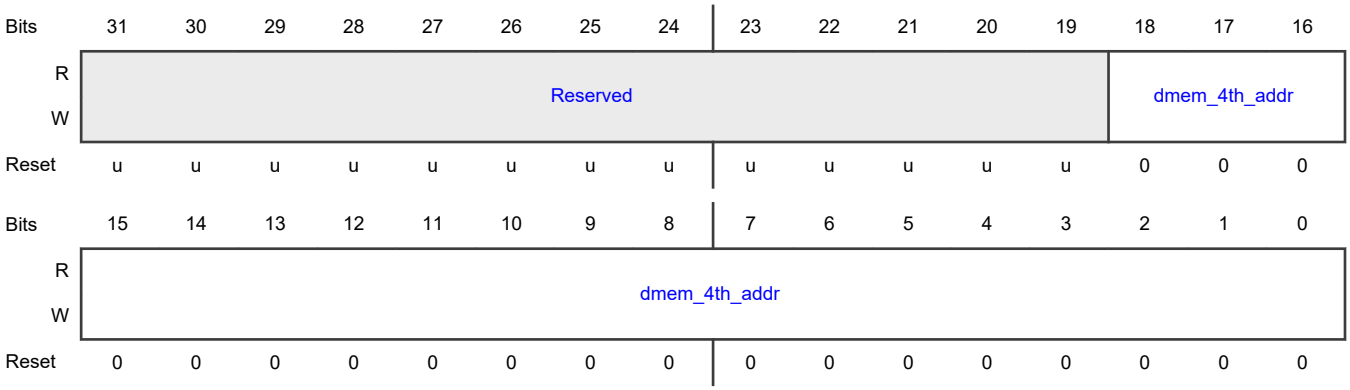
Offset

Register	Offset
FECU_DMEM_4TH_ADR	340h

Function

The half-word (16 bit) DMEM address of the 4th stream.

Diagram



Fields

Field	Function
31-19 —	- Reserved
18-0 dmem_4th_addr	dmem_4th_addr The half-word (16 bit) address used for the 4th stream. Read address when decoding, write when encoding. Only implemented when FECU_MAX_NSS >= 4.

16.2.19 FECU DMEM 5th address register (FECU_DMEM_5TH_ADR)

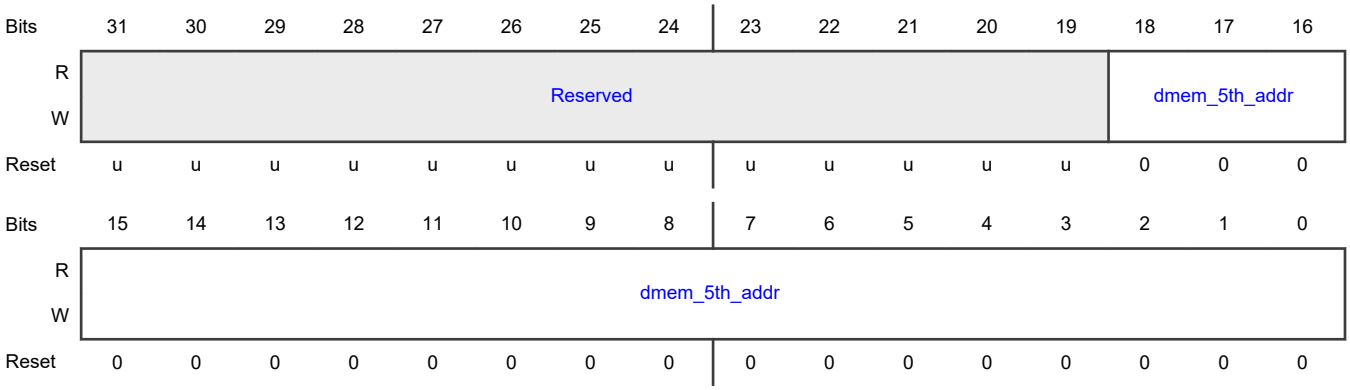
Offset

Register	Offset
FECU_DMEM_5TH_ADR	344h

Function

The half-word (16 bit) DMEM address of the 5th stream.

Diagram



Fields

Field	Function
31-19 —	- Reserved
18-0 dmem_5th_addr	dmem_5th_addr The half-word (16 bit) address used for the 5th stream. Read address when decoding, write when encoding. Only implemented when FECU_MAX_NSS >= 5.

16.2.20 FECU DMEM 6th address register (FECU_DMEM_6TH_ADR)

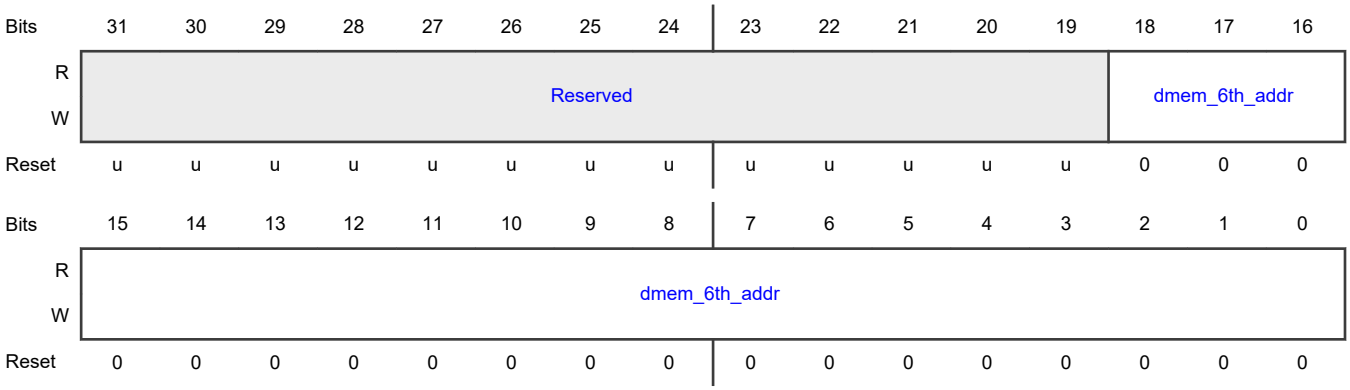
Offset

Register	Offset
FECU_DMEM_6TH_ADR	348h

Function

The half-word (16 bit) DMEM address of the 6th stream.

Diagram



Fields

Field	Function
31-19 —	- Reserved
18-0 dmem_6th_addr	dmem_6th_addr The half-word (16 bit) address used for the 6th stream. Read address when decoding, write when encoding. Only implemented when FECU_MAX_NSS >= 6.

16.2.21 FECU DMEM 7th address register (FECU_DMEM_7TH_ADR)

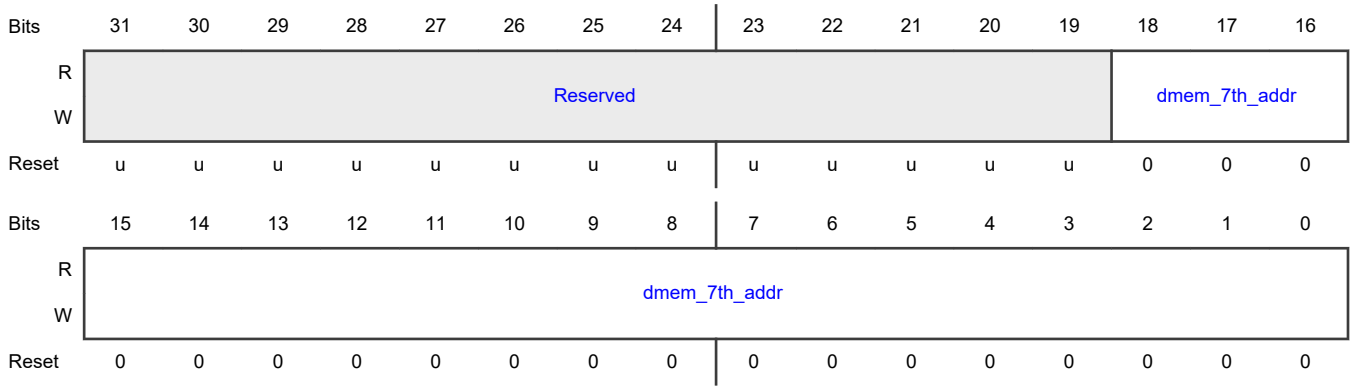
Offset

Register	Offset
FECU_DMEM_7TH_ADR	34Ch

Function

The half-word (16 bit) DMEM address of the 7th stream.

Diagram



Fields

Field	Function
31-19 —	- Reserved
18-0 dmem_7th_addr	dmem_7th_addr The half-word (16 bit) address used for the 7th stream. Read address when decoding, write when encoding. Only implemented when FECU_MAX_NSS >= 7.

16.2.22 FECU DMEM 8th address register (FECU_DMEM_8TH_ADR)

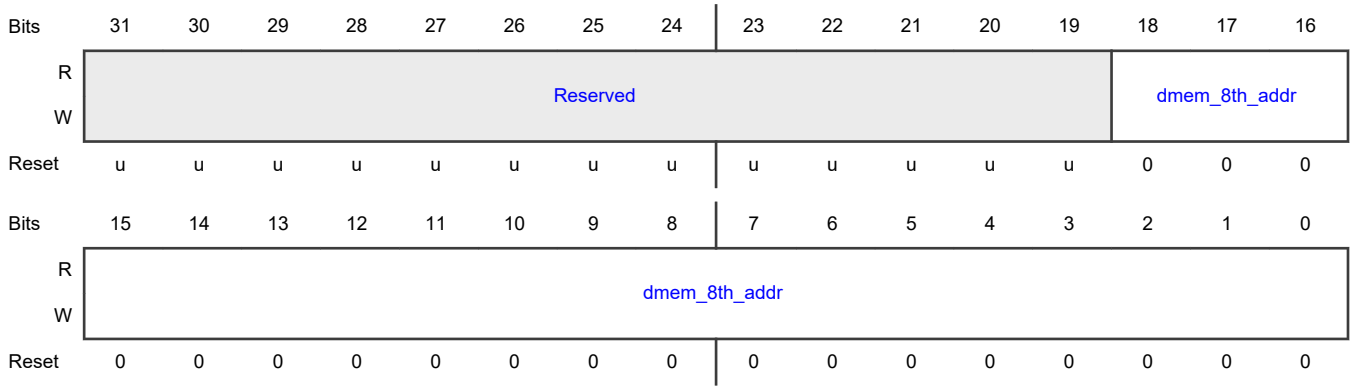
Offset

Register	Offset
FECU_DMEM_8TH_ADR	350h

Function

The half-word (16 bit) DMEM address of the 8th stream.

Diagram



Fields

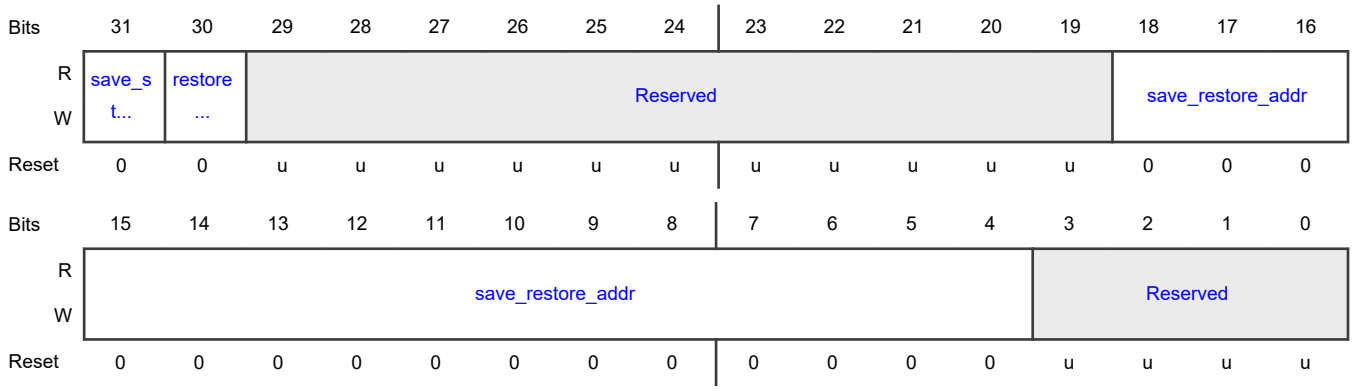
Field	Function
31-19 —	- Reserved
18-0 dmem_8th_addr	dmem_8th_addr The half-word (16 bit) address used for the 8th stream. Read address when decoding, write when encoding. Only implemented when FECU_MAX_NSS >= 8.

16.2.23 FECU Save and restore configuration register (FECU_SAVE_RESTORE)

Offset

Register	Offset
FECU_SAVE_RESTORE	354h

Diagram



Fields

Field	Function
31 save_state	save_state When set, FECU will store FECU state (context) after the current operation completes.
30 restore_state	restore_state When set, FECU will restore FECU state before the current operation starts. When clear, FECU will continue from ending state of the last operation.
29-19 —	- Reserved
18-4 save_restore_addr	save_restore address The pointer to store FECU state after the current operation completes, or restore the state before the current operation starts.
3-0 —	- Reserved

16.2.24 FECU Control register (FECU_CONTROL)

Offset

Register	Offset
FECU_CONTROL	358h

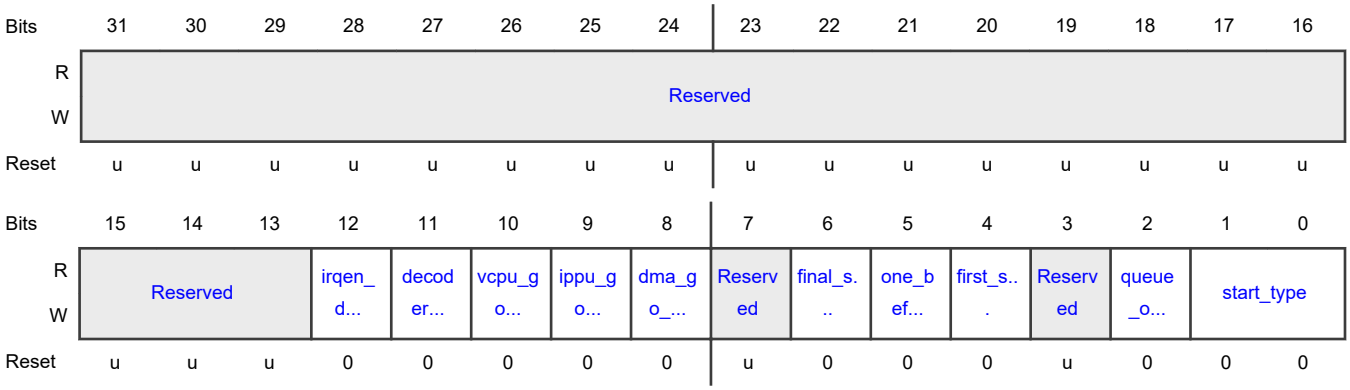
Function

A write to this register will add a pending command in the command FIFO.

Reads return the last value written.

NOTE: The FECU_CONTROL register must be written after all other IP registers. IP register values must not change during a FECU operation.

Diagram



Fields

Field	Function
31-13 —	- Reserved
12 irqen_done	irqen_done When set, FECU will send an interrupt to the host upon a fecu_done event. When cleard, FECU will not generate a host interrupt for this FECU operation.
11 decoder_error_vcpu_go	decoder_error_vcpu_go When set, FECU will send a VCPU go when the decoder gets a decode error, and the ldpc_decode_error bit is set. When cleared, no VCPU go events will be generated as a result of ldpc_decode_error.
10 vcpu_go_enable	vcpu_go_enable When set, FECU will generate a VCPU go when FECU is done with an output buffer. When cleared, FECU will not generate a VCPU go when it is done with an output buffer.
9 ippu_go_enable	ippu_go_enable When set, FECU will generate an IPPU go when FECU is done with an output buffer. When cleared, FECU will not generate an IPPU go when it is done with an output buffer.
8 dma_go_enable	dma_go_enable When set, FECU will trigger a DMA transfer when FECU is done with an output buffer. When cleared, FECU will not trigger a DMA transfer when it is done with an output buffer.
7 —	- Reserved
6 final_symbol	final_symbol Set when this is the last symbol in a code block.
5 one_before_final_symbol	one_before_final_symbol Set when this operation is the one before the last symbol in a code block.
4 first_symbol	first_symbol Set when this is the first symbol in a code block.
3 —	- Reserved
2	queue_output When set, the write to the FECU_CONTROL register creates an output buffer command.

Table continues on the next page...

Table continued from the previous page...

Field	Function
queue_output	When cleared, no output buffer command is created. An input buffer command is always created.
1-0 start_type	start_type Determines when FECU should start. 00b - Start immediately 01b - Wait for an external DMA trigger 10b - Wait for an external IPPU trigger 11b - Reserved

16.2.25 FECU Status register (FECU_STATUS)

Offset

Register	Offset
FECU_STATUS	364h

Function

Reflects the current state of the FECU unit.

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	Reserved															
W																
Reset	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u	u

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	Reserved					start_e...	start_source		Reserved		command_depth		suspend	comm and...	busy_or...	busy
W						W1C										
Reset	u	u	u	u	u	0	0	0	u	u	0	0	0	0	0	0

Fields

Field	Function
31-11 —	- Reserved
10	start_error

Table continues on the next page...

Table continued from the previous page...

Field	Function
start_error	High after a write to the FECU_CONTROL register when the command FIFO is full. Cleared by writing 1 to this bit.
9-8 start_source	start_source Indicates the source of start for the current operation or last completed operation. 00b - Immediate start 01b - dma trigger 10b - ippu trigger 11b - Reserved
7-6 —	- Reserved
5-4 command_depth	command_depth Number of pending operations in the command FIFO. A maximum of 2 operations can be pending. Further attempts to start an operation will set start_error.
3 suspend	suspend High if FECU is suspended by the debugger. In this state, FECU will not access DMEM.
2 command_fifo_full	command_fifo_full High when command_depth equals 2.
1 busy_or_pending	busy_or_pending High when busy is set or there is at least one command pending in the FIFO.
0 busy	busy High while FECU is currently running an operation. Busy will be set until all output buffers have completed.

16.2.26 FECU DMEM Write count register (FECU_DMEM_WRITE_COUNT)

Offset

Register	Offset
FECU_DMEM_WRITE_COUNT	368h

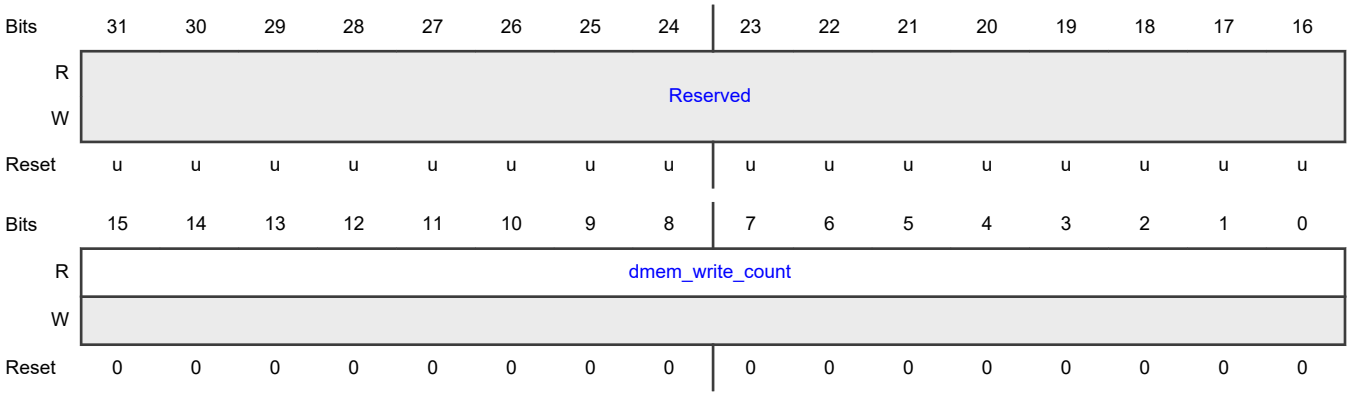
Function

The number of items FECU wrote to DMEM.

Updated as each line is written.

Used for diagnotics / debug.

Diagram



Fields

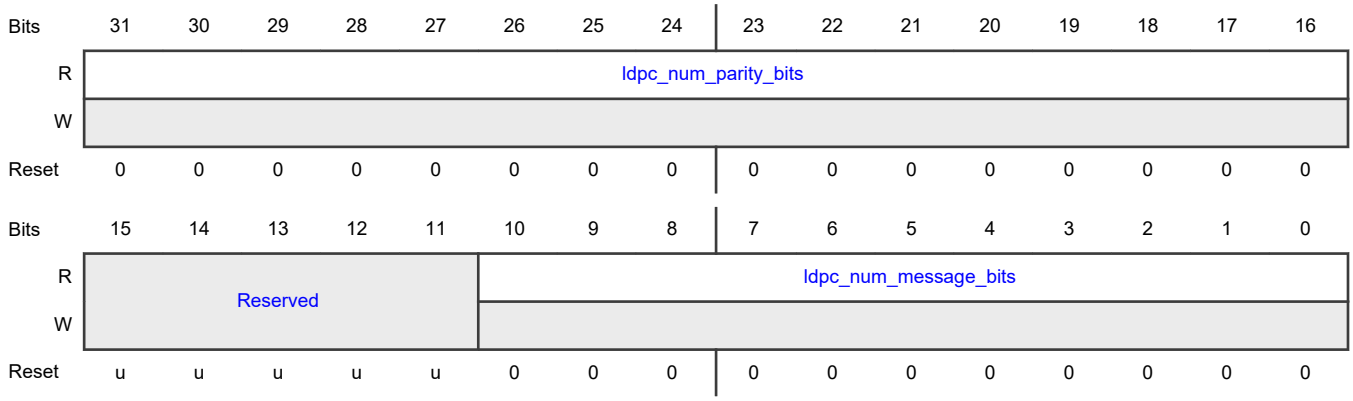
Field	Function
31-16 —	- Reserved
15-0 dmem_write_count	dmem_write_count Number of new LLRs (decode) or new bits (encode) written to DMEM. This count does not include repeated bits.

16.2.27 FECU LDPC encoder block sizes register (FECU_LDPC_ENC_BLOCK)

Offset

Register	Offset
FECU_LDPC_ENC_BLOCK	36Ch

Diagram



Fields

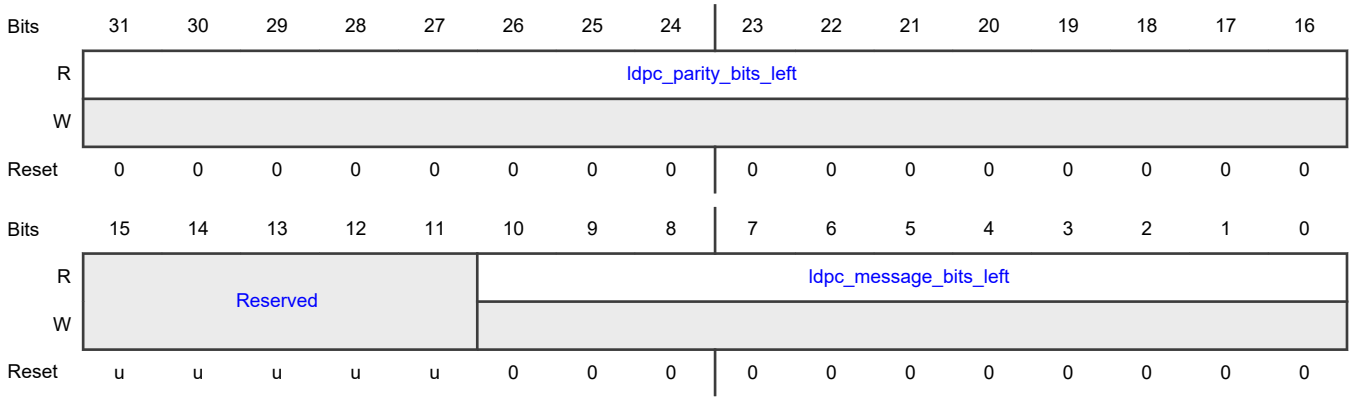
Field	Function
31-16 ldpc_num_parity_bits	ldpc_num_parity_bits The total number of parity plus repetition bits in the block the LDPC encoder is currently processing.
15-11 —	- Reserved
10-0 ldpc_num_message_bits	ldpc_num_message_bits The total number of message bits in the block the LDPC encoder is currently processing.

16.2.28 FECU LDPC encoder status register (FECU_LDPC_ENC_STATUS)

Offset

Register	Offset
FECU_LDPC_ENC_STATUS	370h

Diagram



Fields

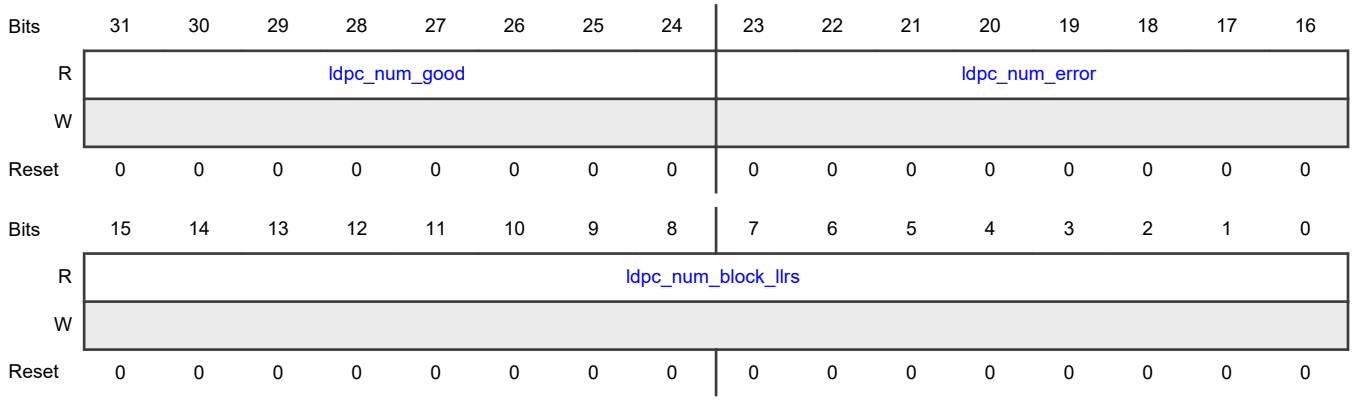
Field	Function
31-16 ldpc_parity_bits_left	ldpc_parity_bits_left The number of parity plus repeat bits that FECU still needs to send out in order to complete the current LDPC block.
15-11 —	- Reserved
10-0 ldpc_message_bits_left	ldpc_message_bits_left The number of input message bits that are still required to complete the current LDPC block.

16.2.29 FECU LDPC decoder block sizes and counts register (FECU_LDPC_DEC_BLOCK)

Offset

Register	Offset
FECU_LDPC_DEC_BLOCK	374h

Diagram



Fields

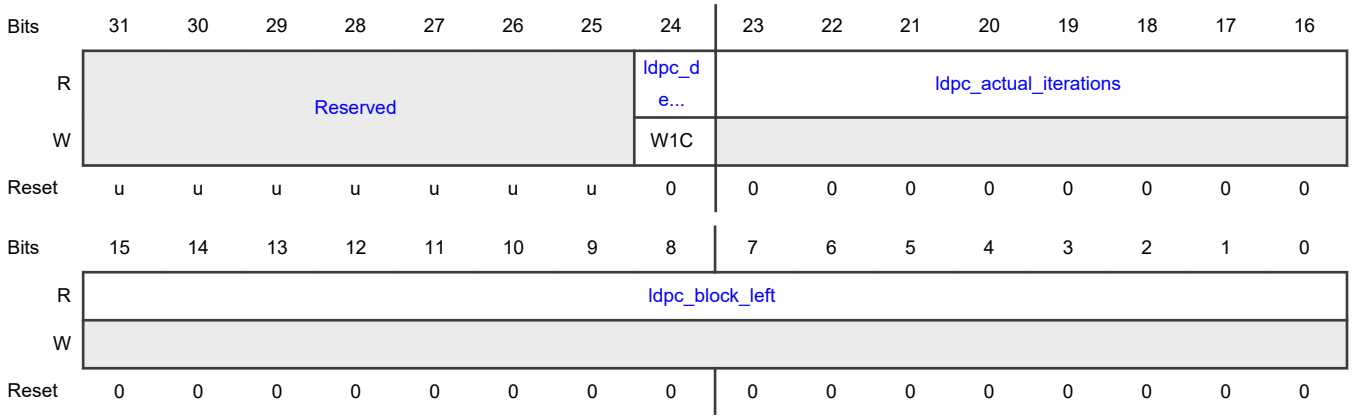
Field	Function
31-24 ldpc_num_good	ldpc_num_good The number of LDPC blocks that the decoder was able to decode, during the last symbol. The decoder is able to decode when its output satisfies the parity check matrix.
23-16 ldpc_num_error	ldpc_num_error The number of LDPC blocks that the decoder failed to decode, during the last symbol. A decoder failure occurs when the LDPC decoder's output does not satisfy the parity check matrix after ldpc_max_iterations number of iterations.
15-0 ldpc_num_block_llrs	ldpc_num_block_llrs The total number of LLRs in the block the LDPC decoder is currently processing.

16.2.30 FECU LDPC decoder status register (FECU_LDPC_DEC_STATUS)

Offset

Register	Offset
FECU_LDPC_DEC_STATUS	378h

Diagram



Fields

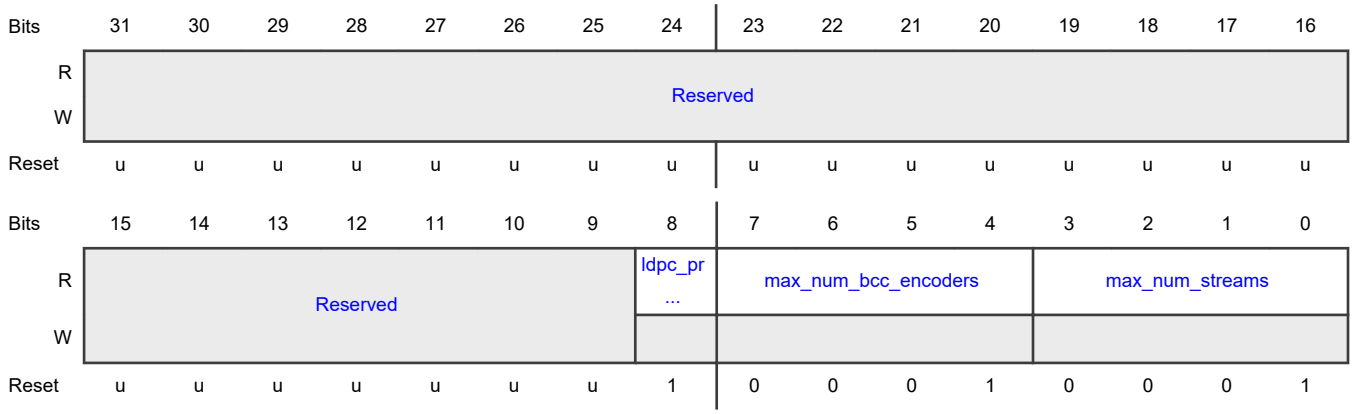
Field	Function
31-25 —	- Reserved
24 ldpc_decode_error	ldpc_decode_error The bit will be set when the LDPC decoder fails to decode a block for ldpc_max_iterations. This bit is cleared by a write to this register with a 1 in this bit position. This bit can be used to send a VCPU go event.
23-16 ldpc_actual_iterations	ldpc_actual_iterations The actual number of iterations used to decode the last LDPC block.
15-0 ldpc_block_left	ldpc_block_left The number of input LLRs that are still required to complete the current LDPC block.

16.2.31 FECU Hardware parameters / capabilities of FECU (FECU_HW_PARAMS)

Offset

Register	Offset
FECU_HW_PARAMS	380h

Diagram



Fields

Field	Function
31-9 —	- Reserved
8 ldpc_present	ldpc_present Set when FECU includes an LDPC encoder and decoder.
7-4 max_num_bcc_encoders	max_num_bcc_encoders The maximum number of BCC encoders the FECU hardware supports that is (1).
3-0 max_num_streams	max_num_streams The maximum number of spatial streams the FECU hardware supports that is (1).

16.2.32 FECU Hardware parameters / capabilities of the LDPC encoder and decoder in FECU (FECU_LDPC_HW_PARAMS)

Offset

Register	Offset
FECU_LDPC_HW_PARAMS	384h

Diagram

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	ldpc_num_dc								ldpc_num_sp							
W																
Reset	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	ldpc_lu_dw								ldpc_io_dw							
W																
Reset	0	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0

Fields

Field	Function
31-24 ldpc_num_dc	ldpc_num_dc Number of decoder cores in the LDPC decoder.
23-16 ldpc_num_sp	ldpc_num_sp Number of sub-matrix processors in a single LDPC decoder core.
15-8 ldpc_lu_dw	ldpc_lu_dw LDPC load / un-load data width. The number of LLRs loaded in the LDPC core every cycle.
7-0 ldpc_io_dw	ldpc_io_dw The number of LLRs loaded into the LDPC decoder per cycle. Also, the number of bits that come out of the LDPC decoder each cycle.

Appendix A
Revision History

A.1 Revision History

This table summarizes changes to this document.

Revision	Date	Change
0	06/2021	Initial NDA release.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors. In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, the Freescale logo, CodeWarrior, Layerscape, QorIQ, QorIQ Qonverge, are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved.

© NXP B.V. 2021. **All rights reserved.**

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 06/2021
Document identifier:
vspa2_ism_16au_la9310RM